**Play-list to watch**:

**Docker**: https://www.youtube.com/watch?v=CYyUCJad954&list=PLRAV69dS1uWTJLvDP4Veld5F05rJAmOcp

**Spring**:

https://www.youtube.com/watch?v=GB8k2-Egfv0&list=PLC97BDEFDCDD169D7 (Java Brains: 289 min : 24 Videos : Avg :13)

&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

Sometimes type casts are unavoidable and in Generic code they happen often without the programmer ever noticing. Therefore, significant effort was made to make type cast quite fast.

Telescoping constructor pattern: One constructor with only required fields and many other constructors with different combinations of optional fields. Difficult to maintain and bug prone.

JavaBeans Pattern: One constructor with only required fields and setters for all other optional patterns. Inconsistent state and supports mutability.

Java Bean Pattern vs Telescoping pattern ?

&&&&&&&&&&&&&&&&&&&&&&&&&&&& Streams &&&&&&&&&&&&&&&&&&&&&&&&&&

Reduction of Backtracking: Parsing from the end can help reduce backtracking, which occurs when the parser encounters an invalid or ambiguous construct and needs to backtrack to explore alternative parsing paths. Parsing from the end may allow for more straightforward decision-making because the parser has more context available to resolve ambiguities or make choices without extensive backtracking.

&&&&&&&&&&&&&&&&&&&&&&&&&&&& 29 May 23 &&&&&&&&&&&&&&&&&&&&&&&&&&

new String(inputStream.readAllBytes())

&&&&&&&&&&&&&&&&&&&&&&&&&&&& 01 Jun 23 &&&&&&&&&&&&&&&&&&&&&&&&&&

Future Interface:

Future, which represents a pending asynchronous event

get(), isDone(), cancel(), isCancelled()

Drawbacks of Future :

1. Cannot be manually completed: There is no way to complete the future. we can only attempt to cancel the task.

2. Blocking operations: Get(), It does not notify when result is available. So if we do get() before task completion, it gets blocked

3. No exception Handling, is it ?

4. Cannot chain or combine the future objects/result

CompletionStage Interface:

It represents a stage of a certain computation which can be done either synchronously or asynchronously. Several CompletionStages can be chained together so that one stage’s completion triggers the execution of another stage,

thenCombine(),whenComplete()

CompletableFuture:

It has the ability to explicitly complete the Future, hence the name CompletableFuture.

CompletableFuture provides a broad set of methods for creating multiple Futures, chaining, and combining. It also has comprehensive exception handling support.

CompletableFuture doesn’t work with callable’s. but it does with runnable’s and supplier functions.

We have to be careful that supplier functions doesn’t throw checked exceptions.

we can utilize them to build a pipeline of asynchronous operation or data pipelines. So these pipelines are blocking... Isnt it?

You can also use methods like get() or join() to block and wait for the completion of a CompletableFuture. However, using these blocking methods defeats the purpose of writing asynchronous code with CompletableFuture and should be avoided whenever possible.

Instead, it is recommended to use the non-blocking methods provided by CompletableFuture such as thenApply, thenCompose, or thenAccept, along with appropriate error handling and exception propagation mechanisms, to write efficient and responsive asynchronous code.

complete()

---------------------------

supplyAsync()

thenAppy()

thenApplySync()

thenAccept()

thenAcceptAsync()

------------------------------

runAsync()

thenRun()

thenRunAsync()

----------------------------------------

he difference has to do with the Executor that is responsible for running the code. Each operator on CompletableFuture generally has 3 versions.

thenApply(fn) - runs fn on a thread defined by the CompleteableFuture on which it is called, so you generally cannot know where this will be executed. It might immediately execute if the result is already available.

thenApplyAsync(fn) - runs fn on a environment-defined executor regardless of circumstances. For CompletableFuture this will generally be ForkJoinPool.commonPool().

thenApplyAsync(fn,exec) - runs fn on exec.

In the end the result is the same, but the scheduling behavior depends on the choice of method.

thenApplyAsync is guarenteed to not block the current thread, whilst thenApply depends on the object it is called on for its behavior

-------------------------

&&&&&&&&&&&&&&&&&&&&&&&&&&&& 21 Jun 23 &&&&&&&&&&&&&&&&&&&&&&&&&&

What does Spring framework do:

1. Application Context and Dependence injection

2. Data Access(hides a little bit of JDBC)

3. Spring MVC

&&&&&&&&&&&&&&&&&&&&&&&&&&&& 09 Aug 23 &&&&&&&&&&&&&&&&&&&&&&&&&&

<?xml version="1.0" encoding="UTF-8"?>

<project

xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">