Systemutveckling Ramverk

25 HP

Kursupplägget

Föreläsning			Innehåll (FM)	Övningar (EM)	Inlämning
1	mån 25 mars	D	Introduktion Typescript (Basic Types, Type Inferrence, Variable Declarations, Iterators and Generators)	TypeScripts Hemsida TypeScript in 5 minutes + övning	
2	ons 27 mars	V	Typescript forts. (Functions, Classes, Interfaces, Modules)		
3	fre 29 mars	V	Introduktion React (SPA, Virtuell DOM, Kap. 1-6)	Reacts Hemsida Main Concepts inklusive övningar i CodePen Kodövning (RP)	
4	mån 1 apr.	D	React Playground (1-initial-setup + 2-layout)		Inlämning 1 ges ut
5	ons 3 apr.	V	React forts. (Kap. 7-12)		
6	fre 5 apr.	V	Create React App TS Intro	Övning ”Todo App”	Handledning
7	tis 9 apr.	D	React Playground (3-navigation-with-state) React Playground (4-navigation-with-routes)	Kodövning (RP)	
8	ons 10 apr.	V	React Playground (5-code-splitting) React Playground (6-error-boundary)	Kodövning (RP)	Inlämning 1 lämnas in
9	fre 12 apr.	V	React Playground (7-portals)	Kodövning (RP)	Inlämning 2 ges ut
10	mån 15 apr.	D	React Playground (8-code-split-app-start)	Kodövning (RP)	
11	tis 16 apr.	D	React Playground (9-api-lib-axios)	Kodövning (RP)	Handledning
12	tis 23 apr.	D	React Playground (10-context)	Kodövning (RP)	
13	tors 25 apr.	V	Tentaplugg		Inlämning 2 lämnas in
14	fre 26 apr.	V	TENTAMEN		

En liten påminnelse

Ni måste själva utanför föreläsningstiden läsa dokumentationen och göra tutorials för att kunna hänga med på föreläsningarna.

Vi kommer ha en relativt högt tempo och hamnar ni efter så kommer det bli tufft för er.

TypeScript

JavaScript that scales.

TypeScript is a typed superset of JavaScript that compiles to plain Javascript.

Any browser. Any host. Any OS. Open spurce.

https://github.com/Microsoft/TypeScript

Funktioner, klasser, interfaces, moduler & övning!

Functions

TypeScript - Functions

Att typa en funktion

Icke typad funktion (JS)

```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

Sparar funktionen i en variabel

Typen som returneras

Typad funktion (TS)

```
function add(x: number, y: number): number {
    return x + y;
}

let myAdd = function(x: number, y: number): number { return x + y; };
```

Parametrars typer

TypeScript - Functions

Bra att veta

Genom ”Type Inference” har Typescript, i många fall, koll på typen en funktion skall returnera

Deklarerade variabler utanför funktionens scope är åtkomstbara.
Bättre att typa variabler som inputparameter för att öka tydlighet i vad funktionen skall utföra.

TypeScript - Functions

Att skriva en funktionstyp

myAdd är en funktion som får in två parametrar av typen number och returnerar ett number.

```
let myAdd: (x: number, y: number) => number =
    function(x: number, y: number): number { return x + y; };
```

Det räcker med typningar, namnen behöver inte stämma överrens (kan dock vara tydligare om namnen stämmer överrens).

```
let myAdd: (baseValue: number, increment: number) => number =
    function(x: number, y: number): number { return x + y; };
```

TypeScript - Functions

Parametrar

I javascript är alla parametrar alltid frivilliga.

En typad funktion måste alltid få in de parametrar den förväntar sig, annars kastas error.

```
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result3 = buildName("Bob", "Adams");         // ah, just right
```

TypeScript - Functions

Parametrar

Vet man inte om en parameter skall skickas in i funktionen eller inte, kan detta anges som en ”Optional parameter”.

Dessa måste alltid läggas som ”sista parametrar” in i funktionen med ett ’?’ efter variabelns namn. Om variabeln inte skickas in blir den *undefined*.

Koll om parametern finns måste göras →

```
function buildName(firstName: string, lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}

let result1 = buildName("Bob");                  // works correctly now
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result3 = buildName("Bob", "Adams");         // ah, just right
```

Optional parameter

TypeScript - Functions

Parametrar

En default-initialized parameter kommer ges ett default värde om den inte skickas in i funktionen.

En default-initialized parameter måste inte deklareras efter samtliga *required* parametrar, däremot måste ett värde skickas in. Är detta värde, i nedanstående exempel, *undefined* kommer variabeln i funktionen bli ”Smith”.

```
function buildName(firstName: string, lastName = "Smith") {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");                  // works correctly now, returns "Bob
 Smith"
let result2 = buildName("Bob", undefined);       // still works, also returns "Bob Smi
th"
let result3 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result4 = buildName("Bob", "Adams");         // ah, just right
```

Default-initialized

TypeScript - Functions

Parametrar

Required, optional och default parameters har en sak gemensamt, alla syftar till en variabel åt gången.

Vill man hantera flera parametrar på samma gång kan Rest Parameters användas. I fallet nedan kommer …restOfName bli en array av ”Samuel”, ”Lucas”, och ”MacKinzie”…

Rest Parameters

↓

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

TypeScript - Functions

Parametrar

Det går även att typa Rest Parameters när en funktion typas.

Rest Parameters

↓

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;
```

↑

Rest Parameters

Classes

TypeScript - Klasser

En simple klass

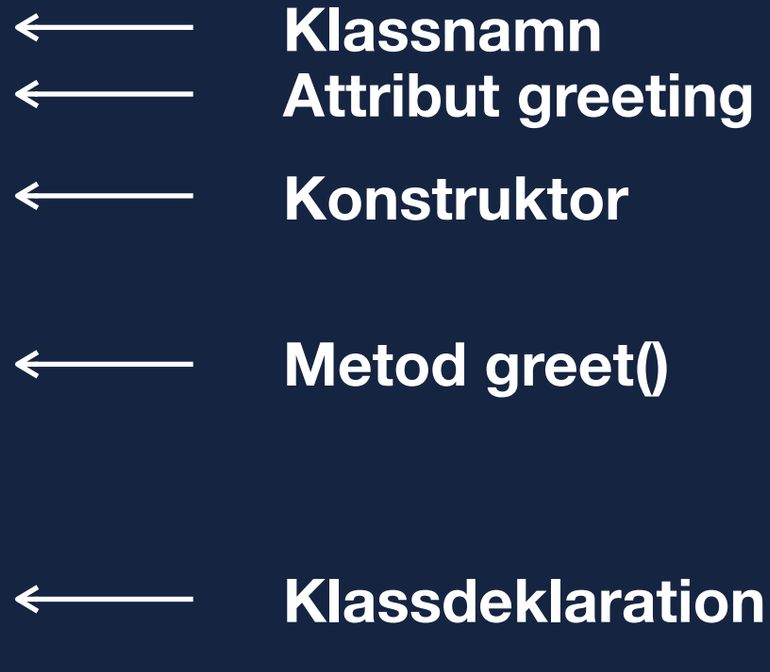
En klass består av attribut och metoder.

Konstruktion är klassens första kod som körs då den deklareras.

Inputparametrar till klassen tas emot i konstruktion.

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}

let greeter = new Greeter("world");
```



TypeScript - Klasser

Arv

Om en klass skall ärva (vara en subklass till en annan superklass) anges detta med keywordet *extends*. Detta betyder i detta exemplet att klassen *Dog* kommer få allt som klassen *Animal* har.

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved ${distanceInMeters}m.`);
    }
}

class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');
    }
}

const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

⟵ **Klassnamn**

⟵ **Metod move()**

⟵ **Arv av Animal**

⟵ **Metod bark()**

⟵ **Instansierar och anrop till klassen Dog (och indirekt klassen Animal)**

TypeScript - Klasser

Arv

Om en arv görs där superklassen har en konstruktor måste man i subklassens konstruktor kalla på metoden *super().* Detta för att köra superklassens konstruktor. I exemplet nedan kallas *super()* men en sträng som inputparameter. Denna parameter är det som tas emot i *Animals* konstruktor.

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}
```

← Superklassens konstruktor

← Kalla på super()

TypeScript - Klasser

Kontext för *this* och *super*

Keywordet *this* är en representation av aktuell kontext och *super* är kontexten för en superklass.

```
class Animal {
    name: string;
    constructor(theName: string) { this.name = theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}
```

this.name är den egna klassens attribut (*this* är här klassens kontext)

Kallar på en metod i superklassen (även då båda heter move)

TypeScript - Klasser

Public, Private & Protected

Public betyder att attributet eller metoden är fullt tillgänglig hos andra klasser.

Alla attribut en klass har är *Public* per default.

Det går att explicit deklarera ett attribut eller metod som *Public*, detta illustreras i exemplet nedan

```
class Animal {
    public name: string;
    public constructor(theName: string) { this.name = theName; }
    public move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }
}
```

TypeScript - Klasser

Public, Private & Protected

Private betyder att attributet eller metoden endast är tillgänglig inom kontexten för den aktuella klassen (genom *this*).

Attribut eller metod som skall vara *Private* måste deklareras som *Private*.

Vill man ha tillgång till ett attribut som är *Private* utifrån klassen attributet existerar bör en metod deklareras för detta som är *Public* (ex. Public getName()) som returnerar *this.name.*

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name = theName; }
}

new Animal("Cat").name; // Error: 'name' is private;
```

→ Instansen av *Animal* har inte tillgång till attributet *name* i klassen *Animal*

←

TypeScript - Klasser

Public, Private & Protected

Protected betyder att attributet eller metoden endast är tillgänglig inom kontexten för den aktuella klassen samt och dess subklasser.

Attribut eller metod som skall vara *Protected* måste deklareras som *Protected*.

ERROR

Attributet *name* i *Person* är tillgänglig ifrån *Employee* (*this.name*)

OK

Attributet *name* i *Person* är inte tillgängligt utanför klassen *Person* eller *Employee*

```
class Person {
    protected name: string;
    constructor(name: string) { this.name = name; }
}

class Employee extends Person {
    private department: string;

    constructor(name: string, department: string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is ${this.name} and I work in ${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error
```

TypeScript - Klasser

Abstact

En *Abstract* klass kan inte instansieras utan endast användas som superklass till andra klasser.

En *Abstract* funktion kan måste, i subklassen, deklareras. En *Abstract* funktion i en *Abstract* klass existerar endast i syfte att tvinga subklasser till specifikt innehåll.

Abstract printMeeting() måste måste definieras i subklasser till klassen Department.

```
abstract class Department {

    constructor(public name: string) {
    }

    printName(): void {
        console.log("Department name: " + this.name);
    }

    abstract printMeeting(): void; // must be implemented in derived classes
}

class AccountingDepartment extends Department {

    constructor() {
        super("Accounting and Auditing"); // constructors in derived classes must call
super()
    }

    printMeeting(): void {
        console.log("The Accounting Department meets each Monday at 10am.");
    }

    generateReports(): void {
        console.log("Generating accounting reports...");
    }
}
```

TypeScript - Klasser

Readonly

Readonly är ett keyword som endast kan appliceras på attribut.

Readonly anger att attributet det står framför endast får läsas och inte ändras (kan jämföras med varibeltypen *const*).

Attribut som är *Readonly* får endast bli tilldelat ett värde i deklarationen eller i klassens konstruktor.

BRA

INTE BRA

```
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong legs");
dad.name = "Man with the 3-piece suit"; // error! name is readonly.
```

Interfaces

TypeScript - Interfaces

Interfaces

Ett *interface* efterliknar en klass, dock utan konstruktor och *Modifiers (public, private etc)*.

Interfaces används för att definiera olika typer. Detta i primärt syfte att säkerställa att rätt data finns tillgänglig när den behövs, men också för att skapa struktur och tydlighet i koden.

Exemplen till höger utför samma sak, skillnaden är att det, i det undre exemplet, finns ett *interface* definierat som säger att en property vid namn ’label’ måste definieras då en *LabelledValue* skapas

```
function printLabel(labelledObj: { label: string }) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

```
interface LabelledValue {
    label: string;
}

function printLabel(labelledObj: LabelledValue) {
    console.log(labelledObj.label);
}

let myObj = {size: 10, label: "Size 10 Object"};
printLabel(myObj);
```

TypeScript - Interfaces

Optional properties

Ett *interface* kan på samma sätt som inputparametrar till en funktion ha *Optional* properties. Med detta menas att ett objekt av typen *SquareConfig* nedan inte måste ha en *color* och *width* för att få existera.

Koller på om *color* och *width* finns måste dock göras i *createSquare()*

```
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
    let newSquare = {color: "white", area: 100};
    if (config.clor) {
        // Error: Property 'clor' does not exist on type 'SquareConfig'
        newSquare.color = config.clor;
    }
    if (config.width) {
        newSquare.area = config.width * config.width;
    }
    return newSquare;
}

let mySquare = createSquare({color: "black"});
```

← Stämmer inte med typningen SquareConfig

TypeScript - Interfaces

Readonly properties

En property som är *Readonly*, kan på samma sätt som i klasser, inte modifieras efter de fått ett värde.

Interface Point (i bild 1) har två properties (x & y) som är readonly vilket gör att de i bild 2 inte får lov att modifieras.

I bild 3 ser vi hur en *Readonly* array av numbers kan skapas samt hur den ej får modifieras

1

```
interface Point {
    readonly x: number;
    readonly y: number;
}
```

2

```
let p1: Point = { x: 10, y: 20 };
p1.x = 5; // error!
```

3

```
let a: number[] = [1, 2, 3, 4];
let ro: ReadonlyArray<number> = a;
ro[0] = 12; // error!
ro.push(5); // error!
ro.length = 100; // error!
a = ro; // error!
```

TypeScript - Interfaces

Excess Property Checks

Ett objekt kan ha fler properties än vad som deklararerats i interfacet

```ts
interface User {
    userName?: string;
    age?: number;
}

let mySquare = { lastName: "Pelle", age: 24, role: "Admin" } as User;
```

Om en icke optional (required) property skulle uteslutas kommer error dock kastas.

```ts
interface User {
    userName: string;
    age?: number;
}

let mySquare = { lastName: "Pelle", age: 24, role: "Admin" } as User;
```

TypeScript - Interfaces

Excess Property Checks

Om en funktion förväntar sig ett objekt av typen *SquareConfig* kan man ej skicka in ett objekt med icke matchande properties.

Man kan med hjälp av *type assertion* komma runt detta (*as SquareConfig*).

```ts
interface SquareConfig {
    color?: string;
    width?: number;
}

function createSquare(config: SquareConfig): { color: string; area: number } {
    // ...
}
```

```ts
// error: 'colour' not expected in type 'SquareConfig'
let mySquare = createSquare({ colour: "red", width: 100 });
```

```ts
let mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);
```

TypeScript - Interfaces

Excess Property Checks

VIKTIGT

Skriv rätt *interfaces* och använd inte *type-assertion* eller *any* om ni inte använder externa bibliotek där typer fattas!

TypeScript - Interfaces

Function types

Vi kan även typa en funktion i ett *interface.*

```
interface SearchFunc {
    (source: string, subString: string): boolean;
}
```

Vi kan sedan skapa och deklarera en variabel av typen *SearchFunc.*

```
let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
    let result = source.search(subString);
    return result > -1;
}
```

För att en funktion skall göra en korrekt *type-check* behöver inte namnen på parametrar stämma överrens med typningens namn. Returntypen är även optional här då den redan finns i *interfacet.*

```
let mySearch: SearchFunc;
mySearch = function(src: string, sub: string): boolean {
    let result = src.search(sub);
    return result > -1;
}
```

TypeScript - Interfaces

Function types

En funktion kan även typas som en arrowfunction i ett *interface.*

```
type Sound = "voff" | "ugh" | "aahhg"
type Race = "Golden" | "Shiba" | "Pudel"

interface Dog {
    name: string
    age: number
    race: Race
    makeSound: (sound: Sound) => void
}
```

Skapa ett objekt av typen *Dog* och fastställ vad funktionen *makeSound()* skall göra

```
const myDog: Dog = {
    name: "Julius",
    age: 8,
    race: "Golden",
    makeSound: (sound) => { console.log(sound) }
}
```

TypeScript - Interfaces

Class types

Med hjälp av *implements* kan vi säga att en *klass* skall få alla properties som ett specifikt *interface* innehåller

```ts
interface ClockInterface {
    currentTime: Date;
    setTime(d: Date);
}

class Clock implements ClockInterface {
    currentTime: Date;
    setTime(d: Date) {
        this.currentTime = d;
    }
    constructor(h: number, m: number) { }
}
```

TypeScript - Interfaces

Extending interfaces

Med hjälp av *extends* kan vi säga att ett *interface* skall få alla properties som ett specifikt *interface* innehåller

```
interface Shape {
    color: string;
}

interface Square extends Shape {
    sideLength: number;
}

let square = <Square>{};
square.color = "blue";
square.sideLength = 10;
```

Modules

TypeScript - Modules

Vad är Modules?

En *modul* är en fil som vars innehåll kan tillgängliggöras till annan fil

JavaScript har ett koncept av *modules*, TypeScript delar detta koncept.

En *modul* exekveras inom sitt egna scope, inte i det globala scopet

TypeScript - Modules

Export

Alla deklarationer (såsom variabler, funktioner, klasser, typer, eller interfaces) kan exporteras genom att lägga till keywordet *export*

```ts
export interface StringValidator {
    isAcceptable(s: string): boolean;
}
```

```ts
export const numberRegexp = /^[0-9]+$/;

export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
```

TypeScript - Modules

Export

Export statements möjliggör namnbyte för imports (där koden skall användas)

```
class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
        return s.length === 5 && numberRegexp.test(s);
    }
}
export { ZipCodeValidator };
export { ZipCodeValidator as mainValidator };
```

TypeScript - Modules

Imports

Att importera en exporterad *modul* görs med hjälp av att lägga till keywordet *import*

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

För att ge nya namn till imports används keywordet *as*

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

TypeScript - Modules

Imports

Att importera en exporterad *modul* görs med hjälp av att lägga till keywordet *import*

```
import { ZipCodeValidator } from "./ZipCodeValidator";

let myValidator = new ZipCodeValidator();
```

För att ge nya namn till imports används keywordet *as*

```
import { ZipCodeValidator as ZCV } from "./ZipCodeValidator";
let myValidator = new ZCV();
```

För att importera en hel modul in i en variabel används * och *as*

```
import * as validator from "./ZipCodeValidator";
let myValidator = new validator.ZipCodeValidator();
```

***Default export* följer inte med i *import* *.**

TypeScript - Modules

Default export

Varje modul kan ha **en** *default export*, men måste inte!

Detta anges med hjälp av att ange *default* innan *export*

Allt som kan exporteras kan exporteras som *default*

TypeScript - Modules

Default export

Exempel: **Klasser**

Exportera en klass som *default* →

```
export default class ZipCodeValidator {
    static numberRegexp = /^[0-9]+$/;
    isAcceptable(s: string) {
        return s.length === 5 && ZipCodeValidator.numberRegexp.test(s);
    }
}
```

```
import validator from "./ZipCodeValidator";

let myValidator = new validator();
```

← Importera en *default* exporterad klass

Importeras utan { … }

TypeScript - Modules

Default export

Exempel: **Funktoner**

Exportera en funktion som *default*

→

```
const numberRegexp = /^[0-9]+$/;

export default function (s: string) {
    return s.length === 5 && numberRegexp.test(s);
}
```

```
import validate from "./StaticZipCodeValidator";

let strings = ["Hello", "98052", "101"];

// Use function validate
strings.forEach(s => {
  console.log(`"${s}" ${validate(s) ? " matches" : " does not match"}`);
});
```

← Importera en *default* exporterad klass

Importeras utan { … }

Läsanvisningar

TypeScript Handbook

https://www.typescriptlang.org/docs/home.html

Kapitel.

Functions

Classes

Interfaces

Modules

Nästa lektion

Introduktion React, SPA & Virtuell DOM

Resten av dagen

Övningsuppgift

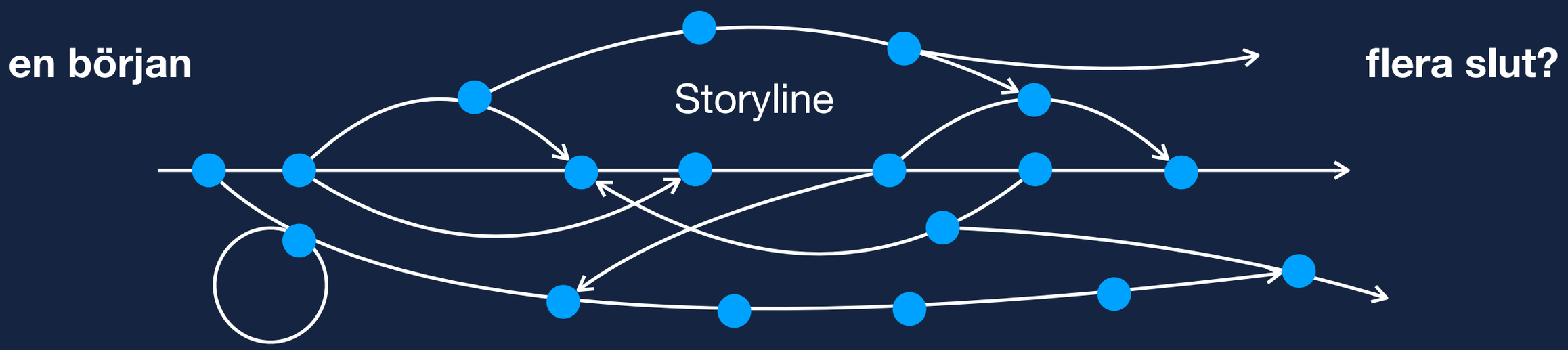
Övningsuppgift (Twisted Stories)

Ni skall i grupp om två eller tre bygga ett textbaserat storyspel där spelaren färdas på välgenomtänka (eller galna) **stigar i er story. För varje punkt i er story skall en text** (och gärna en bild) **presenteras för spelaren och denne ska få ett, två eller tre val. Valen tar spelaren vidare** (eller tillbaka) **på en väg i storyn.**

Börja med att bestämma vad spelet skall handla om och planera sedan gärna ett grovt upplägg av spelet innan ni börjar programmera. Alla spel skall ha ett namn och en början, när spelet tar slut skall man få valet att börja om.

När ni är klara skall minst 3 personer spela erat spel, ni ska själva spela minst 3 andra spel. Därefter ska ni välja ut 2 favoriter. Favoriterna räknas samman för att fastställa de bästa spelen. Lycka till!

Programmera tillsammans med hjälp av VS Code Live Share!



Tack