**Contents**

**Debugging**

**CI / CD**

[**Git**](#Git)

**Frontend**

[HTML](#HTML)

[CSS](#CSS)

[Bootstrap](#Bootstrap)

[PrimeNg](#PrimeNg)

[PrimeFlex](#PrimeFlex)

[ngX](#ngX)

[Typescript](#Typescript)

[Angular](#Angular)

[Rxjs](#RxJs)

[Loadash](#Loadash)

**Backend**

**Java**

1. [Object Oriented Programming](#OOPs)
2. [Input/Output](#IO)
3. [Miscellenous](#Misc)

**Springboot**

1. [Maven](#MAVEN)
2. Quartz
3. [Spring](#spring)
4. Process workflow
5. [Spring security](#springsecurity)
6. Swagger
7. Mapstruct
8. JPA | Hibernate
9. Spring cloud
10. Logging

[**JPA**](#JPA)

[**Database**](#Database)

machine code is platform-specific and directly executable by the CPU, while bytecode is platform-independent and executed by a virtual machine

Scratch file

Emacs

To remove unnecessary imports in a Java file, Ctrl + Alt + O

To format your code, Ctrl + Alt + L

The **null** check operator is available in Java 14 and later. In earlier versions of Java, you can use the **Optional** class or the **null** check ternary operator (**obj == null ? null : obj.toString()**) to achieve a similar effect.

The version number is typically displayed in the format **x.y.z**, where **x** is the major version, **y** is the minor version, and **z** is the patch version.

1.8 = 8

Java 1.8 is still widely used and supported, although the latest version of Java is currently Java 17, which was released in September 2021.

Both **mounted drives** and **virtual drives** allow you to access a data store located elsewhere as if it were a local drive, using familiar commands and tools. The main difference is that a mounted drive is a connection to a data store that is accessible to the entire system, while a virtual drive is a connection to a data store that is specific to a PowerShell session or script.

**Debugging**

Breakpoint

Step over the next function call

Step into the next function call

Step out of current function

Step

Deactivate breakpoints

Pause on exceptions

Chart

Description automatically generated with low confidence

Element tab

Console tab

Network tab

Source tab

Warn

Info

Debug

Error

**Git** [**Contents**](#contents)

Git Clone

Git revert

Git reset

Git push

Git pull = Git fetch + Git merge

Squash

 In Git, the term squash is used to squash the previous commits into one. You can merge several commits into a single commit with the compelling interactive rebase command.

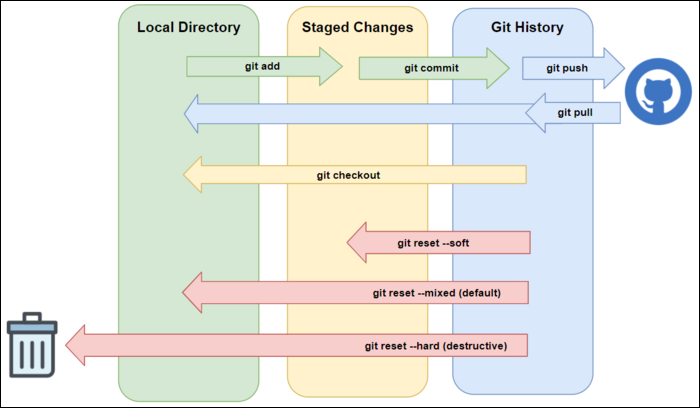
**Step1: Check the commit history**

**git log --oneline**

**Step 2: Choose the commits to squash.**

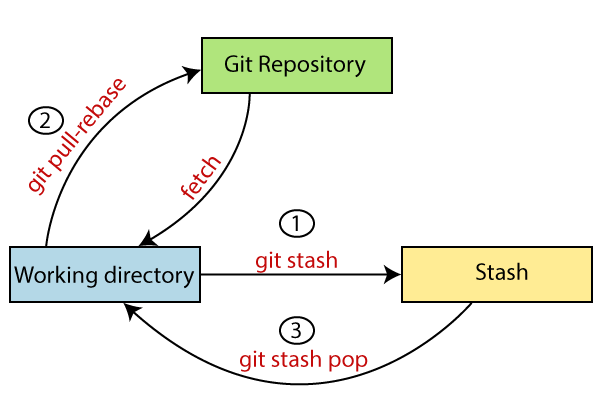
**git rebase -i HEAD ~3**

The above command will open your default text editor and will squash the last three commits



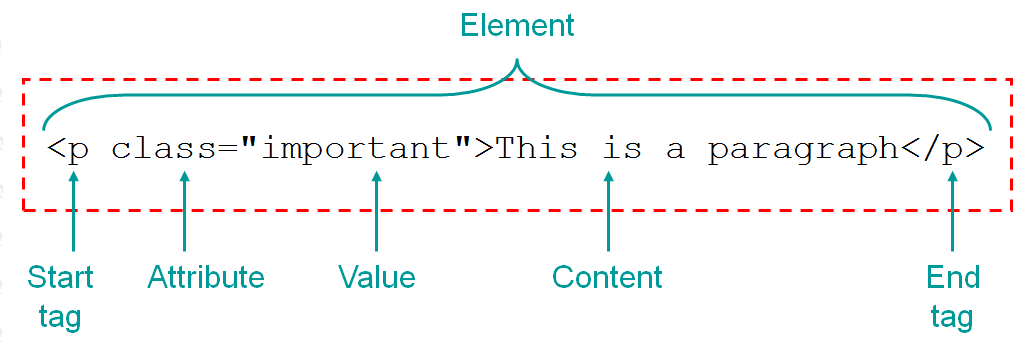
git diff

git diff -staged



Git workflow

**HTML** |[**Contents**](#contents)



When a web page is loaded, the browser creates a **D**ocument **O**bject **M**odel of the page.

The HTML DOM model is constructed as a tree of Objects:

HTML Attributes

* All HTML elements can have **attributes**
* Attributes provide **additional information** about elements
* Attributes are always specified in **the start tag**
* Attributes usually come in name/value pairs like **name="value"**

HTML Block and Inline Elements

Block Elements

* A block-level element always starts on a new line, and the browsers automatically add some space (a margin) before and after the element.

Here are the block-level elements in HTML:

[<div>](https://www.w3schools.com/tags/tag_div.asp)

[<form>](https://www.w3schools.com/tags/tag_form.asp)

<h1>-<h6>

Here are the inline elements in HTML:

[<a>](https://www.w3schools.com/tags/tag_a.asp)

[<button>](https://www.w3schools.com/tags/tag_button.asp)

[<input>](https://www.w3schools.com/tags/tag_input.asp)

[<label>](https://www.w3schools.com/tags/tag_label.asp)

[<span>](https://www.w3schools.com/tags/tag_span.asp)

* There are two display values: block and inline
* A block-level element always starts on a new line and takes up the full width available
* An inline element does not start on a new line and it only takes up as much width as necessary
* The <div> element is a block-level and is often used as a container for other HTML elements
* The <span> element is an inline container used to mark up a part of a text, or a part of a document

Async

HTML DOM

Browser BOM

Web APIs

JSON

JSON WEB API

**CSS** |[**Contents**](#contents)

Margin

padding

Grid

Flexbox

Responsive

* Setting up the viewport
* Using max-width/max-height
* Using media queries
* Using rem/vh/vw units over pixels

**PrimeNg** |[**Contents**](#contents)

**PrimeFlex** |[**Contents**](#contents)

p-field

p-grid

p-col-fixed

pInputText

**flexbox was designed for layout in one dimension - either a row or a column.** Grid was designed for two-dimensional layout - rows, and columns at the same time.PrimeFlex is a CSS utility library featuring various helpers such as a

1. grid system,
2. flexbox,
3. spacing,
4. elevation and more.

**ngX datatable** |[**Contents**](#contents)

Table

Inputs

Outputs

Methods

Internals

Column

Inputs

Modes

Internals

Row Detail

Inputs

Outputs

What does toggle mean in Word?

switch or alternate

**let-\***

**let-\*=”Var”:**

### Offset

Offset classes allow defining a left margin on a column to avoid adding empty columns for spacing.

Ng-content

Parent ---->child

Parent.hmtl

<child> data </child>

Child.html

<ng-content></ng-content>

<ng-template>

<ng-container>

Inline

An inline element does not start on a new line and allocates width as necessary.

Inline Block

Inline block is similar to an inline element but properties like width, height and top bottom paddings/margins are respected.

Flex

Displays the element as a block level flex container.

Inline Flex

Displays the element as an inline level flex container.

OMBP

O offset

M margin

B border

P padding

**Port kill**

netstat -ano | findstr :<PORT>

taskkill /PID <PID> /F

**#To remove registry**

path to intellij  
C:\Users\saroj.sharma\AppData\Roaming\JetBrains\IntelliJIdea2020

rmdir "eval" /s /q  
del "options\other.xml"  
reg delete "HKEY\_CURRENT\_USER\Software\JavaSoft\Prefs\jetbrains\idea" /f

**Typescript** |[**Contents**](#contents)

**Variables**

Put semicolon at the end of every statements !! #Practice

let identifier: type = value;

* let - block scoping
* can re-assign new value to it but cannot re-declare it within the scope

Shadowing

* const – block scoping

- we cannot re-assign any new value to it.

-🡪scope

* Global scope
* Class scope
* Local scope

-🡪hoisting : not hoisted for let and const

-🡪Type Assertion

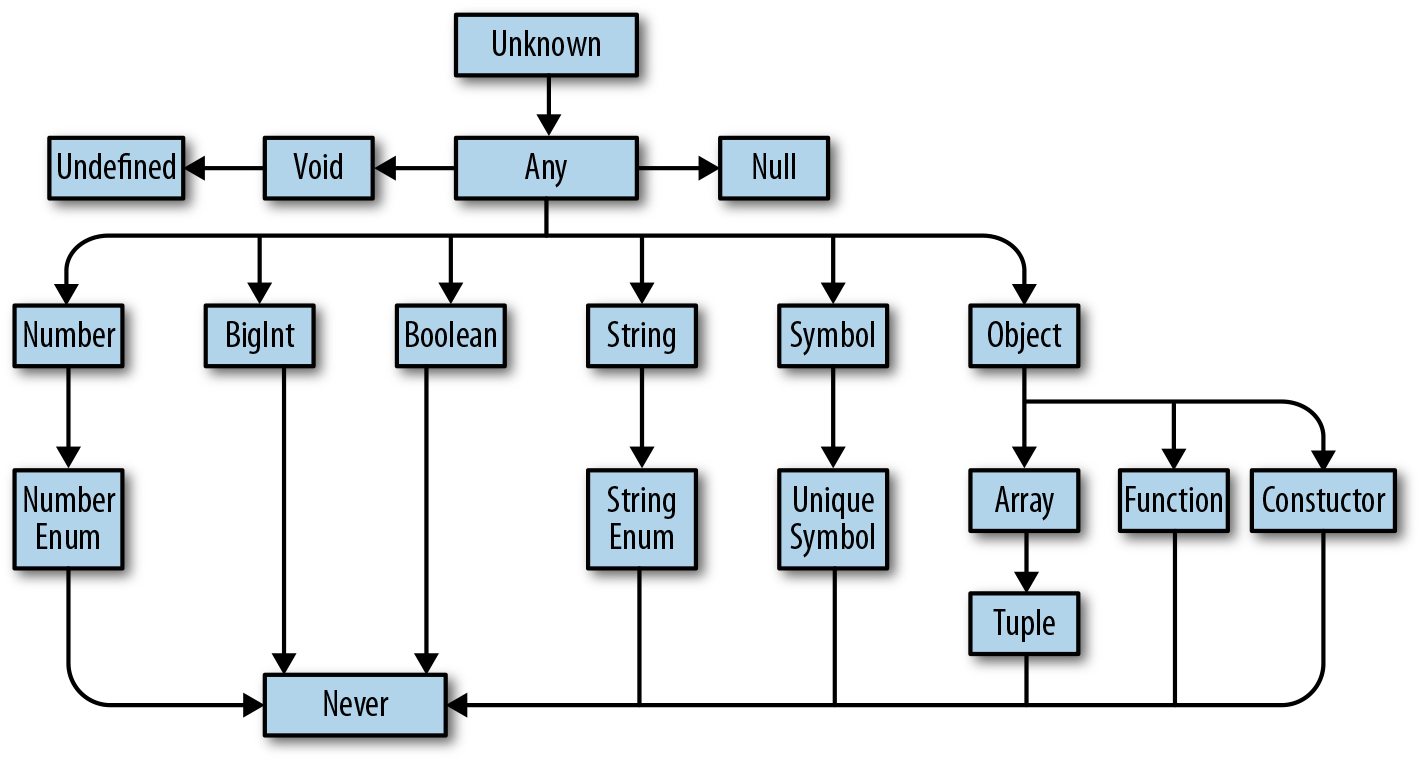
Type assertion works like typecasting, but it does not perform type checking or restructuring of data just like other languages can do like C# and Java.

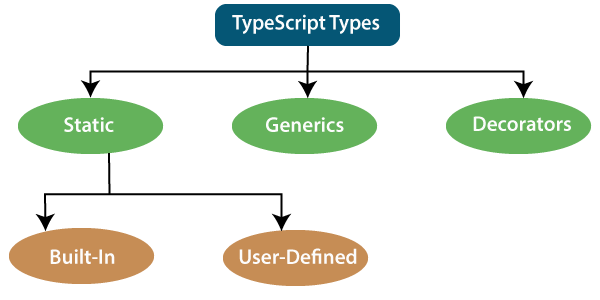
TypeScript provides two ways to do Type Assertion. They are

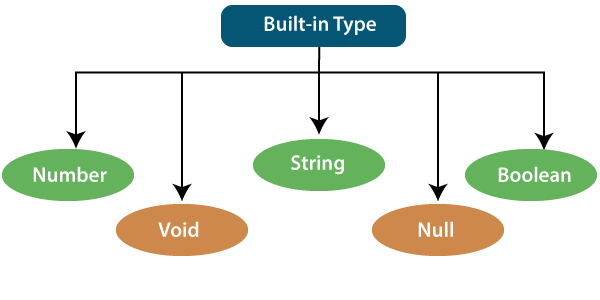
1. Using Angular Bracket <>
2. Using as keyword

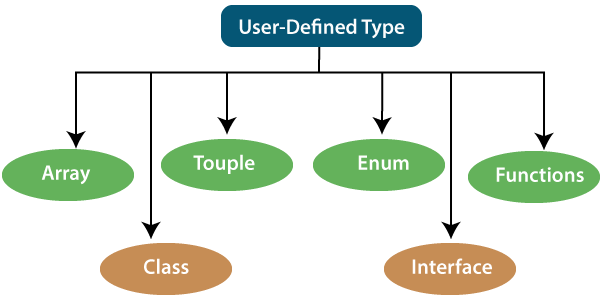
-🡪Inferred Typing let num = 2; //data type inferred as number #Type Inference

**Data Types**









Null | void | Undefined

**Operators**

* Arithmetic operators
* Comparison (Relational) operators

== Is equal to check value

=== Identical(equal and of the same type) check value and type

!= Not equal to

!== Not identical

* Logical operators

&& Logical AND

|| Logical OR

! Logical NOT

* Bitwise operators
* Assignment operators
* Ternary/conditional operator

expression ? expression-1 : expression-2;

* Concatenation operator
* Type Operator

in - It is used to check for the existence of a property on an object.

delete - It is used to delete the properties from the objects.

typeof - It returns the data type of the operand.

instanceof - It is used to check if the object is of a specified type or not.

**Type Annotation**

* Inline type annotation

**Decision Making**

There are various types of Decision making in TypeScript:

* if statement
* if-else statement
* if-else-if ladder
* nested if statement
* switch statement
* There can be N number of cases inside a switch statement.
* The case values must be unique.
* The case values must be constant.
* Each case statement has a break statement at the end of the code. The break statement is optional.
* The switch statement has a default block which is written at the end. The default statement is optional.

**Loops**

We can classify the loops into two types.

1. Idefinite
   1. while loop
   2. do-while loop
2. Definite
   * 1. for loop
     2. for..of loop
     3. for..in loop
3. for loop

for (first expression; second expression; third expression ) {

    // statements to be executed repeatedly

}

1. for..of loop

The for..of loop is used to iterate and access the elements of an array, string, set, map, list, or tuple collection.

for (var val of list) {

   //statements to be executed

}

1. for..in loop

The for..in loop is used with an array, list, or tuple. This loop iterates through a list or collection and returns an index on each iteration

for (var val in list) {

   //statements

}

for (key in object) {  
  // *code block to be executed*  
}

for (variable in array) {  
  code  
}

The **for..in** loop returns a list of indexes on the object being iterated, whereas the **for..of** loop returns a list of values of the object being iterated.

**Function**

When you pass a function as an argument, remember not to user parenthesis. Instead of passing the name of a function as an argument to another function, you can always pass a whole function instead.

Function parameter can be categories into the following:

* Optional Parameter

 The parameters which may or may not receive a value can be appended with a "?" sign to mark them as optional.

* Default Parameter
* Rest Parameter

The rest parameter is used to pass **zero or more** values to a function. We can declare it by prefixing the **three "dot"** characters ('...') before the parameter.

**Rules to follow in rest parameter:**

* Only one rest parameter is allowed in a function.
* It must be an array type.
* It must be the last parameter in a parameter list.
* Named Function with optional parameter

function functionName(identifier1:type, identifier2?:type):returntype {

    //statements

}

* Anonymous Function with variable length arguments

(let, const) functionName =

function(identifier1:type, ...identifier2:type[]):returntype{

    //statements

}

* Arrow function

There is no need to use the **function** keyword

(let, const) functionName =

(identifier1:type, identifier2:type):returntype => {

    //statements

}

**Boolean**

**Numbers**

Converting Variables to Numbers

There are 3 JavaScript methods that can be used to convert variables to numbers:

* The Number() method

Number() can be used to convert variables to numbers:

Number(true);  
Number(false);  
Number("10");  
Number("  10");  
Number("10  ");  
Number(" 10  ");  
Number("10.33");  
Number("10,33");  
Number("10 33");  
Number("John");

If the number cannot be converted, Nan (Not a Number) is returned.

Number() can also convert a date to a number.

* The parseInt() method
* The parseFloat() method

**Strings**

There are three ways in which we can create a string

* single quoted strings
* Double quoted strings
* Back-ticks strings

type variableName = "value1" | "value2" | "value3"; // up to N number of values

Method

Search

* String indexOf()
* String lastIndexOf()
* String startsWith()
* String endsWith()

Templates

Template Literals use back-ticks (``) rather than the quotes ("") to define a string:

Synonyms:

* Template Literals
* Template Strings
* String Templates
* Back-Tics Syntax

**void**

**null**

**Undefined**

**Array**

let identifier: any = []

let identifier: primitiveType[] = []

let identifier: userdefinedType[] = []

**Optional Chaining Operator (?.)**

The ?. operator returns undefined if an object is undefined or null (instead of throwing an error).

obj.val?.prop

obj.val?.[expr]

obj.func?.(args)

Nullish coalescing operator

The **nullish coalescing (**??**)** operator is a logical operator that returns its right-hand side operand when its left-hand side operand is null or undefined, and otherwise returns its left-hand side operand.

const foo = { someFooProp: "hi" };

console.log(foo.someFooProp?.toUpperCase() ?? "not available"); // "HI"

console.log(foo.someBarProp?.toUpperCase() ?? "not available"); // "not available"

## Spread operator (…arg)

The spread operator is used to initialize arrays and objects from another array or object. We can also use it for object de-structuring. It is a part of the ES 6 version.

**Method**

* forEach()

array.forEach(callback[, thisObject]);

The forEach() method executes the provided **callback** once for each element present in the array in **ascending order**.

It only **works** with arrays

Arrayobj.forEach(callbackfun)

Array map set

* splice(start:number, deleteCount:number, …items:number[])

**Tuple**

**Union**

TypeScript can combine one or two different types of data (i.e., number, string, etc.) in a single type, which is called a union type.

(type1 | type2 | type3 | ........ | type-n)

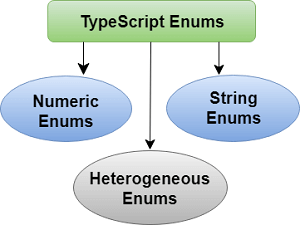
**Map**

 It allows us to store data in a **key-value pair**

**Set**

 It allows us to store **distinct data** (each value occur only once) into the **List** similar to other programming languages.

**Enum**



It is used to define the set of **named constants**, i.e., a collection of related values.

**Object**

An object is an instance which contains set of key value pairs. The values can be scalar values or functions or even array of other objects.

let object\_name = {

key1: “value1”, //scalar value

key2: “value”,

key3: function() {

//functions

},

key4:[“content1”, “content2”] //collection

};

Types of objects

* User defined object
* Native object
* Host object
* Documents object

let obj = {name:'saroj', roll:23, location:'waling'}

let obj:{name: String, roll:number} = {name: "saroj", roll:12}

const obj:userdefinedType = {key: value}

interface Info{

    name:string;

    roll:number;}

let obj:Info ={name:'saroj',roll:23}

let obj:Info[] = [{name:'saroj', roll:23}, { name:'samrat', roll:20}]

console.log(obj[0]['name'])

console.log(obj[0].name)

class om

{

    name:string;

    constructor(name:string)

    {

        this.name = name;

    }

}

let obj2:om = {name:'saroj'}

* can pass object to function as an argument

**Class**

## Creating an object of class

let object\_name = new class\_name(parameter)

Ojbect Initialization

1. By reference variable
2. By method
3. By constructor

constructor()

**Inheritance**

TypeScript supports only single and multilevel inheritance. It doesn't support multiple, hierarchical, and hybrid inheritance.

:extends

**Interface**

interface interface\_name {

          // variables' declaration

          // methods' declaration

}

Use of Interface

We can use the interface for the following things:

* Validating the specific structure of properties
* Objects passed as parameters
* Objects returned from functions.

 We can use the "**extends**" keyword to implement inheritance among interfaces.

**Generics**

In TypeScript, we can create

1. generic functions,
2. generic classes,
3. generic methods,
4. generic interfaces.

### **Advantage of Generics**

There are mainly three advantages of generics. They are as follows:

1. **Type-safety:** We can hold only a single type of objects in generics. It doesn't allow to store other objects.
2. **Typecasting is not required:** There is no need to typecast the object.
3. **Compile-Time Checking:** It is checked at compile time so the problem will not occur at runtime.

**Dates**

**Math Object**

The JavaScript Math object allows you to perform mathematical tasks on numbers.

**Namespaces**

**Map filter reduce forEach**

let a = [1,2,3,4,4]

let b = a.map(obj => obj\*2)

console.log(b)

[2, 4, 6, 8, 8]

let a = [1,2,3,4,4]

let b = a.map((obj) => {obj\*2})

console.log(b)

[undefined, undefined, undefined, undefined, undefined]

let a = [1,2,3,4,4]

let b = a.map((obj) => {return obj\*2})

console.log(b)

[2, 4, 6, 8, 8]

For multiple statement return is required

if there is only one statement, you may or may not use return keyword. You must use return keyword when it contains multiple statements.

**Loadash** |[**Contents**](#contents)

**Array**

**Collection**

**Date**

**Function**

**Lang**

**Math**

**Number**

**Object**

**Seq**

**String**

**Util**

**Properties**

**Methods**

**Angular |** [**Contents**](#contents)

Development Environment setup

Install Nodejs

node - - version

npm - - version

Install Angular CLI

npm install -global @angular/cli@latest globally/latest

npm install @angular/cli@latest

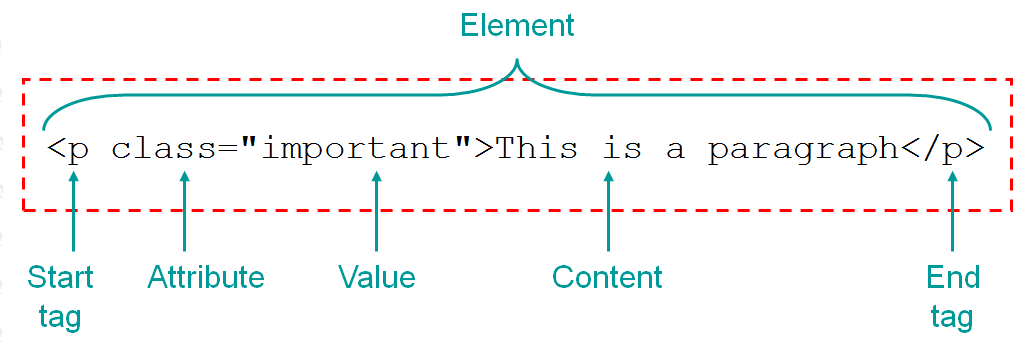
ng --version



**Data binding**

Use to bind the data from component to the template. Data binding works with properties of DOM elements, components, and directives, not HTML attributes. Remember that HTML attributes and DOM properties are different things, even when they have the same name. In Angular, the only role of HTML attributes is to initialize element and directive state. When you write a data binding, you're dealing exclusively with the DOM properties and events of the target object. The HTML attribute value specifies the initial value; the DOM value property is the current value.

HTML attributes vs DOM properties



1. One way Data binding
2. Component to view
3. Interpolation Binding {{}}

Component ---------------------> View

1. Property Binding []

HTML,

Component

Directive

view <--------------------- component

<element [element\_property]=”component\_property or method or exp”></element>

<element [element\_property]<----------”component\_property or method”></element>

[Left child]=”Right parent”

1. Attribute Binding
2. Style Binding
3. Class Binding
4. View to component
5. Event Binding ()

View -----------------------> Component

<element (event)=”view\_method($event)”></element>

<element (event)--------------->”view\_method()”></element>

Event -----emit value ----> $event

1. Two way Data Binding

The brackets, [], cause Angular to evaluate the right-hand side of the assignment as a dynamic expression. Without the brackets, Angular treats the right-hand side as a string literal and sets the property to that static value.

one of the limitations of binding is that dynamic children (ng-content) are not bindable. With the help of template outlet, it is possible to establish a binding-like communication channel which allow data to flow from the parent to its children.

<element [(ngModel)]=”component-property or method”></element>

If we need to inspect the properties of the associated FormControl (like validity state), export the directive into the local template variable using the ngModel as the key (ex: #myVar=” ngModel”).

Events

* Window event
* Form event
* Keyboard events
* Mouse events
* Drag Events
* Clipboard Event
* Media Event

Some useful events

1. (input)="myFunction()"
2. (change)="myFunction()"
3. (focusin)="myFunction()"
4. (focusout)="myFunction()"
5. (keyup)="myFunction()"
6. (keypress)="myFunction()"
7. (keydown)="myFunction()"
8. (click)="myFunction()"
9. (dblclick)="myFunction()"
10. (submit)="myFunction()"
11. (blur)="myFunction()"
12. (scroll)="myFunction()"
13. (cut)="myFunction()"
14. (copy)="myFunction()"
15. (paste)="myFunction()"
16. (mouseup)="myFunction()"
17. (mousedown)="myFunction()"
18. (mouseenter)="myFunction()"
19. (drag)="myFunction()"
20. (drop)="myFunction()"
21. (dragover)="myFunction()"

**Directives**

* Directives can be used as a
  + CSS class
  + HTML element
  + HTML attribute

import { [Directive](https://angular.io/api/core/Directive) } from '@angular/core';

@[Directive](https://angular.io/api/core/Directive)({

selector: '[appHighlight]'

})

export class HighlightDirective {

}

Structural directive

Change the DOM layout by adding and removing DOM elements.

Attribute directive

Change the appearance or behavior of an element, component, or another directive.

Component directive

Used with a template. This type of directive is the most common directive type.

Custom directive

ng g directive highlight

Use to include logic as well as enable creation of complex HTML DOM elements

|  |  |  |  |
| --- | --- | --- | --- |
| Structural directive | Attribute directive | Component directive | Custom |
| \*ngIf | [ngStyle] | Components |  |
| \*ngFor | [ngClass] |  |  |
| ngSwitch | [(ngModel)] |  |  |
| <ng-template> |  |  |  |
| <ng-content> |  |  |  |

Structural ngIf | ngFor | ngSwitch

ngIf

\*ngIf =”typescript\_expression” | expression must be resolved into boolean

<element \*ngIf=”typescript\_expression;else reference\_variable> </element>

ngFor

\*ngFor=”let item of items; let i = index”

ngSwitch

----

ng-template

<ng-template #reference\_variable> </ng-template>

ng-content

<app-component> <p>Hello</p></app-component>

<ng-content></ng-content>

ng-container

Attribute

.item(class){

Property1:value,

Property2:value }

ngStyle

The ngStyle directives is used to set a CSS style dynamically for an HTML element based on a given typescript expression.

Adds and removes a set of HTML styles. An expression which returns an object where the keys are CSS properties, and the values are CSS values.

[ngStyle] = {key:value}

<element [ngStyle]=”{css\_property:css\_value or typescript\_expression}”></element>

Typescript\_expression resolved into css\_value

ngClass

Adds and removes a set of CSS classes.

<element [ngClass]=”{css\_class:typescript\_expression}”></element>

Typescript\_expression must resolved into boolean.

ngModel

Adds two-way data binding to an HTML form element.

Custom Attribute directive

Renderer2

Custom structural directive

### **Element directives**

|  |  |
| --- | --- |
| <ng-container> | A special element that can hold structural directives without adding new elements to the DOM. |
| <ng-content> | The <ng-content> element specifies where to project content inside a component template. |
| <ng-template> | Angular's <ng-template> element defines a template that is not rendered by default. |

@HostListner | @HostBinding

HostBinding and HostListener are written in directives and the other ones (...) and [..] are written inside templates (of components).

Binding Directive Properties

Conditional directive

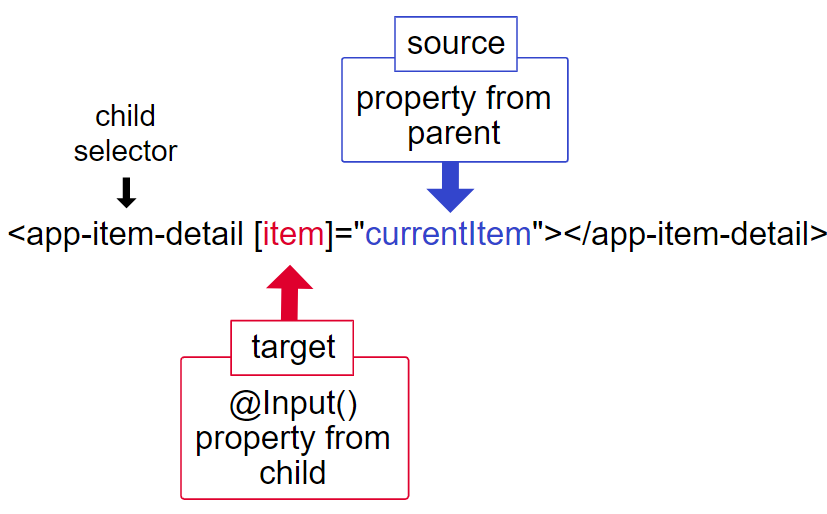
Custom appClass directive

Custom appStyle directive

**Components Interactions**

1. Parent – Child interaction using @Input & @Output properties
2. Parent – Child Interaction using @ViewChild & ViewChildren
3. Bidirectional parent – child interaction through a service

@Input() decorator

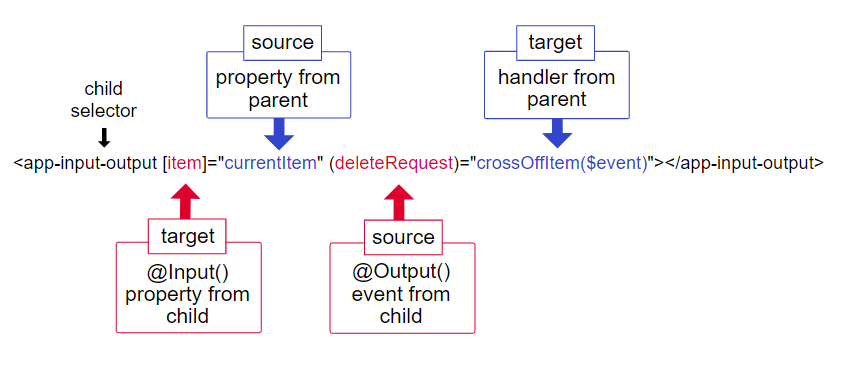


* Also call it custom property binding because here we bind the custom properties of child component class with the property or method of parent component class.

@Input(‘alias’) variable\_name

[alias]=”parent\_property or method”

@Output decorator



* Also call it custom event binding

@Output

variable:EventEmitter<Type> = new EventEmitter<Type>();

method(){

this.variable.emit(value);

}

Call-------->method() --------> event raised--->emit value of <Type> type

$event is a special variable and holds whatever was emitted by EventEmitter

**Template Reference variable**

**Template Reference Variable**in angular is used to access all the properties of any element inside DOM. A template variable can refer to the following:

* a DOM element within a template
* a directive or component
* a TemplateRef from an ng-template
* a web component
* If you declare the variable on a standard HTML tag, the variable refers to the element.
* If you declare the variable on an <ng-template> element, the variable refers to a TemplateRef instance which represents the template.

<element #reference\_variable></element>

type: HTMLInputElement - input tag

type: ComponentName - component tag

<element #reference\_variable (event)=”method(reference\_variable)”></element>

-pass reference variable to the component by calling method

Variable specifying a name

#var=”ngForm”

#var="ngModel",

<input #val>

<button (click)=”onClick(val.value)”>Click me</button>

Template variable scope

**ViewChild | ViewChildren**

We use the ViewChild or ViewChildren to Query and get the reference of any DOM element in the Component. The DOM element can be an HTML element, Child Component, or directive, etc.

#### Some time we might want to access the html element, child component before calling a method in the component.

@ViewChild

 Sometimes, the AppComponent *might* need references to the multiple elements that it contains inside its template, in order to mediate their interaction.

* Accessing template of same component
* Accessing template of child component

<tag #selector> </tag>

@ViewChild(‘selector’) variableName:type

@ViewChild(‘selector’) variableName:ElementRef;

@ViewChild(ComponentName, {static:true}) variableName:Component\_type;

ElementRef

The ElementRef class is a simple wrapper for native elements, which are usually DOM elements in the browser. It provides a way to access the nativeElement object, exposing all the methods and properties of the associated native element.

Each ElementRef has a nativeElement property and this native element property is the property which stores the actual html element

@ViewChildren

Returns the specified elements or directives from the view DOM as QueryList. The QueryList is initialized only before the ngAfterViewInit lifecycle hook, therefore, is available only from this point.

**ContentChild | ContentChildren**

The ContentChild & ContentChildren is very similar to the ViewChild & ViewChildren. We use the ViewChild or ViewChildren to Query and get the reference of any DOM element in the Component. The DOM element can be an HTML element, Child Component, or directive, etc. But We cannot use the ViewChild or ViewChildren to get the reference to the template inserted using the Content projection.

@ContentChild

@ContentChildren

**ViewEncapsulation**

In Angular, a component's styles can be encapsulated within the component's host element so that they don't affect the rest of the application.

The Component decorator provides the encapsulation option which can be used to control how the encapsulation is applied on a per component basis.

Three types of view encapsulation in Angular

* encapsulation:ViewEncapsulation.None
* encapsulation:ViewEncapsulation.Emulated #default
* encapsulation:ViewEncapsulation.ShadowDOM

Change Detection: The process of updating the view (DOM) when the data has changed

**Angular life cycle hooks**

Parent Constructor

Child constructor ascending order as used in HTML

Parent OnInit

Child OnInit ascending order as used in HTML

Constructor | parent--child

ngOnChanges | parent -- child

OnInit | parent -- child

1. When the angular application starts it first creates and renders the root component then it creates and renders its childrens & their children. It forms a tree of components.
2. Once Angular loads the components, it starts the process of rendering the view. To do that, it needs to check the input properties, evaluate the data bindings & expressions, render the projected content etc.

**Change detection cycle**

Change detection is the mechanism by which angular keeps the template in sync with the component.

1. Projected content: Projected content is that HTML content which replaces the <ng-content> directive in child component.
2. Input bound properties: These are those properties of a component class which is decorated with @Input decorator.

Global variable

Constructor

* Suitable for Dependency Injection.

ngOnChanges

* At the beginning and multiple time when a component’s input property change
* Component must have Input properties
* Work with arguments

ngOnInit

* Angular raises the ngOnInit hook, after it creates the component and updates its input properties.
* This hook is fired only once and immediately after its creation
* This is a perfect place where you want to add any initialization logic for your component.
* Here we have access to every input property of the component. You can use them in http get requests to get the data from backend server or run some initialization logic etc.
* But remember that, by the time ngOnInit get’s called, none of the components or projected content are available at this point.Hence any properties we decorate with @ViewChild, @ViewChildren, @ContentChild & @ContentChildren willnot be available to use.

ngDoCheck

Execute multiple times

Typically, you should not use both [DoCheck](https://angular.io/api/core/DoCheck) and [OnChanges](https://angular.io/api/core/OnChanges) to respond to changes on the same input.

ngAfterContentInit

-projected ngcontent

ngAfterContentChecked

ngAfterViewInit

After execution of component view

ngAfterViewChecked

ngOnDestroy

* If the component is destroyed, Angular invokes ngOnDestroy.

**Form**

**Services**

ng g s services/authservice

Services are good for tasks such as fetching data from the server, validating user input, or logging directly to the console. By defining such processing tasks in an *injectable service class*, you make those tasks available to any component.

Service can have 2 scopes.

If service is declared on your module, you have same instance shared for all, this means service will be constructed when the first component/directive/service/Pipe who needs it will be created. Then it will be destroyed when Module itself will be destroyed (most of the time when page is unloaded)

if the service is declared on Component/Directive/Pipe, then 1 instance will be created each time when Component/Directive/Pipe will be created and destroyed when related Component/Directive/Pipe will be destroyed.

Services and Dependency Injection

The @Injectable() decorator defines a class as a service in Angular and allows Angular to inject it into a component as a dependency. Likewise, the @Injectable() decorator indicates that a component, class, pipe, or NgModule has a dependency on a service.

* An injector creates dependencies and maintains a container of dependency instances that it reuses, if possible.
* A provider is an object that tells an injector how to obtain or create a dependency
* A dependency doesn't have to be a service —it could be a function, for example, or a value

When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector makes one use the registered provider and adds it to the injector before returning the service to Angular.

You register providers in the metadata of the service (in the @Injectable() decorator), or in the @NgModule() or @Component() metadata

Hierarchical Injection

* By default, the Angular CLI command [ng generate service](https://angular.io/cli/generate) registers a provider with the root injector for your service by including provider metadata in the @[Injectable](https://angular.io/api/core/Injectable)() decorator.

@[Injectable](https://angular.io/api/core/Injectable)({ providedIn: 'root', })

* When you register a provider with a [specific NgModule](https://angular.io/guide/architecture-modules), the same instance of a service is available to all components in that NgModule.

@[NgModule](https://angular.io/api/core/NgModule)({ providers: [ BackendService, Logger ], … })

* When you register a provider at the component level, you get a new instance of the service with each new instance of that component.

@[Component](https://angular.io/api/core/Component)({ selector: 'app-hero-list', templateUrl: './hero-list.component.html', providers: [ HeroService ] })

Creating Data service

Injecting Service into another service

Should have @Injectable() decorator on receiving service

Components Interaction with services

Your components need to [use a single instance of the service](https://stackoverflow.com/questions/36198785/how-do-i-create-a-singleton-service-in-angular-2), so make sure it's provided at the root level.

#Not completed !!!

**Pipe**

Async pipe

**RxJs** |[**Contents**](#contents)

Observable

An observable is a function that creates an observer and attaches it to the source where values are expected, for example, clicks, mouse events from a DOM element or an Http request, etc.

Observer

It is an object with next(), error() and complete() methods, that will get called when there is interaction to the with the observable i.e., the source interacts for an example button click, Http request, etc.

Subscription

When the observable is created, to execute the observable we need to subscribe to it. It can also be used to cancel the execution.

Operators

An operator is a pure function that takes in observable as input and the output is also an observable.

Subject

A subject is an observable that can multicast i.e., talk to many observers. Consider a button with an event listener, the function attached to the event using addlistener is called every time the user clicks on the button similar functionality goes for subject too.

Schedulers

A scheduler controls the execution of when the subscription has to start and notified.

[of](https://rxjs.dev/api/index/function/of)(1, 2, 3) .pipe([first](https://rxjs.dev/api/index/function/first)()) .subscribe((v) => console.log(`value: ${v}`)

Observables

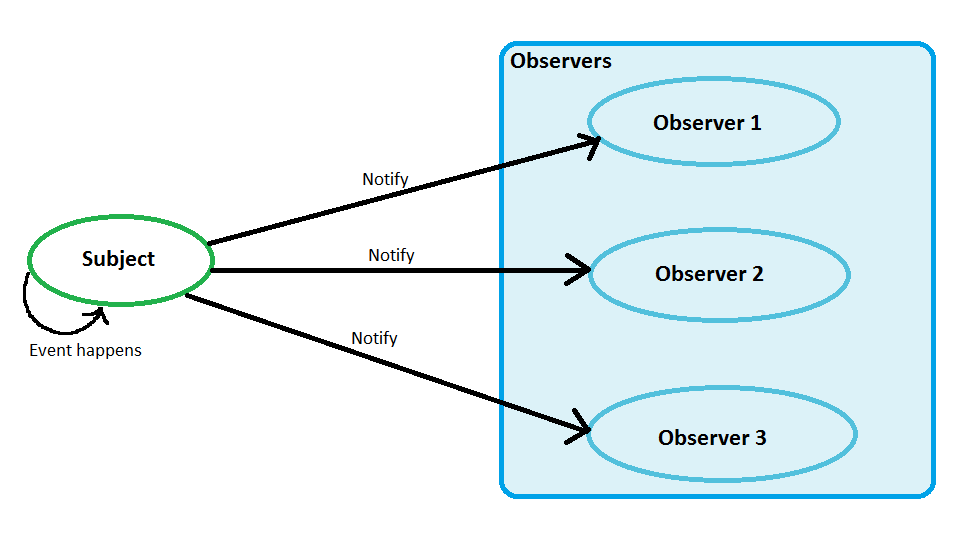
* Create Observable
* Subscribe Observable
* Execute Observable

Satellite Data Types (Observers, Schedulers, Subjects)

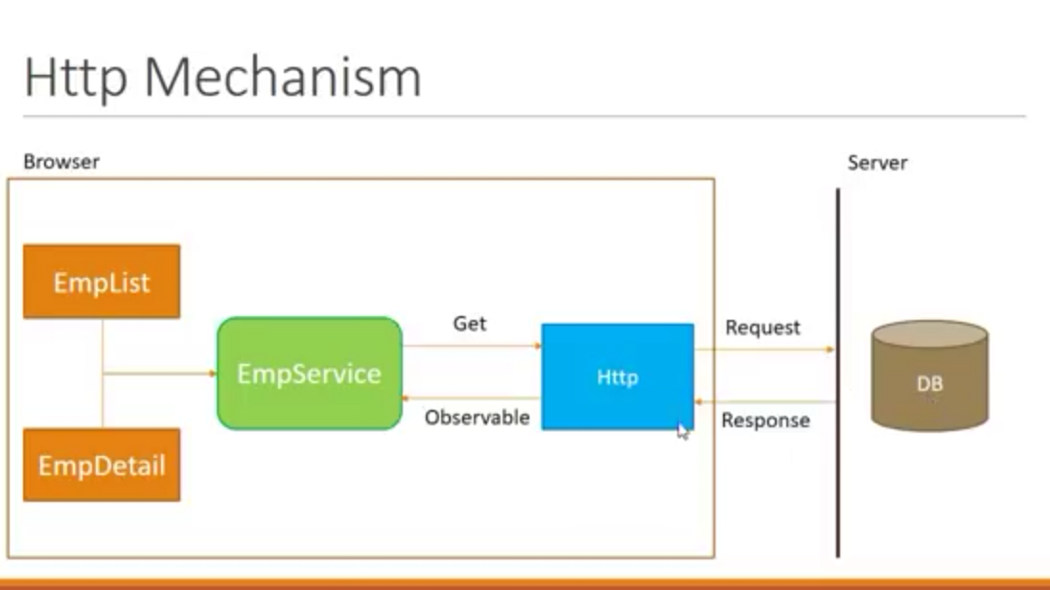
Operators (map, filter, reduce)

* Observer can implement three methods:
  + next(): called whenever a new value is emitted
  + error(): called whenever an observable throws an error
  + complete(): called whenever an observable is done
* Some observable may never finish (click listener).

Subject



**HTTP**



1. HTTP verbs (GET, POST, PUT, PATCH, DELETE) are used to interact with web servers through URLs, and headers can be added to provide metadata for the request.
2. Query parameters can be added to URLs to pass additional information or data to a web server.
3. The HttpClient service is used to send HTTP requests and receive responses, and requires the HttpClientModule to be imported.
4. The HttpClient service offers features such as requesting typed response objects, streamlined error handling, testability, and request/response interception.
5. To specify the response object type, an interface should be defined with the required properties, and used instead of a class since the response is a plain object that cannot be automatically converted to an instance of a class.

**Requesting data from a server**

1. HttpClient.get() is used to fetch data from a server asynchronously by sending an HTTP request and returning an Observable that emits the requested data when the response is received.
2. The observe option specifies how much of the response to return, with possible values of 'body', 'events', or 'response'.
3. The responseType option specifies the format in which to return data, with possible values of 'arraybuffer', 'blob', 'json', or 'text'.
4. The params property is used to configure a request with HTTP URL parameters, which can be specified as an object with string or number values.
5. The reportProgress option can be set to true to listen for progress events when transferring large amounts of data.
6. The HttpClient service can be used to post-process the data, add error handling, and add retry logic to the HTTP requests.
7. To begin the HTTP request for any HttpClient method, you need to call subscribe() on the observable the method returns.
8. It is important to unsubscribe from an observable when a component is destroyed to avoid memory leaks.
9. You can structure your HttpClient request to declare the type of the response object, which makes it easier and more obvious to consume the output.
10. Specifying the response type acts as a type assertion at compile time, which can catch errors earlier in the development process.
11. The type of observable response returned by the get(), post(), and delete() methods varies based on the observe and responseType options passed to the method call.

Path Variables:

const id = 123;

this.http.get(`https://example.com/api/items/${id}`).subscribe((response) => {

console.log(response);

});

In this example, we are sending a GET request to **https://example.com/api/items/123**, where 123 is the value of the **id** variable.

Query Parameters:

const params = { search: 'query', sort: 'asc' };

this.http.get('https://example.com/api/items', { params }).subscribe((response) => {

console.log(response);

});

Query parameters are used for optional parameters, meaning that they are not required to fulfill the request.

Request Body:

const headers = { 'Content-Type': 'application/json' };

const body = { name: 'John', age: 30 };

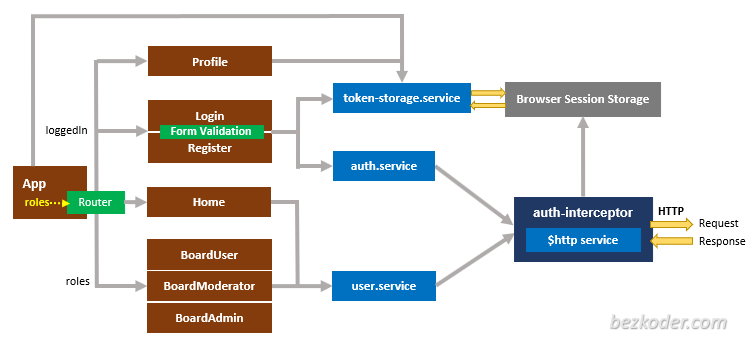
this.http.post('https://example.com/api/items', body, { headers }).subscribe((response) => {

console.log(response);

});

Route Parameter:

HTTP Interceptor



1. To use HttpInterceptors for the entire app, import HttpClientModule only in AppModule and add interceptors to the root application injector.
2. When HttpClient service is used to make an HTTP request, the Interceptor's intercept() method is called, passing a reference to the httpRequest object.
3. The intercept() method can inspect and modify the request, and then calls next.handle to pass the updated request back to the application.
4. Interceptors need to be registered as a multi-provider since there can be multiple interceptors running within an application.
5. Interceptors can be used for authentication and authorization in the application.
6. The original request cannot be modified, so changes need to be made by cloning the original request.

* Modify HTTP Headers
* Modifying the request body
* Set authentication/authorization token
* Mock the backend API
* Modify the response
* Error handling

1. Interceptors can be used to intercept HTTP requests and responses and perform various tasks like adding headers, handling errors, and modifying request or response.
2. Interceptors are registered in the providers array of the app.module.ts file or a specific module.
3. Angular allows the use of multiple interceptors to handle HTTP requests and responses. The order of execution for interceptors is determined by their order in the providers array. The response interceptors are executed in reverse order.
4. Interceptors are applied globally by default, but you can specify specific routes or requests to exclude or include interceptors.
5. When intercepting requests, make sure to clone the original request to avoid modifying it directly.
6. Interceptors can modify the request body, headers, or URL.
7. Interceptors can transform the response body or headers.
8. To handle errors, interceptors can modify the error response, re-throw the error, or return a new error response.
9. Interceptors can be used for various use cases, including authentication, logging, caching, and more.
10. Interceptors can be unit tested using a mock HTTP client and the TestBedTop of Form

Bottom of Form

**Router**

In Angular, there are several types of routes that can be defined in an application.

1. Root Route: This is the top-level or base route that represents the entry point of the application.
2. Child Route: A child route is a route that is a part of another route. It is usually defined as a sub-route of a parent route and is used to display additional content related to the parent route.
3. Current Route: The current route is the route that is currently active or being displayed in the application.
4. Wildcard Route: This is a catch-all route that is used to handle any undefined or unknown routes that may be requested by the user.
5. Redirect Route: A redirect route is used to redirect the user from one route to another route. It is typically used when a user requests a route that has been changed or no longer exists.
6. Lazy-Loaded Route: A lazy-loaded route is a route that is loaded only when it is needed. This can improve the performance of the application by reducing the initial load time and improving the overall user experience.

Lazy loading is a technique used to defer the loading of a module or component until it is needed. In the context of routing in Angular, this means that a module or component is loaded only when a particular route is accessed, rather than being loaded upfront when the application is first loaded.

**Angular Guard**

Angular guards are used to protect routes and prevent unauthorized access to certain parts of your application. They can be used to implement both authentication and authorization in your Angular application.

There are four types of guards in Angular:

1. CanActivate: This guard is used to prevent access to a route. It is used to check if a user is authorized to access a specific route.

canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): This method is used to determine if a user can activate a particular route. The ActivatedRouteSnapshot parameter contains information about the route to be activated, while the RouterStateSnapshot parameter contains information about the current router state.

1. CanActivateChild: This guard is similar to CanActivate, but it applies to child routes. It is used to check if a user is authorized to access a child route of a specific route.

canActivateChild(childRoute: ActivatedRouteSnapshot, state: RouterStateSnapshot): This method is used to determine if a user can activate the child routes of a particular route. The ActivatedRouteSnapshot parameter contains information about the child route to be activated, while the RouterStateSnapshot parameter contains information about the current router state.

1. CanDeactivate: This guard is used to prevent a user from leaving a route. It is used to check if a user has unsaved changes before leaving a route.

canDeactivate(component: any, currentRoute: ActivatedRouteSnapshot, currentState: RouterStateSnapshot, nextState?: RouterStateSnapshot): This method is used to determine if a user can deactivate a particular route. The component parameter is a reference to the component being deactivated, while the ActivatedRouteSnapshot and RouterStateSnapshot parameters contain information about the current route and router state, respectively. The optional nextState parameter contains information about the next router state.

1. CanLoad: This guard is used to prevent a user from loading a feature module. It is used to check if a user is authorized to load a feature module.

canLoad(route: Route, segments: UrlSegment[]): boolean | Observable<boolean> | Promise<boolean>: This method is used to determine if a user can load a particular route lazily (i.e., when the user navigates to the route for the first time). The Route parameter contains information about the route to be loaded, while the UrlSegment[] parameter contains information about the current URL segments. The method can return a boolean value, an observable of a boolean value, or a promise that resolves to a boolean value.

Yes, that's correct. AuthGuard, CanActivateChild, and CanLoad are all classes that implement the CanActivate interface, but they have different use cases and provide different types of route guarding functionality.

1. AuthGuard is used to guard routes based on user authentication. It typically checks whether the user is logged in and has a valid session before allowing access to protected routes.
2. CanActivateChild is used to guard child routes of a parent route. It implements the CanActivate interface and provides a way to define custom logic for guarding child routes.
3. CanLoad is used to guard lazy-loaded modules, which are loaded asynchronously when the user navigates to a route that requires them. It implements the CanActivate interface and provides a way to define custom logic for guarding lazy-loaded modules.

The CanActivate interface has a single method, canActivate(), which takes two arguments: the current activated route and the current router state. This method returns a boolean or a Promise or an Observable of a boolean that determines whether the user is allowed to activate the requested route.

For example, consider the following URL: https://example.com/products/123/details?view=compact. The state of the route for this URL would include:

1. The current URL: https://example.com/products/123/details?view=compact
2. The activated route: products/123/details
3. Query parameters: view=compact

The state of the route can be accessed and manipulated using the Angular Router service.

The names of methods defined within an interface in Java can be the same as the name of the interface itself.

public interface CanActivate {

public boolean CanActivate(route, state);

}

The method is used to determine whether a user is allowed to activate a particular route.

The CanActivate interface is used to determine whether the route should be activated based on the current user's authentication and authorization status. If the canActivate() method returns true, the route is allowed to be activated, and the associated component is displayed. If the method returns false, the route is not activated, and the user is redirected to a different page or shown an error message.

CanActivateChild is another interface in Angular's router library that extends the CanActivate interface. It allows you to implement a guard that determines whether a child route can be activated.It defines a single method canActivateChild().

CanActivateChild ----> canActivate, canActivateChild()

ActivateChild ---------> canActivate()

CanDeactivate ---------> canDeactivate()

CanLoad -----------------> canLoad()

CanDeactivate is another interface provided by Angular that allows you to control whether a user can navigate away from a particular route. It is used to confirm with the user if they want to leave a page when they have unsaved changes.

**Storage**

Encoding and encryption to save token in cookies and localstorage

Cookies

1. When Spring Boot sends a response with a Set-Cookie header, the cookie is automatically stored in the client's browser.
2. The browser will include the cookie in subsequent requests to the same domain until the cookie expires or is deleted.
3. There is no need to write any code to include the cookie in the request header.
4. The browser will handle it automatically.
5. You may need to set up your server-side code to read the cookie from the request header and use its contents in some way, depending on your requirements.
6. Cookies can be used for various purposes such as storing session information, user preferences, and tracking user behavior.
7. It is important to ensure that cookies are transmitted over a secure connection (HTTPS) and include additional security measures such as CSRF tokens to prevent attacks.

Cookie vs local Storage

1. You can store a JWT in a cookie or local storage, depending on your security requirements.
2. Cookies are automatically included in every HTTP request to the domain that set them, so they are useful for storing information that needs to be sent with every request.
3. Local storage is more secure than cookies against certain types of attacks, but it is not immune to other types of attacks.
4. Use cookies if you need to send the JWT with every HTTP request to your server.
5. Use local storage if you don't need to send the JWT with every request, or if you can use a different authentication mechanism.
6. Ensure that the JWT is encrypted and transmitted over a secure connection (HTTPS).
7. Set a short expiration time for the JWT and include additional security measures, such as CSRF tokens and server-side validation of the JWT signature, to prevent attacks.

Local storage

Working with Local storage

The local storage API provides a couple of methods to store, retrieve, remove, and clear the storage.

### localStorage.setItem

The setItem method allows you to add a value to the local storage. It stores data using a key value pair. For example

localStorage.setItem(“token”, “1234”)

Local storage only stores string values. To store objects or arrays you need to use JSON.stringify to convert them into a string first

localStorage.setItem(“token”, JSON.stringify(obj))

localStorage.getItem

getItem allows you to read back the data saved in local storage using the same key. For example, to fetch the data stored using the key token, you can use the following method:

localStorage.getItem(“token”)

localStorage.removeItem

Once you are done using the data in the local storage, you can remove it using the removeItem method. For example:

localStorage.removeItem(“token”)

localStorage.clear

You can clear all of the data out of the local storage using the clear method to wipe the slate clean.

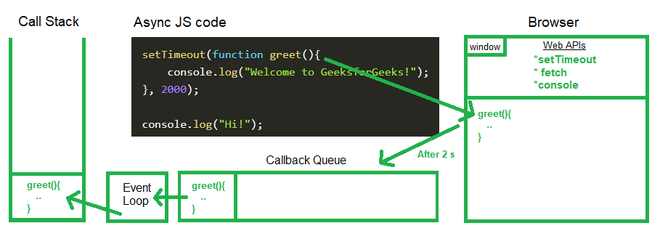
localStorage.clear()

**Note**: Don’t store sensitive information in local storage. It is ok to store JSON web tokens since they are already encrypted. Encrypt the data before storing it in local storage and decrypt it when fetching from local storage. To encrypt and decrypt data in local storage we can use library

Session Storage

**Working of Callbacks**

Callback starvation



**Chrome Dev Tools**

**Testing**

**Debugging**

**Form**

Template driven form

controls

value

valid

Dirty

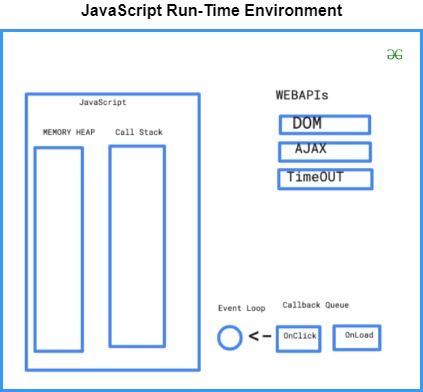
Touched

#reference\_variable = “ngForm”

name=”” ngModel

Form validation

Reactive form



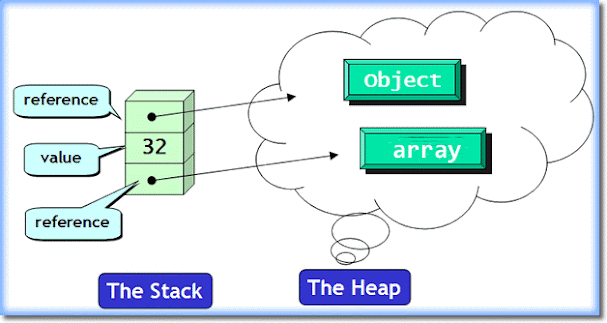
A Basic structure of Java

1. [Simple](https://www.javatpoint.com/features-of-java#Simple)
2. [Object-Oriented](https://www.javatpoint.com/features-of-java#Object-Oriented)
3. [Portable](https://www.javatpoint.com/features-of-java#Portable)
4. [Platform independent](https://www.javatpoint.com/features-of-java#Platform-independent)
5. [Secured](https://www.javatpoint.com/features-of-java#Secured)
6. [Robust](https://www.javatpoint.com/features-of-java#Robust)
7. [Architecture neutral](https://www.javatpoint.com/features-of-java#Architecture-neutral)
8. [Interpreted](https://www.javatpoint.com/features-of-java#Interpreted)
9. [High Performance](https://www.javatpoint.com/features-of-java#High-Performance)
10. [Multithreaded](https://www.javatpoint.com/features-of-java#Multithreaded)
11. [Distributed](https://www.javatpoint.com/features-of-java#Distributed)
12. [Dynamic](https://www.javatpoint.com/features-of-java#Dynamic)



.java file to .class file

Memory model



Stack – LIFO ( Last in First Out)

Heap

### **Q) Can you save a Java source file by another name than the class name?**

Yes, if the class is not public.

How to set path in Java

If you are saving the Java source file inside the JDK/bin directory, the path is not required to be set because all the tools will be available in the current directory.

However, if you have your Java file outside the JDK/bin folder, it is necessary to set the path of JDK.

There are two ways to set the path in Java:

1. Temporary
2. Permanent



Java is **case sensitive**

**Source File Declaration Rules**

* There can be **only one public class** per source file.
* A source file can have multiple non-public classes.
* The public class name should be the name of the source file as well which should be appended by **.java** at the end.
* Main method inside public class

Comment

// one line

/\* \*/ multiline

In Java, a frame (also called activation record or stack frame) is a data structure used by the Java Virtual Machine (JVM) to store information about a method call. Each time a method is called, the JVM creates a new frame on the top of the current call stack to store the local variables, method parameters, return value, and other information related to the method.

The call stack is a data structure used by the JVM to keep track of the active method calls in a Java application. Each method call is represented by a frame on the stack, and the frames are arranged in a last-in-first-out (LIFO) order. When a method call completes, its frame is popped off the stack, and the program control returns to the method that called it.s

A frame typically contains the following information:

* The local variables of the method, including primitive types and object references.
* The method parameters passed to the method.
* The return value of the method.
* The program counter, which keeps track of the current instruction being executed in the method.
* The reference to the class that defines the method.

By creating a new frame on the stack for each method call, the JVM can support nested method calls and maintain a separate scope for each method. When a method returns, its frame is removed from the stack, and the program control returns to the calling method's frame.

Data Types

1. Primitive
2. Numeric
3. Integer

* byte -28 to 28-1 1byte 8bits
* short -216 to 216-1 2byte 16bits
* int -232 to 232-1 4byte 32bits
* long -264 to 264-1 8byte 64bits

1. Floating Point

* float -232 to 232-1 e.g.: 42.22f 4byte 32bits
* double -264 to 264-1 8byte 64bits

1. Non-Numeric
2. Character 2byte 16bits
3. Boolean 1 bits

* true
* false

1. Non-Primitive
2. Class
3. Array
4. Interface

# **Unicode System**

|  |
| --- |
| Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages. In unicode, character holds 2 byte, so java also uses 2 byte for characters. |

**Conversion | Type casting**

## Java Type Casting

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

* **Widening Casting** (automatically) –

converting a smaller type to a larger type size. Data don’t loss.

**Primitive**  
byte -> short -> char -> int -> long -> float -> double

Widening casting is done **automatically** when passing a smaller size type to a larger size type

byte var = 12;

int cast\_var = var

**Reference**

Parent\_type obj = new Child ();

* **Narrowing Casting** (manually) - converting a larger type to a smaller size type. Data loss

**Primitive**  
double -> float -> long -> int -> char -> short -> byte

Narrowing casting must be done **manually** by placing the type in parentheses in front of the value.

double var = 9.79797;

int cast\_var = (int) var;

**Reference**

Child\_type obj = (Parent\_type) new Parent ();

* **Overflow**

**Operators**

Unary Operators

Increment (++) and Decrement (--) operators

|  |  |
| --- | --- |
| **Operators** | **Meaning** |
| !a |  |
| ~a |  |
| ++a | Increment a by 1, then use the new value of a |
| a++ | Use value of a, then increment a by 1 |
| --b | Decrement b by 1, then use the new value of b |
| b-- | Use the current value of b, then decrement by 1 |

 Arithmetic Operators

 Relational Operators

 Logical Operators

* && (logical and)
* || (logical or)
* ! (logical not)

 Assignment Operators

 Misc Operators

* Conditional Operator (? :)

variable = (expression)? Value if true: Value if false

* instanceof Operator

(object reference variable) instanceof (class/interface type)

**Precedence of Java Operators**

**->**

**Keywords**

1. **strictfp:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
2. **transient:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
3. **transient:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.

**Flow control**

Java provides three types of control flow statements.

1. Decision Making statements
   * if statements
   * switch statement
2. Loop statements
   * do while loop
   * while loop
   * for loop
   * for-each loop
3. Jump statements
   * break statement
   * continue statement

**Java Decision Making**

1. simple if statement

if (Boolean\_expression)

{

//Statements will execute if the Boolean expression is true

}

2. if… else statement

if (Boolean\_expression)

{

//Executes when the Boolean expression is true

}

else

{

//Executes when the Boolean expression is false

}

3. if… else if… else statement

if (Boolean\_expression 1)

{

//Executes when the Boolean expression 1 is true

}

else if (Boolean\_expression 2)

{

//Executes when the Boolean expression 2 is true

}

else if (Boolean\_expression 3)

{

//Executes when the Boolean expression 3 is true

}

else

{

//Executes when the none of the above condition is true.

}

4. Nested if… else statement

if (Boolean\_expression 1)

{

//Executes when the Boolean expression 1 is true

if (Boolean\_expression 2)

{

//Executes when the Boolean expression 2 is true

}

}

5. Switch statement

The case expression will be of the same type as the variable

switch(expression)

{

case value:

//Statements

break; //optional

case value:

//Statements

break; //optional

//You can have any number of case statements.

default: //Optional

//Statements

}

* The variable used in a switch statement can only be integers, convertable integers (byte, short, char), strings and enums
* The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.

**Loop control**

1. while

while (Boolean\_expression)

{

//Statements

}

1. do-while

do

{

//Statements

} while (Boolean\_expression);

1. for

for (initialization; Boolean\_expression; update)

{

//Statements

}

**for-each loop**

for(declaration: expression)

{

//Statements

}

**for** (data\_type var: array\_name/collection\_name) {

//statements

}

Loop Control statement

1. break statement

Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.

Syntax: break;

Syntax: label:

break label;

1. continue statement

Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

Syntax: continue;

Syntax: label:

continue label;

Java Object Class

Naming Convention

Class :- ClassName

Interface:- InterfaceName

Method:- methodName

Variable:- variableName

Package:- com.package

Constant:- CONSTANT\_NAME

MODIFIERS IN JAVA

Modifiers in Java

In Java, access modifiers are used to set the visibility of the classes, attributes, methods, etc. We can divide modifiers into two groups:

* Access Modifiers: It controls the access level
* Non- Access Modifiers: It doesn’t control access level but provides other functionality.

Access Modifiers

For classes:

**Visibitly**

| **Modifier** | **Description** |
| --- | --- |
| public | The class is accessible by other class. |
| default | The class is accessible by other class in the same package. |

For properties and methods:

| **Modifier** | **Description** |
| --- | --- |
| public | Accessible by other class. |
| default | Accessible by other class in the same package. |
| private | Accessible within the same class. |
| protected | Accessible within a package and subclass. |

Public > protected > default > private

Non- Access Modifiers

For classes:

| **Modifier** | **Description** |
| --- | --- |
| final | The class cannot be inherited by other classes. |
| abstract | The class cannot be used to create objects. |

For properties and methods:

| **Modifier** | **Description** |
| --- | --- |
| final | Properties and methods cannot be overridden/ modified. |
| static | Properties and methods belong to a class rather than an object. |
| abstract | Can be used in the abstract class and can only be used on methods. |

Class - **C**lass**N**ame

A class in Java can contain:

* **Variables**
* **Static block**
* **Constructors**
* **Methods**
* **Blocks**
* **Nested class and interface**

Object

Ways to create an object in Java?

There are many ways to create an object in java. They are:

* By new keyword
* By newInstance () method
* By clone () method
* By deserialization
* By factory method etc.

Class\_name object\_name; <------------ ---Declaring reference to object

object\_name = new Class\_name (); <-------Allocating Memory

Class\_name object\_name = new Class\_name ();

Accessing class member using object

object\_name. variable\_name = value;

object\_name. method\_name ();

object\_name. method\_name (parameter\_list);

In Java, you can use the this keyword in several different contexts within a class:

1. To distinguish between a class field and a local variable or parameter with the same name:
2. To call a constructor from another constructor in the same class:
3. To pass the current object as a parameter to another method or constructor:
4. To return the current object from a method:
5. To access a static field or method from an instance method:

Ways to initialize object

There are 3 ways to initialize object in Java.

1. By constructor
2. By method
3. By reference variable



Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.If you have to use an object only once, an anonymous object is a good approach.

**new** Calculation();//anonymous object

### Creating multiple objects by one type only

**int** a=10, b=20;

Rectangle r1=**new** Rectangle(), r2=**new** Rectangle();//creating two objects

**Variable**

Declaring single variable

Syntax – Data\_Type Variable\_name;

Declaring multiple variables

Syntax – Data\_Type Variable\_name1, Variable\_name2;

Variable name only starts with

* A-Z, a-z
* \_
* $

Variable Initialization

* After Declaration
* During Declaration / Dynamic Initialization

Access modifier non-access modifier type variable\_name;

Kind of Variables

* **Static / Class variable**

**-** Static variables is created by adding “static” keyword before the variable.

Syntax: ClassName. static\_vairable;

Initialization: ClassName. static\_vairable = value;

* **Instance variable / Data Member**

- Instance variables are declared in a class, but outside a method, constructor or any block.

- We need an object of class to access Instance variable.

Syntax: object. instance\_vairable;

Initialization: object. instance\_vairable = value;

- cannot be reinitialized directly within class

* **Local variable**

- Local variables are declared in methods, constructors, or blocks.

- Access modifiers cannot be used for local variables.

- Local variables are visible only within the declared method, constructor, or block.

**-** There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

**Block**

1. Instance Initialization block

Runs each time you instantiate an object

{

//statement

}

1. Static Initialization block

Runs once (when the class is initialized)

static

{

//statement

}

* Is used to initialize the static data member.
* It is executed before the main method at the time of classloading.

**Order of execution**:

1. static initialization blocks of super classes
2. static initialization blocks of the class
3. instance initialization blocks of super classes
4. **constructors of super classes**
5. instance initialization blocks of the class
6. constructor of the class.

**Constructor**

Initialize

Grandparent constructor ----> parent constructor ------> child constructor

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like void).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

They can’t be inherited. Only members are inherited, and a constructor is not considered as a member.

### **Rules for creating Java constructor**

The first line of a constructor is a call to super() or this(), (a call to a constructor of a super-class or an overloaded constructor), if you don’t type in the call to super in your constructor the compiler will provide you with a non-argument call to super at the first line of your code, the super constructor must be called to create an object:

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

**Constructor Parameters**

Constructors can also take parameters, which is used to initialize attributes

class Student

{

Student ()

{

}

}

Type of constructor

1. Default constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

### **What is the purpose of a default constructor?**

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

2. Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

There is no copy Constructor in Java Programming.

## Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

* By constructor
* By assigning the values of one object into another
* By clone () method of Object class

## Constructor Overloading in Java

Constructor overloading in Java is a technique of having **more than one constructor** with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in **the list and their types.**

### **Can constructor perform other tasks instead of initialization?**

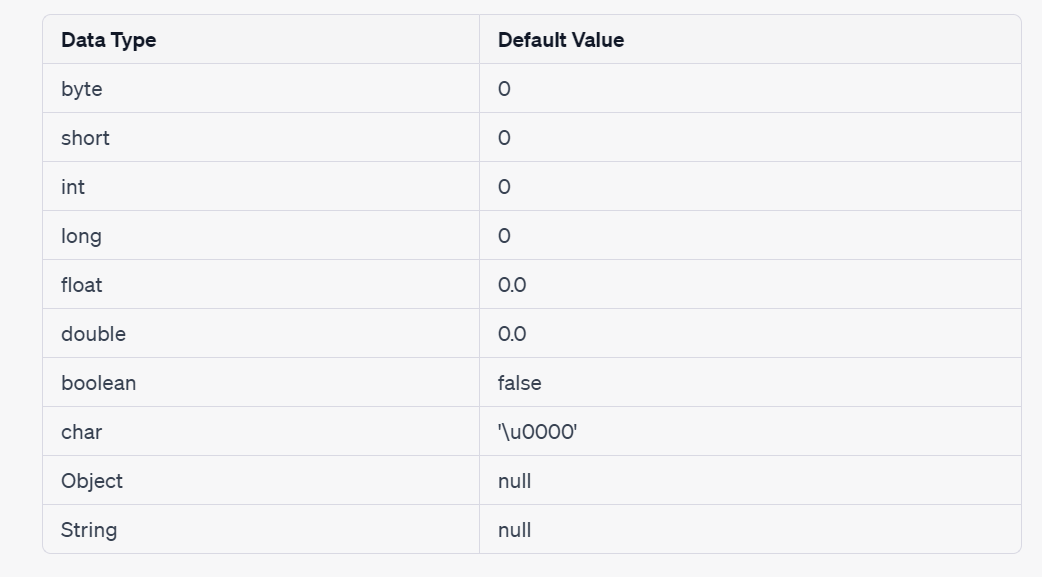
Yes, like object creation, **starting a thread, calling a method**, etc. You can perform any operation in the constructor as you perform in the method.

Destructor - There is no destructor in java.

### **Is there Constructor class in Java?** Yes.

**Does constructor return any value?** - Yes, it is the current class instance (You cannot use return type yet it returns a value).

Default constructor set

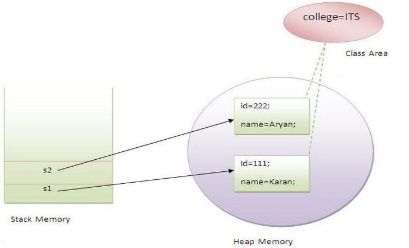
****

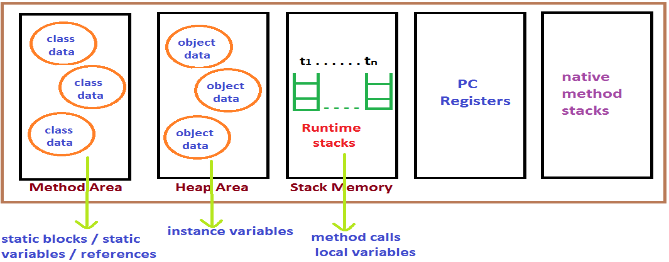
In Java, you can pass an object either in a constructor or in a method, depending on your specific needs and the design of your application. Here are some general guidelines:

1. Pass an object in a constructor when it is an essential dependency of the class and must be available throughout the lifetime of the object. For example, if a class needs to access a database, you might pass a database connection object in the constructor so that it can be used throughout the life of the object.
2. Pass an object in a method when it is only needed for a specific operation or a subset of methods. For example, if you have a method that needs to compute the average of a list of numbers, you might pass the list object in the method.

**Methods**

return





Kind of Methods

* Instance Method

1. Instance method can access the instance variables and instance methods directly.
2. Instance method can access static variables and static methods directly.

* Static / Class Method ----🡪 static keyword

1. Static methods can access the static variables and static methods directly.
2. Static methods can’t access instance variables and instance methods directly. They must use reference to object.
3. static method can’t use this keyword as there is no instance for ‘this’ to refer to.

The JVM runs the static method first, followed by the creation of class instances. Because no objects are accessible when the static method is used. A static method does not have access to instance variables. As a result, a static method can’t access a class’s instance variable.

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

### **Factory method**

It is a method that returns an object to the class to which it belongs. All static methods are factory methods.

**Same class**

**Inside instance method and static method**

|  |  |  |
| --- | --- | --- |
|  | Inside - **Static method** | Inside - **Instance method** |
| **Static variable** | variable  method () | variable  method () |
| **Static method** |
| **Instance variable** | obj\_name. variable  obj\_name. method () | variable  method () |
| **Instance method** |

**Different class**

|  |  |  |
| --- | --- | --- |
|  | Inside - **Static method** | Inside - **Instance method** |
| **Static variable** | ClassName. variable  ClassName.method () | ClassName. variable  ClassName.method () |
| **Static method** |
| **Instance variable** | obj\_name. variable  obj\_name. method () | obj\_name. variable  obj\_name. method () |
| **Instance method** |

**Getting input from user**

java is package, util is sub-package, Scanner is class

import java. util. Scanner;

Data\_Type Variable\_name;

Scanner object\_name = new Scanner (System. in);

Variable\_name = object\_name. nextInt ();

Variable\_name = object\_name. nextFloat ();

Variable\_name = object\_name. nextLine (); --------🡪string

public static void main (String args [])

{

/\* main () has to be declared public because it has to be invoked by the code outside the class when program is executed

Static keyword allows the main () method to be executed without creating an object of that class

String – It is a predefined class

args [] – It is a variable of String class

Note – Variable name can be different \*/

}

OOP **|** [**Contents**](#contents)

# **Inheritance**

Here are some interesting things about inheritance in Java:

Single Inheritance: Java supports single inheritance, which means that a class can only inherit from one superclass. However, it can implement multiple interfaces.

Superclass constructor: When a subclass is instantiated, the constructor of its superclass is called automatically before the constructor of the subclass.

Overriding: A subclass can override the methods of its superclass by providing a new implementation of the same method. This allows the subclass to change the behavior of the method inherited from the superclass.

Polymorphism: The ability to use a superclass reference variable to refer to an instance of a subclass is known as polymorphism. This allows for objects of different classes to be treated as objects of the superclass.

Access modifiers: The private members of a superclass are not accessible from its subclasses. However, protected members can be accessed from subclasses in the same package and public members can be accessed from anywhere.

Object class: All classes in Java inherit from the Object class, which is the root of the class hierarchy in Java. The Object class provides several methods like equals, hashCode, toString, etc.

Abstract class: A class that is declared as abstract cannot be instantiated, but it can be subclassed. An abstract class can have abstract methods, which are methods without a body. Subclasses must provide an implementation for these methods.

Interfaces: Java allows classes to implement one or more interfaces. An interface defines a set of abstract methods that a class must implement.

* + Public
  + Protected: A protected class member can be accessed within the same package, as well as from subclasses in other packages. This allows for inheritance and polymorphism to work as intended.
  + Default (package-private): A class member with default access can be accessed only within the same package, not from other packages. This is the default access level if no access modifier is specified.
  + private

In Java, we have two types of relationship:

1. Inheritance

- Tight coupling

- Is A relation

2. Association

- Loose coupling

- Has A relation

* + 1. Aggregation

Weak bonding

* + 1. Composition

Strong bonding

* **subclass** (child) - the class that inherits from another class
* **superclass** (parent) - the class being inherited from

To inherit from a class, use the **extends** keyword.

## Types of inheritance in java

On the basis of class, there can be three types of inheritance in java:

**single, multilevel and hierarchical.**

****

In java programming, **multiple and hybrid** inheritance is supported through interface only



On the child reference we can call both parent and child class methods. But on the parent reference we can call only methods available in the parent class and we can’t call child specific methods

Child c = new Child (); -----🡪 can access Both parent and child members

Parent p = new Child (); --🡪 can access Only parent members

* + call parent class constructor default / parameter (super()) first statement
  + call child class constructor

Parent p = new Parent ();

Child c = new Parent (); ------------🡪 incompatible

Child Is a Parent

Has a

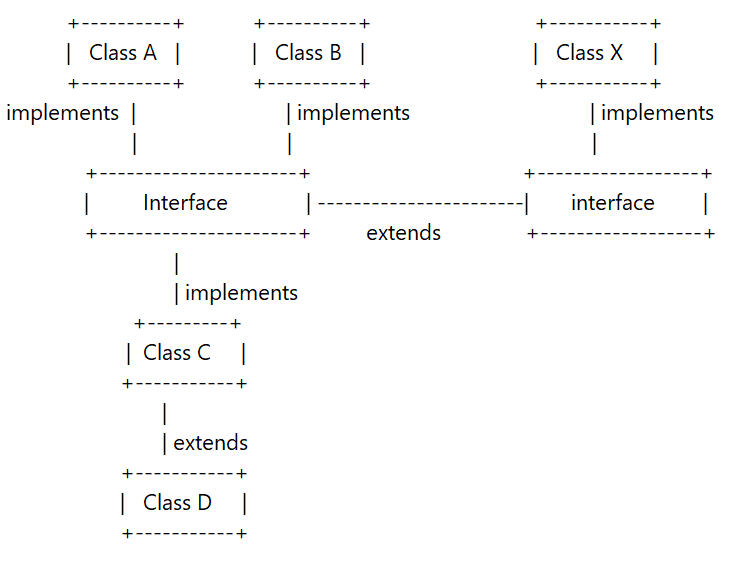
- we can access only parent variables and method and cannot call properties specific to child class.

-overridden method will be called

-Downcast reference variable to call properties specific to child class.

private methods and variables are not inherited by the subclass. This means that a subclass cannot access private methods or variables defined in its superclass. Private methods and variables are only accessible within the class in which they are defined. They cannot be accessed or overridden by any other class, including subclasses.

Multiple inheritance



**Polymorphism**

Method Overloading

Occur within the class

Compile time polymorphism!!!

## Advantage of method overloading

Method overloading increases the readability of the program.

### **Different ways to overload the method**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In Java, Method Overloading is not possible by changing the return type of the method only.

Compile Time Error is better than Run Time Error

### **Can we overload java main () method?**

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main () method which receives string array as arguments only

## Method Overloading and Type Promotion

Automatic promotion in overloading



One type is promoted to another implicitly if no matching datatype is found.

### **Method Overriding**

**Rules for Method overriding are:**

* Performed in two classes with inheritance.
* Run time polymorphism!!!
* final and static methods cannot be overridden but are inherited.
* Constructors cannot be overridden and are not inherited.
* Methods must share the same name in child and parent class.
* It must have the same parameter as in the superclass.
* It must have same return type.

public static int fun(int a)

overloading methods must have same name and different signature.

overriding methods must have same name and same signature.

@Override

Use it every time you override a method

### **Super keyword**

When invoking a superclass version of an overridden method the **super** keyword is used.

Super keyword usage in inheritance, always refers to its immediate as an object.

Usage of Java super Keyword

1. super () can be used to invoke immediate parent class constructor.

super (values);

1. super can be used to invoke immediate parent class method.

super. method ();

1. super can be used to refer immediate parent class instance variable.

super. variable

Final keyword

1. When we use **final** with **primitive types**, it ensures that the value of the variable remains constant and cannot be changed.
2. But when we use final with **reference types** (e.g., objects), it ensures that the reference to the object remains constant and cannot be changed, but the state of the object itself can still be modified if the object is mutable.

Final method is inherited but you cannot override it.

### A final variable that is not initialized at the time of declaration is known as blank final variable. we can initialize blank final variable only in constructor.

### **static blank final variable**

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

We cannot declare constructor final

* 1. final variable
  + If you make any variable as final, you cannot change the value of final variable (It will be constant).
  + Stop value change

final with Instance Variable

final int roll = 101;

final int roll; // we can initialize this using constructor

final with static variable

final static roll = 71;

final static roll; // we can initialize this using static block

final with local variable

final int roll = 71;

final int roll; // we can initialize this with same method or block before using it

final with parameter

* 1. final method
* If you make any method as final, you cannot override it.
* Stop method overriding.
  1. final class
* If you make any class as final, you cannot extend it.
* Stop inheritance.

|  |  |
| --- | --- |
| Final variable | Stop value changes |
| Final method | Stop method overriding |
| Final class | Stop inheritance |

**Abstraction**

abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

Abstract class doesn't support multiple inheritance.

An abstract class can extend another Java class and implement multiple Java interfaces.

* 1. Abstract classes may or may not contain abstract methods, i.e., methods without body (public void get () ;)
  2. But, if a class has at least one abstract method, then the class **must** be declared abstract.
  3. If a class is declared abstract, it cannot be instantiated.
  4. To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
  5. If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.
  6. We cannot declare abstract constructors or abstract static methods.

Abstract Methods

1. **abstract** keyword is used to declare the method as abstract
2. Instead of curly braces, an abstract method will have a semi colon (;) at the end.
3. Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Interface

Why use Java interface?

Interface supports multiple inheritance.

An interface can extend another Java interface only.

It has static constants and abstract methods.

There are mainly three reasons to use interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.

Class C implements A, B

* It can be used to achieve loose coupling.

Variable by default public + static + final

Method by default public + abstract

you cannot define variables without initializing them in an interface in Java.



We can also include default and static methods in an interface, which have concrete implementations and do not need to be implemented by implementing classes.

public interface MathUtils {

int add(int a, int b);

int subtract(int a, int b);

default int multiply(int a, int b) {

return a \* b;

}

static int divide(int a, int b) {

if (b == 0) {

throw new IllegalArgumentException("Cannot divide by zero");

}

return a / b;

}

}

InterfaceName.staticMethodName();

MyInterface instance = new MyClass();

instance.defaultMethodName();

Note that if a class that implements the interface provides its own implementation of a default method, that implementation will be used instead of the one defined in the interface.

InterfaceClass obj = new ImplementedClass();

obj.method();

overridden abstract method can be called but properties specific to implemented class cannot be called.

Loose coupling

public interface Dependency {

void doSomething();

}

public class ConcreteDependency implements Dependency {

public void doSomething() {

// implementation

}

}

public class Client {

private Dependency dependency;

public Client(Dependency dependency) {

this.dependency = dependency;

}

public void doSomethingWithDependency() {

dependency.doSomething();

}

}

Dependency concreteDependency = new ConcreteDependency();

Client client = new Client(concreteDependency);

Marker Interfaces: Some interfaces do not contain any methods and are used only to mark or tag a class as having a certain property or behavior. These interfaces are called marker interfaces. An example of a marker interface in Java is the Serializable interface, which is used to indicate that a class can be serialized and deserialized.

public interface MyMarkerInterface {

// no methods

}

Functional Interfaces: A functional interface is an interface that contains only one abstract method and is used to represent a single unit of behavior. Functional interfaces are often used in lambda expressions and method references in Java 8 and later versions.

@FunctionalInterface

public interface MyFunctionalInterface {

void myMethod();

}

implement

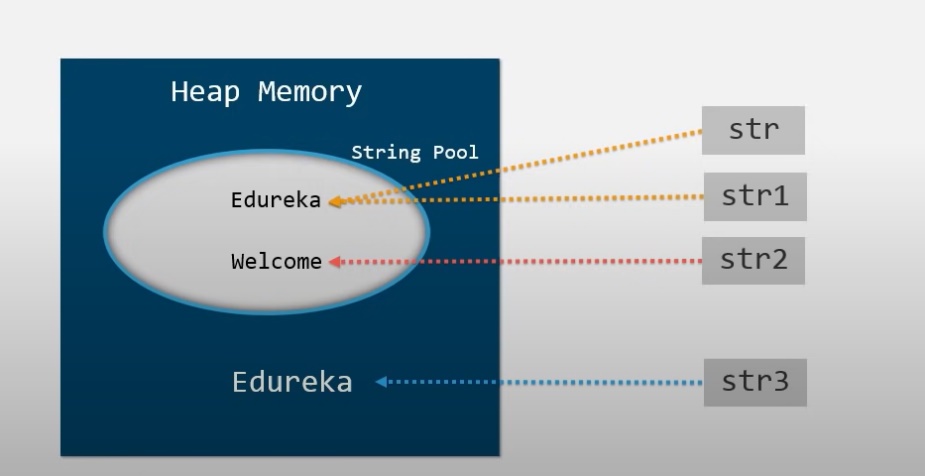
-----------> class A

class O interface -----------> class B

-----------> class C

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | public | default | protected | private | static | final | abstract |
| class | √  (only once in a file) | √ | × | × | × | √ | √ |
| Nested class | √ | √ | √ | √ | √ | √ | √ |
| constructor | √ | √ | √ | √ | × | × | × |
| Static method | √ | √ | √ | √ | -- | √ | × |
| Instance method | √ | √ | √ | √ | √ | √ | √ |
| Abstract  method | √ | √ | √ | × | × | × | -- |
| Interface |  |  |  |  |  |  |  |

**String**





The java.lang.String class implements Serializable, Comparable and CharSequence interfaces

CharSequence interface

The CharSequence interface is used to represent the sequence of characters. String, [StringBuffer](https://www.javatpoint.com/StringBuffer-class)

and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



There are two ways to create String object:

1. By string literal
2. By new keyword

String s1 = “Welcome”;

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

Note: String objects are stored in a special memory area known as the "string constant pool"

String s = new String(“Welcome”)

In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable. Once String object is created its data or state can't be changed but a new String object is created.

### **Why String objects are immutable in Java?**

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

**1. ClassLoader:**

A ClassLoader in Java uses a String object as an argument. Consider, if the String object is modifiable, the value might be changed and the class that is supposed to be loaded might be different.

**2. Thread Safe:**

As the String object is immutable we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

**3. Security**

As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

**4. Heap Space:**

The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

There are three ways to compare String in Java:

1. By Using equals() Method

The String class equals() method compares the original content of the string. It compares values of string for equality.

1. By Using == Operator

The == operator compares references not values.

1. By compareTo() Method

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

* **s1 == s2** : The method returns 0.
* **s1 > s2** : The method returns a positive value.
* **s1 < s2** : The method returns a negative value

String concatenation

In Java, String concatenation forms a new String that is the combination of multiple strings. There are two ways to concatenate strings in Java:

1. By + (String concatenation) operator

 String concatenation operator produces a new String by appending the second operand onto the end of the first operand. The String concatenation operator can concatenate not only String but primitive values also.

String s = 50 + 30 + welcome” + 40 + 40;

Output: 80welcome80

1. By concat() method

There are some other possible ways to concatenate Strings in Java,

1.StringBuilder

StringBuilder is class provides append() method to perform concatenation operation. The append() method accepts arguments of different types like Objects, StringBuilder, int, char, CharSequence, boolean, float, double. StringBuilder is the most popular and fastet way to concatenate strings in Java. It is mutable class which means values stored in StringBuilder objects can be updated or changed.

String Buffer class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

String Methods

**Array**

- an array is a collection of similar type of elements which has contiguous memory location.

**Syntax to Declare an Array in Java**

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

**Instantiation of an Array in Java**

arrayRefVar=**new** datatype[size];

**int** a[]={33,3,4,5};//declaration, instantiation and initialization

### **Advantages**

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
* **Random access:** We can get any data located at an index position.

### **Disadvantages**

* **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

### **Types of Array in java**

There are two types of array.

* Single Dimensional Array
* Multidimensional Array

**Collections**

****

**HashMap**

**Exceptions**

### In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

### **Types of Java Exceptions**

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error



## Difference between Checked and Unchecked Exceptions

### **1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### **2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### **3) Error**

Error is irrecoverable. Some example of errors is OutOfMemoryError, VirtualMachineError, AssertionError etc.

try

{//statements}

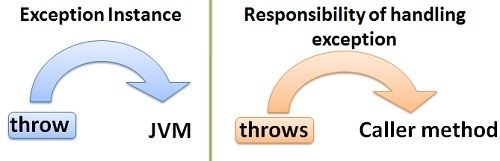
// throw new ExceptionType()

catch (ExceptionType e) //ExceptionType e = new ExceptionType()

{//statements}

finally

{//statements that always get executed}



throw

The “throw” keyword is used to throw an exception

throws

The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signatures.

## Points to remember

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

## Java throw keyword

## It is mainly used to throw a custom exception.

**throw** **new** exception\_class ("error message");

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

### **Throwing User-defined Exception**

**class** UserDefinedException **extends** Exception

{

**public** UserDefinedException (String str)

    {

        // Calling constructor of parent Exception

**super**(str);

  }

}

**throw** **new** UserDefinedException ("This is user-defined exception");

# **Java throws keyword**

return\_type method\_name () **throws** exception\_class\_name(multiple)//Checked exception only

{

//method code

}

NullPointerException

NullPointerException is a RuntimeException. In Java, a special null value can be assigned to an object reference. NullPointerException is thrown when program attempts to use an object reference that has the null value.

These can be:

Invoking a method from a null object.

Accessing or modifying a null object’s field.

Taking the length of null, as if it were an array.

Accessing or modifying the slots of null object, as if it were an array.

Throwing null, as if it were a Throwable value.

When you try to synchronize over a null object.

**Generics**

Generics Work Only with Reference Types

# Generics in Java

**1) Type-safety:** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

**2) Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime

## Defining class / Creating object

## Defining method / calling method

## Generic class

A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.

**class** MyGen<T> {

T obj;

MyGen(T obj)

{

this.obj = obj

}

**void** add (T obj)

{

**this**.obj=this.obj+2;

}

}

BaseType<ParameterType> obj = new BaseType<>()

Note: In Parameter type we can not use primitives like ‘int’,’char’ or ‘double’.

The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

## Type Parameters

1. T - Type
2. E - Element
3. K - Key
4. N - Number
5. V - Value

## Generic Method

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to the generic method.

public <T> List<T> fromArrayToList(T[] a)

{ return Arrays.stream(a).collect(Collectors.toList());

}

**Syntax** to use generic collection

ClassOrInterface<Type>

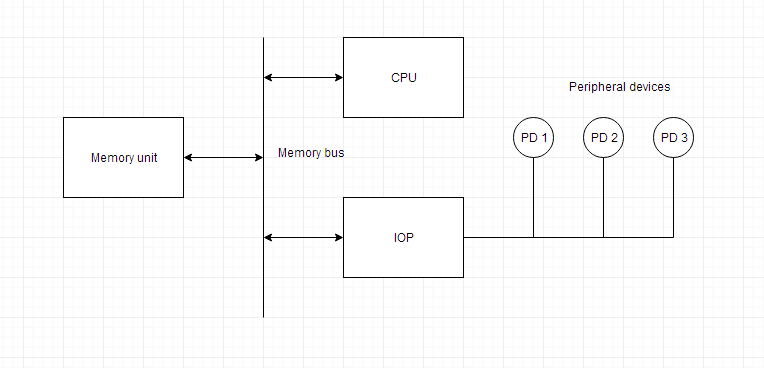
**DATE/TIME**

Java.time

**I/O |** [**Contents**](#contents)

**CPU and Memory bus**

HardDisk ----------> buffer ---------> mainMemory



**FILE / DIRECTORY**

Java.io.file

Its all method take or use file object.

If you need to perform simple file operations, such as creating or deleting files and directories, or checking their properties, you can use th File class

For directory For File

File dir = new File(“c\user\dir”) File file = new File(“c\user\dir\file.text”)

If(!dir.exists()){ If(!file.exists()){

dir.mkdirs(); file.createNewFile();

}else{ }else{

//already exists //already exists

} }

File class provide method for creating, deleting and renaming files and directories as well as checking their properties such as whether they exist or are readable and writeable

Java.nio.file

Its all method take or use path object.

If you need to work with the contents of a file or directory, or perform more advanced file system operations, such as reading and writing files, or copying or moving files and directories, you can use the Path class.

For directory For File

Path path = Paths.get(“c\user\dir”) Path path = Paths.get(“c\user\dir\file.text”)

If(!Files.exists(path)){ if(!Files.exists(path)){

Files.createDirectory(path); Files.createFile(path);

}else{ }else{

//already exists //already exists

} }

**Java.io**

**Java I/O** (Input and Output) is used to process the input and produce the output. java.io package contains all the classes required for input and output operations.

Stream

-Byte stream (InputStream / OutputStream) 8bits

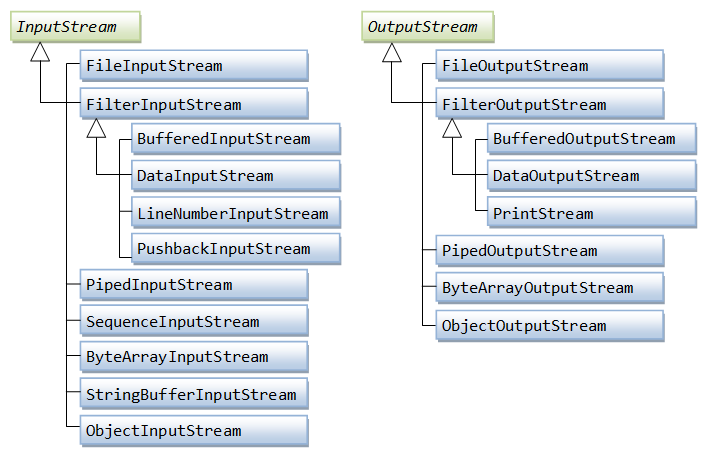
-Character stream (Reader / Writer) 16bits

Input/read Output/write

InputStream OutputStream

Source-------------------> program ------------------------->Destination

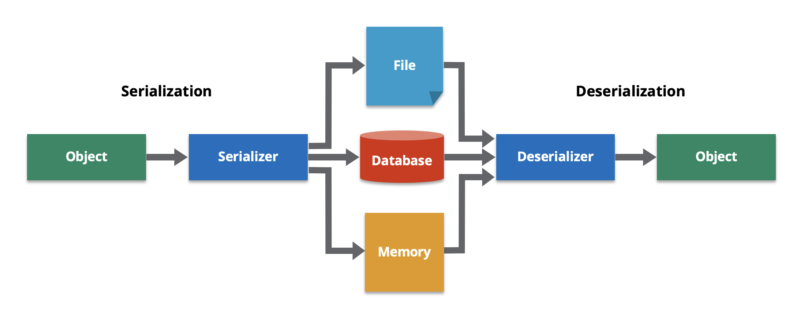
- For character-oriented data, it is preferred to use FileWriter than FileOutputStream.



I/O Methods

**Serialization and Deserialization**

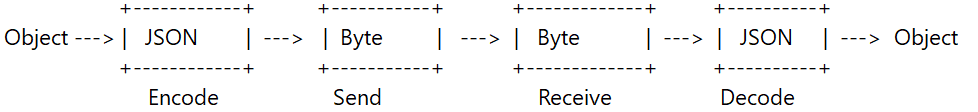
Java.io.Serializable



Serialization in Java is the process of converting an object into a stream of bytes so that it can be easily stored in a file, sent over a network, or persisted in a database. Deserialization is the reverse process of reconstructing the object from the serialized form.

In order to serialize an object in Java, the class must implement the Serializable interface, which is a marker interface indicating that the class can be serialized.

In general, if you're working with Java-specific data within a Java application, Java object serialization might be the better choice. If you need to exchange data with other platforms or programming languages, or if you're working with data that doesn't have a specific Java-centric structure, JSON might be a better choice.

****

**Multithread**

### Concurrency vs Parallelism | Baeldung on Computer Science

### Advantages of Java Multithreading

Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

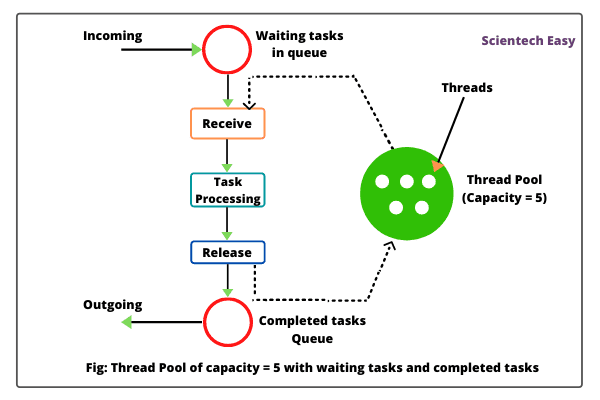
* Process-based Multitasking (Multiprocessing)
* Thread-based Multitasking (Multithreading)

# **Life cycle of a Thread (Thread States)**

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

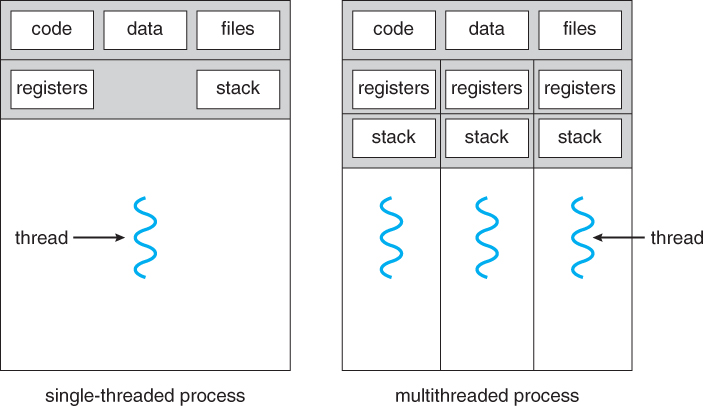
ThreadPool



**Context Switching** involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier.

**Task scheduler**

**Translation Lookaside Buffer**



**Reflection**

**OOP Miscellenous |** [**Contents**](#contents)

# **Wrapper classes in Java**

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* **Synchronization:** Java synchronization works with objects in Multithreading.
* **java.util package:** The java.util package provides the utility classes to deal with objects.
* **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

you need to cast an object reference to an interface reference. Whenever you need to call the methods of the interface only.

The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive.

it is important to note that using wrapper classes can be less efficient than using primitive types, since they require additional memory and processing overhead. In cases where performance is critical, it is usually best to use primitive types directly.

**Lamda Expression**

(argument-list) -> {body}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

@FunctionalInterface

**Anonymous class**

**Recursive function**

Design pattern

1. Creational design patterns

1. Factory Method Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern
6. Object Pool Pattern

2. Structural Design Pattern

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

3. Behavioral Design Pattern

1. Chain Of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern
    * Inversion of control
    * Dependency Injection
    * MVC

* Builder pattern
* Factory method pattern

Duck Typing

Functional programming

Sure, here's a more detailed overview of the features introduced in Java 8 to support functional programming:

1. Lambda expressions: Lambda expressions are a way to define anonymous functions in Java. They provide a concise syntax for defining functions that can be passed as arguments to other functions or stored in variables. Lambda expressions can be used in place of anonymous inner classes in many cases, and can make code more concise and expressive.
2. Functional interfaces: A functional interface is an interface that has exactly one abstract method. Functional interfaces are used to represent functions or actions, and can be used as the type of a lambda expression or method reference. Java 8 introduced several new functional interfaces in the java.util.function package, including Predicate, Function, and Consumer, among others.
3. Default methods: A default method is a method defined in an interface that provides a default implementation. Default methods allow interfaces to provide concrete implementations of methods without breaking existing implementations. Default methods can be overridden by classes that implement the interface, or can be called directly on the interface if they are not overridden.
4. Static methods: A static method is a method defined in an interface that can be called directly on the interface without needing an instance of a class that implements the interface. Static methods can provide utility functions or other functionality that does not depend on the state of an implementing class.
5. Streams: The java.util.stream package provides a powerful API for processing collections of data using functional programming constructs. Streams allow you to chain together a series of operations on a collection, such as filtering, mapping, and reducing, in a concise and expressive way. Streams can improve code readability and maintainability by reducing the amount of boilerplate code needed for common collection operations.

Functional Programming

* Lambdas and Streams
* Functional Interfaces
* Consumer, Predicate, Function & Supplier
* Stream Intermediate/Terminal Operations
* Learn using FP with Files and Threads
* Parallelized Functional Code

**Maven |** [**Contents**](#contents)

If you have not specified a version of Spring Cloud in your project, Maven will use the default version of Spring Cloud that corresponds to the version of Spring Boot that you are using.

For example, if you are using Spring Boot 2.5.0, the default version of Spring Cloud will be 2020.0.3, and the version of spring-cloud-starter-sleuth that will be picked up will be 2020.0.3.

POM stands for Project Object Model. It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml.

POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal. Some of the configuration that can be specified in the POM are following −

* project dependencies
* plugins
* goals
* build profiles
* project version
* developers
* mailing list

What are Maven Plugins?

Maven is actually a plugin execution framework where every task is actually done by plugins. Maven Plugins are generally used to −

* create jar file
* create war file
* compile code files
* unit testing of code
* create project documentation
* create project reports

**Maven Life Cycle**

**Maven Lifecycle:**Below is a representation of the default Maven lifecycle and its 8 steps: Validate, Compile, Test, Package, Integration test, Verify, Install and Deploy.

1. **Validate:** This step validates if the project structure is correct. For example – It checks if all the dependencies have been downloaded and are available in the local repository.
2. **Compile:** It compiles the source code, converts the .java files to .class and stores the classes in target/classes folder.
3. **Test:** It runs unit tests for the project.
4. **Package:** This step packages the compiled code in distributable format like JAR or WAR.
5. **Integration test:** It runs the integration tests for the project.
6. **Verify:** This step runs checks to verify that the project is valid and meets the quality standards.
7. **Install:** This step installs the packaged code to the local Maven repository.
8. **Deploy:** It copies the packaged code to the remote repository for sharing it with other developers.

Maven follows a sequential order to execute the commands where if you run step *n*, all steps preceding it (Step 1 to *n-1*) are also executed. For example – if we run the Installation step (Step 7), it will validate, compile, package and verify the project along with running unit and integration tests (Step 1 to 6) before installing the built package to the local repository.

**Maven Commands:**

* **mvn clean:** Cleans the project and removes all files generated by the previous build.
* **mvn compile:** Compiles source code of the project.
* **mvn test-compile:** Compiles the test source code.
* **mvn test:** Runs tests for the project.
* **mvn package:** Creates JAR or WAR file for the project to convert it into a distributable format.
* **mvn install:** Deploys the packaged JAR/ WAR file to the local repository.
* **mvn deploy:** Copies the packaged JAR/ WAR file to the remote repository after compiling, running tests, and building the project.
* Generally, when we run any of the above commands, we add the **mvn clean** step so that the target folder generated from the previous build is removed before running a newer build. This is how the command would look on integrating the *clean* step with *install* phase: **mvn clean install**
* Similarly, if we want to run the step-in debug mode for more detailed build information and logs, we will add **-X** to the actual command. Hence, the *install* step with debug mode on will have the following command: **mvn -X install**
* Consider a scenario where we do not want to run the tests while packaging or installing the Java project. In this case, we use **-DskipTests** along with the actual command. If we need to run the *install* step by skipping the tests associated with the project, the command would be: **mvn install -DskipTests**

Maven Repositories

In Maven terminology, a repository is a directory where all the project jars, library jar, plugins or any other project specific artifacts are stored and can be used by Maven easily.

Maven repository are of three types. The following illustration will give an idea regarding these three types.

* local
* central
* remote

Maven local repository by default get created by Maven in %USER\_HOME% directory. To override the default location, mention another path in Maven settings.xml file available at %M2\_HOME%\conf directory.

Central Repository

Maven central repository is repository provided by Maven community. It contains many commonly used libraries.

When Maven does not find any dependency in local repository, it starts searching in central repository using following URL − <https://repo1.maven.org/maven2/>

Key concepts of Central repository are as follows −

* This repository is managed by Maven community.
* It is not required to be configured.
* It requires internet access to be searched.

## Remote Repository

Sometimes, Maven does not find a mentioned dependency in central repository as well. It then stops the build process and output error message to console. To prevent such situation, Maven provides concept of **Remote Repository**, which is developer's own custom repository containing required libraries or other project jars.

For example, using below mentioned POM.xml, Maven will download dependency (not available in central repository) from Remote Repositories mentioned in the same pom.xml.

Scope: compile, provided, runtime, test and system

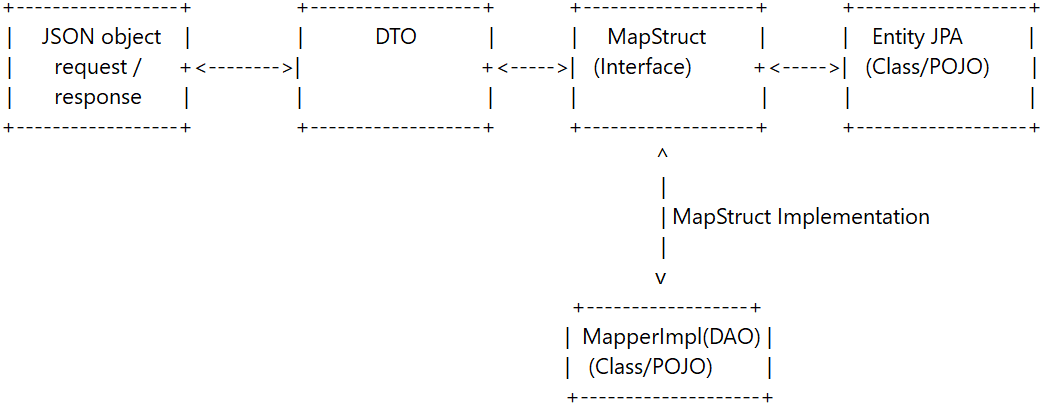
**SpringBoot framework |** [**Contents**](#contents)

Data Access Object

Data Transfer Object

JPA Entity

JSON object



In MapStruct, you define an interface that specifies how the mapping should be done between two classes. MapStruct generates an implementation of that interface for you, which you can then use to perform the actual mapping.

Working of Spring boot

In a Spring Boot application, when a request is made to a web controller, the Spring framework creates an instance of the controller class and then calls the appropriate method to handle the request. Here's a high-level overview of how this process works:

1. When a request is made to the web application, it is intercepted by the Spring DispatcherServlet.
2. The DispatcherServlet analyzes the request and determines which controller class should handle it, based on the request URL and other factors.
3. The DispatcherServlet then checks if an instance of the controller class already exists in the Spring application context. If an instance does not exist, it creates one.
4. Once the controller instance is created (or retrieved from the context), the DispatcherServlet calls the appropriate method in the controller class to handle the request. This method is typically annotated with @RequestMapping or a related annotation that maps the method to a specific request URL or pattern.
5. The controller method executes and returns a response to the DispatcherServlet, which then sends the response back to the client.

So, to answer your question, the instance of the controller class is created automatically by Spring when a request is made, and the appropriate controller method is then called to handle the request. You do not need to create the instance of the controller class manually.

**Life cycle of bean**

In Spring Boot, a bean's lifecycle can be divided into several stages, each with its own corresponding method callback. Here is a brief overview of the different stages and their respective methods:

1. Instantiation: During this stage, the bean instance is created by the container. This is usually done using a no-argument constructor.
2. Dependency Injection: Once the bean instance has been created, any dependencies it requires are injected into it. This is typically done using setters or constructor injection.
3. Initialization: After the dependencies have been injected, the bean's initialization methods are called. There are two types of initialization methods: @PostConstruct and InitializingBean. The @PostConstruct method is called immediately after all dependencies have been injected, while the InitializingBean method is called right before the bean is ready for use.
4. Bean Usage: The bean is now ready to be used by the application. Any business logic that the bean is responsible for can now be executed.
5. Destruction: When the application is shutting down, the container calls the bean's destruction methods to release any resources it might be holding onto. There are two types of destruction methods: @PreDestroy and DisposableBean. The @PreDestroy method is called right before the bean is destroyed, while the DisposableBean method is called just before the bean is removed from the container.

Here's a list of all the stereotype annotations in Spring Boot that can be used to create a bean:

* @Component: Indicates that a class is a Spring component. This is a general-purpose annotation that can be used for any class that needs to be managed by the Spring container.
* @Service: Indicates that a class is a business service. This annotation is typically used for classes that provide business logic in a Spring application.
* @Repository: Indicates that a class is a data repository. This annotation is typically used for classes that provide data access in a Spring application.
* @Controller: Indicates that a class is a Spring MVC controller. This annotation is typically used for classes that handle HTTP requests in a Spring application.
* @RestController: Indicates that a class is a RESTful controller. This annotation is a specialized form of @Controller that is used for building RESTful web services in a Spring application.
* @Configuration: Indicates that a class is a Spring configuration class. This annotation is used for classes that provide configuration information to the Spring container.
* @Bean: Indicates that a method produces a bean to be managed by the Spring container. This annotation is typically used in configuration classes to define beans that are not automatically detected by Spring's component scanning.

All of these annotations tell Spring Boot that a class should be treated as a bean and that an instance of the class should be created and managed by the Spring container.

When a Spring Boot application starts up, the container performs the following steps:

1. Component Scanning: The container scans the application's classpath for classes annotated with @Component, @Service, @Repository, @Controller, and other stereotype annotations. It then creates instances of these classes and registers them as beans in the container.

@ComponentScan

1. Bean Creation: The container creates a bean for each class that has been registered in the previous step. This involves invoking the class's constructor and any setter or initialization methods, and injecting any dependencies that the bean may have.
2. Dependency Injection: The container then wires the beans together by injecting dependencies into each bean. This is done by inspecting the properties and constructor parameters of each bean and looking for matching dependencies in the container.
3. Bean Post-Processing: After the beans have been created and wired together, the container may apply additional post-processing to the beans. This can include invoking any @PostConstruct methods or applying custom AOP advice to the beans.

The Spring container is a runtime environment that provides a number of components that are used to manage and configure Spring components. Some of the key components of the Spring container, along with their uses, are:

1. BeanFactory: This is the core interface for accessing and managing Spring beans. It provides support for lazy loading, scope management, and more.
2. ApplicationContext: This is a superset of BeanFactory and provides additional functionality such as support for internationalization, event handling, and more.
3. Spring IoC (Inversion of Control) container: This is responsible for managing the lifecycle of beans in a Spring application, including instantiation, configuration, and dependency injection.
4. Spring AOP (Aspect-Oriented Programming) framework: This provides support for cross-cutting concerns such as transaction management, security, and logging.
5. Spring MVC (Model-View-Controller) framework: This provides support for building web applications using the MVC pattern.
6. Spring Data Access: This provides support for working with data sources such as databases and XML files.
7. Spring Testing framework: This provides support for testing Spring-based applications.

In Spring Boot, there are several types of scopes that can be used to manage the lifecycle of a bean:

1. Singleton: This is the default scope in Spring Boot. In this scope, a single instance of the bean is created for the entire application context.
2. Prototype: In this scope, a new instance of the bean is created every time it is requested.
3. Request: In this scope, a new instance of the bean is created for each HTTP request that is processed by the Spring MVC application.
4. Session: In this scope, a single instance of the bean is created for each user session. The bean is stored in the HTTP session and is available for the duration of the user's session.
5. Global Session: This scope is similar to the session scope, but it is used in a Portlet context. In this scope, a single instance of the bean is created for each user's global session.
6. WebSocket: This scope is used in WebSocket applications. In this scope, a single instance of the bean is created for each WebSocket connection.
7. Application: This scope is used in a Servlet context. In this scope, a single instance of the bean is created for the entire Servlet context.

The @Scope annotation in Spring Boot can be used to specify the scope of a bean. This annotation can be applied to a class or a method, depending on where the bean is defined.

If the bean is defined in a class, the @Scope annotation can be applied to the class itself, like this:

@Component

@Scope("prototype")

public class MyPrototypeBean {

// class implementation

}

In this example, the MyPrototypeBean class is defined as a Spring component and has a scope of "prototype", which means that a new instance of the bean will be created every time it is requested.

It's important to keep this in mind when working with singleton-scoped beans in Spring Boot. If you need to ensure that each class gets its own instance of a bean, you can change the scope of the bean to prototype.

**Repository and Mapstruct**

MapStruct:

* We define a mapper interface.
* MapStruct generates the implementation classes for the mapper interface at compile time.

Repository:

* We define a repository interface.
* Spring Boot generates the implementation classes for the repository interface at runtime using Spring Data JPA or another underlying data access technology.

Lombok

| Annotation | Description | Used with |
| --- | --- | --- |
| @Data | Generates boilerplate code for **getters, setters, equals, hashCode, and toString** methods. | Class |
| @NoArgsConstructor | Generates a no-argument constructor. | Class |
| @AllArgsConstructor | Generates a constructor with arguments for all fields. | Class |
| @Builder | Generates a builder pattern for creating instances of the class. | Class |
| @Getter | Generates getters for all fields. | Field |
| @Setter | Generates setters for all fields. | Field |
| @Slf4j | Generates a logger field for the class using SLF4J. | Class |

**Controller Layer**

| Annotation | Description | Where to use |
| --- | --- | --- |
| @RestController | Marks the class as a REST controller that returns data in the response body. | Class |
| @RequestMapping | Maps HTTP requests to handler methods based on the URL pattern. Can also specify HTTP methods and request parameters. | Class or Method |
| @GetMapping | Shortcut for @RequestMapping with method = RequestMethod.GET. Maps GET requests to handler methods. | Method |
| @PostMapping | Shortcut for @RequestMapping with method = RequestMethod.POST. Maps POST requests to handler methods. | Method |
| @PutMapping | Shortcut for @RequestMapping with method = RequestMethod.PUT. Maps PUT requests to handler methods. | Method |
| @DeleteMapping | Shortcut for @RequestMapping with method = RequestMethod.DELETE. Maps DELETE requests to handler methods. | Method |
| @PathVariable | Maps a URL path variable to a method parameter. | Method Parameter |
| @RequestParam | Maps a request parameter to a method parameter. Can specify default values and required parameters. | Method Parameter |
| @RequestBody | Binds the request body to a method parameter or object. Used for POST and PUT requests. | Method Parameter |
| @ResponseBody | Marks a method or return value as the response body. Used for RESTful APIs that return data in JSON, XML, etc. | Method or Return Value |
| @ResponseStatus | Sets the HTTP response status code for a handler method. | Method |
| @ExceptionHandler | Handles exceptions thrown by a handler method or any of its submethods. | Method |
| @Valid | Enables validation of method parameters or objects using validation constraints. | Method Parameter or Object |
| @ModelAttribute | Maps a model attribute to a method parameter. Used for form data binding. | Method Parameter |

Bottom of Form

Controller Annotation

@RestController()

= @Controller + @ResponseBody (serialization)

@RequestMapping()

@ResponseStatus

The annotation is used to map web requests to Spring Controller methods.

* @GetMapping
* @PostMapping
* @PutMapping
* @DeleteMapping
* @PatchMapping

@PathVariable Method Parameter

Path variables are typically used for dynamic portions of the URL that change with each request, such as an ID or a username. They are denoted in the URL by placing a variable name in curly braces, such as /**users/{id}.**

For path variables, the URL typically follows this pattern:

**http://example.com/path/to/endpoint/{variable1}/{variable2}**

Where the path variables are specified within curly braces {} in the URL path.

@RequestParam Method Parameter

1. @RequestParam is used to retrieve query parameters from the request URL
2. These parameters are usually added to the URL as key=value pairs separated by &, following a ? character.
3. For example, in the URL http://example.com/path?key1=value1&key2=value2, the key1 and key2 parameters can be retrieved using @RequestParam in a Spring controller method.

For request parameters, the URL format typically follows this pattern:

**http://example.com/path/to/endpoint?param1=value1&param2=value2**

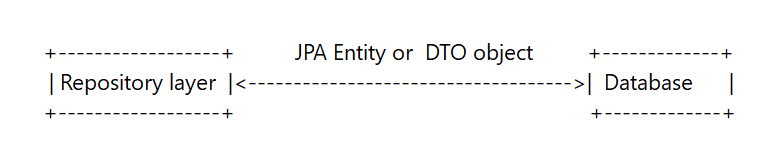
Where the path to the endpoint is specified first, followed by a ? separator, and then the request parameters are specified as key-value pairs separated by &.

@RequestBody Method Parameter

**Service layer**

| Annotation | Description | Used with |
| --- | --- | --- |
| @Service | Indicates that the class is a service that holds the business logic of the application. | Class |
| @Transactional | Indicates that a method should be executed within a transaction. | Method |
| @Async | Indicates that a method should be executed asynchronously. | Method |
| @PostConstruct | Indicates that a method should be called after the bean is initialized. | Method |
| @PreDestroy | Indicates that a method should be called before the bean is destroyed. | Method |
| @Qualifier | Specifies which bean should be injected when multiple beans of the same type are present. | Parameter |
| @RequestParam | Binds a request parameter to a method parameter. | Parameter |
| @PathVariable | Binds a URI template variable to a method parameter. | Parameter |
| @RequestBody | Binds the request body to a method parameter. | Parameter |
| @Valid | Validates the method parameter. | Parameter |

**Repository Layer**



* 1. When to define method and when not?
  2. Can we pass dto to repository interface = update case
  3. Can we convert one dto to another

| Annotation | Description | Used with |
| --- | --- | --- |
| @Repository | Indicates that the class is a repository that provides CRUD operations on the database. | Class |
| @Transactional | Indicates that a method should be executed within a transaction. | Method |
| @Query | Defines a custom query that should be executed on the database. | Method |
| @Param | Binds a method parameter to a named parameter in a query. | Parameter |
| @EntityGraph | Specifies the entity graph to use for a query. | Method |
| @Lock | Specifies the lock mode to use for a query. | Method |
| @Modifying | Indicates that a query should modify the database. | Method |
| @Procedure | Defines a stored procedure that should be executed on the database. | Method |
| @NamedStoredProcedureQuery | Specifies a named stored procedure query. | Method |
| @SqlResultSetMapping | Maps the result set of a native query to a class or a set of classes. | Method |

if you search for an object on your repo, for example by Id or Name the named query method returns an object of type T, but if no results are found from your repo, it will return null.

Methods that can return more than one element, will produce an empty collection List<T>(not null)

| Method | Parameters | Return Type |
| --- | --- | --- |
| save/persist | Entity | Entity |
| saveAll/persistAll | Collection of Entities | Collection of Entities |
| findById | ID (Long, Integer, etc.) | Optional of Entity |
| findAll | N/A | Collection of Entities |
| findAllById | Collection of IDs | Collection of Entities |
| count | N/A | Long |
| deleteById | ID (Long, Integer, etc.) | N/A |
| delete | Entity | N/A |
| deleteAll | N/A | N/A |
| existsById | ID (Long, Integer, etc.) | Boolean |
| flush | N/A | N/A |
| getOne | ID (Long, Integer, etc.) | Entity |
| refresh | Entity | N/A |
| findBy{Property} | Property Value | Collection of Entities |
| findFirstBy{Property} | Property Value | Optional of Entity |
| findTopBy{Property} | Property Value | Optional of Entity |
| findBy{Property}And{Property} | Property Values | Collection of Entities |
| findBy{Property}OrderBy{Property} | Property Values | Collection of Entities |
| findBy{Property}And{Property}OrderBy{Property} | Property Values | Collection of Entities |
| findBy{Property}In | Collection of Property Values | Collection of Entities |
| findBy{Property}NotIn | Collection of Property Values | Collection of Entities |
| findBy{Property}IsNull | N/A | Collection of Entities |
| findBy{Property}IsNotNull | N/A | Collection of Entities |

custom\_interface extends jpa\_interface;

custom method implementation

@Repository

public class MyCustomRepositoryImpl extends SimpleJpaRepository<MyEntity, Long> implements MyCustomRepository {

public MyCustomRepositoryImpl(EntityManager entityManager) {

super(MyEntity.class, entityManager);

}

@Override

public List<MyEntity> findByQueryString(String queryString) {

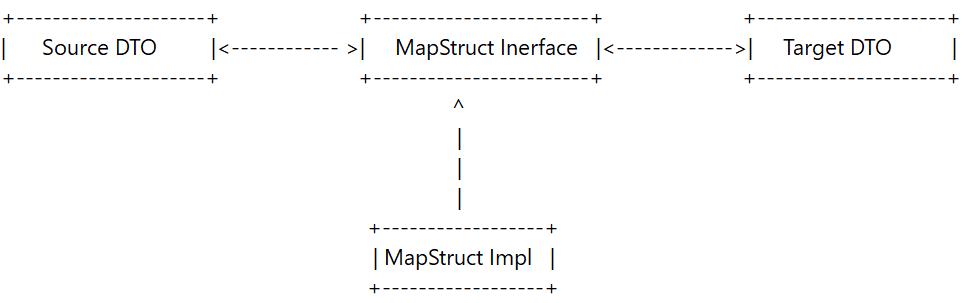
// custom implementation goes here

}

}

| Interface | Description |
| --- | --- |
| CrudRepository | Provides basic CRUD (Create, Read, Update, Delete) operations for entities. |
| PagingAndSortingRepository | Extends CrudRepository and adds methods for pagination and sorting of query results. |
| JpaRepository | Extends PagingAndSortingRepository and adds additional methods for managing entities, such as flushing changes to the database and batch operations. |
| QueryByExampleExecutor | Provides methods for querying entities using an example entity. This interface is useful for creating dynamic queries that are based on the structure of an existing entity. |
| JpaSpecificationExecutor | Provides methods for creating dynamic queries using JPA criteria queries. This interface is useful for building complex queries that cannot be expressed using the repository method naming conventions. |
| JpaContext | Provides access to the underlying EntityManager and JPA Metamodel. This interface can be used to perform low-level JPA operations that are not supported by the higher-level repository interfaces. |
| JpaRepositoryFactory | A factory for creating instances of JpaRepository or its subinterfaces. This interface is used internally by Spring Data JPA to create repository instances. |
| JpaRepositoryImplementation | An implementation of JpaRepository that provides default implementations for all methods defined in the interface. This class is used internally by Spring Data JPA to provide default implementations of repository methods. You can extend this class instead of JpaRepository to provide custom implementations of the default methods. Note that this class is not meant to be used directly in application code. |
| SimpleJpaRepository | An implementation of JpaRepository that provides default implementations for all methods defined in the interface. This class is used internally by Spring Data JPA to provide default implementations of repository methods. You can extend this class instead of JpaRepository to provide custom implementations of the default methods. Note that this class is not meant to be used directly in application code. |

Mapstruct



When mapping different field names, we will need to configure its source field to its target field and to do that, we will need to add *@Mapping* annotation for each field.

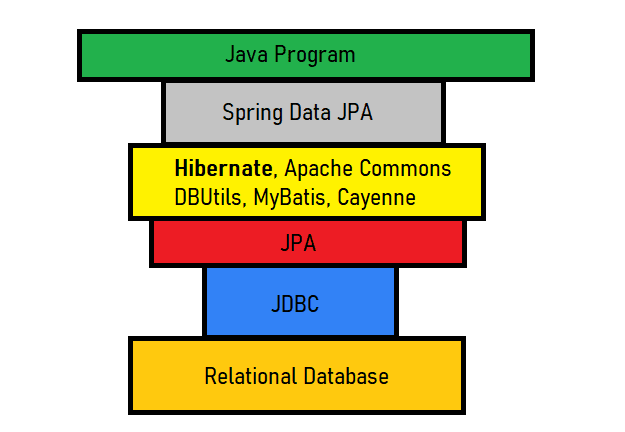
| Annotation | Description | Used with |
| --- | --- | --- |
| @Mapper | Marks the class as a MapStruct mapper interface. | Interface |
| @Mapping | Specifies how to map a source property to a target property. | Method |
| @InheritInverseConfiguration | Inverts the source and target properties of a mapping. | Method |
| @IterableMapping | Specifies how to map an iterable of source objects to an iterable of target objects. | Method |
| @AfterMapping | Provides custom code to be executed after a mapping. | Method |

DTO (Data Transfer object)

JPA Entity

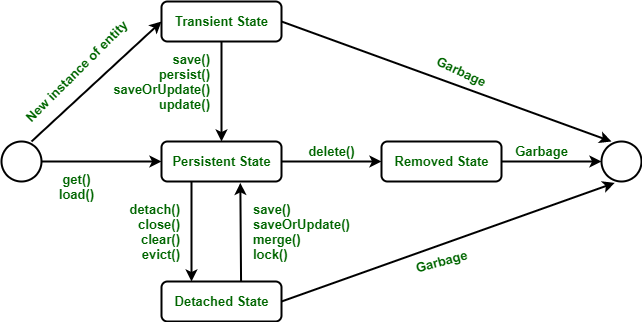
| Annotation | Description | Used with |
| --- | --- | --- |
| @Entity | Marks a Java class as a persistent entity that can be stored in a database. | Class |
| @Table | Specifies the name of the database table that the entity maps to. Can be used to override the default naming strategy. | Class |
| @Id | Marks a field or getter method as the primary key of the entity. | Field or Getter Method |
| @GeneratedValue | Specifies how the primary key value should be generated. Typically used in conjunction with @Id. | Field or Getter Method |
| @Column | Specifies the mapping between a Java object property and a database column. Can be used to override the default naming strategy or to specify a different data type. | Field or Getter Method |
| @OneToOne | Defines a one-to-one relationship between two entities. | Field or Getter Method |
| @OneToMany | Defines a one-to-many relationship between two entities. | Field or Getter Method |
| @ManyToOne | Defines a many-to-one relationship between two entities. | Field or Getter Method |
| @ManyToMany | Defines a many-to-many relationship between two entities. | Field or Getter Method |

**JDBC/JPA/Hibernate**  **|** [**Contents**](#contents)



JPA

JPA is not a tool or framework. It defines a set concepts that can be implemented by another tool or framework. JPA is the interface and Hibernate is the implementation.



JDBC

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

* Register the Driver class
* Create connection
* Create statement
* Execute queries
* Close connection

**Querying Data**

1. Query Methods: These are methods defined on a repository interface that are automatically implemented by Spring Data JPA. The method name determines the query that will be executed. For example, the method name findByLastName(String lastName) will generate a query to find all entities with the specified last name.
2. JPQL (Java Persistence Query Language): This is a type-safe query language that is similar to SQL but operates on entities rather than database tables. JPQL queries can be defined using the @Query annotation on a repository method.
3. Native SQL: Spring Data JPA also supports executing native SQL queries using the @Query annotation. This allows you to execute any SQL query, including complex joins and subqueries.
4. Criteria API: This is a programmatic way of building queries using a type-safe API. It provides a fluent interface for building queries that can be used to dynamically construct complex queries.
5. Named Queries: These are static queries that are defined in the entity class using the @NamedQuery annotation. They can be executed using the EntityManager or by defining a repository method with the same name as the named query.
6. Query by Example: This is a dynamic query mechanism that allows you to create queries based on the properties of an example entity. This can be useful for creating search functionality or building dynamic filters.

DB inpendent

Query Methods > Named Queries > Criteria API > JPQL > Native SQL > Query by Example

**Entity Relationships**

**Pagination and Sorting**

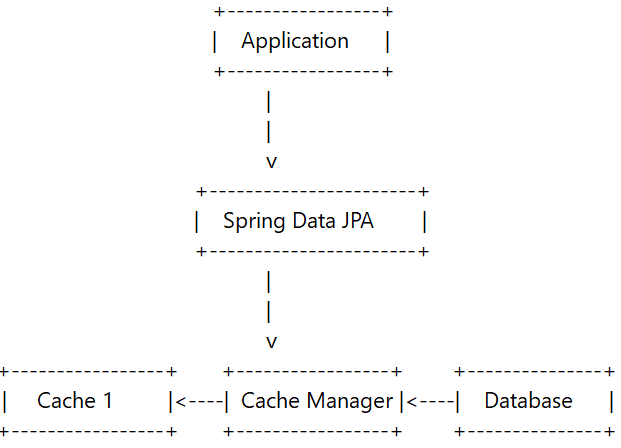
* Pagination and sorting are important features in web applications that deal with large datasets.
* To perform pagination, use the findAll(Pageable pageable) method with a Pageable parameter specifying the page number, page size, and sorting order.
* To perform sorting, use the Sort object with the findAll(Sort sort) method.
* To combine pagination and sorting, pass a Sort object to the PageRequest constructor.
* The Page object returned from pagination contains the requested data and information about the current page, total pages, and total records.

the parameters that can be passed to a Pageable object:

* pageNumber - the number of the page you want to retrieve (starting from 0).
* pageSize - the maximum number of items to be returned per page.
* sort - an optional Sort object that specifies the sorting order of the returned data.
* offset - the offset of the first item to be returned, used in place of pageNumber and pageSize. If offset is used, then pageSize must also be specified.

Page<Type> findAll(Pageable pageable);

**Caching**



* Add the necessary dependencies: First, you'll need to add the Spring Data JPA and caching dependencies to your project. These dependencies include the spring-data-jpa and spring-cache modules.
* Enable caching: Next, you'll need to enable caching in your Spring application by annotating your configuration class with @EnableCaching. This annotation will enable caching support in Spring and activate the caching annotations.
* Configure caching: You can configure the caching behavior by adding the @CacheConfig annotation to your repository interface or implementation class. This annotation allows you to specify the name of the cache and any additional caching properties.
* Annotate repository methods: Finally, you can annotate individual repository methods with the @Cacheable, @CachePut, or @CacheEvict annotations to specify the caching behavior for each method.
* @Cacheable indicates that the method results should be cached. If the same method is called again with the same parameters, the cached result will be returned instead of invoking the method again.
* @CachePut indicates that the method results should be cached, but the method should also be invoked and the returned value should be updated in the cache.
* @CacheEvict indicates that the cached results associated with a given method or a given set of cache entries should be evicted (removed) from the cache.

@Configuration

@EnableCaching

public class AppConfig {

// ...

}

**Transactions and Concurrency**

**Auditing and Versioning**

**Projections**

import lombok.Data;

@Data

public interface CustomerNameProjection {

String getFirstName();

String getLastName();

}

@Repository

public interface CustomerRepository extends JpaRepository<Customer, Long> {

List<CustomerNameProjection> findAllProjectedBy();

}

In this example, the findAllProjectedBy() method returns a list of CustomerNameProjection instances, which contain only the firstName and lastName fields of the Customer entity. When this method is called, Spring Data JPA will automatically generate a query to select only these fields from the database, improving performance and reducing network traffic.

Connection Pooling

* Connection pooling is a cache of database connections maintained by an application server or middleware application.
* Connection pooling is used to improve the performance of database-intensive applications by reusing database connections rather than creating a new connection each time a request is made to the database.
* Creating a new connection to a database can be an expensive operation, involving network connection establishment, authentication, and setting up the database session.
* Connection pooling maintains a set of pre-initialized database connections ready to be used by the application.
* By borrowing a connection from the pool, the application avoids the overhead of creating new connections.
* Connection pools have configurable parameters such as the minimum and maximum number of connections, maximum idle time of a connection, and time interval for checking the health of connections.
* Configuring these parameters can optimize the performance of the application and prevent the pool from running out of resources.
* Spring Boot supports several connection pool implementations, including HikariCP, Apache Tomcat JDBC Connection Pool, and Commons DBCP2.

# HikariCP configuration

spring.datasource.hikari.minimum-idle=5

spring.datasource.hikari.maximum-pool-size=10

# HikariCP connection pool idle and health check configuration

spring.datasource.hikari.idle-timeout=60000

spring.datasource.hikari.connection-timeout=30000

Yes, you can call a stored procedure using Spring Data JPA. Spring Data JPA provides the @Procedure annotation to map a repository method to a stored procedure.

Here's an example of how to call a stored procedure using Spring Data JPA:

1. Define the stored procedure in the Oracle database.
2. Create a repository interface that extends JpaRepository.
3. Add a method to the repository interface with the @Procedure annotation and the name of the stored procedure, like this:

public interface EmployeeRepository extends JpaRepository<Employee, Long> {

@Procedure(name = "update\_salary")

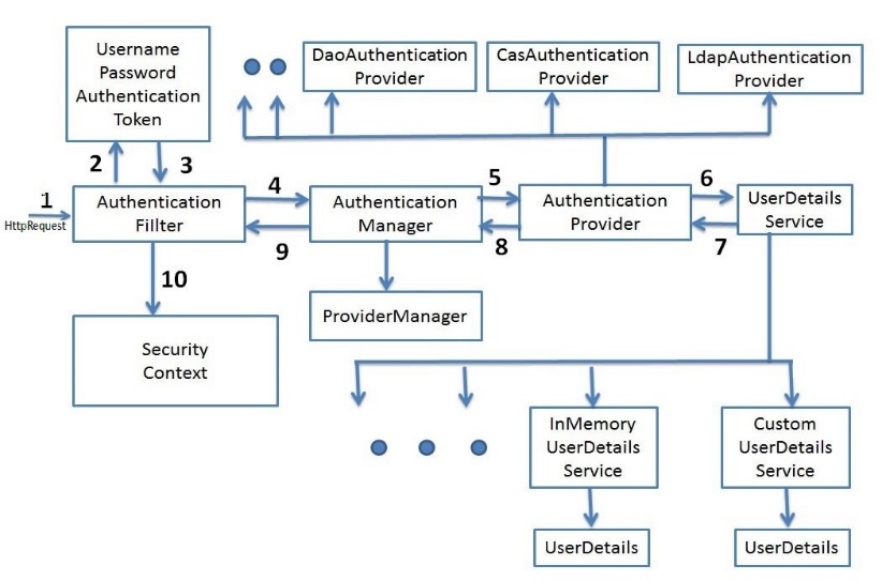
void updateSalary(@Param("employee\_id") Long employeeId, @Param("new\_salary") BigDecimal newSalary);

}

Connecting spring boot with database

How sql and plsql execute

Spring security **|** [**Contents**](#contents)



1. Authentication and Authorization
2. OAuth2 and OpenID Connect
3. CSRF and XSS protection
4. Session management
5. Customization and extensibility
6. Testing Spring Security
7. Best practices and security considerations
8. Cookies are small text files stored on a user's computer by a web browser for storing user preferences and authentication.
9. Sessions are similar to cookies, but stored on the server-side and contain user information. Sessions are not used in RESTful APIs because they violate the statelessness constraint.
10. Tokens are a unique identifier exchanged between client and server for authentication.
11. Cookies and sessions are used for server-side state management, while tokens are used for stateless authentication.
12. Cookies and tokens are stored on the client-side, while sessions are stored on the server-side.

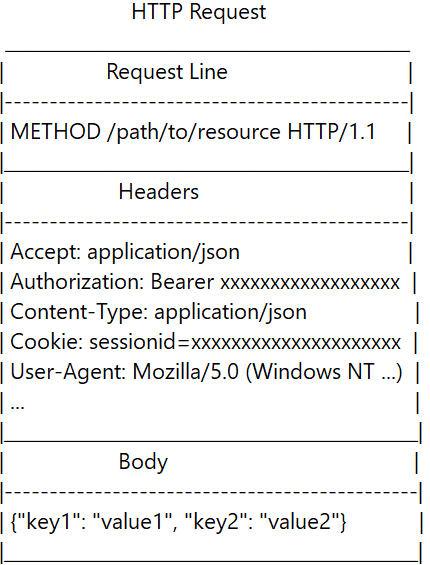
Restful API

1. It follows the client-server model, meaning the client initiates a request and the server responds accordingly.
2. RESTful APIs are stateless, meaning that the server does not store any information about the client session. Instead, each request includes all the necessary information for the server to complete it.
3. RESTful APIs typically use JSON (JavaScript Object Notation) or XML (Extensible Markup Language) as the data format for requests and responses.

In the case of a RESTful API, the backend is usually designed to be stateless, meaning that each request from the client contains all of the necessary information for the server to process the request and generate a response, without relying on any stored state or context on the server-side.

Response Header

HTTP is stateless



A request to a server typically includes the following information:

1. HTTP method (such as GET, POST, PUT, DELETE)
2. Request URL
3. Request headers (such as Accept, Content-Type, Authorization)
4. Request body (for POST, PUT and DELETE requests)
5. Query parameters (for GET requests)
6. Path parameters (for RESTful APIs)
7. Cookies (for session management)

Request Header

Authentication header

The Authentication header typically consists of two main parts:

1. Authentication Scheme: It specifies the type of authentication being used, such as Basic, Bearer, or Digest.
2. Credentials: It contains the user's credentials in the form of a token or other authentication data, such as a username and password, encrypted or hashed for security reasons.

Authorization: Bearer <token>

In this case, "Bearer" is the key and "<token>" is the string value.

Bearer is a type of token-based authentication mechanism used in web applications, particularly in RESTful APIs. In the context of HTTP authentication, a bearer token is an encoded string that is typically included in the Authorization header of a request, preceded by the word "Bearer".

A JWT token consists of three parts separated by dots:

1. The header contains information about how the JWT is signed and encrypted, such as the signing algorithm and the type of token.

{

"alg": "HS512",

"typ": "JWT"

}

1. The payload contains the data that is being transmitted, such as user ID or access permissions.cli

{

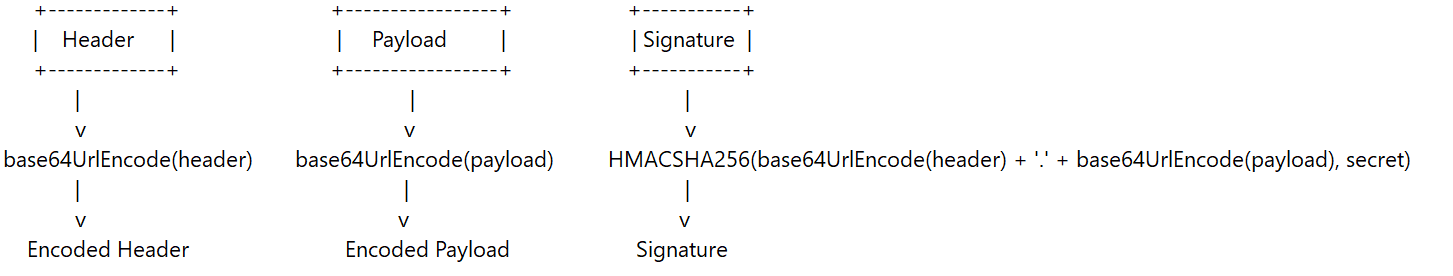
"sub": "1234567890",

"name": "John Doe",

"iat": 1516239022

}

1. The signature is used to verify that the sender of the JWT is who it claims to be and to ensure that the message has not been tampered with. The signature is generated by hashing the header and payload along with a secret key that only the server knows



{

"header": {

"alg": "HS256",

"typ": "JWT"

},

"payload": {

"sub": "anonymous",

"aud": ["ROLE\_USER", "ROLE\_USER2"],

"iss": "admin",

"iat": 1623677827,

"exp": 1623678427

},

"signature": "c9Rvs4jXGATPP6lSh2t6HlWs5Vx2wsOTFs4hUHYvcX0"

}

Verify the signature:

1. The server extracts the signature from the received JWT token.
2. The server applies the same signing algorithm and uses its own secret key to generate a new signature based on the extracted header and payload.
3. The server compares the newly generated signature with the extracted signature from the token.
4. If the two signatures match, it confirms that the token has not been tampered with and the signature is valid.

Check the header and payload:

1. After verifying the signature, the server decodes the Base64Url-encoded header and payload parts of the token.
2. The server can access the information contained in the header and payload, such as the subject, expiration time, issuer, and other custom claims.
3. The server can perform additional checks on the decoded header and payload, such as validating the expiration time or issuer.

User Registration:

1. User provides necessary information (username, email, password).

2. Server securely stores the user's details in a database.

User Login:

1. User enters credentials (username, password).

2. Server authenticates the provided credentials against the stored data.

Authentication and Token Generation:

1. Server verifies the user's identity and permissions.

2. If authentication is successful:

- Server generates a JWT token containing user information (username, authority/role).

- The token is signed using a secret key known only to the server.

- The server returns the generated JWT token to the client.

Client Navigation and Request:

1. Client navigates through the application.

2. Client includes the JWT token in the request headers for subsequent requests.

JWT Token Validation:

1. Server receives the request and extracts the JWT token from the request headers.

2. Server validates the authenticity of the token by verifying the signature using the secret key.

Extracting Role and Authorization:

1. Once the token is validated:

- Server extracts the user information (username, authority/role) from the token.

- The extracted information helps determine the user's authorization level.

Resource Access Control:

1. Server performs further authorization checks based on the extracted role.

2. If the user has the necessary authority:

- Server allows the requested action or resource access.

3. If the user lacks the necessary authority:

- Server returns an appropriate error response.

Secret\_Key

claim

1. OAuth is an authorization framework that enables third-party applications to access restricted resources on behalf of a resource owner.
2. OpenID Connect is an authentication layer that sits on top of OAuth and provides a standard way for applications to authenticate users.
3. JWT is a format for representing claims securely between two parties, which is often used as a token format for OAuth and OpenID Connect.

Here's a brief explanation of CSRF, XSS, and CORS:

1. CSRF (Cross-Site Request Forgery): An attack that tricks a user into unintentionally executing an unwanted action on a website where they are authenticated. This can be prevented by adding a unique token to each form or request that only the server knows.
2. XSS (Cross-Site Scripting): An attack that injects malicious code into a web page, which can then be executed by other users who visit the page. This can be prevented by properly sanitizing user input and encoding output.
3. CORS (Cross-Origin Resource Sharing): A security feature that allows a web page from one domain to request resources from another domain. It involves adding headers to HTTP requests and responses to specify which origins are allowed to access resources.

OAuth

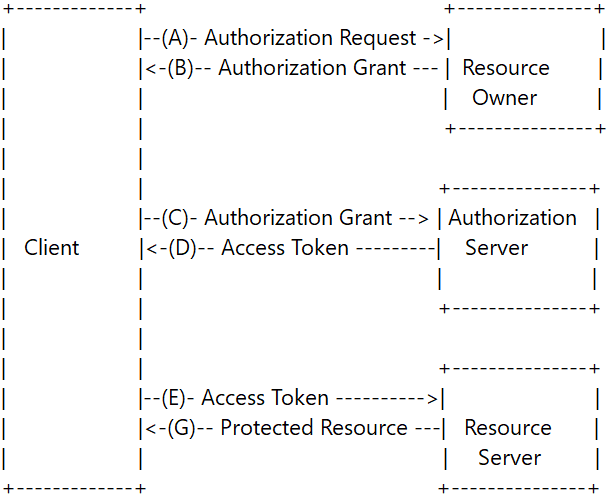
1. A user wants to access a protected resource on a client application (e.g., a mobile app or a web app).
2. The client application redirects the user to an OAuth server to request authorization to access the protected resource.
3. The OAuth server asks the user to log in and authenticate themselves.
4. Once the user is authenticated, the OAuth server asks the user to grant permission to the client application to access the protected resource.
5. If the user grants permission, the OAuth server issues an access token and sends it back to the client application.
6. The client application sends the access token to the resource server to access the protected resource.
7. The resource server verifies the access token and either grants or denies access to the protected resource.
8. If access is granted, the resource server sends the requested resource back to the client application.

when dealing with more complex use cases involving multiple clients, multiple resources, and different authorization requirements, OAuth2 provides a more standardized and robust solution.

So, in short, while both OAuth and JWT can be used for authentication, they have different approaches to generating and handling tokens. With OAuth, the authorization server generates and signs the access token, while with JWT, the token is generated and signed by the server.

The OAuth authorization code flow works as follows:

1. The client application initiates the flow by redirecting the user to an authorization server.
2. The user authenticates with the authorization server and grants permission to the client application to access their resources.
3. The authorization server generates an authorization code and returns it to the client application via a callback URL.
4. The client application exchanges the authorization code for an access token by sending a request to the authorization server with the authorization code and client credentials.
5. The authorization server validates the authorization code and returns an access token to the client application.



***JWT***

To implement JWT authentication and authorization in an Angular and Spring Boot application, you can follow the following steps:

1. Implement a JWT authentication service in Angular. This service should handle the authentication process by sending the user's credentials to the server and receiving a JWT token in response. The JWT token should be stored in local storage or session storage and included in the Authorization header of all subsequent requests to the server.
2. Implement a JWT authentication filter in Spring Boot. This filter should intercept incoming requests and validate the JWT token in the Authorization header. If the token is valid, the filter should extract the user's identity from the token and set it in the security context of the request.
3. Implement a JWT authorization filter in Spring Boot. This filter should intercept incoming requests and check the user's identity in the security context against the required roles or permissions for the requested resource. If the user is authorized, the filter should allow the request to proceed. If the user is not authorized, the filter should return an HTTP error response.
4. Secure the endpoints of your Spring Boot application using the JWT authorization filter. You can do this by adding the @PreAuthorize annotation to your controller methods, along with the required roles or permissions for that endpoint.
5. Use the JWT authentication service in Angular to handle user login and logout. When the user logs in, the service should send their credentials to the server and receive a JWT token in response. The token should be stored in local storage or session storage and included in the Authorization header of all subsequent requests. When the user logs out, the token should be removed from storage and the Authorization header cleared.
6. Use Angular guards to secure routes and components that require authentication or authorization. You can create a guard that checks whether the user is logged in and has the required roles or permissions to access the requested route or component. If the user is not authorized, the guard can redirect them to a login page or show an error message.

In Spring Boot, you can create a JWT token using the Java JWT library, which provides a set of classes and methods for generating, signing, and verifying JWT tokens.

Here's an example of how to create a JWT token in Spring Boot:

1. Add the Java JWT library as a dependency in your Spring Boot project. You can add the following dependency to your pom.xml file if you're using Maven.

2. Create a method that generates a JWT token with the required claims

3. Use the generateToken method to create a JWT token in your Spring Boot application. You can then include this token in the response to a successful authentication request or send it to the client as a cookie or header value.

Authentication

Angular:

1. Use a secure method to transmit user credentials over the network, such as HTTPS.
2. Store the JWT token securely in the browser, such as in a HttpOnly cookie or local storage with proper precautions against XSS attacks.
3. Implement an authentication interceptor that attaches the JWT token to the Authorization header of each HTTP request.
4. Handle authentication errors and display appropriate error messages to the user.
5. Provide a logout functionality that clears the JWT token from the browser.

Spring Boot:

1. Validate user credentials and generate a JWT token upon successful authentication.
2. Store the user's password securely using a password hashing function such as BCrypt.
3. Ensure that JWT tokens are not susceptible to common attacks, such as tampering or replay attacks.
4. Configure Spring Security to use the JWT token for authentication and provide appropriate access control rules based on user roles.
5. Implement a refresh token mechanism to issue a new JWT token when the old one is about to expire.

Angular:

1. Create a login component that displays a form for users to enter their credentials (username and password).
2. Implement a login service that sends an HTTP POST request to the Spring Boot backend to authenticate the user and retrieve a JWT token.
3. Store the JWT token securely in the browser, such as in a HttpOnly cookie or local storage with proper precautions against XSS attacks.
4. Implement an authentication interceptor that attaches the JWT token to the Authorization header of each HTTP request.
5. Use guards to restrict access to certain routes or features based on the user's authentication status.

Spring Boot:

1. Configure Spring Security to use JWT for authentication.
2. Create a user entity that stores user information, such as username and encoded password.
3. Create a UserDetailsService implementation that loads user information from the database or another data source.
4. Implement an authentication endpoint that accepts user credentials, validates them, and returns a JWT token upon successful authentication.
5. Define access control rules using Spring Security based on user roles and permissions.
6. Implement a refresh token mechanism to issue a new JWT token when the old one is about to expire.

Authorization

Angular:

1. Define user roles and permissions in your application.
2. Use guards to restrict access to certain routes or features based on the user's role and permissions.
3. Display appropriate messages or UI elements to the user based on their role and permissions.

Spring Boot:

1. Define user roles and permissions in your application.
2. Implement access control rules using Spring Security, based on user roles and permissions.
3. Configure method-level security to restrict access to certain methods based on user roles and permissions.
4. Use Spring Security annotations, such as @PreAuthorize and @PostAuthorize, to secure individual methods based on custom conditions.

using guards on the frontend and access control rules on the backend,

1. The user enters their registration information (such as username, email, and password) in a registration form in the Angular frontend.
2. The registration form data is sent as a POST request from the Angular frontend to the Spring Boot backend over HTTPS.(string format??)
3. The Spring Boot backend receives the registration data and validates it, checking that the data is complete and valid, and that the username and email are not already registered in the system.
4. If the registration data is valid, the Spring Boot backend generates a secure password hash using a password hashing algorithm (such as bcrypt or PBKDF2) and stores the user's information (including the password hash) in the database.
5. The Spring Boot backend sends a response back to the Angular frontend indicating that the user has been registered successfully.
6. The user can then log in to the system using their registered username and password.

Angular:

1. The user enters their login credentials (e.g., username and password) in the Angular login form.
2. When the user clicks the login button, the Angular frontend sends an HTTP POST request to the Spring Boot backend API, containing the user's credentials in the request body.
3. The backend API receives the login request and verifies the user's credentials. If the credentials are invalid, the backend returns an error response to the frontend. If the credentials are valid, the backend generates a JSON Web Token (JWT) and sends it back to the frontend in the response.
4. The frontend receives the JWT and stores it in the browser's local storage or a cookie.
5. The frontend updates the application state to indicate that the user is logged in, and navigates to the appropriate authenticated view.

Spring Boot:

1. The Spring Boot backend API receives the login request from the Angular frontend.
2. The API uses the user's credentials to authenticate the user. This may involve retrieving the user's hashed password from the database and comparing it to the entered password.
3. If the user's credentials are valid, the API generates a JWT using a secure secret key and a payload containing information about the user and their permissions.
4. The API sends the JWT back to the Angular frontend in an HTTP response.
5. The Angular frontend receives the JWT and stores it in the browser's local storage or a cookie, so that it can be included in subsequent requests to the backend API as an authorization header.
6. The backend API validates the JWT on subsequent requests from the Angular frontend, to ensure that the user is authorized to access the requested resource.

Top of Form

Bottom of Form

In Spring Boot, authorization can be implemented at different levels depending on the requirements of the application:

1. Method-level authorization: This involves securing individual methods within a class. In Spring Security, you can use the @PreAuthorize and @PostAuthorize annotations to specify access control policies for methods. These annotations allow you to specify conditions under which a method can be called, based on the user's role, permissions, or other attributes.
2. Class-level authorization: This involves securing an entire class of methods. In Spring Security, you can use the @Secured annotation to specify a list of roles that are allowed to access the methods in the class.
3. URL-level authorization: This involves securing individual URLs or URL patterns. In Spring Security, you can use the antMatchers() method to specify URL patterns and access control policies for each pattern. You can specify conditions under which a user is allowed to access a URL, based on their role, permissions, or other attributes.
4. Global authorization: This involves specifying global access control policies for the entire application. In Spring Security, you can use the WebSecurityConfigurerAdapter to configure global security policies for the application.

Order of granularity method > class > url > global

1. After the user logs in, the Spring Boot back-end generates a JWT token and sends it to the Angular front-end.
2. The Angular front-end stores the JWT token in local storage or session storage for future use.
3. When the user attempts to access a protected resource, such as a page or functionality that requires certain roles or permissions, the front-end retrieves the JWT from local storage or session storage.
4. The front-end decodes the JWT to extract the user's roles and other claims.
5. The front-end checks whether the user's roles match the required roles for accessing that resource.
6. If the user's roles match the required roles, the front-end grants access to the resource. If not, the front-end redirects the user to a login page or shows an error message.
7. Once the Angular front-end decodes the JWT and extracts the user's role, it can use an Angular guard to check if the user is authorized to access a particular route or component.
8. When the user interacts with the front-end to make an API request, the front-end sends the JWT token with each request as an "Authorization" header.
9. The Spring Boot back-end receives the request and extracts the JWT token from the "Authorization" header.
10. The Spring Boot back-end decodes the JWT to extract the user's roles and other claims.
11. The Spring Boot back-end checks whether the user's roles match the required roles for accessing the protected resource.
12. If the user's roles match the required roles, the Spring Boot back-end allows the request to proceed. If not, the Spring Boot back-end returns a 401 Unauthorized response or redirects the user to a login page or shows an error message.

Authentication:

1. Verifies the identity of a user and ensures they are who they claim to be.
2. Achieved using an authentication filter.
3. Intercepts incoming requests and checks whether the user has provided valid credentials.
4. Spring Boot provides several built-in authentication filters, including Basic Authentication, Form-Based Authentication, and OAuth2 Authentication.

Type of Authentication Filter

1. Basic Authentication Filter: This filter uses HTTP Basic Authentication to authenticate the user. The user's credentials are sent in the HTTP header in an encoded format.
2. Form-Based Authentication Filter: This filter authenticates the user using a login form that collects the user's credentials. It uses cookies or URL rewriting to maintain the user's session.
3. OAuth2 Authentication Filter: This filter uses OAuth2 to authenticate the user. The user is redirected to a third-party authentication provider to log in and grant access to the application.
4. JWT Authentication Filter: This filter uses JSON Web Tokens (JWT) to authenticate the user. The user's credentials are stored in the JWT and sent with each request.
5. LDAP Authentication Filter: This filter authenticates the user against an LDAP directory.
6. SAML Authentication Filter: This filter uses the Security Assertion Markup Language (SAML) to authenticate the user. The user is redirected to a SAML identity provider to log in and grant access to the application.

Authorization:

1. Determines whether a user is allowed to perform a particular action or access a particular resource.
2. Achieved using an authorization filter.
3. Examines the user's credentials and determines whether they have the necessary permissions to perform the requested action.
4. Spring Boot provides several built-in authorization filters, including Role-Based Access Control (RBAC) and Permission-Based Access Control (PBAC).

Type of Authorization filter

1. Role-Based Access Control (RBAC): This filter restricts access to certain resources based on the user's role or group membership. Users are assigned roles, and permissions are granted to roles.
2. Permission-Based Access Control (PBAC): This filter restricts access to certain resources based on the user's permissions. Users are assigned individual permissions, and permissions are granted to users.
3. IP Address Filtering: This filter restricts access to certain resources based on the user's IP address.
4. Session Management: This filter manages user sessions and controls access to resources based on the user's session state.
5. Token-Based Access Control: This filter restricts access to certain resources based on a token that is passed in with the request. The token contains the user's permissions or roles.
6. Attribute-Based Access Control: This filter restricts access to certain resources based on attributes of the user or the resource being accessed.

**Role vs Permission**

In general, roles are higher-level constructs that are used to group users with similar access requirements, while permissions are lower-level constructs that determine specific access rights. Roles are often used to simplify the management of access control, while permissions are used to fine-tune access control and provide more granular control over resources.

list of the main beans that you may need to configure in a typical Spring Security setup and what they're used for:

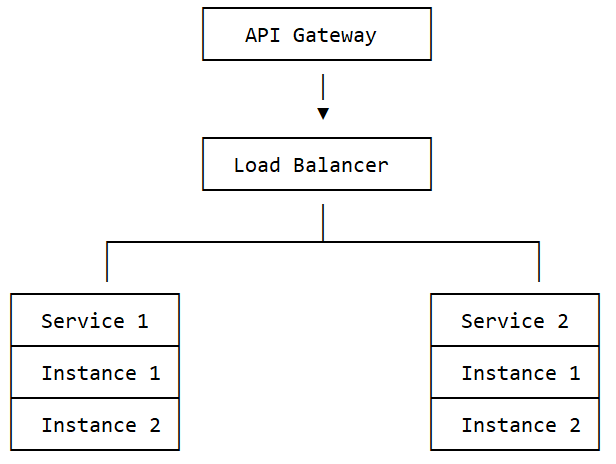
1. **WebSecurityConfigurerAdapter**: This is the main class for configuring Spring Security. You can extend this class and override its methods to configure various aspects of security, such as authorization rules, authentication providers, and form login settings.
2. **UserDetailsService**: This interface is used to load user information from a data source. You can implement this interface to provide your own user details service, or you can use one of the pre-built implementations provided by Spring Security.
3. **PasswordEncoder**: This interface is used to encode and decode passwords. Spring Security provides several built-in implementations, such as BCryptPasswordEncoder, Pbkdf2PasswordEncoder, and StandardPasswordEncoder. You can also provide your own implementation if needed.
4. **AuthenticationManagerBuilder**: This class is used to configure the AuthenticationManager, which is responsible for authenticating users. You can use this class to specify the user details service, password encoder, and authentication providers.
5. **AuthenticationProvider**: This interface is used to define a strategy for authenticating users. Spring Security provides several built-in implementations, such as DaoAuthenticationProvider, LdapAuthenticationProvider, and JaasAuthenticationProvider. You can also provide your own implementation if needed.
6. **SecurityContextHolder**: This class is used to store the Authentication object for the current user. You can use this class to retrieve the current user's authentication information, such as their username, password, and roles.
7. **FilterChainProxy**: This class is used to manage the filters in the Spring Security filter chain. The filter chain is responsible for processing HTTP requests and enforcing security constraints. You can configure the filter chain by adding or removing filters as needed.

.

Spring Cloud

1. Service registration and discovery
2. Distributed configuration management
3. Circuit Breakers
4. Service-to-service communication
5. API gateway
6. Distributed tracing and logging
7. Best practices and architecture patterns

Microservices



API gateway

-Zuul

-spring cloud gateway

Load balancer

-Ribbon

service discovery

-Eureka discovery client

-Eureka discovery server

circuit breaker, Fault tolerance

-Hystrix

Microservices communication

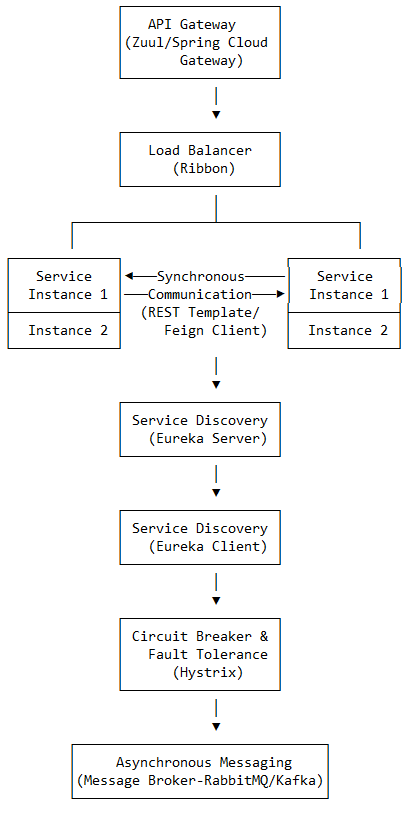
Synchronous Way

* REST Template - spring.framework.web.client
* Feign

Asynchronous Way

* Message-Broker
* RabbitMQ
* ActiveMQ
* Kafka

RabbitMQ is a message queue. Message queues make sure messages reach their destination, persisting during downtimes and load-balancing between multiple worker processes. You generally want a message queue if you have several processes doing different types of work, and you need a way to distribute the workload.



Logger

@Slf4j is a Lombok annotation that automatically generates a logger field in a Java class. It can be used to quickly and easily add logging capabilities to a class without the need to write boilerplate code for a logger object.

* + Configurable
  + Log levels
  + Log Destination

Swagger

| Annotation | Description | Used with |
| --- | --- | --- |
| @Api | Provides metadata about the API. | Class |
| @ApiOperation | Provides metadata about an API operation (i.e., a method). | Method |
| @ApiParam | Provides metadata about a parameter in an API operation. | Parameter |
| @ApiResponse | Provides metadata about a possible response from an API operation. | Method |
| @ApiModel | Provides metadata about a model (i.e., a class). | Class |
| @ApiModelProperty | Provides metadata about a property in a model. | Field |

* + Develop APIs
  + Interact with APIs
  + Document APIs

Scheduler

*\* CRON EXPRESSION DESCRIPTION  
\* ┌───────────── second (0-59)  
\* │ ┌───────────── minute (0 - 59)  
\* │ │ ┌───────────── hour (0 - 23)  
\* │ │ │ ┌───────────── day of the month (1 - 31)  
\* │ │ │ │ ┌───────────── month (1 - 12) (or JAN-DEC)  
\* │ │ │ │ │ ┌───────────── day of the week (0 - 7)  
\* │ │ │ │ │ │ (or MON-SUN -- 0 or 7 is Sunday)  
\* │ │ │ │ │ │  
\* \* \* \* \* \* \*  
\** ***@Scheduled(cron*** *= "\* \* \* \* \* \*")*

“\*” represents all values

“,” separates values

“-“ represents a range of value

“/” represents increaments

Second:minutes:hour:day of month:month:day of week

By default, Spring will use the server's local time zone for the cron expression. However, we can use the zone attribute to change this timezone:

@Scheduled(cron = "0 15 10 15 \* ?", zone = "Europe/Paris")

Yml vs properties file

| Feature | YAML | Properties |
| --- | --- | --- |
| Syntax | Structured, indentation-based | Simple key-value pairs |
| Comments | Yes, starts with '#' | Yes, starts with '#' |
| Nesting | Yes | No |
| Lists | Yes, using '-' | No |
| Multi-line values | Yes, using ' | ' or '>' |
| Profiles | Multiple documents or '---' separator | Separate files or 'spring.profiles.active' |
| Spring Boot placeholders | Yes, using '${}' | Yes, using '${}' |

Yaml is best

caching, lazy loading, and optimistic locking.

Spring Boot reads configuration properties from a variety of sources, and it has a predefined order in which it looks for them. The order of precedence is as follows:

1. Command line arguments - Properties passed as command line arguments, using the format --property=value.
2. Java System properties - Properties set using System.setProperty(property, value) in your application code or passed as system properties when starting the JVM, using the format -Dproperty=value.
3. OS environment variables - Properties set as environment variables in the operating system.
4. application.properties or application.yml file - Configuration files located in the classpath of your application, which can be customized for different profiles.
5. application-{profile}.properties or application-{profile}.yml file - Profile-specific configuration files, where {profile} is the name of the profile. For example, application-dev.properties for the "dev" profile.
6. @PropertySource annotations - Configuration properties defined in external property files, using @PropertySource annotations in your Spring Boot application code.
7. @ConfigurationProperties annotations - Configuration properties defined using @ConfigurationProperties annotations in your Spring Boot application code.

By default, Spring Boot looks for application.properties or application.yml files in the root of the classpath of your application. However, you can also specify a different location for these files by setting the spring.config.name and spring.config.location properties in your application.properties or application.yml file or as command line arguments.

For example, to look for myapp.properties and myapp.yml files in the config directory of your application, you can use the following properties:

# Set the name of the configuration files

spring.config.name=myapp

# Set the location of the configuration files

spring.config.location=classpath:/config/

Note that the order of precedence can be modified by setting the spring.config.additional-location property to specify additional locations for configuration files.

Oracle **|** [**Contents**](#contents)

Command line interface

(Interface) RDBMS File

SQL(like) ---> Oracle SQL developer tool ----------------------------> oracle database (SQLPlus) -----> database

MySQL workbench ------------------------------------------> MySQL(mysql) -----------------> database

Microsoft SQL server management studio ----> Microsoft SQL server(sqlcmd) ----> database

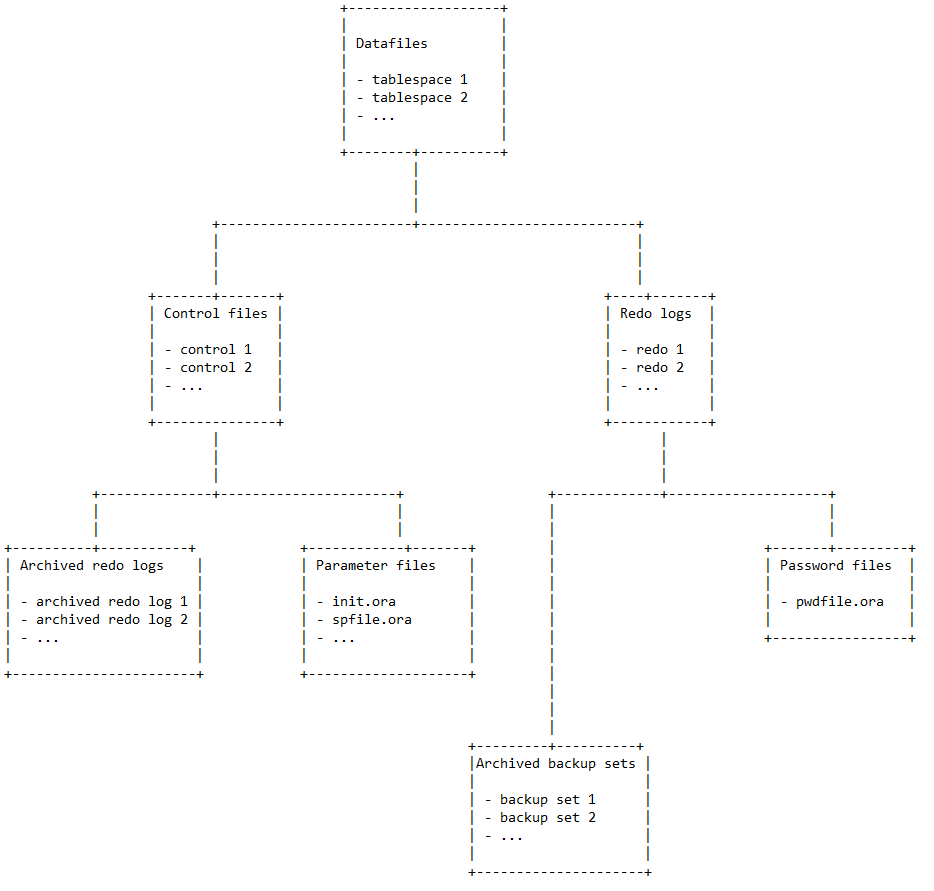
No SQL

NoSQL ------> Mongo dB compass -----------------------------------> Mongo dB(mongo shell)--------> database

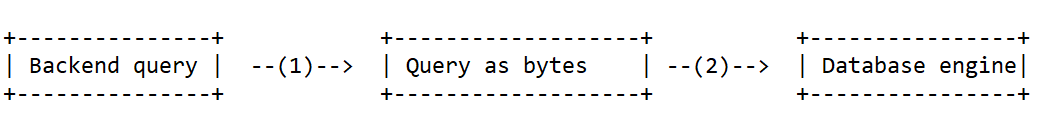
“Oracle Database should be installed in a directory that does not have any spaces in its name. Otherwise, the setup.exe file may not open”.

Oracle database files are physical files used by the Oracle Database to store data, metadata, and configuration information. These files include:

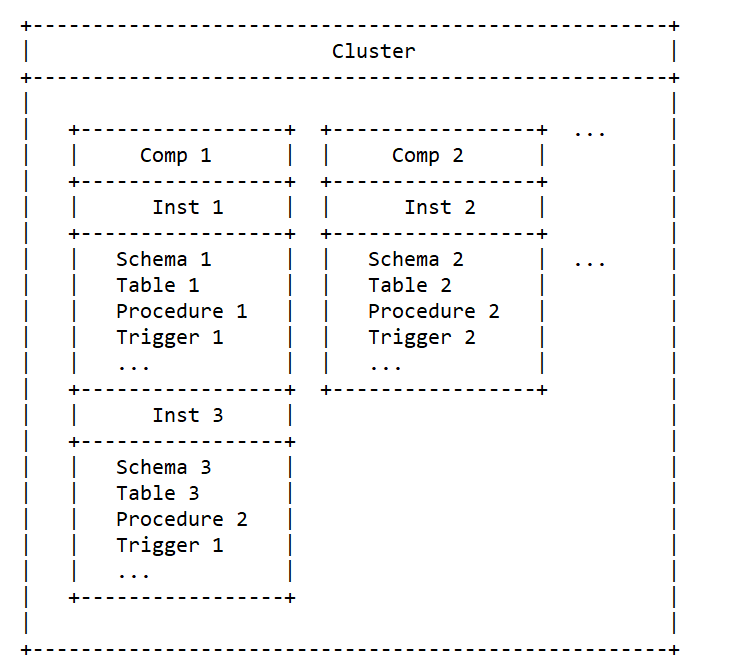
1. Datafiles: These files contain the actual data stored in the database. Each tablespace in the database has one or more datafiles associated with it.
2. Control files: These files contain metadata about the database, such as the database name, the datafiles associated with the database, and the redo log files.
3. Redo log files: These files contain a record of all changes made to the database. They are used to recover the database in the event of a failure.
4. Archived redo log files: These are copies of the redo log files that have been archived and stored in a separate location. They are used for point-in-time recovery and disaster recovery.
5. Parameter files: These files contain configuration information for the database instance, such as the location of the control files and the initialization parameters.
6. Password files: These files contain encrypted passwords for database users with administrative privileges.
7. Archived backup sets: These are backups of the database that have been archived and stored in a separate location. They are used for disaster recovery and point-in-time recovery.



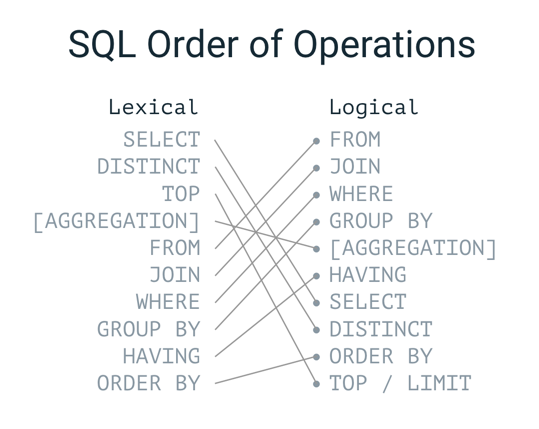
Oracle database save database in oradata\sample by



In Oracle, a datafile can be created with a specific size limit, and when that limit is reached, the file can be set to automatically extend in size, up to a maximum specified size. If the maximum size is reached, another datafile can be added to the tablespace to continue storing data. This allows for the efficient use of disk space and helps to prevent data loss due to running out of space.



SQL is used for database querying and manipulation, while PL/SQL is used for programming and automation within the database.



From - table level

where - row level

select - column level

**Oracle Clauses**

Distinct

Oracle DISTINCT clause is used to remove the duplicate records from the result set. It is only used with SELECT statement.

From

FROM clause is a mandatory clause in SELECT expression. It specifies the tables from which data is to be retrieved.

Order BY

In Oracle, ORDER BY Clause is used to sort or re-arrange the records in the result set. The ORDER BY clause is only used with SELECT statement.

**SELECT** expressions

**FROM** tables

**WHERE** conditions

**ORDER** **BY** expression [ **ASC** | **DESC** ];

**expressions:** It specifies columns that you want to retrieve.

**tables:** It specifies the table name from where you want to retrieve records.

Group By

In Oracle GROUP BY clause is used with SELECT statement to collect data from multiple records and group the results by one or more columns.

Having By

In Oracle, HAVING Clause is used with GROUP BY Clause to restrict the groups of returned rows where condition is TRUE.

**Oracle set Operators**

Union

In Oracle, UNION operator is used to combine the result sets of two or more Oracle SELECT statements. It combines both SELECT statement and removes duplicate rows between them. If you don’t want to remove duplicates, uses Oracle Union All operator.

Each SELECT statement within the UNION operator must have the same number of fields in the result sets with similar data types.

Union all

Intersect

In Oracle, INTERSECT Operator is used to return the results of 2 or more SELECT statement. It picks the common or intersecting records from compound SELECT queries.

Minus

In Oracle, MINUS operator is used to return all rows in the first SELECT statement that are not returned by the second SELECT statement.

**Oracle Joins**

There may be at least one join condition either in the FROM clause or in the WHERE clause for joining two tables. It compares two columns from different tables and combines pair of rows, each containing one row from each table, for which join condition is true.

## Types of Joins

Cross product + conditions

* Inner Joins (Simple Join)
* Outer Joins
  + Left Outer Join (Left Join)

Natural join + left table data

* + Right Outer Join (Right Join)

Natural join + Right table data

* + Full Outer Join (Full Join)

Natural join + Full table data

* Equijoins
* Self Joins
* Cross Joins (Cartesian Products)
* Anti joins
* Semi joins

Inner join

Inner Join is the simplest and most common type of join. It is also known as simple join. It returns all rows from multiple tables where the join condition is met.

In Oracle, there are several types of database objects that can be used to encapsulate and execute complex database operations:

1. Functions: A function is a named PL/SQL block that returns a single value. A function can accept input parameters, execute one or more SQL statements, and return a single value as a result. Functions are commonly used to perform calculations or transformations on data, and they can be called from other SQL statements, PL/SQL blocks, or client applications.
2. Procedures: A procedure is a named PL/SQL block that performs one or more specific tasks or operations. A procedure can accept input parameters, execute one or more SQL statements, and does not necessarily return a value. Procedures are commonly used to encapsulate and execute complex database operations that involve multiple SQL statements, control structures, and error handling logic.
3. Triggers: A trigger is a named PL/SQL block that is executed automatically in response to certain database events, such as an insert, update, or delete operation on a table. Triggers can be used to enforce business rules, perform data validation or transformation, or generate audit trails.
4. Packages: A package is a named collection of related database objects, such as functions, procedures, and variables, that can be used together as a single unit. Packages can be used to encapsulate and organize complex database logic, and they can be shared across multiple applications or database modules.
5. Views: A view is a named query that produces a result set based on one or more tables or other views. Views can be used to simplify complex queries, provide a consistent interface to data, or enforce data security policies.

In Oracle, you can create functions, procedures, triggers, packages, and views using the SQL CREATE statement. The syntax of the CREATE statement depends on the type of object you want to create

Table

**PLSQL**

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling>

END;

To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code.

PL/SQL Program Units

A PL/SQL unit is any one of the following −

* PL/SQL block
* Function
* Package
* Package body
* Procedure
* Trigger
* Type
* Type body
* Cursor

Data type

Scalar

1. Numeric
2. Character
3. Boolean
4. Datetime

Large Object

Composite

Reference

Variable

## Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following −

* The **DEFAULT** keyword
* The **assignment** operator

variable\_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial\_value]

counter binary\_integer := 0;

greetings varchar2(20) DEFAULT 'Have a Good Day';

**Local variables** − Variables declared in an inner block and not accessible to outer blocks.

**Global variables** − Variables declared in the outermost block or a package

## PL/SQL Literals

PL/SQL, literals are case-sensitive. PL/SQL supports the following kinds of literals −

* Numeric Literals
* Character Literals
* String Literals
* BOOLEAN Literals
* Date and Time Literals

PL/SQL – Operators

* Arithmetic operators
* Relational operators
* Comparison operators
* Logical operators
* String operators

PL/SQL – Conditions

A subprogram can be created −

* At the schema level
* Inside a package
* Inside a PL/SQL block

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

* **Functions** − These subprograms return a single value; mainly used to compute and return a value.
* **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

## Procedures

## CREATE [OR REPLACE] PROCEDURE procedure\_name

## [ (parameter\_name [IN | OUT | IN OUT] type [, ...]) ]

## { IS | AS }

## [declaration\_section]

## BEGIN

## executable\_section

## [EXCEPTION

## exception\_section]

## END [procedure\_name];

The AS keyword is used instead of the IS keyword for creating a standalone procedure

## Executing a Standalone Procedure

A standalone procedure can be called in two ways −

* Using the **EXECUTE** keyword
* Calling the name of the procedure from a PL/SQL block

IN: It is a default parameter. It passes the value to the subprogram.

OUT: It must be specified. It returns a value to the caller.

IN OUT: It must be specified. It passes an initial value to the subprogram and returns an updated value to the caller.

We cannot call a procedure in a SQL Query.

The SELECT statements can never have procedure calls.

A procedure cannot be called using any function

Function

We can call a function in a SQL Query.

The SELECT statements can have function calls.

A function can be called using a procedure

Functions can be used in typical SQL statements like SELECT, INSERT, UPDATE, DELETE, MERGE, while procedures can't.

CREATE [OR REPLACE] FUNCTION function\_name

[ (parameter\_name [IN | OUT | IN OUT] type [, ...]) ]

RETURN return\_datatype

IS | AS

[declaration\_section]

BEGIN

executable\_section

[EXCEPTION

exception\_section]

END [function\_name];

Cursor

A cursor is a pointer to a private SQL area that stores information about the processing of a SELECT or DML statements like INSERT, UPDATE, DELETE or MERGE. Cursor is a mechanism which facilitates you to assign a name to a SELECT statement and manipulate the information within that SQL statement.

CURSOR cursor\_name

IS

SELECT\_statement;

After the declaration of the cursor, you have to use the open statement to open the cursor.

OPEN cursor\_name;

fetch rows from cursor

FETCH cursor\_name INTO variable\_list;

1) cursor\_name:It specifies the name of the cursor that you wish to fetch rows.

2) variable\_list: It specifies the list of variables that you wish to store the cursor result set in.

close cursor

CLOSE cursor\_name;

Oracle Trigger

In Oracle, you can define procedures that are implicitly executed when an INSERT, UPDATE or DELETE statement is issued against the associated table. These procedures are called database triggers.

Records

Exceptions

Triggers

BEFORE INSERT TRIGGER

BEFORE UPDATE TRIGGER

BEFORE DELETE TRIGGER

AFTER INSERT TRIGGER

AFTER UPDATE TRIGGER

AFTER DELETE TRIGGER

CREATE [ OR REPLACE ] TRIGGER trigger\_name

( BEFORE | AFTER) (INSERT | UPDATE | DELETE)

ON table\_name

[ FOR EACH ROW ]

DECLARE

-- variable declarations

BEGIN

-- trigger code

EXCEPTION

WHEN ...

-- exception handling

END;

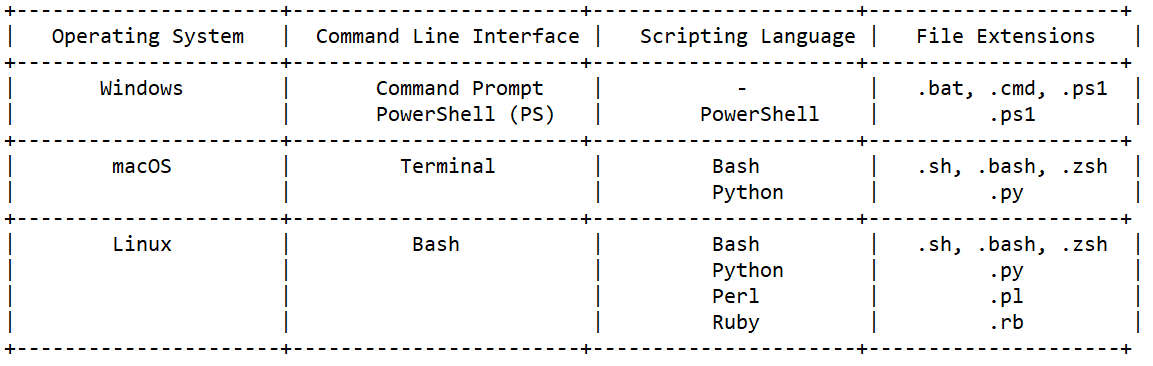
Object Oriented

**VPD**

**Dbconnection pooling**

Powershell scripting

CMD is a basic command-line interface that is suitable for performing simple tasks, PowerShell offers more advanced features and is better suited for more complex tasks and automation.



cmdlet

cmdlet – comman-let is a lightweight command that is used in the windows powershell environment. Cmdlets are way different from commands

passing parameters to cmdlet

variable

$location = Get-Location

Using variable

$location

Getting information of variable

$location | Get-Member

Special variable

Operators

Comparison operator

eq, ne, gt, ge, it, le

Logical operator

AND, OR, NOT

Looping

CLOUD COMPUTING

AWS

Networking

Devops

CI/CD

* + Bamboo, Jenkins

Docker

Kubernetes

Miscellaneous

Winscp

WinSCP is a free and open-source SFTP, FTP, and SCP client for Windows. It enables users to securely transfer files between a local and a remote computer using various protocols such as SFTP, FTP, SCP, and WebDAV. Some of the key features of WinSCP include:

Graphical user interface (GUI) for easy navigation and file management

Support for both basic and advanced file transfer operations such as upload, download, and synchronization

Built-in text editor for editing files directly on the remote server

Support for various authentication methods including password, public key, and keyboard-interactive

WinSCP also has a scripting interface that allows you to automate file transfer tasks, you can use it to create scripts that can be scheduled to run at specific times or triggered by certain events.

**JSCH**

GraphQl

Redis: Redis is an in-memory data store that is often used as a cache or message broker in web applications.

RabbitMQ: RabbitMQ is a message broker that is often used in web applications to handle asynchronous tasks and messaging

Kubernetes: Kubernetes is a container orchestration platform that allows you to manage and deploy containerized applications at scale.

Apache Kafka: Apache Kafka is a distributed streaming platform that is often used in web applications to handle real-time data streams

Redux: Redux is a predictable state container for JavaScript apps. It provides a centralized store for managing application state and allows you to easily manage complex state changes in your application.

How to deploy website to different server

In a typical authentication flow, the token is generated during the login process, not during the registration process. Let's break down the steps:

1. Registration:
   * During the registration process, a user provides their information and creates an account in the system.
   * The user's information is typically stored in a user database or repository.
   * At this stage, no token is generated yet.
2. Login:
   * After registering, the user can log in using their credentials (e.g., username and password).
   * The server validates the provided credentials.
   * If the credentials are valid, the server generates a token, typically a JWT (JSON Web Token), for the user.
   * The token contains encoded information about the user's identity and possibly other claims (e.g., expiration time, permissions).
   * The server returns the token to the client as part of the login response.

Once the token is generated and provided to the client during the login process, the client (e.g., browser) can store the token, usually in local storage or a cookie, and include it in the **Authorization** header of subsequent requests to authenticate the user.

It's worth noting that the exact implementation may vary depending on the specific authentication system or framework used in your application. However, the general practice is to generate the token during the login process after successful validation of the user's credentials.

During the navigation process or subsequent requests after the initial login, the token generated during the login process is used for authentication. Here's how it typically works:

1. Initial login and token generation:
   * The user logs in and receives a token (JWT) from the server.
   * The token is typically stored on the client-side, such as in local storage or a cookie.
2. Subsequent requests:
   * As the user navigates through the application or makes subsequent requests, the stored token is included in the **Authorization** header of each request.
   * The client adds the token to the request header in the format "Bearer <token>" (e.g., "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...").
3. Server-side authentication:
   * The server receives the request and extracts the token from the **Authorization** header.
   * The server then verifies the token's validity and integrity, checking the token's signature and expiration time, and possibly performing additional custom checks.
   * If the token is valid, the server extracts the user's identity and other relevant information from the token.
4. Authentication and authorization:
   * Using the extracted user information, the server authenticates the user based on the token.
   * The server may also perform authorization checks to determine if the user has the necessary permissions to access the requested resource.
5. Processing the request:
   * If the authentication and authorization are successful, the server processes the request and returns the requested data or performs the requested operation.

This process repeats for each subsequent request made by the user. The stored token is included in the **Authorization** header to authenticate and authorize the user on the server side.

It's important to note that the token's expiration time should be set appropriately to ensure the user remains authenticated for a reasonable duration. If the token expires, the user will need to log in again to obtain a new token for further navigation and requests.

In summary, during the registration process, user details are securely stored in the database. During login, the server validates the credentials, generates a token, and sends it to the client. During navigation, the client includes the token in subsequent requests, and the server validates and extracts the user

**Builder Pattern**

public class Product {

private String attribute1;

private String attribute2;

// ...

private Product(Builder builder) {

this.attribute1 = builder.attribute1;

this.attribute2 = builder.attribute2;

// ...

}

// Getters for attributes

public static class Builder {

private String attribute1;

private String attribute2;

// ...

public Builder() {

// Initialize default attribute values if needed

}

public Builder setAttribute1(String value) {

this.attribute1 = value;

return this;

}

public Builder setAttribute2(String value) {

this.attribute2 = value;

return this;

}

// Additional methods for setting other attributes or configurations

public Product build() {

return new Product(this);

}

}

}

Product.Builder builder = new Product.Builder();

Product product = builder.setAttribute1("value1")

.setAttribute2("value2")

.build();