===============MISLLANEOUS===========================================

1. **Resizing** of **array** should be done **exponenetially** to get insertion time complexity as O(1)

3. In computer programming, boilerplate code or just boilerplate refers to sections of code that have to be included in many places with little or no alteration. It is often used when referring to languages that are considered verbose, i.e. the programmer must write a lot of code to do minimal jobs.

5. B-trees are used to manage indexes in DBMS.

6. AOP or Aspect Oriented programming is setting tasks as aspects and instead of calling them specifically in you function configuring

them to run before/after a task Execution across classes in application.

9. The main class for a Simple springboot application

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

Note: In Java, two dates can be compared using the compareTo() method of Comparable interface.

10. The @ControllerAdvice annotation allows us to consolidate our multiple, scattered @ExceptionHandlers from before into

a single, global error handling component.

@RestControllerAdvice

public class GlobalExceptionMapper extends ResponseEntityExceptionHandler {

@ExceptionHandler(value = UserNotFoundException.class)

public ResponseEntity<Object> userNotFound(UserNotFoundException exception) {

String errorMessage = "Error: " + exception.getLocalizedMessage()

+ " occured for user "

+ exception.getUserName()

+ " in call at: "

+ LocalDate.now();

ExceptionMapper exceptionMapper = new ExceptionMapper(errorMessage);

return new ResponseEntity<>(exceptionMapper, HttpStatus.NOT\_FOUND);

}

11. LocalDate myObj = LocalDate.now(); // Create a date object

12. Overriding equals and hasCode function in java class

class java\_rough {

int age;

String name;

@Override

public boolean equals(Object obj) {

if(this == obj) {

return true;

}

if(this == null || obj == null) {

return false;

}

java\_rough newObj = (java\_rough) obj;

return this.age == newObj.age && this.name.equals(newObj.name);

}

@Override

public int hashCode() {

return Objects.hash(this.property); // Objects.hash implementation iss part of java.util package

}

}

13. Types of Memories in C:

- Code: Code/instructions are stored here

- Static: Stores Global/Static variables

- Stack: Stores function context(address space, stack space, virtual address space, register set image (e.g. Program Counter

(PC), Stack Pointer (SP), Instruction Register (IR), Program Status Word (PSW) and other general processor registers)

- Heap: Dynamic memory allocation is done from this memory segment(malloc)

14. Classes used for JWT authentiaction with customer Id :

SecurityContextHolder -> Authentication -> Principal -> Extract details

Input/Output streams, Java Multithreading, Serialization in JSON, RMI, HttpUrlConnection, socket

====JAVA-INTERVIEW====

1. Static vs Non-Static: Bonded during compile time vs runtime

=====Annotations DropWizard vs SpringBoot====

1. public @interface Test : Is used to defien an interface where @interface denotes a interface for annotation defination.

2. Controller class : DropWizard :: SpringBoot

- Class level : @Controller :: @Controller

- Path mapping : @Path :: @RequestMapping

- Dependancy Injection : @Autowired/@Inject :: @Autowired/@Inject

- HTTP Method mapping : @Get :: @GetMapping

- Getting path parameters from request : @PathParam :: @PathVariable

- Getting query parameters from request : @QueryParam :: @RequestParam

3. Dropwizard does not use Spring

===== JAX-RS ====

1. REST services borrow the below mentioned mechanisms from HTTP :-

- Resource based URL's

- HTTP methods are used {GET, PUT, POST, DELETE}

- Responce responds back with appropriate HTTP status codes.

- Responce should have appropriate headers like content-type.

2. The REST Resource URI's are of 2 types

- Instance URI - Get details for single Resource. EX - /messages/{messageId}

- Collection URI - Get a collection of details. EX - /messages

3. Query parameters can be used for Pagination and Filtering. EX - /messages/{messageId}?results=10&year=2014.

@GET

public String getOrders(@QueryParam("stateCode") String stateCode) {

return stateCode != null ? getOrdersByState(stateCode)

: getAllOrders();

}

4. When creating a new resource instance we use the POST method with collections URI. EX - /messages

5. Idempotent methods - GET, PUT, DELETE {Can be repeated}

Non-Idempotent methods - POST

6. The different error codes classes are mentioned below :-

- 1XX - Informational(100: Continue, 101: Switching Protocols)

- 2XX - Success(200: OK, 201: created, 202: Accepted)

- 3XX - Redirection(301 Moved Permanently, 302 Found, 305 Use Proxy)

- 4XX - Client error(400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found)

- 5XX - Server error(500 Internal Server Error, 501 Not Implemented, 502 Bad Gateway, 503 Service Unavailable)

7. HATEOAS is a way to provide client link to various resources as a responce to an api call so that no URL creation has to be done at client side.

public Message getMessage(@PathParam("messageId") long messageId, @Context UriInfo uriInfo) {

Message message = MessageService.getMessage(messageId);

String uri = uriInfo.getBaseUriBuilder()

.path(MessageResource.class)

.path(Long.toString(messageId))

.build()

.toString();

message.addLink(uri, "self");

8. In SOAP services only one API end point is exposed and the request body contains details for what operation is to be performed.

9. According to Richardson Maturity Model below mentioned are the levels of REST service achived :-

- Lvl 0 - Using single API endpoint

- Lvl 1 - Using resource based URI'service

- Lvl 2 - Using HTTP methods {POST, PUT etc}

- Lvl 3 - HATEOAS

10. JAX-RS is a set of interfaces and annotations that has api's for REST services. Library classes like jersey or RESTeasy

implement these interfaces and make sense of the annotations wherever the JAX-RS api's are used. Every JAX-RS Library comes

with a copy of JAX-RS. Methods you get from Jersey for basic rest API are:

@GET

@Path("/{messageId}")

@Consumes(MediaType.APPLICATION\_JSON)

@Produces(MediaType.APPLICATION\_JSON)

@QueryParam("year") int year,

@BeanParam MessageFilterBean messageFilterBean

public Message getMessage(@PathParam("messageId") long messageId) {

return MessageService.getMessage(messageId);

}

11. jaxb is used by java to convert response to XML file.

12. Always have a no-arg constructor when returning XML/JSON response as the associated libraries uses it internally.

13. @QueryParam URL design = http://localhost:8080/messanger/webapi/messages?start=0&size=2

14. Some important params that we can extract from incoming request :-

- @MatrixParam -> Similar as @QueryParam but uses ; to seprate values;

- @HeaderParam -> Get token value sent in header.

- @CookieParam -> Get token value sent in cookie.

- @Context -> This cannot annotate to any datatype but only few selected ones like 'class:UriInfo, HttpHeaders' and

is used to get details about the request without specifying details if you don't know them before hand,

like AbsolutePath, ContextPath, QueryParams, Cookies etc

15. Responce object can be returned instead of actual object as Responce class has methods to return any class entity along with

editing HTTP status codes (.status(Status.CREATED)), return an object/value (.entity), sending URI along with status code 201

(.created(new URI("/messanger/webapi/messages/ + message.getId())));

EX -

@POST

public Response addMessage(Message message, @Context UriInfo uriInfo) {

Message newMessage = messageService.addMessage(message);

String newId = String.valueOf(newMessage.getId());

URI uri = uriInfo.getAbsolutePathBuilder().path(newId).build();

return Response.created(uri)

.entity(newMessage)

.build();

}

EX from SpringBoot:

return ResponseEntity.status(HttpStatus.OK).body(some\_object);

16. A basic exception handlind example for REST calls :-

//method that may have exception scenario

public Message getMessage(long id) {

Message message = messages.get(id);

if (message == null) {

throw new DataNotFoundException("Message with id " + id + " not found");

}

return message;

}

public class DataNotFoundException extends RuntimeException { //Custom exception handling class

private static final long serialVersionUID = -6328286661536343936L;

public DataNotFoundException(String message) {

super(message);

}

//Error object Class

@XmlRootElement

public class ErrorMessage {

private String errorMessage;

private int errorCode;

private String documentation;

public ErrorMessage() {

}

public ErrorMessage(String errorMessage, int errorCode, String documentation) {

super();

this.errorMessage = errorMessage;

this.errorCode = errorCode;

this.documentation = documentation;

}

// Class to implement and Map exception to response when thrown

@Provider //annotation to let jax.rs know that this class exists

public class DataNotFoundExceptionMapper implements ExceptionMapper<DataNotFoundException> {

@Override

public Response toResponse(DataNotFoundException ex) {

ErrorMessage errorMessage = new ErrorMessage(ex.getMessage(), 404, "http://javabrains.koushik.org");

return Response.status(Status.NOT\_FOUND)

.entity(errorMessage)

.build();

}

//Providing generic exception handler class

@Provider

public class GenericExceptionMapper implements ExceptionMapper<Throwable> {

@Override

public Response toResponse(Throwable ex) {

ErrorMessage errorMessage = new ErrorMessage(ex.getMessage(), 500, "http://javabrains.koushik.org");

return Response.status(Status.INTERNAL\_SERVER\_ERROR)

.entity(errorMessage)

.build();

}

17. The Jersey Library provides you with 'WebApplicationException' where you can set from a set of constructor option

on how to handle an error scenario that may happen in a web-application.

EX -

public Comment getComment(long messageId, long commentId) {

ErrorMessage errorMessage = new ErrorMessage("Not found", 404, "http://javabrains.koushik.org");

Response response = Response.status(Status.NOT\_FOUND)

.entity(errorMessage)

.build();

Message message = messages.get(messageId);

if (message == null) {

throw new WebApplicationException(response); //Creating your responce and sending

}

Map<Long, Comment> comments = messages.get(messageId).getComments();

Comment comment = comments.get(commentId);

if (comment == null) {

throw new NotFoundException(response); //Using a standard error code part of WebApplicationException

}

return comment;

}

18. Content negociation is client and server using header values to communicate type (XML, JSON etc) that will be sent and

received amongst them. One method to enable multiple types in server side in jersey is :

@Consumes(value = { MediaType.APPLICATION\_JSON, MediaType.TEXT\_XML })

You can set @Consumes or @Produces at method levels as well, based on this request of same HTTP method will select method on type

sent from client.

=====MAVEN Fundamentals=====

1. POM has project name and its group and artifact ids

2. POM needs only entry level dependancy

3. COMMANDS :

- mvn compile (uses compile plugin, no need to mention in pom if no custom config is required)

- mvn package -> creates JAR/WAR

- mvn clean -> removes target folder from project

- java -cp \*.jar class\_name -> To run main method in class\_name from jar

- mvn jetty:run(Uses jetty plugin, you can use scan interval for getting new changes deployed)

4. Maven creates a local repo and checks it first for artifacts then goes to internet maven repo

5. PHASES OF MAVEN BUILDCYCLE:

Validate (check if all information necessary for the build is available) -> Compile -> Test -> package ->

install(pushes artifact to local repo) -> deploy (pushes artifact to remote repo)

6. Common maven dependency scopes are :

1. Compile : jar will be available all the time and is packaged

2. Provided : jar will be available for build and is not packaged

3. Runtime : jar will be available only during running application and is packaged

4. Test : jar will be available for test/build and is not packaged

7. For adding Repository details use <repositories><repository> tag in pom

====SPRING Fundamentals(IOC)====

Summary :-

Two distinct types of Spring IoC Containers (IoC is also known as dependency injection (DI). It is a process whereby objects define

their dependencies, that is, the other objects they work with, only through constructor arguments, arguments to a factory method,

or properties that are set on the object instance after it is constructed or returned from a factory method. The container then

injects those dependencies when it creates the bean. This process is fundamentally the inverse, hence the name Inversion of Control

(IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes,

or a mechanism such as the Service Locator pattern.) which are a BeanFactory Container and ApplicationContext Container in Spring

Framework:-

- Spring BeanFactory Container is the simplest container which provides basic support for DI. It is defined by org.springframework

.beans.factory.BeanFactory interface. A BeanFactory is essentially nothing more than the interface for an advanced factory capable

of maintaining a registry of different beans and their dependencies. The BeanFactory enables you to read bean definitions and

access them using the bean factory.

- ApplicationContext container adds more enterprise-specific functionality such as the ability to resolve textual messages from a

properties file and the ability to publish application events to interested event listeners. This container is defined by the org

.springframework.context.ApplicationContext interface.

1. Namespace declaration in applicationContext.xml which is located in src/main/resources file which is used for Spring configuration

are basically libraries that help in configuration and Validation operations.

2. Beans are POJOs that Spring configures. Simply put mentioning in file that we need a bean allows you to use it instead of

hard-coding it with new keybord in a class. Also use interfaces while configuring beans as principal of 'Sepration of Concerns' helps

you to update the implementation class whenever you want without need of changing applicationContext.xml.

3. To enable annotation based bean configuration for your project add the below in applicationContext.xml :-

<context:annotation-config/> //telling that we use annotations

<context:component-scan base-package="com.project"/> //telling where is the root folder for the files that are bound by annotations

4. Annotation are of 3 types at class level which are symantically same but have different significance by convention as mentioned :-

- @Component -> Any POJO, A Java class decorated with @Component is found during classpath scanning and registered in the

context as a Spring bean.

- @Service -> Class that implements business logic

- @Repository -> Classes that interact with DB.

5. Using @Autowire annotation you can add any class as dependency in your class in 3 ways as shown below :-

- Member level :-

@Autowire

private RequiredClass1 requiredClass1;

- Setter Injection :-

@Autowire

public void setRequiredClass(RequiredClass1 requiredClass1) {

this.requiredClass1 = requiredClass1;

}

- Constructor Injection :-

@Autowire

public CurrentClass(RequiredClass1 requiredClass1) {

this.requiredClass1 = requiredClass1;

}

6. Using java you can also create you spring configuration file as shown below. You can name your file anything just use the same

name in your main method when running application and giving name of config file.

@Configuration

@ComponentScan("com.packageRoot")

public class AppConfig {

@Bean(name = "serviceClass")

public MyServiceInterface getServiceClassBean() {

return new MyServiceImpl();

}

}

7. Any bean has 5 scopes as described below :-

- Singleton - Single instance per Spring container and is the default bean scope. Bean is created before code asks for it.

Below is an code snippet.

@Service

@Scope("singleton")

public class CurrentClass() {

//functions

}

- Prototype - This scopes a single bean definition to have any number of object instances. Bean is created after code asks for it.

Below is an code snippet.

@Service

@Scope("prototype")

public class CurrentClass() {

//functions

}

\*\* As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.\*\*

For web projects

- Request - This scopes a bean definition to an HTTP reques t; that is each and every HTTP request will have its own

instance of a bean created off the back of a single bean definition.

- Session - This scopes a bean definition to an HTTP session.

- Global - This scopes a bean definition to a global HTTP session

=====SPRING MVC FRAMEWORK(SOC)=====

1. Common Vocabulary :-

- Dispatcher Servlet - The entry / configuration point of application. Front controller class maintained by Spring MVC.

Uses @Controller annotation to identify all controller classes.

- Handles the request and determines which view to route to.

- Request Mapping - The URL and the HTTP method type the function is bind to.

- View Resolver - Used to locate JSP pages

- Servlet - config - configuration file per Dispatcher Servlet

- POJO - Plain Old Java Object, which has no arg constructor and has private fields with associated getter/setter methods.

- Bean - It is a POJO that is configured by Spring to be used within the application.

- Common Annotations:

@Controller

@RequestMapping

@PathVariable

@RequestParam

@ModelAttribute

@RequestBody and @ResponseBody:

EX: public @ResponseBody List<User> listAllUsers(@RequestBody UserList users) {

return userService.findAllUsers();

}

@RequestHeader and @ResponseHeader

2. Basic Components of Spring MVC project :-

- web.xml

- servlet-config.xml

- Controller class - Class that has the annotations

- View - JSP pages to display rendered datatype

3. Declaring your Dispatcher Servlet in web.xml:-

<servlet>

<servlet-name>fitTrackerServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/config/servlet-config.xml</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>fitTrackerServlet</servlet-name>

<url-pattern>\*.html</url-pattern>

</servlet-mapping>

4. Adding servlet-config.xml under WEB-INF to tell that we will use annotations for configuration and where classes with annotations

will be.

<mvc:annotation-driven />

<context:component-scan base-package="com.pluralsight.controller" />

5. Telling about location of your JSP pages using 'p' which stands for property namespace. Namespaces are internal utility methods.

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" p:prefix="/WEB-INF/jsp/" p:suffix=".jsp" />

6. For Validation of data sent from client you can use JSR-303 standards which involve use of dependancy 'hibernate-validator'.

Setting custom annotations for validation also uses org.hibernate.validator dependancy

public String updateGoal(@Valid @ModelAttribute("goal")Goal goal, BindingResult result) { //@Valid annotation starts hibernate-validator rules check

System.out.println("result has errors: " + result.hasErrors()); //result object tells if the constraints are violated

System.out.println("Minutes updated: " + goal.getMinutes());

return "redirect:addMinutes.html";

}

====JUNIT and MOKITO====

1. A unit test has 4 phases :-

- Setup

- Execution of method under test

- Verification by asserting result of the test

- Teardown of resources created

2. Junit key points :-

- @Test annotation marks a function as test classes

- Assert class has many static functions to verify output

- @Before and @After runs before and after every @Test respectively, whereas @BeforeClass and

@AfterClass runs once per class.

3. Basic Mokito functions :-

- Mockito.mock(sample.class) to create mock of a class done during setup phase usually.

- Mokito.when(className.functionName()).thenReturn(output) in setup to provide mocked output.

- Mokito.verify(mockOrderDao).findById(idValue)

4. You can also use @Mock for your dependancy classes, but remember to add below in @Before part :-

- MokitoAnnotations.initMocks(this);

5. Common when/then constraints on mocked methods are :-

- thenReturn -> Returns an object/value

- thenThrow -> Returns an exception type

- doThrow -> In case of void methods use first get the doThrow output in a Stubber object and then

on that object apply your when condition.

- thenCallRealMethod -> calls the actual method for the mocked class.

6. Mokito allows you to test if your method was called :-

- Mokito.verify(className, VerificationSettings.times(number\_of\_times\_invoked)).functionName();

- You can also use times, atleast(n), atleastOnce(), atmost(n), never

7. Argument matching while mocking functions uses common Matcher classes below functions :-

EX: when(mockFoo.bool(anyString(), anyInt(), any(Object.class))).thenReturn(true);

- Matchers.eq() -> Used for equals as you have to use implicit or explicit matching only for all fields.

- (data\_type) Matchers.any()

- Matchers.anyInt() and other generic datatypes

- Matchers.any(String.class)

- Matchers.eq/contains/startsWith/EndsWith/match(regex) can be used for strings

8. To verify that services are executed in order we can use the below mentioned functionalities :-

- InOrder inOrderVerifier = Mockito.inOrder(mockService1, mockService2);

inOrderVerifier.verify(mockService1).function1();

inOrderVerifier.verify(mockService2).function2();

- When return for same function needs to be different

when(mockFoo.someMethod())

.thenReturn(0)

.thenReturn(1)

.thenReturn(-1); //any subsequent call will return -1

9. Mockito provides a utility to capture arguments that are being passed to function calls to improve the scope

of what we can to test. The below code snippet demos that approach :-

ArgumentCaptor<className> classArgumentCaptor = ArgumentCaptor.forClass(className.class);

Mokito.verify(mockOrderDao).methodName(classArgumentCaptor.capture());

Assert.assertEquals(expectedValue, classArgumentCaptor.someFunction());

10. @Rule : Sort of using @After and @Before to set up mock services.  
EX :

@Rule

public TemporaryFolder tmpFolder = new TemporaryFolder();

@Test

public void givenTempFolderRule\_whenNewFile\_thenFileIsCreated() throws IOException {

File testFile = tmpFolder.newFile("test-file.txt");

assertTrue("The file should have been created: ", testFile.isFile());

assertEquals("Temp folder and test file should match: ",

tmpFolder.getRoot(), testFile.getParentFile());

}

=====Lambda Expressions====

ABOUT:

1. Advantages of using Lambdas include:

Memory saved in intermidiate collections requirements

Conciseness

Reduction in code bloat

Readability

Code reuse

Parallel processing opportunities

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

import java.util.function.Predicate;

import java.util.stream.Collectors;

class java\_rough {

public static class Person {

int age;

String name;

public Person(int age, String name) {

this.age = age;

this.name = name;

}

}

public static void main(String[] args) {

List<Person> students = new ArrayList<>();

Person person\_1 = new Person(10, "dude-1");

Person person\_2 = new Person(16, "dude-5");

Person person\_3 = new Person(7, "dude-2");

Person person\_4 = new Person(15, "dude-6");

Person person\_5 = new Person(13, "dude-4");

students.add(person\_1);

students.add(person\_2);

students.add(person\_3);

students.add(person\_4);

students.add(person\_5);

Collections.sort(students, (p1, p2) -> (p1.age - p2.age));

List<String> names = students.stream().filter(getTeenAgers()).filter(getEvenNames()).map(person -> person.name).collect(Collectors.toList());

names.forEach(System.out::println);

}

public static Predicate<Person> getTeenAgers() {

return person -> person.age > 12;

}

public static Predicate<Person> getEvenNames() {

return person -> {

String[] names = person.name.split("-");

int number = (int) names[1].charAt(0);

return number % 2 == 0 ? true : false;

};

}

}

Passing function as method reference(EX: System.out::println) or (parameters -> Function body(Single line or multiple lines

function body contained inside {} and separated by ;))

Functional interface: A functional interface is an interface that contains only one abstract method. Comparator is a functional interface because there's only one unimplemented abstract method: compare(T o1, T o2).

EX: Implementing the compare method:

personList.sort(new Comparator<Employee>() {

@Override

public int compare(Employee e1, Employee e2) {

return e1.getName().compareTo(e2.getName());

});

EX : FUNCTIONAL INTERFACE

@FunctionalInterface

public interface

8. Using lambda expression the declaration can be implified to: personList.sort(Comparator.comparing((Person p) -> p.name));

Note: for String class compareTo returns int (0 if equal +ve if first arg is bigger else -ve)and equals returns boolean.

2. Lambdas can only be defined for functional interfaces that are interfaces with exactly one abstract method.

3. The declaration for filter looks like: Stream<T> filter(Predicate<? super T> predicate);

4. A predicate function can be defined as:

public static Predicate<Employee> isAdultMale()

{

return p -> p.getAge() > 21 && p.getGender().equalsIgnoreCase("M");

}

EX :

class java\_rough {

public static void main(String[] args) {

List<String> names = new ArrayList<>(Arrays.asList("viv", "sh", "vis"));

names = names.stream().filter(oddLength(3)).collect(Collectors.toList());

System.out.println(names.toString());

}

public static Predicate<String> oddLength(int nameLength) {

return thisName -> thisName.length() == nameLength;

}

}

5. Optional is a container object used to contain not-null objects. Optional object is used to represent null with absent value,

which helps in avoiding null pointer checks. Common methods are listed below :

-optional.empty() : Returns an empty instance. No value not even null is present for this Optional. Used to intialize an

optional as using null won't make sense in this case.

-optional.of() : Passes the value under () to the optional variable.

-optional.ofNullable() : Returns describing the specified value, if non-null otherwise returns an empty.

-optional.isPresent() : Checks if optional variable is null. Even empty will return true.

-optional.get() : get value inside the optional container.

6. Example of important methods:

import java.util.Arrays;

import java.util.List;

import java.util.Optional;

class java\_rough {

public static void main(String[] args) {

Optional<String> s1 = Optional.empty();

Optional<String> s2 = Optional.of("value");

Optional<String> s3 = Optional.ofNullable("value\_1");

Optional<String> s4 = Optional.ofNullable(null);

Optional<List<String>> s5 = Optional.of(Arrays.asList(new String[]{"hi", "there"}));

if(s1.isPresent()) {

System.out.println(s1.get());

} else {

System.out.println(s1);

}

if(s2.isPresent()) {

System.out.println(s2.get());

} else {

System.out.println(s2);

}

if(s3.isPresent()) {

System.out.println(s3.get());

} else {

System.out.println(s3);

}

if(s4.isPresent()) {

System.out.println(s4.get());

} else {

System.out.println(s4);

}

if(s5.isPresent()) {

System.out.println(s5.get());

} else {

System.out.println(s5);

}

}

}

O/P: Optional.empty

value

value\_1

Optional.empty

[hi, there]

1. Common methods for Iterable and Collection interfaces :-

- List<Person> person = ...;

person.forEach(System.out::println); //This method of calling functions is called method reference

- person.removeIf(person -> person.getAge < 18); //The condition passed as function is called predicate

2. For List interface :-

- person.replaceAll(String::toUpperCase);

- person.sort(

Comparator.comparing(Person::getName).reversed()

.thenComparing(Person::getAge)

);

3. For Map interface :-

- Map<City, List<Names>> = ...;

map.forEach((city, list) ->

System.out.println(city + ":" + list.size() + "people");

)

- map.getOrDefault(boston, emptyList()); //It will either give associated value or return value passed as second parameter

- map.putIfAbsent(boston, new ArrayList<Names>()); //It will add second parameter as value if key does not exist.

- map.replace(key, existingValue<optional>, newValue) //Replaces value for given key if matches provided existing value

- numbers.replaceAll((key, oldValue) -> oldValue \* oldValue);

- courseMap.entrySet().removeIf(e - > e.getKey().equals("Java"));

- public static void main(String[] args)

{

// Create a Map and add some values

Map<String, String> map = new HashMap<>();

map.put("Name", "Aman");

map.put("Address", "Kolkata");

// Print the map

System.out.println("Map: " + map);

// remap the values using compute() method

map.compute("Name", (key, val)

-> val.concat(" Singh"));

map.compute("Address", (key, val)

-> val.concat(" West-Bengal"));

// print new mapping

System.out.println("New Map: " + map);

}

- Similar functions are computeIfAbsent and computeIfPresent with condition being that the passed function and BiFunction

respectively will be called to compute new associated value with passed parameter key if it is absent or is it is present.

- map2.forEach(

(key, value) ->

map1.merge(key, value,

(existingValue, newValue) -> {

existingValue.addAll(newValue);

return existingValue;

})

);

4. When you create a stream, it is always a serial stream unless otherwise specified. To create a parallel stream, invoke the

operation Collection.parallelStream. Alternatively, invoke the operation BaseStream.parallel. For example, the following statement

calculates the average age of all male members in parallel:

double average = roster

.parallelStream()

.filter(p -> p.getGender() == Person.Sex.MALE)

.mapToInt(Person::getAge)

.average()

.getAsDouble();

For such use cases you should use predicate in filter function that is associative.

5. Optional is a wrapper class like Integer that might be empty. Optional may contain a list as well.

6. Stream is a new interface in Java8 to perform map/filter/reduce operations on collections without creating new intermidiate collections

reducing the memory and processing overhead.

- Stream does not hold any data, it just processes data from a source.

- The size of the source is not known at runtime.

- EX: personList.stream()

.map(p -> p.getAge()) //Converts objects list to integer list

.filter(age -> age > 20)

.forEach(System.out::println);

- Without a terminal operation like forEach or any call that does not retur stream the stream will never process the data.

- EX: personList.stream()

.skip(2) //Does not considers first 2 elements

.limit(3) //Takes only 3 elements for stream manipulation

.map(p -> p.getAge()) //Converts objects list to integer list

.filter(age -> age > 20)

.forEach(System.out::println);

EX: class java\_rough {

public static void main(String[] args) {

List<String> numbers = new ArrayList<>(Arrays.asList("sdss","dfgfgfg","fgf"));

List<Integer> lengths = numbers.stream().map(entry -> entry.length()).filter(length -> length >= 4)

.collect(Collectors.toList());

System.out.println(lengths.toString());

}

}

- Match Reduction EX :

boolean b = people.stream.allMatch(p -> p.getAge() > 20); //Return true only if all elements are true for the condition.

Similalry we have anyMatch() and noneMatch().

- Find Reduction EX :

Optional<Person> persons = persons.stream()

.findAny(p -> p.getName().length() > 10);

Similalry we have findFirst().

- Reduce Reduction EX :

The reduce reduction predicate should be associative in nature as this error will not be caught and only the answer will be

wrong. Below are its EX :

int sumOfAges = people.stream()

.reduce(0, (p1, p2) -> p1.getAge() + p2.getAge()); //here 0 is the identity element if no identity

//element is present then we can provide output

//in optional.

EX : Third type is using an Accumulator

List<Integer> ages = people.stream()

.reduce(

new ArrayList<Integer>(),

(list, p) -> { list.add(p.getAge()) ; return list;},

(list1, list2) -> { list1.addAll(list2); return list1;}

);

7. toMap method : Creates a new map to store input entries from operations, and if key values are not unique IllegalStateException()

is thrown.

Example : Map<String, String> map = str.collect(Collectors.toMap(p -> p[0], p -> p[1]));

====SPRING - BOOT====

1. @EnableAutoConfiguration - Spring Boot automatically configures your application based on the dependencies you have added to the project

2. The entry point of the spring boot application is the class contains @SpringBootApplication annotation and the main method.

3. Spring Boot automatically scans all the components included in the project by using @ComponentScan annotation. You can add :-

@ComponentScan(

basePackages = {name of base package}

excludeFilters = {Classes you want to exclude}

)

====FILE-HANDLING====

1. Streams are unidirectional flow of data to Read or write data from or to a file, both cannot be done simultaneously.

2. Streams have 2 categories :

1. Byte Streams : Interact as binary data

2. Text Stream : Interact as Unicode Characters.

3. For byte stream to read data we use InputStream interface and int read() method. The int value returned by read() method should be type casted to byte before being used. If -1 is returned then it means there is nothing more to read.

4. For Text stream to read data we use Reader interface and int read() method. The int value returned by read() method should be type casted to char before being used. If -1 is returned then it means there is nothing more to read.

5. Another method for byte stream to read data we use InputStream interface and int read(byte[]) method.

6. Another method for text stream to read data we use Reader interface and int read(char[]) method.

7. Similar methods are available for OutputStream and Writer class to write byte and text stream respectively. These methods are void write(int value) and for array void write(byte[] or char[] or String).

8. Common concrete implementation of InputStream and OutputStream are :

- ByteArrayInputStream :

- PipedInputStream :

- FileInputStream : Most commonly used.

- ByteArrayOutputStream :

- PipedOutputStream :

- FileOutputStream : Most commonly used.

1. java.io package contains all classes related to getting value in and from a variable.Java I/O is organized into 4 base classes:

- Reader(read from file) and Writer(write to a file)

- i/p (read as binary) and o/p (write as a binary)stream

- Utility classes: File and Path

2. File file = new File("dummyFile.txt"); //does not create file in disk just a file object

Common methods for File class are:-

file.isDirectory();

file.isFile();

file.exists();

file.canRead();

file.canWrite();

file.canExecute();

file.createNewFile();

file.mkdir();

file.delete();

file.getName(); //file name

file.getParent(); //path where file is

file.getPath(); //complete path with file name

file.getAbsolutePath();

file.getCanonicalPath(); //unique for each file

3. Path interface also has above mentioned methods with new addition like:

- path.normalize(): Return value: This method returns the resulting path or this path if it does not

contain redundant name elements; an empty path is returned if this path does not have a root

component and all name elements are redundant.

- path.toRealPath()

- Files.isSameFile(path1, path2): Checks if both paths point to same location.

- path.get(""): Converts string into path instance.

4. Java.io.Reader class in Java is an abstract class for reading character streams. The only methods

that a subclass must implement are read(char[], int, int) and close().

public static void main(String[] args) throws Exception{

Reader reader = new FileReader("./dummy.txt");

char[] buffer = new char[10];

if(reader.ready()) {

reader.read(buffer, 0, 10);

System.out.println(Arrays.toString(buffer));

}

reader.close();

}

}

5. The reset() method of Reader Class in Java is used to reset the stream. After reset, if the stream

has been marked, then this method attempts to reposition it at the mark, else it will try to position

it to the starting.

class GFG {

public static void main(String[] args)

{

try {

String str = "GeeksForGeeks";

Reader reader = new StringReader(str);

int ch;

for (int i = 0; i < 10; i++) {

ch = reader.read();

System.out.print((char)ch);

}

System.out.println();

reader.mark(5);

reader.reset();

for (int i = 0; i < 5; i++) {

ch = reader.read();

reader.skip(1); //Skip one character every time

System.out.print((char)ch);

}

}

6. Sample program for creating files dynamically according to file size:

import java.io.File;

import java.io.FileWriter;

import java.io.Writer;

class java\_rough {

public static void main(String[] args) throws Exception{

File file = new File("./dummy.txt");

if(!file.exists()) {

file.createNewFile();

System.out.println("Created " + file.getName());

}

int count = 0;

long size = 0l;

String fileName;

CharSequence fileContent = "Sample data to be written"; //CharSequence an interface while String is a concrete implementation

//of that interface.

Writer writer = new FileWriter(file);

while(count < 10) {

size = file.length();

if(size > 10l) {

count++;

fileName = "dummy" + count;

file = new File("./" + fileName + ".txt");

file.createNewFile();

writer = new FileWriter(file);

System.out.println("Created new file" + fileName + ".txt" + " as size is " + size);

size = 0;

} else {

writer.append(fileContent);

writer.flush();

}

}

writer.close();

}

}

7. The Properties class represents a persistent set of properties. The Properties can be saved to a stream or loaded from a stream.

- Properties is a subclass of Hashtable.

- It is used to maintain list of value in which the key is a string and the value is also a string.

- One useful capability of the Properties class is that you can specify a default property that will be returned if no value is

associated with a certain key.

- Multiple thread can share a single properties object without the need of external synchronisation.

8. EX: Setting/Retrieving properties

import java.util.Properties;

class java\_rough {

public static void main(String[] args) {

Properties properties = new Properties();

properties.setProperty("name", "vishal");

properties.setProperty("age", "24");

System.out.println("details are: " + properties.getProperty("name")

+ ", " + properties.getProperty("age")

+ ", " + properties.getProperty("job", "Engineer"));

}

}

9. EX: Storing properties in file

public static void main(String[] args) throws Exception {

Properties properties = new Properties();

properties.setProperty("name", "vishal");

properties.setProperty("age", "24");

//Paths creates files automatically

try(Writer writer = Files.newBufferedWriter(Paths.get("./newProperties.txt"))) {

properties.store(writer, "My details");

}

System.out.println("details are: " + properties.getProperty("name")

+ ", " + properties.getProperty("age")

+ ", " + properties.getProperty("job", "Engineer"));

}

File created : #My details

#Sun Jan 26 10:09:48 IST 2020

name=vishal

age=24

10. EX: Reading from a file

public static void main(String[] args) throws Exception {

Properties properties = new Properties();

try(Reader reader = Files.newBufferedReader(Paths.get("./newProperties.txt"))) {

properties.load(reader);

}

System.out.println("details are: " + properties.getProperty("name")

+ ", " + properties.getProperty("age")

+ ", " + properties.getProperty("job", "Engineer"));

}

=====MULTI-THREADDING=======

- States for a thread are: Born/ New → Runnable → Running → Dead

- States of a process are: new -> ready -> running -> terminated

| |

<- waiting <-

1. Runnable Interface: Enables a classes method to run as thread in calling method

EX: class java\_rough {

static class Example implements Runnable {

int instanceNumber;

public Example(int n) {

this.instanceNumber = n;

}

@Override

public void run() {

System.out.println("Testing " + this.instanceNumber);

}

}

public static void main(String[] args) {

Example first = new Example(1);

Example second = new Example(2);

Thread threadFirst = new Thread(first);

Thread threadSecond = new Thread(second);

threadFirst.start();

threadSecond.start();

}

}

2. Callable: The Callable interface is designed to define a task that returns a result and may throw an exception. This interface

also contains a single, no-argument method, called call (), to be overridden by the implementors of this interface. This method

is similar to the run () method of the Runnable interface, except that it returns a value and can throw a checked exception.

3. Future: The Future interface is a generic interface that represents the value returned from an asynchronous computation.

It contains methods to check if the computation has been completed or wait for it, retrieve the result.

4. A java.util.concurrent.ExecutorService interface is a subinterface of Executor interface, and adds features to manage

the lifecycle, both of the individual tasks and of the executor itself.

EX:

public class CallableMain {

public static void main(String[] args)

throws InterruptedException, ExecutionException {

ExecutorService service = Executors.newSingleThreadExecutor();

CallableFactorialTask task = new CallableFactorialTask(5);

Future<Integer> f = service.submit(task);

Integer val = f.get();

System.out.println(val);

service.shutdown();

}

}

class CallableFactorialTask implements Callable<Integer> {

private int num = 0;

public CallableFactorialTask(int num){

this.num = num;

}

@Override

public Integer call() throws Exception {

int prod = 1;

for (int i = 2; i <= num; i++)

prod \*= i;

return prod;

}

}

5. Synchronized Method: If you declare any method as synchronized (synchronized void printTable(int n)), it is known as

synchronized method. Synchronized method is used to lock an object for any shared resource. When a thread invokes a

synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

6. Synchronized statements: Synchronized block is used to lock an object for any shared resource. Scope of synchronized block is

smaller than the method.

EX:

void printTable(int n){

synchronized(this){//synchronized block

for(int i=1;i<=5;i++){

System.out.println(n\*i);

try{

Thread.sleep(400);

}catch(Exception e){System.out.println(e);}

}

}

}//end of the method

=====SQL AND NO-SQL=========

BASIC MONGO QUERIES : <https://docs.mongodb.com/manual/tutorial/query-documents/>

it also contains an mongo shell to practice

1. **db.inventory.find( {} )** : returns all documents in the collection

2. d**b.inventory.find( { status: "D" } )** : Quering using a coumn value

3. **db.inventory.find( { status: { $in: [ "A", "D" ] } } )** : Using query on multiple values

4. **db.inventory.find( { status: "A", qty: { $lt: 30 } } )** : Using AND conditions

5. **db.inventory.find( { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] } )** : Using OR conditions

6. **db.inventory.find( {**

**status: "A",**

**$or: [ { qty: { $lt: 30 } }, { item: /^p/ } ]**

**} )**

: For quering using both AND and OR conditions.

7. **db.inventory.find( { size: { h: 14, w: 21, uom: "cm" } } )** : To query an object use all its feilds and in the order in which they are saved in the DB.

8. **db.inventory.find( { "size.uom": "in" } )** : Using a particular object property to find records.

9. **db.inventory.find( { tags: ["red", "blank"] } ) :** Quering feild that contains array with exact itesm and in the queried order.

10. **db.inventory.find( { tags: { $all: ["red", "blank"] } } )** : Same as above but gets results where array has more itesm and in a different order.

11. **db.inventory.find( { tags: "red" } )** : Finding result where you are trying to find array that contains red as an element

12. **db.inventory.find( { dim\_cm: { $gte: 25 } } )** : Finding result where you are trying to find array that contains an element with value greater than 25.

13. **db.inventory.find( { dim\_cm: { $elemMatch: { $gt: 22, $lt: 30 } } } )** : same as above but multiple conditions

14. **db.inventory.find( { "tags": { $size: 3 } } )** : Same as above but using size of array to query.

15. **db.inventory.find( { "instock": { $elemMatch: { qty: 5, warehouse: "A" } } } )** : The following example queries for documents where the instock array has at least one embedded document that contains both the field qty equal to 5 and the field warehouse equal to A.

16. **db.inventory.find( { status: "A" }, { item: 1, status: 1 } )** : Selecting only required feild values. The above is equivalent to **SELECT \_id, item, status from inventory WHERE status = "A".** You can remove the \_id field from the results by setting it to 0 in the projection

17. **db.inventory.find( { status: "A" }, { item: 1, status: 1, instock: { $slice: -1 } } )** : For fields that contain arrays, MongoDB provides the following projection operators for manipulating arrays: $elemMatch, $slice, and $. For slice :[https://docs.mongodb.com/manual/reference/operator/projection/slice/#proj.\_S\_slice](https://docs.mongodb.com/manual/reference/operator/projection/slice/" \l "proj._S_slice)

Here Both the $ operator and the $elemMatch operator project the first matching element from an array based on a condition, like To return the first array element that matches the specified query condition on the array:

**db.collection.find( { <array>: <condition> ... },**

**{ "<array>.$": 1 } )**

**db.collection.find( { <array.field>: <condition> ...},**

**{ "<array>.$": 1 } )**

18. While quering for record with null, value we need specific queries :

EX : <https://docs.mongodb.com/manual/tutorial/query-for-null-fields/>

**db.inventory.insertMany([**

**{ \_id: 1, item: null },**

**{ \_id: 2 }**

**])**

**db.inventory.find( { item: null } )** : gets both records

**db.inventory.find( { item : { $exists: false } } )** : gets only second record

**db.inventory.find( { item : { $type: 10 } } )** : gets only first record

19. **db.students.updateOne( { \_id: 3 }, [ { $set: { "test3": 98, modified: "$$NOW"} } ] )** : Update calls for one/many adds properties or records if they don not already exist.

20. **db.students.find().pretty()** : For getting output as JSON

21. **db.students2.updateMany( {},**

**[**

**{ $replaceRoot: { newRoot:**

**{ $mergeObjects: [ { quiz1: 0, quiz2: 0, test1: 0, test2: 0 }, "$$ROOT" ] }**

**} },**

**{ $set: { modified: "$$NOW"} }**

**]**

**) :**

Specifically, the pipeline consists of:

a $replaceRoot stage with an $mergeObjects expression to set default values for the quiz1, quiz2, test1 and test2 fields. The aggregation variable ROOT refers to the current document being modified (to access the variable, prefix with $$ and enclose in quotes). The current document fields will override the default values.

a $set stage to update the modified field to the current datetime. For the current datetime, the operation uses the aggregation variable NOW for the (to access the variable, prefix with $$ and enclose in quotes).

22. **db.students3.updateMany(**

**{ },**

**[**

**{ $set: { average : { $trunc: [ { $avg: "$tests" }, 0 ] }, modified: "$$NOW" } },**

**{ $set: { grade: { $switch: {**

**branches: [**

**{ case: { $gte: [ "$average", 90 ] }, then: "A" },**

**{ case: { $gte: [ "$average", 80 ] }, then: "B" },**

**{ case: { $gte: [ "$average", 70 ] }, then: "C" },**

**{ case: { $gte: [ "$average", 60 ] }, then: "D" }**

**],**

**default: "F"**

**} } } }**

**]**

**) :**

The following db.collection.updateMany() operation uses an aggregation pipeline to update the documents with the calculated grade average and letter grade.

23. <https://docs.mongodb.com/manual/aggregation/>

Aggregation operations process data records and return computed results. Aggregation operations group values from multiple documents together, and can perform a variety of operations on the grouped data to return a single result. MongoDB provides three ways to perform aggregation: the aggregation pipeline, the map-reduce function, and single purpose aggregation methods.

1. Analogies MySQL and Mongo

Table : Collection

Primary Key : \_id field

Tuple : BSON Document

Column : Field

Join : Embedding and Linking

2. Differences MySQL and Mongo

Uses Schema : Schema Less

Relies on relations : Does no rely on relations

Stores data in multiple related tables : Stores data in a merged/consolidated manner as BSON Document

Horizontal scaling is difficult : Horizontal scaling is easy

Limitations for large number read and write queries per second : Good performance for large number read and write queries per second

3. Adding mongo db to SpringBoot App: https://springframework.guru/configuring-spring-boot-for-mongo/

4. https://www.javaworld.com/article/3373652/java-persistence-with-jpa-and-hibernate-part-1-entities-and-relationships.html

JPA: Java Persistence API Hibernate is one implementation of JPA. Hibernate is an object-relational mapping (ORM) tool. JPA offers a

standard way to annotate objects so that they can be mapped and stored in a relational database.

5. The Java Persistence API is a specification, not an implementation: it defines a common abstraction that you can use in your code to

interact with ORM products.

6. The @Entity annotation is a marker annotation, which is used to discover persistent entities. If you wanted to map this entity to

another table (and, optionally, a specific schema) you could use the @Table annotation to do that. able Schema is a specification

for providing a “schema” (similar to a database schema) for tabular data. This information includes the expected type of each

value in a column (“string”, “number”, “date”, etc.), constraints on the value and the expected format of the data

EX: @Entity

@Table(name="BOOKS", schema="PUBLISHING")

public class Book {

...

}

7. Fields are mapped to columns of the table. To override the column value use @Column annotation.

EX: @Entity

@Table(name="BOOKS")

public class Book {

private String name;

@Column(name="ISBN\_NUMBER")

private String isbn;

...

}

8. In JPA, we use the @Id annotation to designate a field to be the table's primary key. The primary key is required to be a

Java primitive type, a primitive wrapper, such as Integer or Long, a String, a Date, a BigInteger, or a BigDecimal.

EX: @Entity

@Table(name="BOOKS")

public class Book {

@Id

private Integer id;

private String name;

@Column(name="ISBN\_NUMBER")

private String isbn;

...

}

9. Annotations for mapping for Entity relations:

@OneToOne: @Entity

public class User {

@Id

private Integer id;

private String email;

private String name;

private String password;

@OneToOne(mappedBy="user")

private UserProfile profile;

...

}

Mapped to

@Entity

public class UserProfile {

@Id

private Integer id;

private int age;

private String gender;

private String favoriteColor;

@OneToOne

private User user;

...

}

@ManyToOne: @Entity

public class Book {

@Id

private Integer id;

private String name;

@ManyToOne

@JoinColumn(name="AUTHOR\_ID")

private Author author;

...

}

Author can have multiple books but book has one Author scenario

@Entity

public class Author {

@Id

@GeneratedValue

private Integer id;

private String name;

@OneToMany(mappedBy = "author")

private List<Book> books = new ArrayList<>();

...

}

@ManyToMany:@Entity

public class Book {

@Id

private Integer id;

private String name;

@ManyToMany

@JoinTable(name="BOOK\_AUTHORS",

joinColumns=@JoinColumn(name="BOOK\_ID"),

inverseJoinColumns=@JoinColumn(name="AUTHOR\_ID"))

private Set<Author> authors = new HashSet<>();

...

}

Author can have multiple books and book can have multiple Authors scenario

@Entity

public class Author {

@Id

@GeneratedValue

private Integer id;

private String name;

@ManyToMany(mappedBy = "author")

private Set<Book> books = new HashSet<>();

...

}

======GIT-COMMANDS==========

1. Running the following command will revert the last two commits: git revert HEAD~2..HEAD

Alternatively, one can always checkout the state of a particular commit from the past, and commit it anew.

2. To squash the last N commits of the current branch, run the following command (with {N} replaced with the number of commits

that you want to squash): git rebase -i HEAD~{N}

Upon running this command, an editor will open with a list of these N commit messages, one per line. Each of these lines will

begin with the word “pick”. Replacing “pick” with “squash” or “s” will tell Git to combine the commit with the commit before it.

To combine all N commits into one, set every commit in the list to be squash except the first one. Upon exiting the editor, and

if no conflict arises, git rebase will allow you to create a new commit message for the new combined commit.

3. git diff-tree -r {hash}: Given the commit hash, this will list all the files that were changed or added in that commit.

The -r flag makes the command list individual files, rather than collapsing them into root directory names only.

4. git cherry-pick {hash of that commit}: In spite of applying the same changes, it will be a new commit with a new hash

because the changes are applied to a different destination.

5. git fetch only downloads new data from a remote repository, but it doesn’t integrate any of the downloaded data into your

working files. All it does is provide a view of this data.

6. git pull downloads as well as merges the data from a remote repository into your local working files. It may also lead to

merge conflicts if your local changes are not yet committed. Use the git stash command to hide your local changes.

7. git --amend -m "Updates last commit message" : to update last commit message

8. Git commands :

1. Change message : git commit --amend -m "new message"

2. Remove a staged file not yet commited : git rm --cached src/main/java/com/project/ccs/routes/FileTransferRoute.java

3. Creating a new branch and switching to it : git checkout -b demo-branch

4. Setting upstream for new branch : git push --set-upstream origin demo-branch

5. Merge master to checked out branch : git merge master

6. Revert merging in case of conflicts : git merge --abort

7. Open file with conflicts : vim README.md

8. Add the edited file : git add README.md

9. Commit the change : git commit -m "Resolving Conflicts"

=====DOCKER COMMANDS======

COMMON: docker ps, docker ps -a, docker stop, docker images, docker rm <container id>, docker rmi <image-id>

1. docker run -it -d <image name>: This command is used to create a container from an image , here -d : detached, -i : keep

container interactive and -t : enable a psuedo terminal

2. Usage: docker exec -it <container id> bash: This command is used to access the running container

3. docker kill <container id>: The difference between ‘docker kill’ and ‘docker stop’ is that ‘docker stop’ gives the container

time to shutdown gracefully, in situations when it is taking too much time for getting the container to stop, one can opt to

kill it

4. docker commit <conatainer id> <username/imagename>: This command creates a new image of an edited container on the local system

5. docker push <username/image name>: This command is used to push an image to the docker hub repository

6. docker build <path to docker file>: This command is used to build an image from a specified docker file

=====KAFKA=======

1. Topics in Kafka are data streams, identified by Topic-name. Each topic has partitions and messages are devided in these

partitions for a topic. Topics stored data in ordered manner by using a increasing Offset value for each message.

2. Data is randomly assigned among partitions in for a topic, until a key is used. Data order is only guarenteed for a partition

using offsets. Data once written on a partition cannot be changes ie it is immutable.

3. Kafka broker is the server that stores partitions of kafka topics. A collection of brokers is called a kafka cluster and

once connecting to a bootstrap broker which can be any one of the cluster, connection will be made for the while cluster.

Each broker is identified by its ID (integer).

4. At a time only one broker is leader for a partition. The same broker will not contain its In Sync Replica(ISR) so that if it

goes down the broker with ISR becomes leader.

5. Producer can opt for ACKS in the following 3 ways:

- acks = 0: No acks waiting done by producer.

- acks = 1: Producer waits for leader acks.

- acks = all: Producer waits for leader and replicas acks.

6. Producer can use keys + Message to ensure partition to which the message is sent, else it is sent using Round Robin.

7. Consumers read as groups, where each consumer is reading from fixed partition. If your config has more consumers in group than

partitions from the topic then one in the group will be left idle on the flip side if more number of partition are there then

some consumer will consume from more than one partition.

8. Consumers have 3 delivery semantics for commiting offsets:

- At most once: commiting offset as soon as message is received. Message will be lost if processing fails.

- At least once: commiting offset after processing message. Message is reprocessed if processing fails, so system should be

Idempotent.

- Exactly once: Is achieved using kafka stream APIs

9. Github: https://github.com/simplesteph/kafka-beginners-course/tree/master/kafka-basics/src/main/java/kafka/tutorial1

10. Using below method you can add Kafka metadata for your logging:

producer.send(record, new Callback() { //A callback interface that the user can implement to allow code to

// execute when the request is complete.

public void onCompletion(RecordMetadata recordMetadata, Exception e) {

// executes every time a record is successfully sent or an exception is thrown

if (e == null) {

// the record was successfully sent

logger.info("Received new metadata. \n" +

"Topic:" + recordMetadata.topic() + "\n" +

"Partition: " + recordMetadata.partition() + "\n" +

"Offset: " + recordMetadata.offset() + "\n" +

"Timestamp: " + recordMetadata.timestamp());

} else {

logger.error("Error while producing", e);

}

}

11. You can add key to your kafka message by:

ProducerRecord<String, String> record = new ProducerRecord<String, String>(topic, key, value);

Then sending this record as shown above.

12. After setting Consumer properties you can set consumer to subscribe to topic and log the messages and metadata with it as

shown below:

KafkaConsumer<String, String> consumer = new KafkaConsumer<String, String>(properties);

// subscribe consumer to our topic(s)

consumer.subscribe(Arrays.asList(topic));

// poll for new data

while(true){

ConsumerRecords<String, String> records =

consumer.poll(Duration.ofMillis(100)); // new in Kafka 2.0.0

for (ConsumerRecord<String, String> record : records){

logger.info("Key: " + record.key() + ", Value: " + record.value());

logger.info("Partition: " + record.partition() + ", Offset:" + record.offset());

}

}

13. Kafka allows you to Assign a topic and partition and seek messages for it from a particular offset.

EX: https://github.com/simplesteph/kafka-beginners-course/blob/master/kafka-basics/src/main/java/kafka/tutorial1/ConsumerDemoAssignSeek.java

====OSI LAYERS========

1. The structure of the Internet is modeled on the Open Systems Interconnection (OSI) model.

2. 7 Layers are described below:

1. Physical(Bits, Media) : This layer includes the physical equipment involved in the data transfer, such as the cables and

switches. This is also the layer where the data gets converted into a bit stream, which is a string of 1s and 0s. The

physical layer of both devices must also agree on a signal convention so that the 1s can be distinguished from the 0s on

both devices.

2. Data Link(Frames, Media) : This layer facilitates data transfer between two devices on the SAME network. The data link layer

takes packets from the network layer and breaks them into smaller pieces called frames.

3. Network(Packetes, Media) : This layer is responsible for facilitating data transfer between two different networks. The

network layer breaks up segments from the transport layer into smaller units, called packets, on the sender’s device, and

reassembling these packets on the receiving device. The network layer also finds the best physical path for the data to

reach its destination; this is known as routing.

4. Transport(Segments, Host) : This layers takes data from the session layer and breaks it up into chunks called segments

before sending it to layer 3. The transport layer on the receiving device is responsible for reassembling the segments into

data the session layer can consume. The transport layer is also responsible for flow control and error control. Flow control

determines an optimal speed of transmission to ensure that a sender with a fast connection doesn’t overwhelm a receiver with

a slow connection. The transport layer performs error control on the receiving end by ensuring that the data received is

complete, and requesting a retransmission if it isn’t.

5. Session(Data, Host) : This is the layer responsible for opening and closing communication between the two devices. The

time between when the communication is opened and closed is known as the session. The session layer ensures that the

session stays open long enough to transfer all the data being exchanged, and then promptly closes the session in order to

avoid wasting resources. The session layer also synchronizes data transfer with checkpoints. For example, if a 100 megabyte

file is being transferred, the session layer could set a checkpoint every 5 megabytes. In the case of a disconnect or a

crash after 52 megabytes have been transferred, the session could be resumed from the last checkpoint, meaning only 50 more

megabytes of data need to be transferred. Without the checkpoints, the entire transfer would have to begin again from scratch.

6. Presentation(Data, Host) : This layer is primarily responsible for preparing data so that it can be used by the

application layer; in other words, layer 6 makes the data presentable for applications to consume. The presentation layer

is responsible for translation, encryption, and compression of data it receives from the application layer before delivering

it to layer 5.

7. Application(Data, Host) : This is the only layer that directly interacts with data from the user. Software applications like

web browsers and email clients rely on the application layer to initiate communications.

=====SL4J-LOGGING=======

1. The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks, such as

java.util.logging, logback and log4j. SLF4J allows the end-user to plug in the desired logging framework at deployment time.

Note that SLF4J-enabling your library/application implies the addition of only a single mandatory dependency, namely

slf4j-api-2.0.0-alpha2-SNAPSHOT.jar.

EX: import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

public class HelloWorld {

public static void main(String[] args) {

Logger logger = LoggerFactory.getLogger(HelloWorld.class);

logger.info("Hello World");

}

}

====APACHE-CAMEL====

1. Apache camel support routing of :

- Files

- API calls

-Messages

2. Apache camel has 2 methods for maintaining messages :

- queue : FIFO, point-to-point

- Topic : Pub-Sub

3. Apache messages are of 2 types:

- Message : Simple message

- Exchange : Container of message with more meta data, like if server is waiting after sending message.

4. Adding Apache-Camel to a projects requires below steps :

STEP-1 : Adding maven dependencies :

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-spring</artifactId>

<version>3.1.0</version>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-spring-javaconfig</artifactId>

<version>3.4.2</version>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-test-spring</artifactId>

<version>3.3.0</version>

<scope>test</scope>

</dependency>

STEP-2: Adding camel context in project(Camel context is actually an interface that has methods for camel route lifecycle like start,stop,configure etc)

- Create a new config class that will extend CamelConfiguration.

STEP-3: Creating Routes classes(Routes can be defined by java DSL or XML files)

- ?noop=true means do not change the actual file

1. Apache camel is a complete production-ready framework for people who want to implement their solution to follow the EIP

(Enterprise Integration Patterns).

2. The Apache camel functionality can be used in a class by extending the class RouteBuilder and overriding the function configure.

EX: public class SimpleRouteBuilder extends RouteBuilder {

@Override

public void configure() throws Exception {

String topicName = "topic=javainuse-topic";

String kafkaServer = "kafka:localhost:9092";

String zooKeeperHost = "zookeeperHost=localhost&zookeeperPort=2181";

String serializerClass = "serializerClass=kafka.serializer.StringEncoder";

String toKafka = new StringBuilder().append(kafkaServer).append("?").append(topicName).append("&")

.append(zooKeeperHost).append("&").append(serializerClass).toString();

from("file:C:/inbox?noop=true").split().tokenize("\n").to(toKafka);

}

}

3. Once the route class is ready with configure function, configure the main method of Application to load the camel routes once

application starts.

EX: import org.apache.camel.CamelContext;

import org.apache.camel.impl.DefaultCamelContext;

public class MainApp {

public static void main(String[] args) {

SimpleRouteBuilder routeBuilder = new SimpleRouteBuilder();

CamelContext ctx = new DefaultCamelContext();

try {

ctx.addRoutes(routeBuilder);

ctx.start();

Thread.sleep(5 \* 60 \* 1000);

ctx.stop();

}

catch (Exception e) {

e.printStackTrace();

}

}

}

4. In CBES we use camel as:

- Managed directory keeps track of directories and call respective processor for processing file.

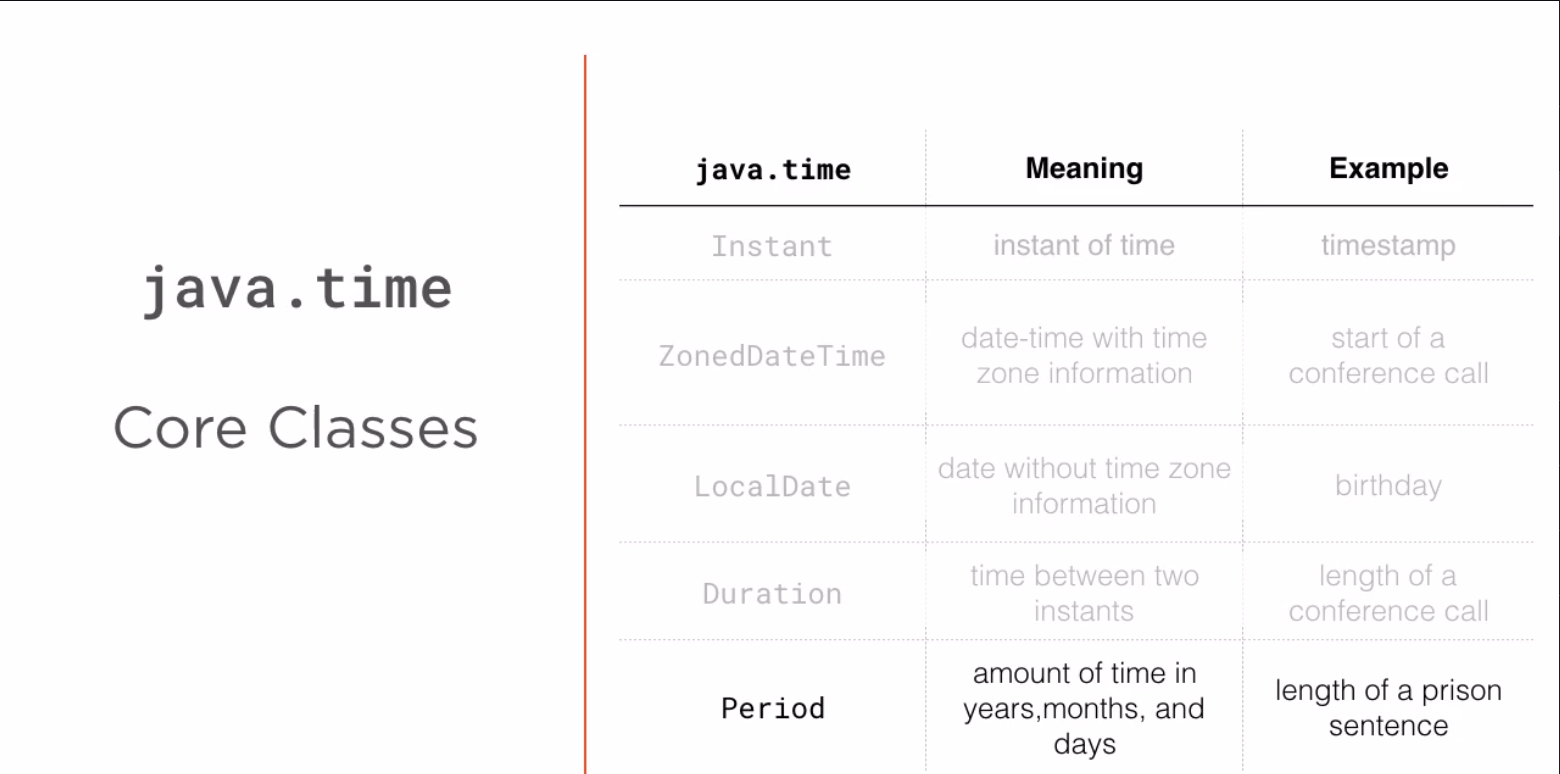
- The processor reads file and creates a file of messages and places it into processed directory

- Camel route takes file from processed and moves to output directory.

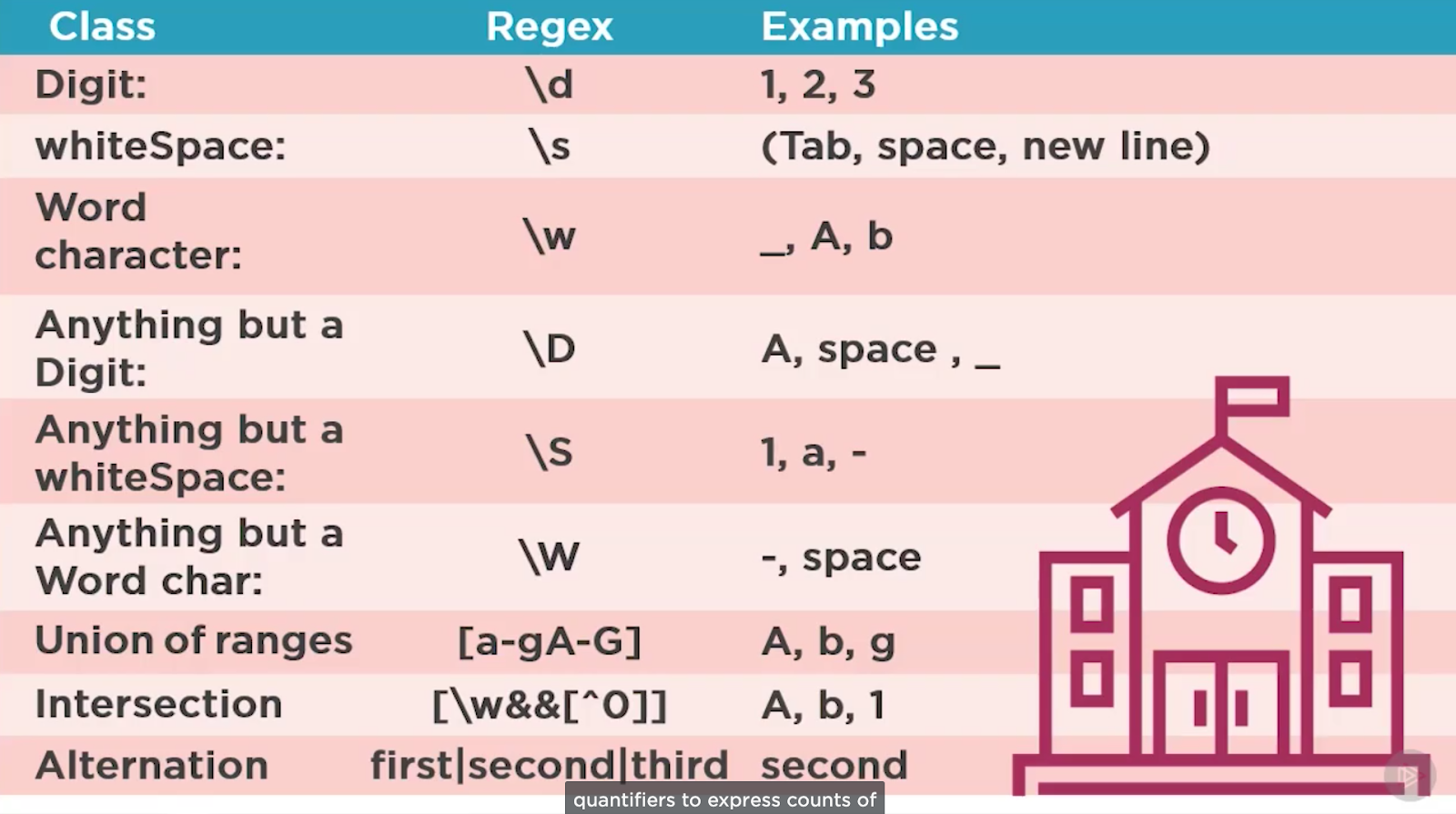
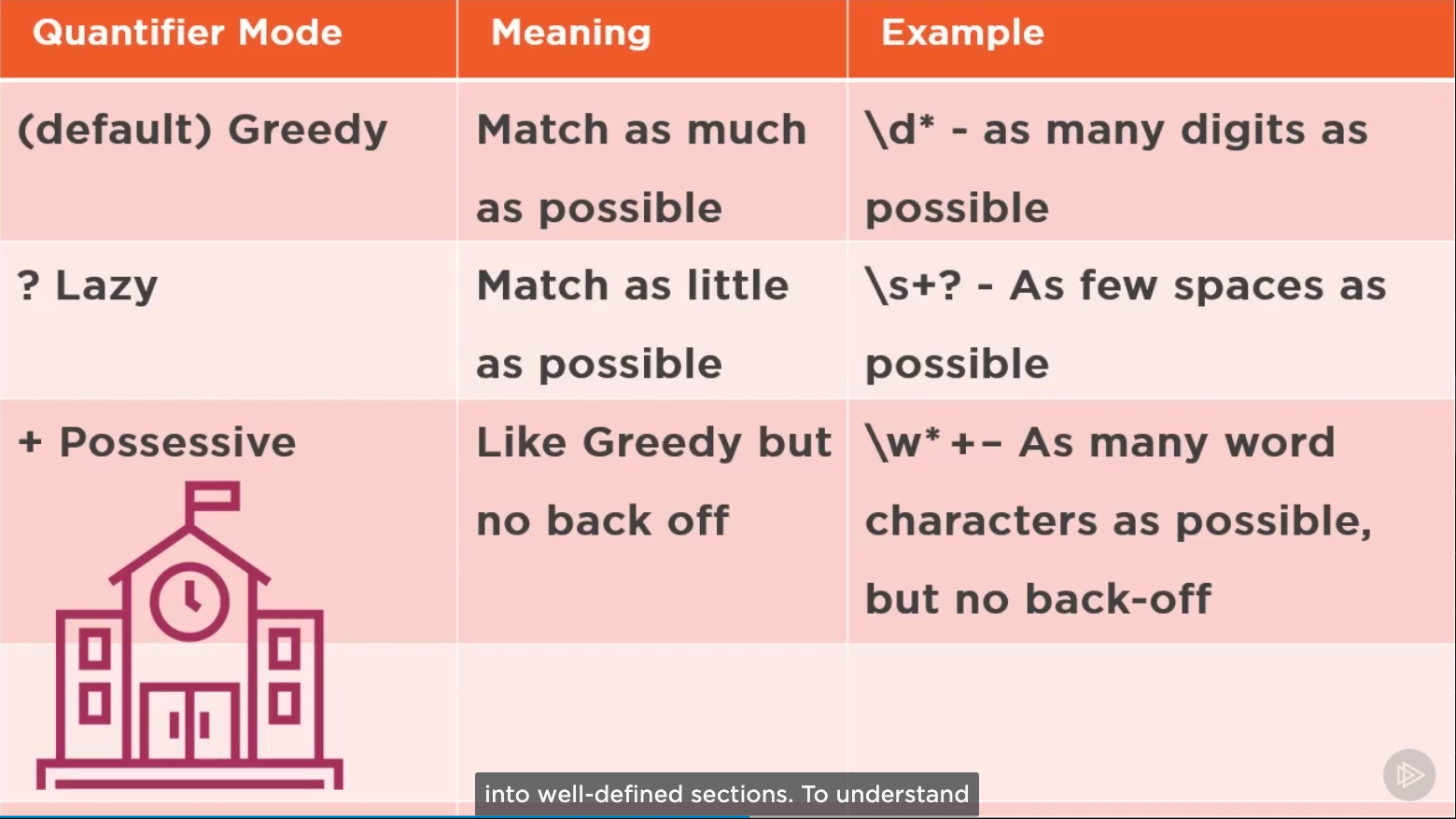
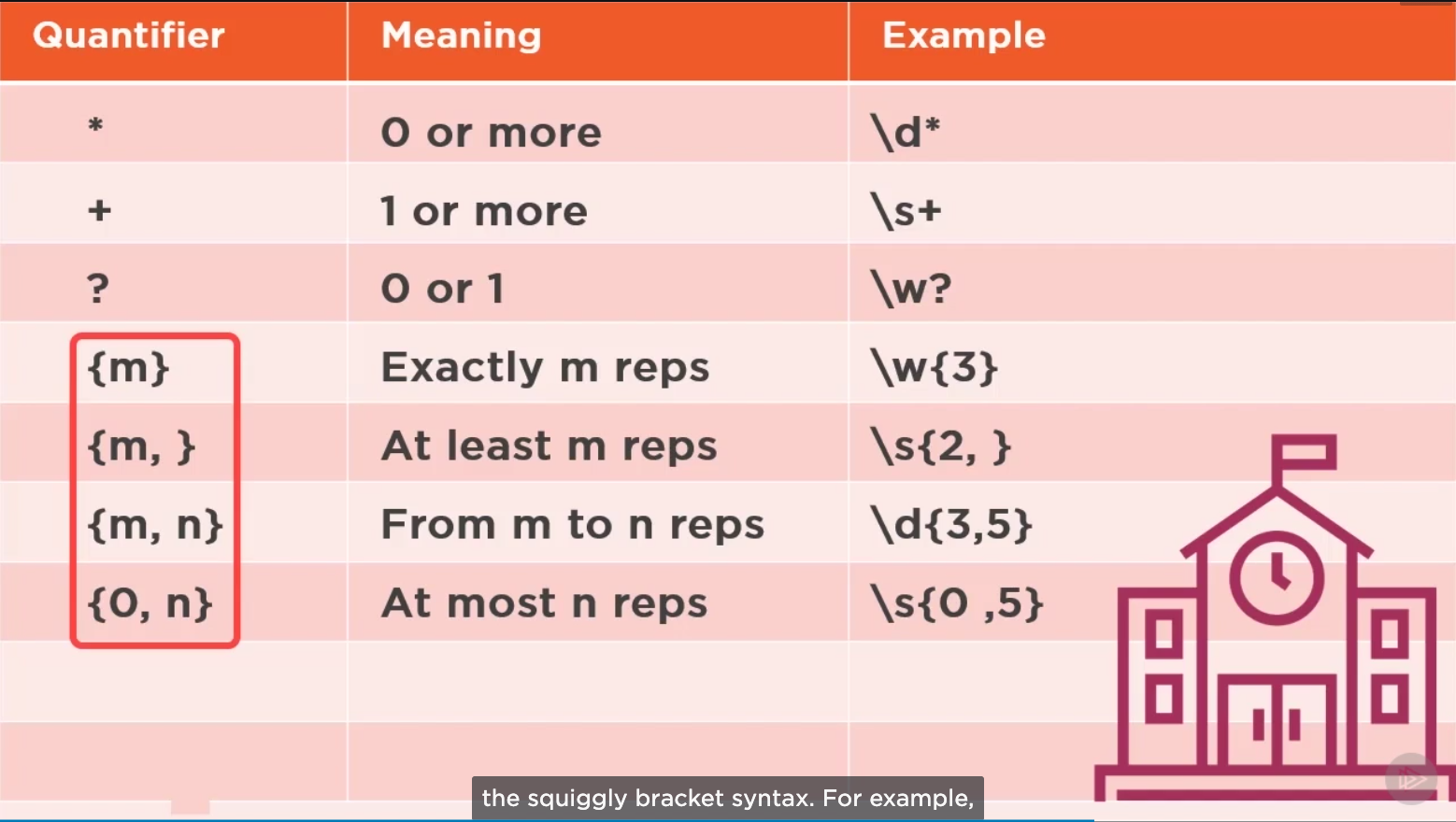
- Camel route then takes the file from output and pushes it to kafka as messages.

====JAVA-DATE-TIME====

1. Common Date time terms :



**=====RegEx Notes=====**

1. Meta characters in regex are < ( { [ \ ^ - = $ ! | ] } ) ? \* . + >
2. To match any of the above mentioned meta characters as well in a string you have to add before them \
3. To consider all characters as normal characters and not as meta characters you have can surround your string with **\Q**<pattern>**\E**
4. . is a wildcard that matches any character where you place the dot. Number of dots = number of characters matched.
5. \d : numbers
6. \D : Anything but a number
7. \s : whitespace
8. \S anything but white space
9. \w : word (a-z , 0 - 9 or \_)
10. \W : Anything but word (space or special characters)
11. When inputting the regex pattern in java as a string you have to use \ infront of “ and \ that might be part of the pattern
12. [ABCDEFG] : Character classes that will get you all the characters present inside the class.
13. [^ABCDEFG] : Character classes that will get you all the except characters present inside the class.
14. [\w&&[^1-3]] : This character class will match any word but not 1 to 3.
15. [\w | [@£]] : This character class will match any word or @£
16. [0-9a-z] : word
17. 
18. 
19. 

==== SOLID Coding Principal====