THEORY QUESTIONS ASSIGNMENT
DANIELA TRIPON
Full stack 1

Full Stack Stream
(Maximum Score: 100)

KEY NOTES

- This assignment is to be completed at the student's own pace and submitted before the given deadline.
- There are **8** questions in total and each question is marked on a scale 1 to 20. The maximum possible grade for this assignment is 100 points.
- Students are welcome to use any online or written resources to answer these questions.
- The answers need to be explained clearly and illustrated with relevant examples where necessary. Your examples can include code snippets, diagrams or any other evidence-based representation of your answer.

Theory questions	**Points allocated per Question**

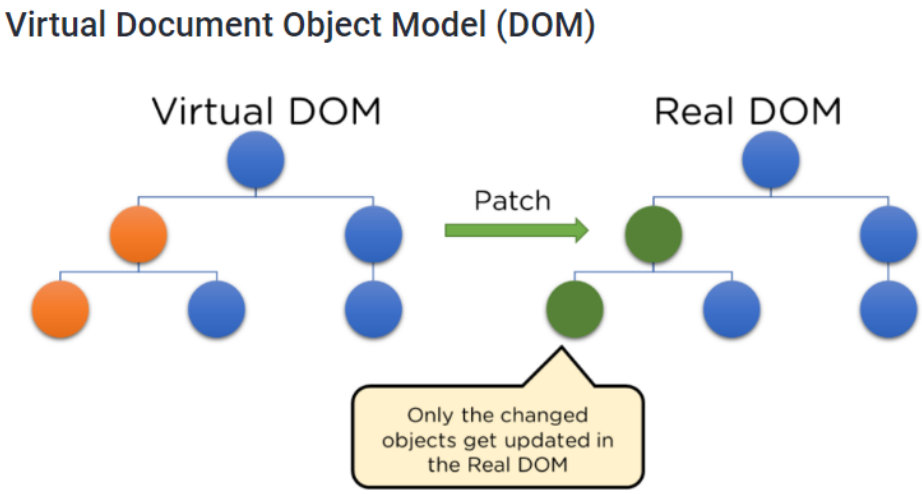
1. What is React? (*E.g. Consider: what is it? What is the benefit of using it? What is its virtual DOM? Why would someone choose it over the standard HTML / CSS stack?)*(15 marks)

Answer:

React is one of the most popular JavaScript libraries for front-end development and it was developed by Facebook in 2011 - 2013. One of the famous apps built entirely on React is Instagram. There are other big companies that used React on part of their apps, such as Netflix, Facebook, Airbnb, Reddit, HelloSign, Feedly, Imgur, and others. Among the benefits of using it in our projects, is its scalability and practicability, meaning that by using only JavaScript, we can code the client/server (Node) and mobile (React Native) side, 360 views, VR (virtual reality), resulting in easy to maintain code, great productivity, code division between team members, and leads to a cleaner architecture comparative to other JavaScript frameworks, and to a greater performance.

React is an open-source, declarative, component-based library used especially for creating interactive UIs (user interface). Declarative means that is easy to debug, update, and render only the components where there were changes, making the code more predictable. Each component's logic is encapsulated, and each state is managed individually. Thanks to this fact, that means that we can have complex data in our app and the state is not passed to the Real DOM (Document Object Model). Once the components are connected, we can build intricate UIs. Best part about components is that they are reusable.

React creates and uses Virtual DOM to render the components. This checks the components' state, and if there was any change in it, then it will update only those components in the Real DOM. This leads to a great performance and is based on complex algorithms that work behind the scenes to calculate the lesser number of updates required. Besides this, Virtual DOM also helps to reduce CPU power and mobile phone's battery consumption.



The components are written using JSX (JavaScript XML) syntax which is a JavaScript extension that supports writing HTML code straight in React. Coding in HTML and styling with CSS only is very repetitive and for a big project is time consuming and hard to manage. We can import and use other libraries with React for styling, formatting dates, codes, and others that speed up the development. Based on the facts mentioned in this answer, React is an easy choice over the standard HTML / CSS stack.

To create a React app, we need to use the command "`npx create-react-app my-react-app`" in our terminal. Once the app and all the packages are installed (including ESLint, Babel, webpack), we can select the app as the current directory (cd) and start run it in our browser by using the command "npm start". At this stage, we can start making modifications on the files, add new files/ folders, and create our own components.

2. What are Props? What is State? What is the difference between them? (10 marks)

Answer

Props are properties for components in React and are used to pass information or data to a component to make it dynamic. They can not be changed as are read-only, and a component can have many props, called with different values.

Example code:

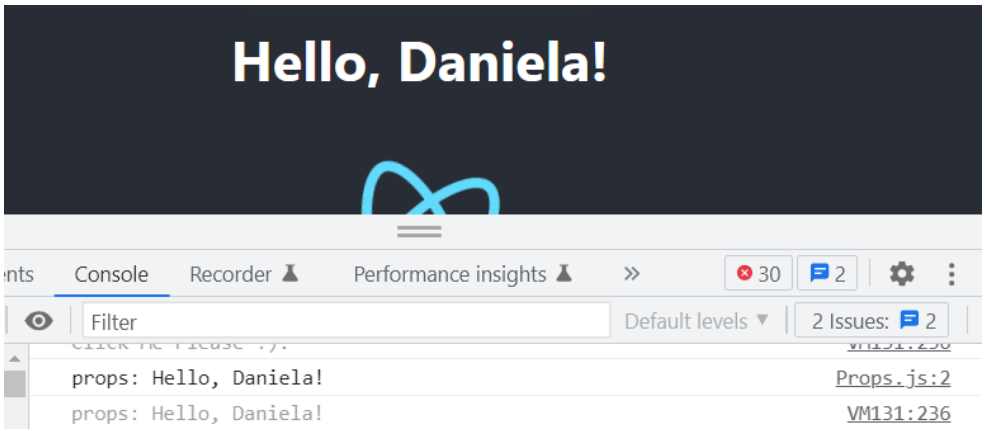
- We create a new component called Props and export it

```
JS Props.js U ×   JS App.js M ●
src > JS Props.js > Props
1  function Props(props) {
2    console.log(`props: ` + props.greeting);
3
4    return <h1>{props.greeting}</h1>;
5  }
6
7  export default Props;
8
```

- in App.js we import the new component called Props and add it in return () method. We also add a props named "greeting" with a value "Hello, Daniela!"

```
<Props greeting="Hello, Daniela!" />
```

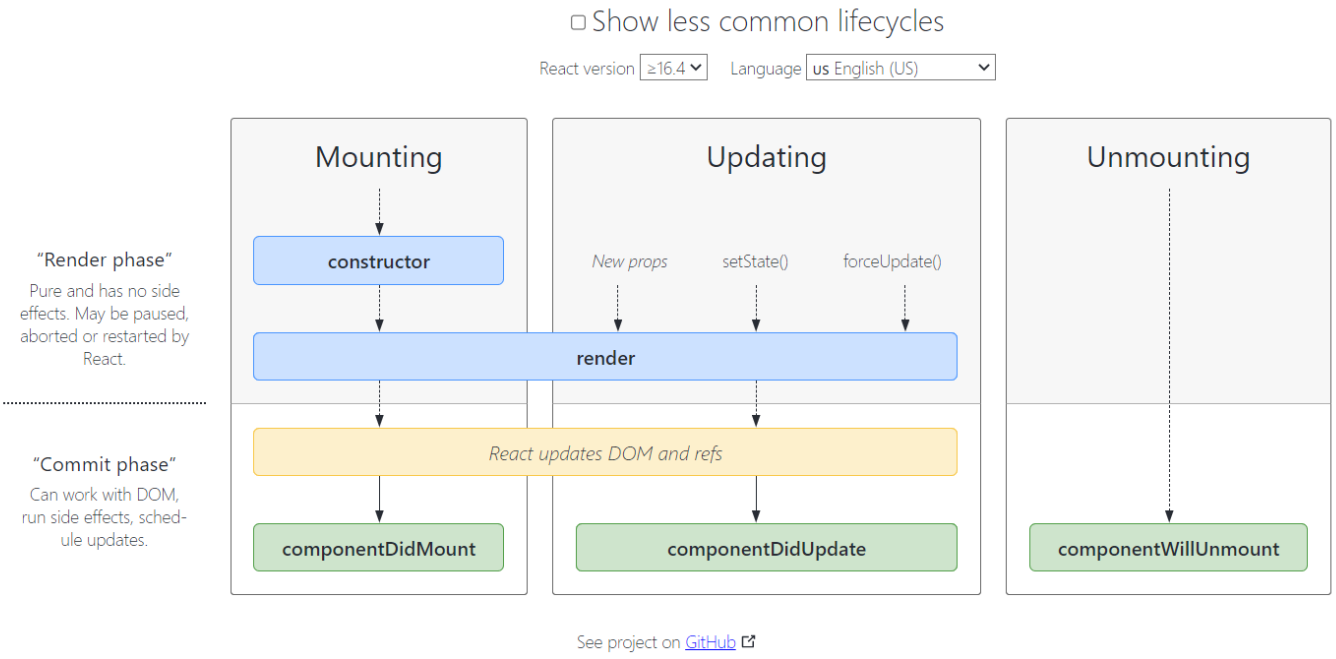
- We start the server and run the app. This is the output in the console and on the browser:



State is an object created in the function Object() and stores the properties values of a component. These states can change values upon a change in the network or a user's action. When this happens, React renders the component again. State can be used only in class components as it is. If we want to use state in functional components, we will have to use hooks which will be discussed in the next answer.

States are managed with lifecycles methods which will mount the component when runs for the first time, update on re-rendering it, and unmount when whenever the DOM created by the State component is removed. The most used Lifecycle methods are:

a) componentDidMount() – Mounting
b) componentDidUpdate() - Updating
c) componentWillUnmount() - Unmounting



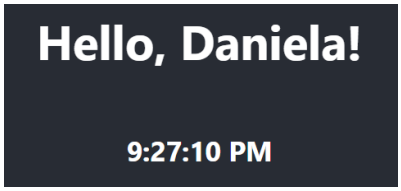
- We create a new component called State and export it

```
src > JS State.js > State > render
1  import React, { Component } from "react";
2
3  export default class State extends Component {
4    constructor(props) {
5      super(props);
6      this.state = { date: new Date() };
7    }
8    render() {
9      return (
10       <div>
11         <h3>{this.state.date.toLocaleTimeString()}</h3>
12       </div>
13     );
14   }
15 }
16
```

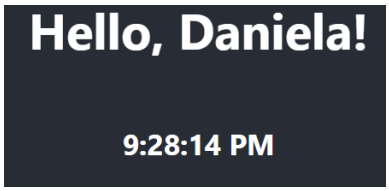
- in App.js we import the new component called State and add it in return () method.

```
<State />
```

- We start the server and run the app. This is the output in the browser (initial state):



- This is the output in the browser (updated state - on browser refresh):



Differences between props and state:

Props	State
Once they have been set, they are immutable.	Data can change and is stored in the state.
These are used to pass events and data to component's children.	It stores the view in the state of the components it needs to present to it.
Props can be used in class and functional components.	State can be used in class components. To use state in a functional component, we need to import useState which is a React Hook.
The props are defined in the parent component.	State is usually updated by event handlers.

3. What are React Hooks? How do they differ from existing lifecycle methods? (10 marks)

Answer

Hooks allows us to use states without using a class and are a new feature of React 16.8 version. React comes with built in hooks such as useState, but it also allows us to create new hooks. React hook useState is imported from React and comes in a pair that is made of the current state and a setter function setSate() that manages the state of the component. This allows using states in functional components as well.

Hooks example:

- We create a new component called Hooks and export it

Props.js U | Hooks.js U | App.js M

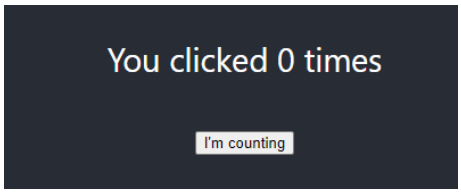
src > Hooks.js > default

```
import React, { useState } from "react";

function Hooks() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>I'm counting</button>
    </div>
  );
}

export default Hooks;
```

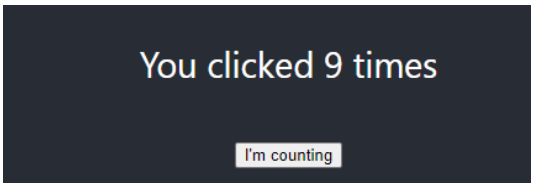
- in App.js we import the new component called Hooks and add it in return () method.

```
<Hooks />
```

- We start the server and run the app. This is the output in the browser (initial state):



- This is the output in the browser (updated state):



Class example with lifecycle methods:

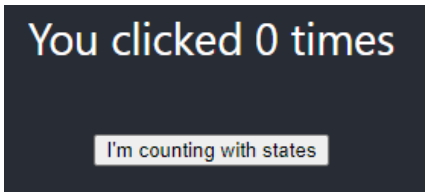
- We create a new component called Classes and export it

```
src > JS Classes.js > Classes > render
1  import React, { Component } from "react";
2
3  export default class Classes extends Component {
4    constructor(props) {
5      super(props);
6      this.state = { count: 0 };
7    }
8    render() {
9      return (
10       <div>
11         <p>You clicked {this.state.count} times</p>
12         <button onClick={() => this.setState({ count: this.state.count + 1 })}>
13           I'm counting with states
14         </button>
15       </div>
16     );
17   }
18 }
```

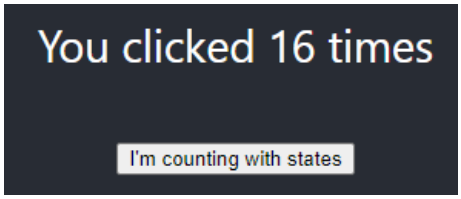
- in App.js we import the new component called Hooks and add it in return () method.

```
<Classes />
```

- We start the server and run the app. This is the output in the browser (initial state):



- This is the output in the browser (updated state):



React Hooks are better to use than the lifecycle methods for the following reasons:

- Without hooks, it's difficult to reuse stateful logic between components because components require the use of patterns (for example render props) which change the component's structure and make the code very complicated. Hooks solve this problem by allowing the components to share hooks, to keep its hierarchy, to reuse stateful logic, and to test the logic independently.
- By using the useState hook and a functional component, the size of our code is significantly reduced comparative to a class component and the lifecycle methods used to manage the states.
- Components can get complex with lots of logic that lifecycle methods can make it hard to manage. Hooks supports dividing a component into several functions based on relationships between them (for example fetching data, or a subscription) and gives the

option to use a reducer (Using the Effect hook) to manage the local state in a predictable way.
- By using lifecycle methods and classes, many people find it hard to learn and to understand concepts such as "this" keyword, OOP concepts, and the binding process for event handlers. By using hooks, people can access imperative escape hatches without learning complicated functional or reactive programming.
- useEffect hook is an effect hook that can affect other components, does the same thing as componentDidMount, componentDidUpdate, and componentWillUnmount combined, and cannot be done during rendering.

4. Design the perfect door - what should it look like, what are the components for it? What design heuristics should it follow, and how does your design match? What made you choose this design? (20 marks).

    a. *Consider in particular (likely need to do independent learning): who are your stakeholders? What is their personas? What is the doors requirements and how will your stakeholders benefit from your solution?*

Answer

First, we need to establish the purpose of the door and its use. Some of the options we need to consider are:

- Is this a door used in a commercial setting or a home?
- Is it a main door which requires a lock as well or is a door for a room/ bathroom?
- Does it require any glass?
- Is it a standard door or used for emergencies?
- Does it require a handle of any kind?

For this example, we assume we design the front door for an office. To achieve a good design, I am following Norman's 7 principles/heuristics for design and Nielsen's 10 general design principles. I consider this design to be a design that people are used to it, and it makes them feel natural users. It is the expected behaviour of a door (recognition rather than recall), offering consistency and following standards, and the decision was made based on brainstorming and researching doors and the design principles and norms mentioned above. By understanding human-behaviour and their expectations, I anticipated this design based on users' possible actions.

Door requirements:

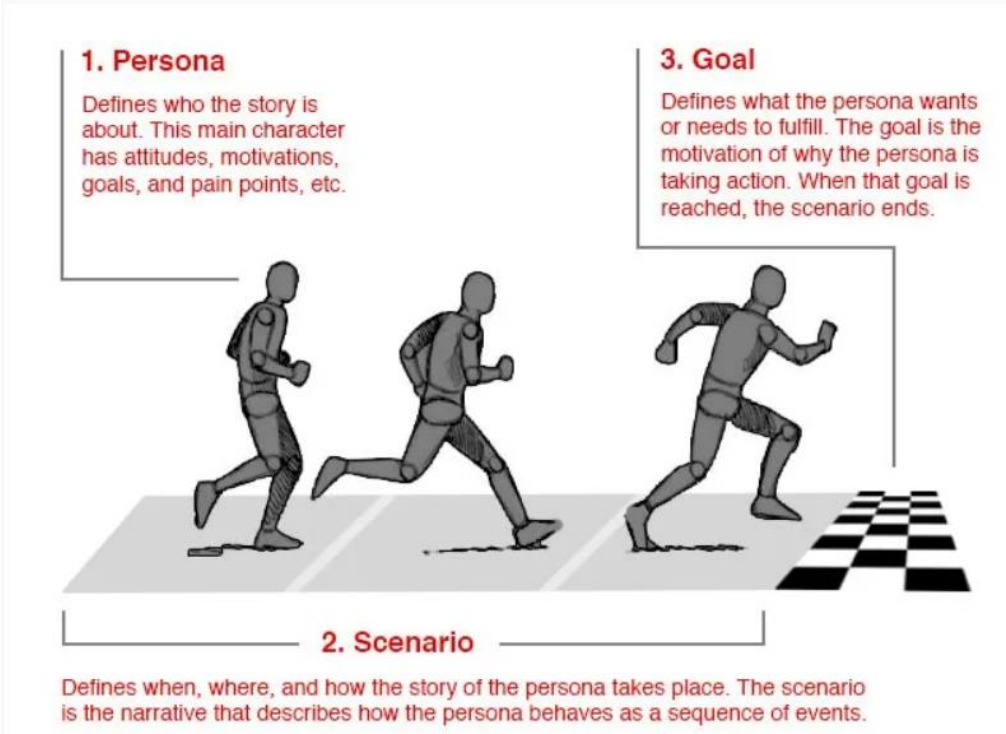
- Because it is the front door, the door needs a lock component, a handle component, hinges components, a frame component, keys component, and no glass.
- The door should be made of wood, be water resistant, and be easy to manipulate.
- The three hinges should be strong and durable, and placed at equal distances to each other.
- The users are used to open by pushing the door with their right hand/ right side of the body.
- The lock mechanism should allow the key to enter easily, and the process of locking/unlocking the door should be smooth.
- There will be two categories of users: those who have a key to lock/unlock the door, and visitors. The regular users will have a good understanding of the door opening direction and is

manipulation. On the other side, visitors will need to feel the process of opening and manipulating the door as naturally as possible even if this will be the first time they will use it.
- It's common use for an entrance door to open towards inside and not outside.
- It's common use to lock the door towards the frame and unlock it on the opposite direction.
- Add a message on each side of the door with a clear text, telling users how to operate the door to remove confusion (design for error): push → enter, pull → exit, just above the handle. Users scan the door from left → right to check for any messages/information. For this reason, the handle is positioned on the left side, with the message above, in capital letters.

It is important to have a good understanding of the stakeholders and a great communication with them to avoid issues, delays, or failed deadlines. Not all stakeholders play the same importance, however, the ones that are more important, are labelled as key stakeholders and they have a great impact on the design. The stakeholders are divided into internal and external stakeholders; the internal stakeholders have a straight relationship with the project and the result affects them directly. The same ones can have personas associated to them which are a set of information linked to each stakeholder label and can be related to their way of seeing things, feeling, thinking, or prefer to communicate.

In this case scenario the internal stakeholders are the client/ sponsor, the project manager, and the design team and are grouped by their roles. Each of these personas contain information such as background/ core information about them, their motivations, barriers/ blockages that might intervene, and communication preferences. According to Lene Nielsen, there are four types of personas:

- **Goal oriented persona** is based on a good understanding of user needs acquired through a thorough user research and has a clear understanding of what the user wants and how the user will use the door. This persona is associated with Alan Cooper (The Father of Visual Basic) perspective.



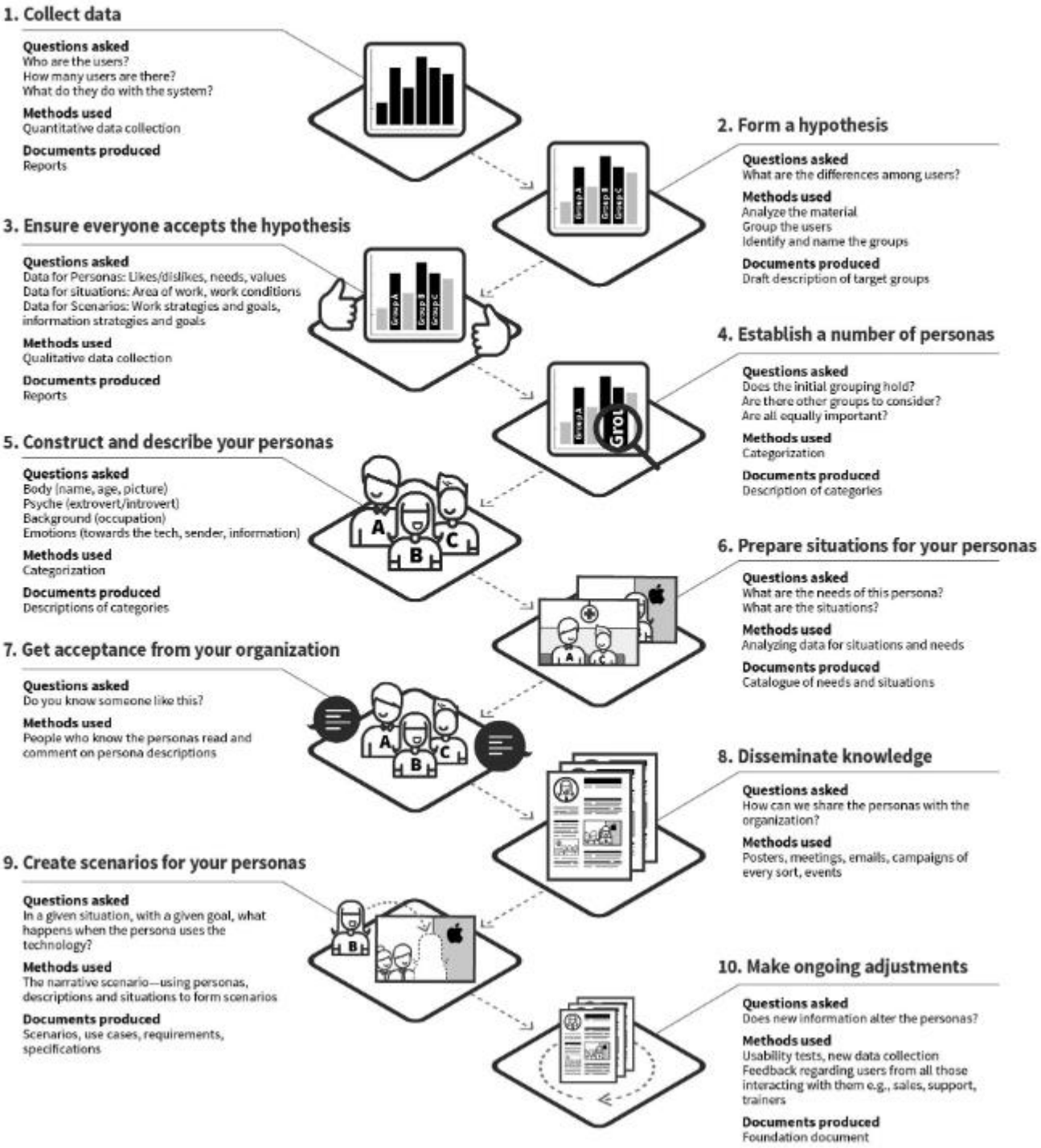
- **Role-based persona** is also goal oriented, but it emphasises on behaviour too. This persona is running lots of data-driven research based on quantitative and qualitative sources to ensure a successful design. Advocated for this persona are Tamara Adlin, John Pruitt, and Jonathan Grudin.
- **Engaging personas** is a combination between the previous two personas plus a traditional rounded persona and helps to visualise the persona as a 3D figure. It emphasises on the examination of the feelings of the user, the background, the psychology, and how these personas based on stories come to life. Again, the advocate is Lene Nielsen, and you can see a sample persona below.

- **Fictional personas** comes from the UI/UX design team experience and is build on the team's past experience from other projects.

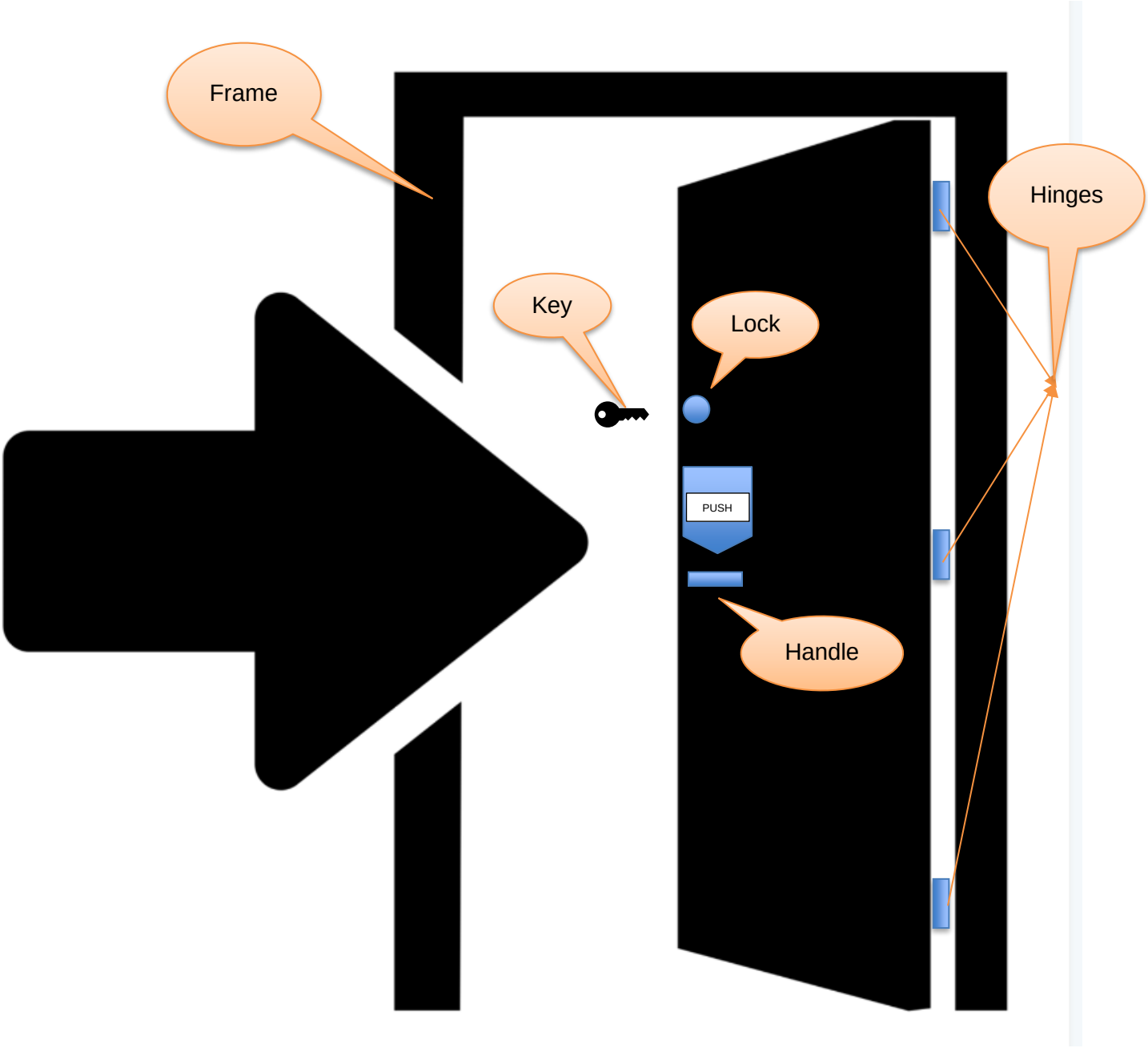
Stakeholders and their personas:

- The client who pays for the door design /sponsor – goal-oriented persona – key stakeholder
- Project manager – role-based persona – key stakeholder
- Users – key stakeholder
- Design team – fictional persona
- Suppliers

10 Steps to Personas



Lene Nielsen 10 steps to personas



Door Design

5. What is Angular, and how does it differ from React? *You may need to conduct independent research and learning for this* (10 marks)

Answer

Angular was created by Google in 2010 and the latest version, Angular 2+, was released in 2019. Nowadays, Google, Upwork, and Microsoft Office use its development services. Other companies that use Angular 5 are Udemy, Telegram, YouTube, Nike, PayPal, Freelancer, AWS, Weather, and others.

According to the official documentation from angular.io, Angular is built on TypeScript and is a development platform that includes a component-based framework, libraries for client-server communication, forms management, routing, and others, but also tools to help developers to develop, update, test, and build code. Is great to build scalable web projects, with a great community of developers, content creators and library authors available.

Angular is different from many points of view from React and these differences are covered as follows.

Components in Angular differ from the ones in React and are built using the @Component() decorator in a TypeScript class, an HTML template, and specific styles. The decorator contains information specific to Angular, such as:

- CSS selector that describes the way a component is utilised in a template. All HTML elements from the template that match this selector convert into its instances.
- A template (HTML) that specifies the component's rendering.
- Optional CSS styles for the HTML elements defined in the template.

Example Hello World Component from Angular's official documentation:

```
import { Component } from '@angular/core';

@Component({
  selector: 'hello-world',
  template: `
    <h2>Hello World</h2>
    <p>This is my first component!</p>
  `
})
export class HelloWorldComponent {
  // The code in this class drives the component's behavior.
}
```

The following template will run the component:

```
<hello-world></hello-world>
```

Once the component is rendered, DOM looks like this:

```
<hello-world>
    <h2>Hello World</h2>
    <p>This is my first component!</p>
</hello-world>
```

To create an Angular app, we need to run the command "**ng new angular-app-name**". Once the app is created, we will see lots of packages installed, like React. To use it, we need to cd the name of the app and run the server with the command "**ng serve --open**". AppComponent controls the application and the page that we see is called Application Shell. Components are the fundament of any Angular

app and they are used to display data, listen to user input, and act based on the input. AppComponent comes as a set of three files:

- app.component.ts– the component's class code, written in TypeScript.
- app.component.html– the component's template, written in HTML.
- app.component.css– the component's private CSS styles.

Every time we create a new component, this new component is created with the same files as AppComponent. If we are looking at the Angular given example **Tour of Heroes app,** we see the following:

- app.component.ts (class title property) → title = 'Tour of Heroes';
- app.component.html (template) → <h1> {{title}} </h1>; where {{}} is Angular's interpolation binding syntax
- src/styles.css - general styles for the application

To create a new component, we use the command "**ng generate component heroes**". The CLI creates a new folder, src/app/heroes/, and generates the three files of the HeroesComponent along with a test file. A component needs to be exported always. Angular apps require the use of interface, which is created like in the example below.

```
export interface Hero {
id: number;
name: string;
}
```

*ngIf adds a condition to a section of the code which can be shown or hidden based on the condition and recreates or removes that section of the DOM.

```
<section *ngIf="showSection">
```

*ngFor works like a for loop but is used to create views and instantiate them for each element from a list and transforms the elements and their content into templates.

```
<li *ngFor="let item of list">
```

Besides these, we also have [ngSwitch], which switches the content based on a condition that is true in an <ng-template>, [ngClass] binds CSS classes to the element based on mapping the truthiness of the map values, [ngStyle] used for styling HTML elements.

We can format the code using pipe, for example we can write the name with uppercase like this:

```
<h2>{{hero.name | uppercase}} Details</h2>
```

ngOnInit() is Angular's lifecycle hook. This is called as soon as a component is created and is the place where all the logic is initiated.

[(ngModel)] is Angular's two-way data binding syntax.

```
<div>
  <label for="name">Hero name: </label>
  <input id="name" [(ngModel)]="hero.name"
  placeholder="name">
</div>
```

In this example [(ngModel)] binds the hero.name property can be bound to HTML textbox to flow data both ways: from hero.name property to textbox and from textbox to hero.name property. [(ngModel)] needs to be imported even though it is a valid directive as is not available by default. To use it, we must opt-in to the FormsModule:

```
app.module.ts (FormsModule symbol import) ----> import { FormsModule } from '@angular/forms';
// <-- NgModel lives here

imports: [
  BrowserModule,
  FormsModule
],
```

Other advantages for using Angular:

- Uses RXJS (Reactive Extensions for JavaScript) library to create observables and which helps the management of the asynchronous code; HttpClient launch, and fast compilation which is less than 3 seconds.
- Great documentation, however it takes a long time to learn.
- Two-way data binding.
- MVVM (Model-View-View-Model) allows developers to work on the same section, with the same data, separately.
- Dependency injection allows the use of external services and is a design pattern that is created upon a class instantiation.

Its disadvantages include the use of TypeScript from its initial version and its complexity, but also the issues related to migration to a newer version.

6. Please describe Redux in as much detail - especially consider: *why would someone use it? What is it? What's the benefit of using it? Are there any potential drawbacks to using it? How can it be added to a project? What is dispatch, provider, actions, etc?*(15 marks)

Answer

Redux official documentation defines Redux as being "a Predictable State Container for JS Apps" and is a standalone library. It works well with a multitude of libraires for example Angular, React, Ember, Vue, and vanilla JS, and is only 2kB including addons. Benefits of using it when writing applications includes:

- Applications run in different environments (native, client, and server), **behave consistently**, and are easy to test.
- By **centralizing** your application's logic and state, you can take advantage of powerful features such as undo/redo, persistence, state, and many others.
- It comes with a Redux DevTools that helps to **debug** the application and keeps a trace of state changes such as why, when, how, and where it changed.

- Redux benefits of a multitude of addons and is compatible with any UI layer; in this way it brings a great **flexibility** to the application.
- Includes **utilities** such as reducers, store setup, create slicing of state, immutable update logic, which simplifies countless use cases.
- Redux-Team created additional libraries such as **Redux Toolkit** used for Redux development, and the React binding library **React-Redux**.
- It helps to **optimize** the code and improve **performance** by re-rendering the component only if there was a change.

How can it be added to a project?

- We can add Redux Toolkit to our project

```
# NPM
npm install @reduxjs/toolkit

# Yarn
yarn add @reduxjs/toolkit
```

- We can create a new Redux app in React by installing one of the official templates that use JavaScript or TypeScript

```
# Redux + Plain JS template
npx create-react-app my-app --template redux

# Redux + TypeScript template
npx create-react-app my-app --template redux-typescript
```

- We can add the core library

```
# NPM
npm install redux

# Yarn
yarn add redux
```

What is dispatch, provider, actions, etc?

The app stores in an object tree, in a single store, the global state of the app. To be able to modify the change, we need to create an **action** which is a plain object with a **type** field that describes what happened, and to **dispatch** this action to the store. Dispatch is a method that belongs to the store, and which updates the state by passing the action object. It behaves like an event trigger based on actions. **Reducers** are pure functions that get the current state value and the action object and returns a new state value based on them. A reducer looks like this: (state, action) => newState. **Selectors** are functions used to retrieve data from a store state value and is useful in big applications as it reduces the need of repeating logic for parts of code that requires the same data.

The state needs to contain only JS primitives, arrays, and objects. The state object (the root) should not be mutated. In case of changes in an object state, we need to return a new object. We can also

add any conditional logic to a reducer. Please see the example below from the official Redux documentation. The switch conditional is optional.

```
import { createStore } from 'redux'


// The following code is the reducer:


function counterReducer(state = { value: 0 }, action) {
  switch (action.type) {
    case 'counter/incremented':
      return { value: state.value + 1 }
    case 'counter/decremented':
      return { value: state.value - 1 }
    default:
      return state
  }
}

// "store" is created and used for the state of the app, and its API is {subscribe, dispatch, getState}.


let store = createStore(counterReducer)

// subscribe() is an alternative to React Redux binding library and is updating the app's UI in case there are any changes in the state of the component.

store.subscribe(() => console.log(store.getState()))



// we use dispatch on an action if we want to mutate the internal state. Actions can be logged, serialized, or stored for replays.

store.dispatch({ type: 'counter/incremented' })
// {value: 1}
store.dispatch({ type: 'counter/incremented' })
// {value: 2}
store.dispatch({ type: 'counter/decremented' })
// {value: 1}
```

Same code from above can be re-written using Redux Toolkit and looks this way:

```
import { createSlice, configureStore } from '@reduxjs/toolkit'

const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0
  },
  reducers: {
    incremented: state => {
      // Redux Toolkit allows us to write "mutating" logic in reducers. It
      // doesn't actually mutate the state because it uses the Immer library,
      // which detects changes to a "draft state" and produces a brand new
      // immutable state based off those changes
      state.value += 1
    },
    decremented: state => {
      state.value -= 1
    }
  }
})

export const { incremented, decremented } = counterSlice.actions

const store = configureStore({
  reducer: counterSlice.reducer
})

// Can still subscribe to the store
store.subscribe(() => console.log(store.getState()))

// Still pass action objects to `dispatch`, but they're created for us
store.dispatch(incremented())
// {value: 1}
store.dispatch(incremented())
// {value: 2}
store.dispatch(decremented())
// {value: 1}
```

Provider is a component that needs to be imported and is used to pass the store as an attribute. This way, we avoid using the store as props in each component. By using it, we have clean code, our code is more readable, and we save time.

```
//index.js

import { Provider } from "react-redux"

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider >,
  document.getElementById('root')
)
```

Are there any potential drawbacks to using it?

There are also a few disadvantages that during the years developers have pinpointed them:

- Redux does not provide data encapsulation which leads to security issues in an application because any component can access it.
- Has limited use and is quite rigid. Is most suited for trivial applications where state management will be too complex without it.
- Since states are immutable, reducer needs to return the updated state on every change which can lead to excessive memory usage in the long run.
- Many developers find it hard to learn and to understand it.

7. Please describe Linux in as much detail as possible (feel free to use notes made during lessons or draw from the lesson directly!). Especially consider: *what is its history? Why would someone use it over other existing operating systems? How does Windows and Mac OSX differ to Linux? How does Linux function, what are some unique features to it? How can it be installed today?*(10 marks)

Answer

UNIX was one of the first operating systems created in 1969 and released in 1970, that allowed people to use computers in a facile way comparative to others such as "Multics" created by AT&T, which was its predecessor. Onyx Systems and Sun Microsystems started selling in 1980 Unix based workstations. In 1983 GNU's Not Unix (GNU) project took life and it was created by Richard Stallman. The scope of it was to create a free system like UNIX, however the GNU kernel Hurd didn't attract enough resources and by 1990 it remained incomplete.

In 1991, whilst he was still studying Computer Science at University of Helsinki, Linus Torvalds (21 years old at the time) started working on what's to become the Linux kernel. He wanted to create this as he mentions in his book "Just for fun" as an independent program from an OS (Operating System), specifically for his hardware and his 80386 processor from his new PC. To achieve this, he used the GNU C Compiler and developed it on MINIX.

Torvalds named the files "Freax" for about six months, until the autumn of 1991 when Ari Lemmke, a volunteer administrator and student at the Helsinki University of Technology, named the project LINUX on the FTP server where they were uploading it, without Torvalds' knowledge. Torvalds agreed

to the new name and the new free operating system grew in popularity and became the new GNU/Linux (because it integrated GNU components).

In 1994, one of the versions was integrated in the Debian project and the project was named Debian GNU/Linux. People continued to use the name Linux to reflect the combination between GNU and Debian. There were many alternate versions and the nowadays most popular one is Ubuntu, overtaking UNIX in terms of preferences. Other Linux options are Fedora and Mint.

Nowadays there are three operating systems: Linux, MAC, and Windows. We can see the following differences between Linux and Windows and MAC:

- Apple MAC OSX, developed by Apple (1984), and with a GUI and special kernel, shares similar features with Linux as they have both evolved from UNIX. It focuses on GUI (graphic user interface) and was intended for Macintosh systems. Windows OS (1985) was developed my Microsoft to overcome MS-DOS limitations. Linux is great for multi-tasking operations and offers full memory protection.
- Linux saves all drives in a single tree form and the file structure is completely different from Windows and MAC. Windows has files, folders, logical drives, and cabinet drawers, and can store data in various extensions, whilst MAC OS X uses directories.
- In terms of registry, Linux doesn't have one and all settings of the app are saved hierarchy, under users, without a centralized database comparative to the other two who store data in database (Windows) or .plist (MAC).
- In Linux is very easy to change interfaces thanks to utilities such as KDE or GNOME. MAC and Windows also offer the possibility to change interface through system interfaces.
- Linux is the least prone to malware and cyberattacks.
- Linux is free, whilst the other two cost money.

Why people use it over the other two operating systems?

- Its open source and we have access to code under GPL (General Public License).
- Customization and diversity
- Facile installation and use
- Security against malware and viruses
- Performance and reliability: Goobuntu – Linux used by Google on their desktops and servers due to its reliability; same with Microsoft Azure
- Great for students and education
- Automation tools and on-line commands through ZSH, BASH, FISH, SH shells.
- Versatility
- Linux USB Live – allows its use from an USB drive and boot up the system. Otherwise, it can be installed and run as a firewall, file server, or web server, without changing the hard drive.

If we want to install Linux on a Windows or MAC operating system, we need to create a partition for it first on the main hard drive. Then, we can select to install it from a USB drive (if we have it saved like that) or download it from Ubuntu official website and follow the instructions on screen. Ubuntu can be installed on a Virtual Machine with VirtualBox, but also on Raspberry Pi 4. Once we download and save an Ubuntu image, we need to add it to a USB stick using one of the tools such as balenaEtcher (compatible to all OS), Rufus (Windows), or Etcher (MAC) that will create the installation media, and click Flash!.

Once this is completed, we can insert the USB drive into the laptop/PC and restart/ boot the device. At this stage we will be given the option to install it or to try it. We will proceed with install Ubuntu, and we will be prompted to select a language for our keyboard. On the next page we are asked to select the type of installation and we will stick to the normal installation option. An internet connection is mandatory for a successful installation and if we are not connected, we are asked to do so at this point. On next screen, we are asked to chose whether Ubuntu will be the only operating system and erase the existing one or install it on the partition created initially. At this stage we can add encryption for security. Our location will be added automatically if we are connected to the internet, otherwise we will be prompted to select it. Next screen will ask us to create our login details and once is done, we can complete the installation and enjoy Ubuntu!

8. What are they, and which is better between Class components and Functional components? Provide a discussion. Consider: *Go deep - how does each one work? What is the unique properties or behaviours to each one? Why would someone use one over the other? What are the advantages and disadvantages of each one? Who benefits from these advantages and disadvantages, who is it suitable for?*(10 marks)

Answer

Functional components are the most preferred and the way forward in React. They are JavaScript functions that are easier to learn and to understand, and the syntax is easier to grasp than class components. These are written as normal JS functions and return an JSX element. Hooks like useState add a state to component.

```
JS FunctionalComponent.js > ...
import React, { useState } from "react";

const FunctionalComponent = () => {
  const [count, setCount] = useState(0);

  const increase = () => {
    setCount(count + 1);
  };

  return (
    <div style={{ margin: "50px" }}>
      <h2>This is a Functional Component rendered here </h2>
      <h3>Counter App using Functional Component : </h3>
      <h2>{count}</h2>
      <button onClick={increase}>Add</button>
    </div>
  );
};

export default FunctionalComponent;
```

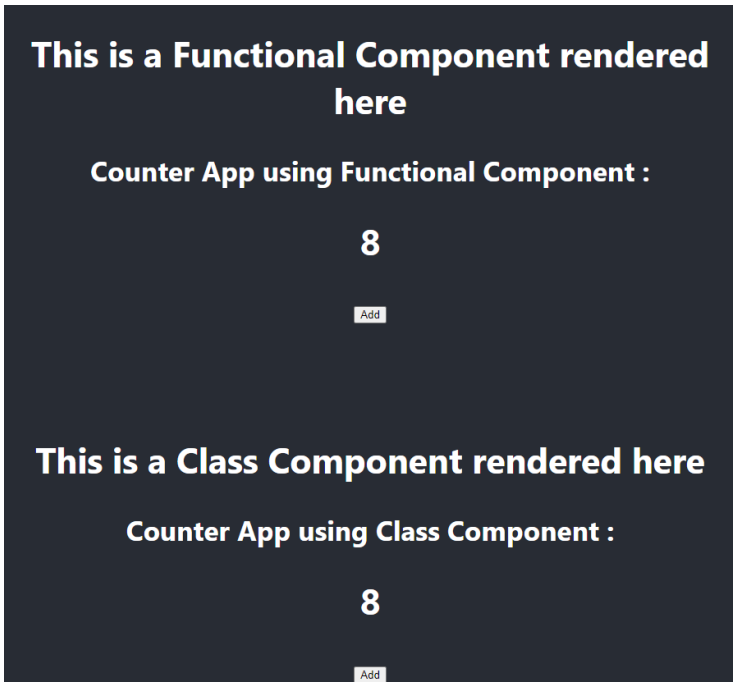
Functional component

React started using **class components** and there are still many applications written that way. We can add many methods to our classes for enhanced functionality. It extends React.Component and requires a constructor() with a super() method for inheritance, and a render() method. Also, we need to use "this" keyword inside a class and data binding to a state so we can use events handlers.

```
ClassComponent.js > default
1  import React from "react";
2
3  class ClassComponent extends React.Component {
4    constructor() {
5      super();
6      this.state = {
7        count: 0,
8      };
9      this.increase = this.increase.bind(this);
10   }
11
12   increase() {
13     this.setState({ count: this.state.count + 1 });
14   }
15
16   render() {
17     return (
18       <div style={{ margin: "50px" }}>
19         <h2>This is a Class Component rendered here </h2>
20         <h3>Counter App using Class Component : </h3>
21         <h2> {this.state.count}</h2>
22         <button onClick={this.increase}> Add</button>
23       </div>
24     );
25   }
26 }
27
28 export default ClassComponent;
```

Class component

Example rendering of the components:



Previously, we could manage the state of a component only if this was a class component. Now, with the React 16.8 moving forward, Hooks were added to it which allows an easy manipulation of component states through useState hook.

Comparison between functional and class components/ advantages and disadvantages:

Functional components	Class components
A pure function that returns a JSX element and takes props as arguments. It has no render() method.	Necessitates to extend from React.Component, must have a render() method and returns a React element.
A functional component runs when is called, from top to bottom. Once the return method is reached, the function dies.	This is instantiated and kept alive through various lifecycles methods.
Are also called Stateless components because their focus is rendering UI, and data passed to them is used and displayed as is.	Also called Stateful components because they can add state and logic to data.
Lifecycle methods are not supported by functional components. We can use useEffect hook instead to achieve the same functionality, however it can get confusing.	They use Lifecycle methods to manipulate data and states. (for example ComponentDidMount())

We can add Hooks to make them stateful.
`const [count, setCount] = useState(0);`
	To implement hooks, it needs to write a different syntax in the constructor.
	`this.state = {`
	`  count: 0,`
	`};`

Functional component does not require the use of "this" keyword, and it doesn't have a constructor.	Uses "this" keyword and requires a constructor to store state.
Functional components use props.name.	They uses this.props.name of the props.
props.children is used to access the children's content for a component	this.props.children is used for class components to access the children's content.
Higher order components (HOC) are functions that return a new component based on a component.	Syntax for HOC used with a class component is complex and can get confusing.
Functional components do not have Error Boundaries to manage errors.	Class components can manage errors with Error Boundaries created with ComponentDidCatch() → like catch{} block in JS, and can use getDerivedStateFromError() to re-render in case the UI has a fallback.

In conclusion, both functional and class components have advantages and disadvantages, and both can be used in a project. The choice is based on what should that component do, the complexity of the application, Error Boundary necessity, state management, and lifecycle methods/ hooks. The standard recommendation is to start learning React with class components and as soon as there is a good understanding of them, we can move to the functional components.