

Assignment 2

Q1. Pull any image from the docker hub, create its container, and execute it, showing the output.

Docker hub is an online image hosting service by Docker, from where users can access a large variety of images and create containers from them.

First, we will pull an image, let's say hello world image, using the command "docker pull hello-world".

```
\pratik@DESKTOP-V4HRCK7 MINGW64 ~  
$ docker pull hello-world  
Using default tag: latest  
latest: Pulling from library/hello-world  
Digest: sha256:6e8b6f026e0b9c419ea0fd02d3905dd0952ad1feea67543f525c73a0a790fefb  
Status: Image is up to date for hello-world:latest  
docker.io/library/hello-world:latest
```

The image is successfully pulled from docker hub, and ready to use on our local machine.

To create a container for this image, we use the command "docker create hello-world".

```
\pratik@DESKTOP-V4HRCK7 MINGW64 ~  
$ docker create hello-world  
fe6917312866997a5a89d5bbfac1e9ff6f7a87909266f1daf0b4b16a153cafbe
```

The container is created, and the output we have received, is the container ID.

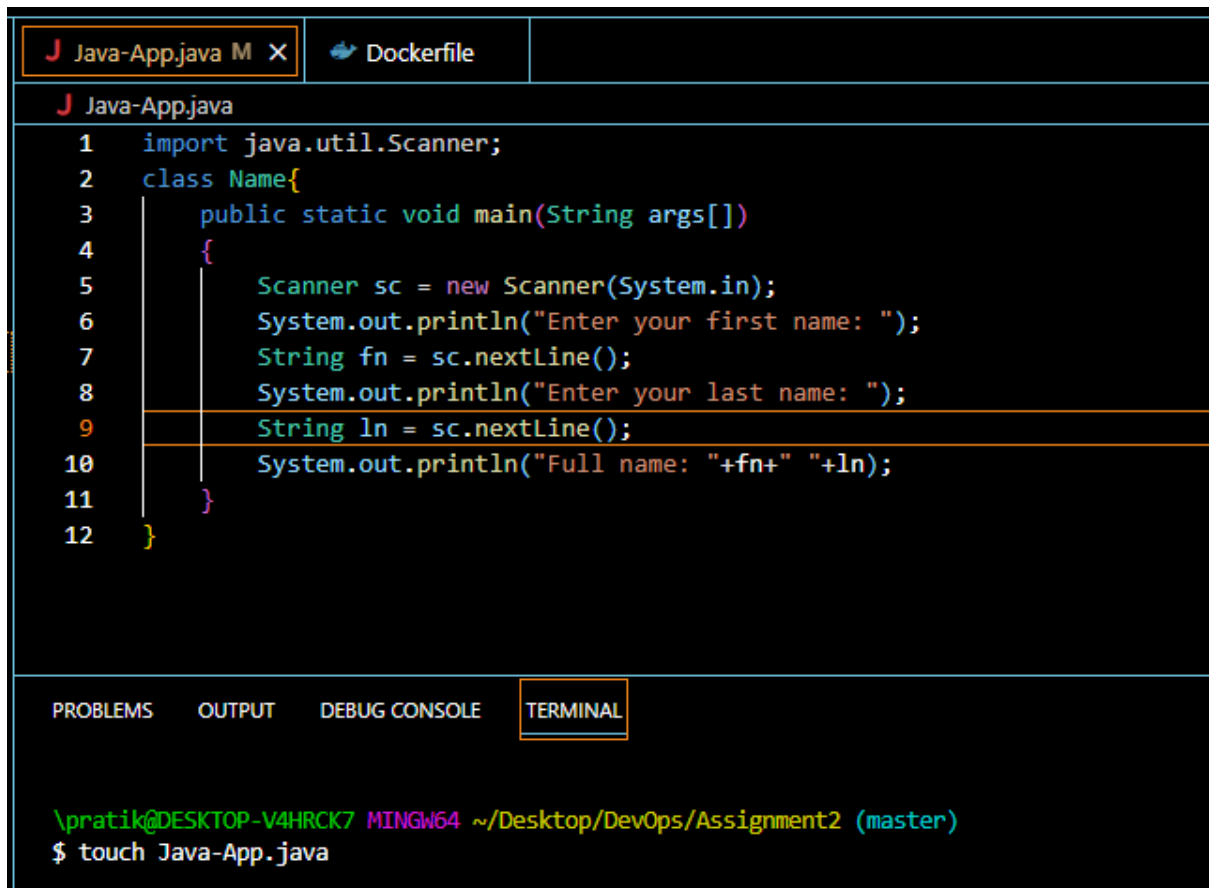
Now, to start this container, we use the command "docker start [container ID]".

```
\pratik@DESKTOP-V4HRCK7 MINGW64 ~  
$ docker start -a fe6917312866997a5a89d5bbfac1e9ff6f7a87909266f1daf0b4b16a153cafbe  
  
Hello from Docker!  
This message shows that your installation appears to be working correctly.  
  
To generate this message, Docker took the following steps:  
1. The Docker client contacted the Docker daemon.  
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.  
   (amd64)  
3. The Docker daemon created a new container from that image which runs the  
   executable that produces the output you are currently reading.  
4. The Docker daemon streamed that output to the Docker client, which sent it  
   to your terminal.  
  
To try something more ambitious, you can run an Ubuntu container with:  
$ docker run -it ubuntu bash  
  
Share images, automate workflows, and more with a free Docker ID:  
https://hub.docker.com/  
  
For more examples and ideas, visit:  
https://docs.docker.com/get-started/
```

Our container is started, and we can see the output.

Q2. Create a basic Java application, generate its image with necessary files, and execute it with docker.

First, we will create our Java program.



The screenshot shows an IDE window with two tabs: 'Java-App.java M' and 'Dockerfile'. The 'Java-App.java' tab is active, displaying the following code:


```
1 import java.util.Scanner;
2 class Name{
3     public static void main(String args[])
4     {
5         Scanner sc = new Scanner(System.in);
6         System.out.println("Enter your first name: ");
7         String fn = sc.nextLine();
8         System.out.println("Enter your last name: ");
9         String ln = sc.nextLine();
10        System.out.println("Full name: "+fn+" "+ln);
11    }
12 }
```

Below the code editor, there are four tabs: 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is active, showing the following command prompt output:

```
\pratik@DESKTOP-V4HRCK7 MINGW64 ~/Desktop/DevOps/Assignment2 (master)
$ touch Java-App.java
```

We have created a simple application that takes first name and last name as input and returns the full name as the output.

Next, we will create the Dockerfile, which will enable our application to become an image



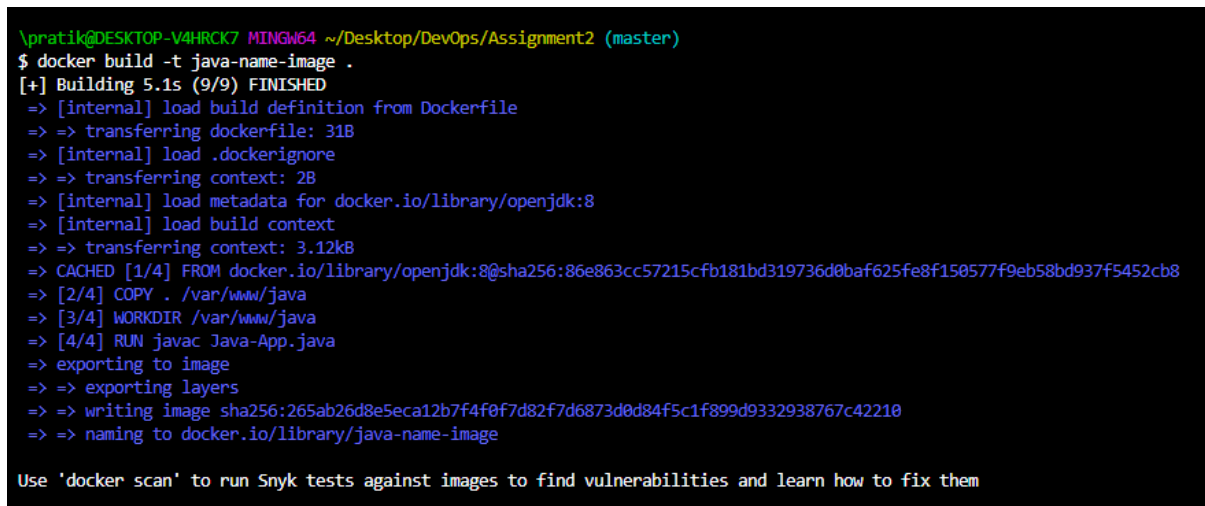
The screenshot shows an IDE interface. At the top, there are two tabs: 'Java-App.java M' and 'Dockerfile X'. The 'Dockerfile' tab is active, displaying the following content:

```
1 FROM openjdk:8
2 COPY . /var/www/java
3 WORKDIR /var/www/java
4 RUN javac Java-App.java
5 CMD ["java", "Name"]
```

Below the editor, there is a panel with four tabs: 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'TERMINAL'. The 'TERMINAL' tab is active, showing the following command prompt and output:

```
\pratik@DESKTOP-V4HRCK7 MINGW64 ~/Desktop/DevOps/Assignment2 (master)
$ touch Dockerfile
```

Now, we will build the image, using the command “docker build –t [image name] [path]”.



The screenshot shows a terminal window with the following output:

```
\pratik@DESKTOP-V4HRCK7 MINGW64 ~/Desktop/DevOps/Assignment2 (master)
$ docker build -t java-name-image .
[+] Building 5.1s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 31B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/openjdk:8
=> [internal] load build context
=> => transferring context: 3.12kB
=> CACHED [1/4] FROM docker.io/library/openjdk:8@sha256:86e863cc57215cfb181bd319736d0baf625fe8f150577f9eb58bd937f5452cb8
=> [2/4] COPY . /var/www/java
=> [3/4] WORKDIR /var/www/java
=> [4/4] RUN javac Java-App.java
=> exporting to image
=> => exporting layers
=> => writing image sha256:265ab26d8e5eca12b7f4f0f7d82f7d6873d0d84f5c1f899d9332938767c42210
=> => naming to docker.io/library/java-name-image

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them
```

We can run the image container using command “docker run –i java-name-image”. The –i option indicates that the container has to be run in interactive mode, as it has to accept input.

```
\pratik@DESKTOP-V4HRCK7 MINGW64 ~/Desktop/DevOps/Assignment2 (master)
$ docker run -i java-name-image
Enter your first name:
Pratik
Enter your last name:
Raj
Full name: Pratik Raj
```

We have successfully run our Java application.