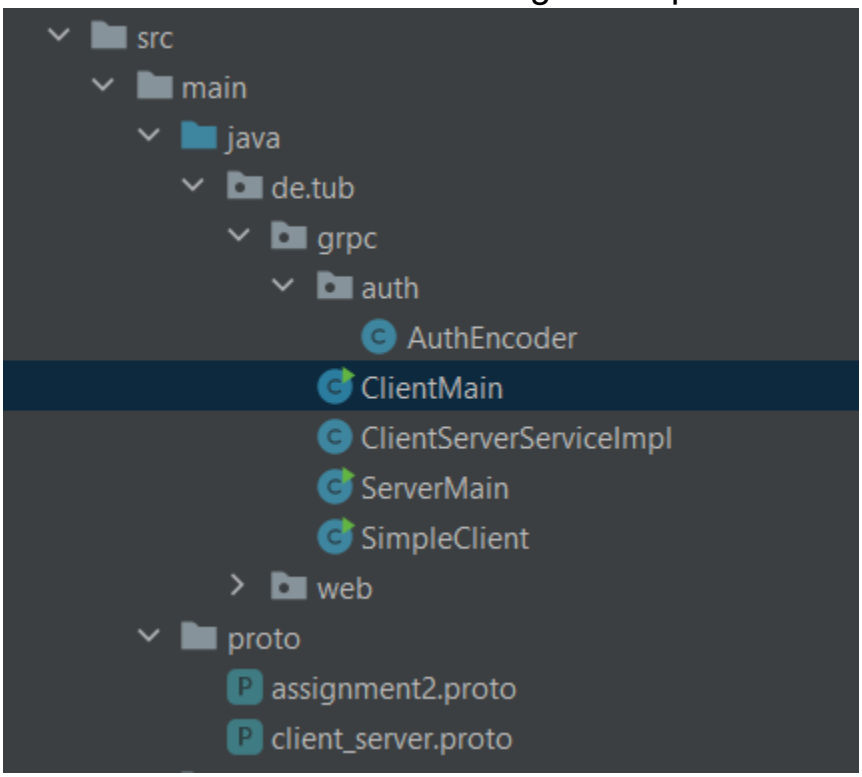
Anwendungssysteme Übungsaufgabe 2

Dimitar Dimov

ASSIGNMENT 1:

I created 3 new classes including a new proto file for this assignment.



Quick Note:
I tried to implement things as simply as possible and I choose to work with the Zombie Apocalypse Use Case. I haven't edited the pom.xml at all, this means no new dependencies. After I read the README, the maven build gave out no issues and I started to work on the assignment.

Technical Requirement 4:
Starting off from the last requirement is strange but that's the recommended workflow when coding with gRPC – create a proto file, generate stubs, and work from there with the implementation.

The first thing I did was write my own proto file called client_server.proto since that was stated in the last technical requirement. It's as simple as I could make it and only uses SurvivalRequest and SurvivalResponse as messages for the service.

```
service ClientServerService {
  rpc GetSurvivalProbability (SurvivalRequest) returns (SurvivalResponse);
}

message SurvivalRequest {
  string name = 1;
  repeated string abilities = 2;
  int32 strength = 3;
}

message SurvivalResponse {
  int32 probabilityToSurviveOneYear = 1;
  int32 probabilityToSurviveTwoYears = 2;
  string evaluation = 3;
}
```

Technical requirement 1:
I created 3 new classes for this. The first is called ClientMain and, as stated in the requirement, I communicate with the servers through the Scanner class. Prompting the user to write their name, choose their abilities and enter their strength is required in order to calculate the chances of us surviving in the apocalypse.

```
Scanner scanner = new Scanner(System.in);

System.out.print("Enter your name: ");
String name = scanner.nextLine();

System.out.print("Enter your abilities (comma separated): ");
String[] abilitiesArray = scanner.nextLine().split( regex: ",");
List<String> abilities = Arrays.asList(abilitiesArray);

System.out.print("Enter your strength (1-10): ");
int strength = scanner.nextInt();
```

The second class is the ServerMain class and that handles basic functionality on the server's side, for example failure handling and of course – the port.

```
        // Wait for le termination of the server
        server.awaitTermination();
    } catch (IOException e) {
        System.err.println("Server failed to start: " + e.getMessage());
        e.printStackTrace();
    } catch (InterruptedException e) {
        System.err.println("Server interrupted: " + e.getMessage());
        e.printStackTrace();
    }
```

And the last new class is the ClientServerServiceImpl for implementing the communication between the Client and the API Server through gRPC requests and the AuthEncoder class.

```
private static final String API_SERVER_TARGET = "34.32.36.165:8080";
1 usage
private static final String ISIS_EMAIL = "dimov@campus.tu-berlin.de";
1 usage
private static final String ISIS_MATRIKEL_NR = "481052";
1 usage
private static final String AUTH_STRING = AuthEncoder.generateAuthString(ISIS_EMAIL, ISIS_MATRIKEL_NR);
```

Technical requirement 2:

After requesting the string from the AuthEncoder class we send a request to the API Server (using streams because I know streams get more points xD)

```
// Create and send a request to the API-SERVER
Assignment2OuterClass.ZombieApocalypseRequest apiRequest = Assignment2OuterClass.ZombieApocalypseRequest.newBuilder()
        .setAuth(AUTH_STRING)
        .setName(request.getName())
        .addAllAbilities(abilities)
        .setStrength(request.getStrength())
        .build();
```

and process the response from the server. Finally, we send the response back to the client (the evaluation variable is for the third requirement)

```
ClientServer.SurvivalResponse response = ClientServer.SurvivalResponse.newBuilder()
        .setProbabilityToSurviveOneYear(apiResponse.getProbabilityToSurviveOneYear())
        .setProbabilityToSurviveTwoYears(apiResponse.getProbabilityToSurviveTwoYears())
        .setEvaluation(evaluation)
        .build();
```

Technical requirement 3:

As shown above, the communication is exactly as shown in the graphic. Request from Client -> Request from Server to API Server -> Response back to the Client

Sadly, I couldn't get the data "transformation" part to work. Like I said, I choose to define a evaluation String variable so that with its help we return to the client what the chances of survival look like in a more "normal" manner.

With an If-statement, we return these strings back to the client.

```
System.out.println("Sending request to API-SERVER: " + apiRequest);
Assignment2OuterClass.ZombieApocalypseResponse apiResponse = apiStub.zombieApocalypse(apiRequest);
System.out.println("Received response from API-SERVER: " + apiResponse);

// Process the API-SERVER response and tell the client how grim it's looking
String evaluation = apiResponse.getEvaluation();
if (apiResponse.getProbabilityToSurviveOneYear() > 80) {
    evaluation = "Excellent survival chances!";
} else if (apiResponse.getProbabilityToSurviveOneYear() < 50) {
    evaluation = "Your survival chances are low, consider improving your abilities.";
}
```

This was the idea, but it just doesn't return it how I'd like (with the strings above).

Additionally, I had some troubles with the abilities and I converted them from strings to enums.

```
try {
    // Convert abilities from String to ZombieApocalypseAbility enum
    List<ZombieApocalypseAbility> abilities = new ArrayList<>();
    for (String ability : request.getAbilitiesList()) {
        try {
            abilities.add(ZombieApocalypseAbility.valueOf(ability));
        } catch (IllegalArgumentException e) {
            System.err.println("Invalid ability: " + ability);
            responseObserver.onError(new IllegalArgumentException("Invalid ability: " + ability));
            return;
        }
    }
```

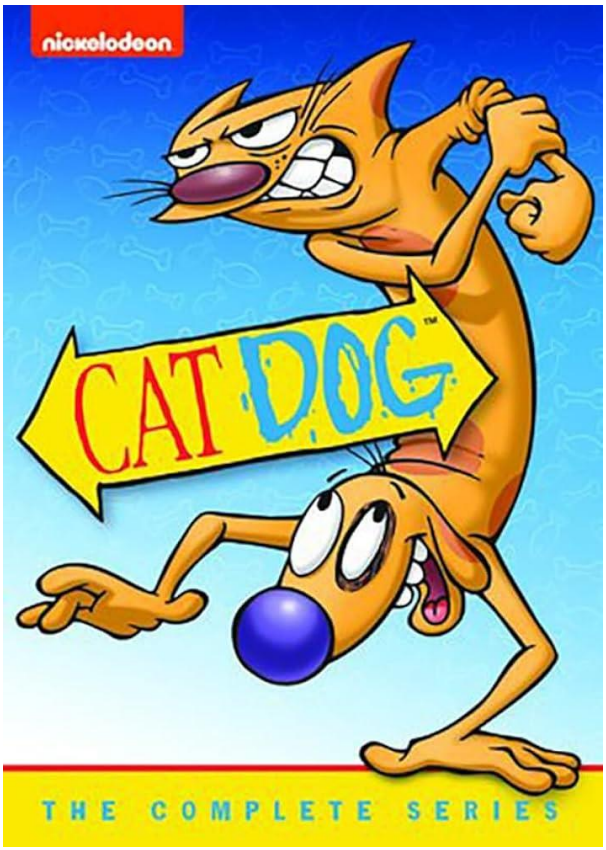
If you want to write multiple abilities in the console you'll have to write them with commas and no space inbetween them (the console infroms you of this as well)

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Enter your name: bibi
Enter your abilities (comma separated): IMMUNE_TO_BITES,AWESOME_ZOMBIE_DANCE_MOVES
Enter your strength (1-10): 5
Probability to survive one year: 39
Probability to survive two years: 18
Evaluation: Your survival chances are low, consider improving your abilities.
```

(No idea how to fix the SLF4J error, I even tried working with basic sout commands)

To summarize:
This was waaaay harder than expected, it took me 3-4 days just reading through the VL slides, the protobuff site and gRPC site, AND YouTube videos just to understand the assignment. Was fun tho!

ASSIGNMENT 2: CatDog



Hear me out… I wanted something easy-to-understand and easy-to-implement as a solution to this assignment. For my two API's I choose these two:

https://dog.ceo/api/breeds/image/random

https://api.thecatapi.com/v1/images/search

The idea is that I (the Client) fetch random images from these two API's (hence the name CatDog 😉). My code is heavily influenced by the SimpleAPICall class, as I wanted something that works without fail.

Technical requirement 1:

```
private static void fetchCatImage() throws Exception {
    URI uri = new URIBuilder()
            .setScheme("https")
            .setHost("api.thecatapi.com")
            .setPath("/v1/images/search")
            .build();
```

```
private static String fetchDogImage(String breed) throws Exception {
    URIBuilder uriBuilder = new URIBuilder()
            .setScheme("https")
            .setHost("dog.ceo")
            .setPath("/api/breeds/image/random");
```

The two API's in my client.

Technical requirement 2:

I completed this requirement by prompting the user how many images he/she wants to receive. This is done at the very beginning with the console input and a for-loop.

```
public static void main(String[] args) throws Exception {
    Scanner scanner = new Scanner(System.in);

    // Prompt user for the number of cat images
    System.out.print("Enter the number of cat images you want to see: ");
    int catImageCount = scanner.nextInt();

    // Prompt user for the number of dog images
    System.out.print("Enter the number of dog images you want to see: ");
    int dogImageCount = scanner.nextInt();

    // Fetch and display cat images
    System.out.println("Fetching cat images...");
    for (int i = 0; i < catImageCount; i++) {
        try {
            fetchCatImage();
        } catch (Exception e) {
            System.err.println("Failed to fetch cat image: " + e.getMessage());
        }
    }
```

Technical requirement 3:

I understand this as the option to use different values when requesting a response from the two API's. Since we're using the console for user input, the parameter is not static. Every time you could want a different number of images.

```
Enter the number of cat images you want to see: 2
Enter the number of dog images you want to see: 4
```

There's no limit to the number of images you ask for, als Hinweis nur.

Technical requirement 4:

At the beginning of the CatDogClient you're met with this list of dog breeds

```
//I got tired, sorry if your dog breed isn't in here. Tbh I didn't know most of these...
2 usages
private static final List<String> validBreeds = Arrays.asList(
        "affenpinscher", "african", "airedale", "akita", "appenzeller", "australian",
        "basenji", "beagle", "bluetick", "borzoi", "bouvier", "boxer", "brabancon",
        "briard", "buhund", "bulldog", "bullterrier", "cairn", "cattledog", "chihuahua",
        "chow", "clumber", "cockapoo", "collie", "coonhound", "corgi", "cotondetulear",
        "dachshund", "dalmatian", "dane", "deerhound", "dhole", "dingo", "doberman",
        "elkhound", "entlebucher", "eskimo", "finnish", "frise", "germanshepherd",
        "greyhound", "groenendael", "hound", "husky", "keeshond", "kelpie", "komondor",
        "kuvasz", "labrador", "leonberg", "lhasa", "malamute", "malinois", "maltese",
        "mastiff", "mexicanhairless", "mix", "mountain", "newfoundland", "otterhound",
        "papillon", "pekinese", "pembroke", "pinscher", "pitbull", "pointer", "pomeranian",
        "poodle", "pug", "puggle", "pyrenees", "redbone", "retriever", "ridgeback",
        "rottweiler", "saluki", "samoyed", "schipperke", "schnauzer", "setter", "sheepdog",
        "shiba", "shihtzu", "spaniel", "spitz", "springer", "stbernard", "terrier",
        "vizsla", "weimaraner", "whippet", "wolfhound"
);
```

The idea is that when you choose the number of dog images, the first random resulting dog is checked for its breed. After establishing the breed, every other image will be of the same breed. This way we use that response from the API to determine the rest of the output.

I defined a String called previousDogBreed and set it to null, after which the for-loop iterates

```
String previousDogBreed = null;
for (int i = 0; i < dogImageCount; i++) {
    try {
        previousDogBreed = fetchDogImage(previousDogBreed);
    } catch (Exception e) {
        System.err.println("Failed to fetch dog image: " + e.getMessage());
        previousDogBreed = null; // Reset breed if fetching fails
    }
}

scanner.close();
```

We check the available “valid” breeds from the ArrayList at the beginning

```
if (breed != null && validBreeds.contains(breed)) {
    uriBuilder.setPath("/api/breed/" + breed + "/images/random");
}
```

The rest is error handling. As additional output however you will receive the name of the breed (added that because like I said, I don’t know most of these breeds)

```
// Validate the new breed
if (!validBreeds.contains(newBreed)) {
    System.out.println("Invalid breed extracted: " + newBreed);
    return null;
}

System.out.println("Extracted breed: " + newBreed);
return newBreed;
```

To summarize:
I found this one easier than the first assignment, probably because a lot of thought went into choosing the API’s to make my life a bit easier.