Oppgaver labøving - (1. September) - TypeScript

Under finner du 15 oppgaver som omhandler TypeScript. Noen av oppgavene krever at det runnes i et React Native prosjekt, mens andre kan kjøres på vanlig måte med node. Det er 5 enkle, 5 medium, og 5 vanskelige oppgaver. Løsninger vil bli vist å labøvings-timen, men vil også bli lagt ut etter vi er ferdige. Anbefaler alle å møte opp, da forklaringer på løsnigene ikke vil bli lagt ut. Du kan bare begynne nå, men det er disse oppgavene som vil bli gjort på labøvingen.

Enkle Oppgaver:

Oppgave 1: Type Inference and Interfaces

Definer et `interface` eller `type` Student med egenskapene `name` (string), `age` (nuimber) og `grade` (string). Opprett en `array` med Student-objekter. Skriv en funksjon som tar et Student-objekt som parameter og logger detaljene med `console.log`.

```
const students: Student[] = [
  { name: "Alice", age: 20, grade: "A" },
  { name: "Bob", age: 22, grade: "B" },
];

// Prints:
// Alice er 20 gammel. Karakter: A
// Bob er 22 gammel. Karakter: B
```

Oppgave 2: Union and Intersection typer

Definer typene `Triangle` og `Rectangle` som representerer figurer med egenskapene `type` og `width`, og `height`. Opprett en `array` som kun inneholder typen `Shape` og beregn det totale arealet basert på alle figurene i listen.

```
const shapes: Shape[] = [
  { type: "triangle", width: 5, height: 8 },
  { type: "rectangle", width: 4, height: 6 },
];

// Prints: Total area: 44
```

Oppgave 3: Enum and Function Signatures

Opprett en enum `ButtonType` med verdiene `Primary`, `Secondary` og `Danger`. Skriv en funksjon `createButton` som genererer en html `button` basert på typen og textinnhold som blir sendt som argumenter. Her kan du bare printe HTML-innholdet som en string.

```
createButton(ButtonType.Primary, "Click me");

// Prints: <button class="primary">Click me</button>
```

Oppgave 4: Optional and Default parametre

Implementer en funksjon som beregner arealet til en rektangel. Utvid funksjonen til å akseptere valgfrie parametere for bredden og høyden til rektangelet. Hvis ikke gitt, bruk standardverdier på 5 for begge dimensjoner.

Utvid denne funksjonen til å støtte sirkler også. Du trenger ikke definere typer, men du er velkommen til å gjøre det. Du må derimot klare å differensiere mellom rektangler og sirkler.

```
console.log("Rectangle area:", calculateArea("rectangle", 4, 6));
console.log("Circle area:", calculateArea("circle", 5));

// Prints:
// Rectangle area: 24
// Circle area: 78.53981633974483
```

Oppgave 5: Type guards and Discriminated unions

Definer en unionstype `MyResponse` med egenskapene `status` (number) og `data` (string eller number) for å representere API-responser. Skriv en funksjon `handleResponse`som tar et `MyResponse`-objekt og logger innholdet basert på statusen. Du kan se på alt som ikke gir `status = 200` som en error.

Utvid funksjonen til å bruke `fetch` for å hente data fra et API. F.eks. pokeapi.co.

```
const successResponse: MyResponse = {
  status: 200,
  data: "Data fetched successfully",
};
const errorResponse: MyResponse = { status: 400, data: 432233 };

handleResponse(successResponse);
handleResponse(errorResponse);

// Prints:
// Success: Data fetched successfully
// Error: 432233
```

Medium Oppgaver:

Oppgave 1: Advanced Type Mapping

Opprett typene `PartialProps` og `RequiredProps` som endrer optional og required egenskaper i et objekt. Bruk disse typene på en interface og logg de resulterende typene.

Oppgave 2: Type Inferens

Definer en type `Product` med egenskapene `name` (string), `price` (number), og `description` (string). Opprett et objekt av typen `Product` uten `description`, og uten å spesifisere typen eksplisit. Skriv en funksjon som tar en `array` av produkter og logger detaljene. Hvordan kan du få denne funksjonen til å kjøre uten å endre typen?

Oppgave 3: Avanserte Funksjonssignaturer

Definer en type `FilterFunction` for en funksjon som filtrerer en `array` basert på resultattypen til en callback. Implementer en generisk funksjon `filterArray` ved hjelp av denne typen. Test funksjonen med ulike datatyper.

Oppgave 4: Avansert React Native Komponent

Design en egendefinert knapp i React Native (bruk `TouchableOpacity`) som støtter ulike stiler basert på egenskaper som størrelse (size), farge (color) og deaktivert tilstand (isDisabled). Knappen bør også ha tekst (label) og bør utføre en funksjon (onPress) når den trykkes på. Implementer komponenten ved hjelp av TypeScript-grensesnitt.

Oppgave 5: Type-Sikker context i React Native (Medium ++)

Opprett en context provider i Native med TypeScript-støtte. Definer kontekst-typer og interfaces for consumer components. Implementer en test av dette for å demonstrere type-sikker kommunikasjon. Stikkord: `AuthProvider`, `useAuth`, `AuthContext`.

Målet er å late som vi har fått informasjon om en bruker som har logget inn, og at vi skal kunne bruke denne informasjonen i andre komponenter. Kult om du bruker denne funksjonaliteten for å vise navnet på din testbruker når du klikker på knappen du lagde i oppgave 4.

Vanskelige Oppgaver:

Oppgave 1: Rekursive types

Definer en type `NestedArray` som representerer en `array` med potensielt uendelig dype nested arrays. Opprett en `array` av denne typen med flere nivåer av

nesting. Implementer en funksjon som flater ut den innebygde array-strukturen. Det vil si: `[1, [2, [3, 4], 5], 6]` blir til `[1, 2, 3, 4, 5, 6]`.

```
console.log(flattenArray([1, [2, [3, 4], 5], 6]));

// Prints: [1, 2, 3, 4, 5, 6]
```

Oppgave 2: Avanserte Contitional types

Opprett en betinget type `UnwrapArray` som pakker ut innebygde arrays. Bruk den på et `NestedArray`-object og logg den resulterende verdien.

Oppgave 3: Rekursive typer

I denne oppgaven skal du fokusere på å implementere en datastruktur som ligner på en trestruktur, samt en funksjon for å traversere den.

Definer en `interface` eller `type TreeNode` som representerer en node i et tre. Hver node skal ha en `value` av hvilken som helst type og en array med child-nodes (`TreeNode`) kalt `children`. Lag et objekt av typen `TreeNode` med flere nivåer av nesting. Lag en funksjon `traverseTree` som tar et `TreeNode`-objekt som bruker en variant av BFS for å traversere treet og returnere en `array` med alle verdiene i treet.

```
const rootNode: TreeNode<number> = {
  value: 1,
  children: [
    {
      value: 2,
      children: [
        { value: 3, children: [] },
        { value: 4, children: [] },
      ],
    },
    {
      value: 5,
      children: [],
    },
  ],
};

console.log(traverseTree(rootNode));

// Prints: [1, 2, 3, 4, 5]
```

Oppgave 4: Avanserte HOCs (Higher Order Components)

Design en høyere ordens komponent (HOC) som utvider en komponent med ekstra egenskaper og styling. Bruk TypeScript-generics for å sikre typenøyaktighet. Test ut og vis den ekstra funksjonaliteten.

Eksempel: Lag en HOC som legger til en prop `title` med en tekstverdi. Denne teksten skal vises i en `<Text>`-komponent som en overskrift over den innkapslede komponenten.

Oppgave 5: Avanserte Egendefinerte Hooks

Opprett en egendefinert hook som håndterer henting av data fra et API. Hooken skal ta en URL som parameter og returnere en type `ApiResponse` med egenskapene `data`, `isLoading` og `error`. Hvis det oppstår en feil skal `error`-egenskapen inneholde feilmeldingen, ellers skal `data`-egenskapen inneholde dataene som ble hentet fra API-et.

Du kan igjen bruke pokeapi.co for å teste ut funksjonaliteten - eller et annet API du ønsker.