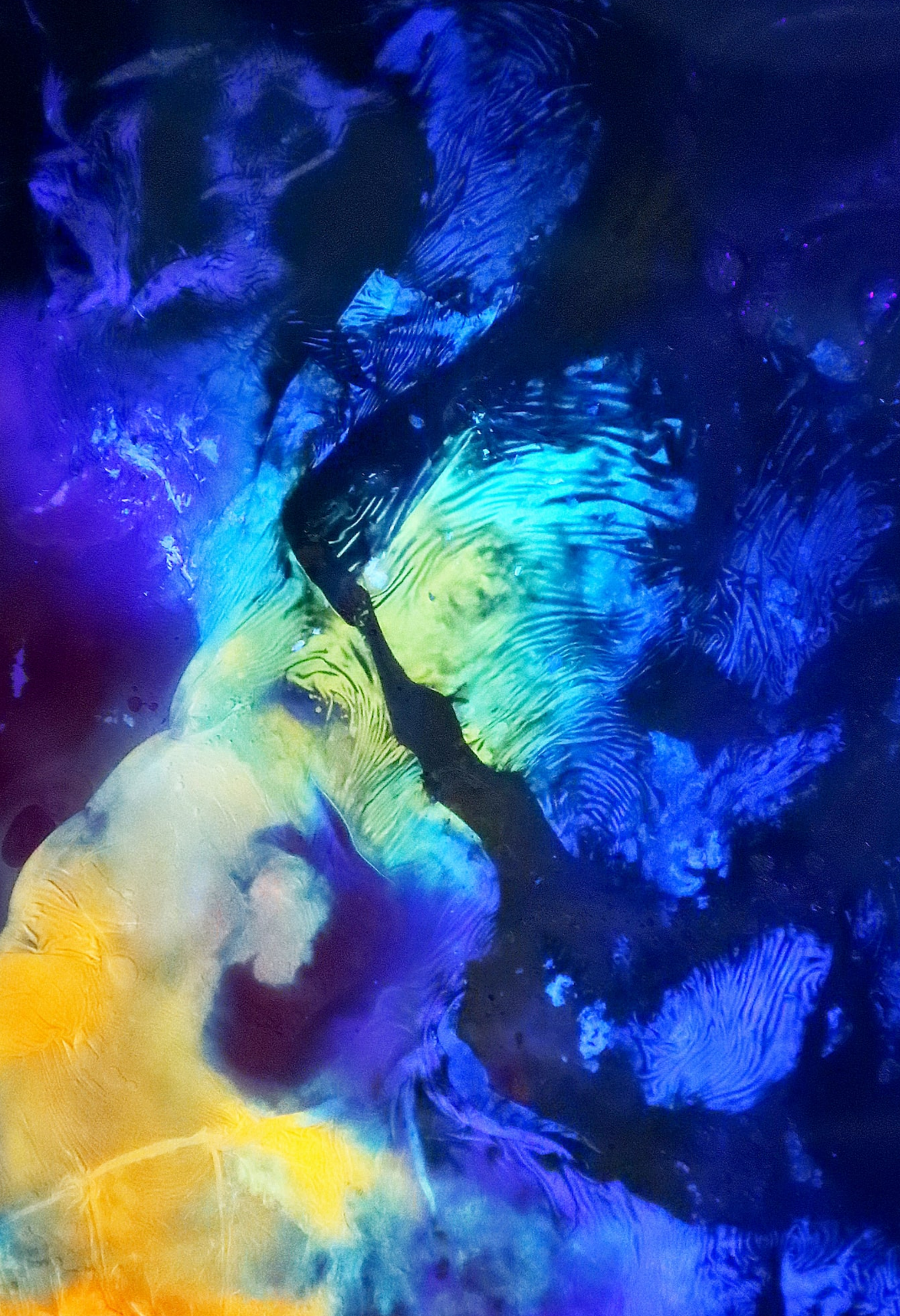




# TypeScript 補足資料 その1





# TypeScript概要



# 講座作成の動機



グーグルなどでVue3やReactを検索すると  
TypeScriptで書かれたコードが増えてきている

TypeScriptでできる事は非常に多い・・・

Vue3やReactで使われているTypeScript構文を厳選

最短距離で実務に使えるスキルを習得できる

# 講座の構成



前半 TypeScriptの基本

中盤 Vue3 x TypeScript, React x TypeScript

後半 それ以外のTypeScript

# TypeScriptの概要



Microsoftが開発 (VB, C#)  
JavaScriptのスーパーセット (上位互換)  
静的型付け言語

型をつけ、開発中にエラー発見できることで  
開発効率が向上

大規模開発にも向いている

# TypeScriptの歴史



2012年 初公開 ver.0.8

2015年 JavaScript ECMAScript2015(ES6)

2017年 ver2.2 Google標準言語の一つに

2018年 ver3.0

2020年 ver4.0

2023年1月 ver5.0 ベータ版

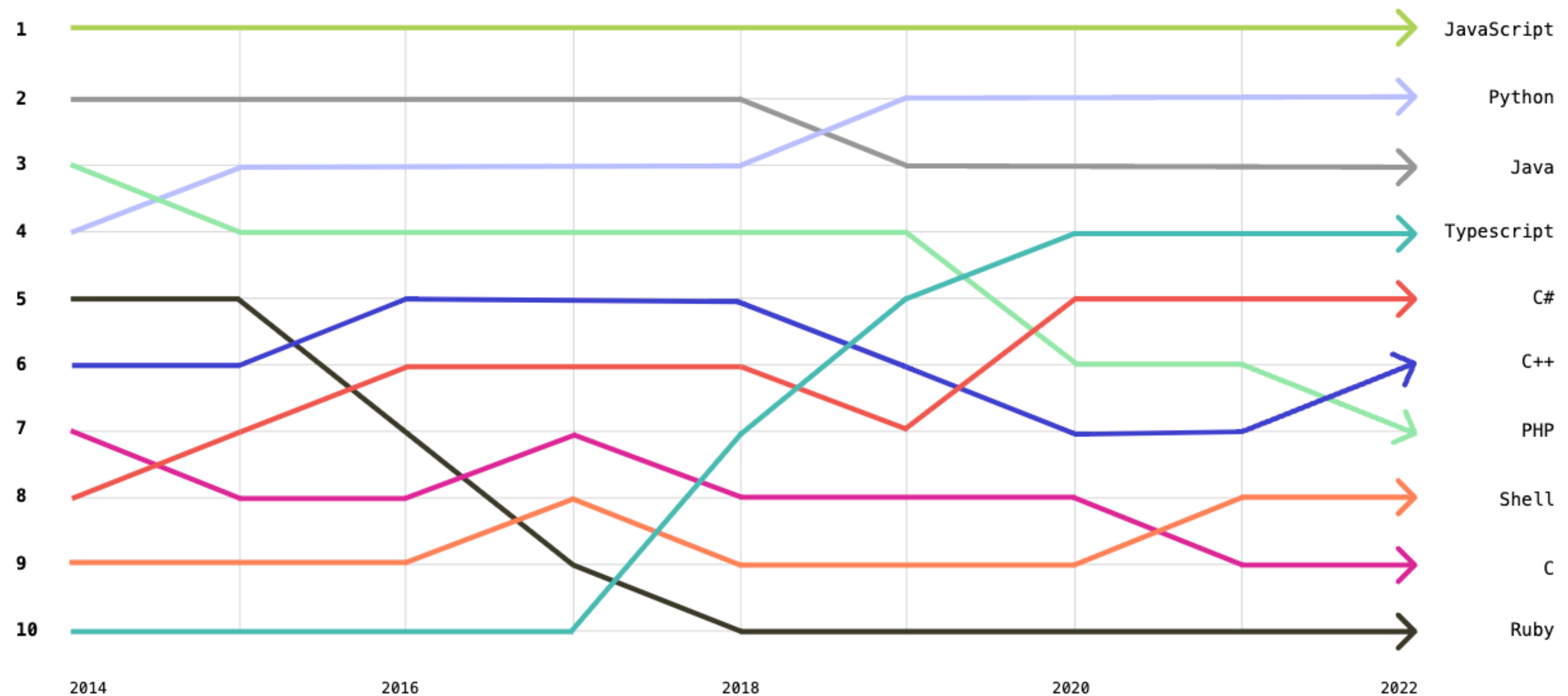
# AltJS トレンド





# GitHub言語人気ランキング

2022年 使用言語4位 勢い3位 (38%増)





# 関連する講座



JavaScript



Vue.js 2 & 3



React





# TypeScript

## 環境構築



# トランスパイル (言語の変換)



TypeScriptで書かれたコードは  
そのままではブラウザで表示できない


JSに変換が必要 (トランスパイル)

app.ts -> トランスパイル -> app.js

tsc ファイル名 // トランスパイル



# 前提: Node.js インストール



<https://nodejs.org/ja/>

JSをサーバー側で使えるようにした仕組み  
フロントエンド開発には必須

```
$ node -v
```

```
$ npm -v
```



# TypeScript環境構築



1. TypeScript単独でインストール

2. 開発環境構築用のツールを使う

Vue3 ・ ・ create-vue (Vue-CLI)


React ・ ・ create-react-app

3. 必要ライブラリをそれぞれインストール

webpack or Vite



# TypeScript単独でインストール



```
npm init -y // 初期化 package.json生成  
npm install typescript --save-dev //インストール  
npm install -g typescript // グローバル環境  
tsc -v // バージョン表示  
  
tsc --init // tsconfig.json ひながた生成
```



# tsconfig.json サンプル



tsconfig.json スターターファイル

<https://github.com/microsoft/TypeScript-Node-Starter/blob/master/tsconfig.json>



# 準備1



src フォルダ作成

tsconfig.json

// 除外フォルダ (追加)

```
"exclude": ["node_modules", "dist"]
```

package.json 追加

```
"scripts": {
```

```
  "build": "tsc",
```

```
  "watch": "tsc --watch"
```



# 準備2 & 実行

**dist/index.html**

```
<script src="./app.js"></script>
```

**src/app.ts**

```
const message: string = 'あああ'  
console.log(message)
```

npm run build // トランスパイル

npm run watch // 監視モード Ctrl+Cで解除

node dist/xxx.js // ファイル実行



# VSCode 拡張機能



## Material Icon Theme

.ts ファイルの場合

typescriptのアイコンが表示される





# TypeScript

## 基本の型



# 変数に型をつける

基本の型 ・ ・ string, number, boolean

変数名:型

```
let numberTest: number = 123
```

```
numberTest = 'あああ' // エラー発生
```

```
const stringTest: string = 'あああ'
```

```
const isValid: boolean = true
```

(const は値の再代入はできない

配列やオブジェクトなら変更可能)



# 配列に型をつける

3通り

(同じ、いずれか(**ユニオン(Union)型**)、順番通り(**タプル(Tuple)型**))

1. `const arrayTest1: string[]` = ['あああ', 'いいい']

`// arrayTest1.push(123) // エラー`

2. `const arrayTest2: ( string | number )[]` = ['あああ', 123]

`// arrayTest2.push(true) // エラー`

3. `const arrayTest3: [string, number, boolean]`  
`= ['あああ', 123, false]`

# 関数に型をつける

引数(インプット・パラメータ) と 戻り値(アウトプット) に型をつける  
戻り値がない場合は void とつける

```
function 関数名(引数:型) : 戻り値の型{ 処理 }
```

```
function funcTest(str: string, int: number) : void {  
  console.log(`文字は${str}, 値は${int}です`)  
}
```

```
const funcTest2 = function(引数: 型) : 戻り値の型 { 処理 }
```

テンプレートリテラル・・・`文字 \${変数名}`



# アロー関数に型をつける

```
const 関数名 = (引数 : 型) : 戻り値の型 => {}
```

```
const ArrowFuncTest = (str : string, int : number) : void => {  
  console.log(`文字は${str}, 値は${int}です`)  
}
```

```
ArrowFuncTest('ああ', 123)
```

```
ファイル実行は $ node dist/function.js
```

# 引数(パラメータ)あれこれ

?をつけることで引数を省略できる ?をつける場合は後半にもってくる

```
const funcTestQ = (str : string, int? : number) : void => {  
  console.log(str, int)  
}
```

デフォルトパラメータは省略の後でもok

```
const funcTestD = (int? : number, str : string = 'aaa' ) : void => {  
  console.log(int, str)  
}
```

可変長パラメータ(...)の場合

```
const eachNumber = (...items: number[]) => {  
  for(const item of items){ console.log(item) }  
}
```



# 型の変換

キャストと呼ばれたりする

文字 -> 数字に変換 など

```
const number1: number = 123
```

```
const inputValue: string = '123'
```

```
const changedValue = Number(inputValue)
```

```
const sum = number1 + changedValue
```

# その他の基本の型

基本(プリミティブ)の型

undefined 定義されていない

null 値が空

any なんでもok(極力使わない)

unknown 型不明 (実装時に修正する)

symbol (JS本体のアップデートのために導入)

bigint (ざっくり 9000兆よりも大きい値(整数限定)を使いたい場合)





tsconfig.json



# tscconfig.json

## compilerOptions

module: 出力されるJSがモジュールを読み込む方法

esModuleInterop: デフォルトインポートできるようにする

allowSyntheticDefaultImports: デフォルトimport時に型エラーにしない

target: 出力するJSのバージョン指定

noImplicitAny: 暗黙的にanyになる値をエラーにする

moduleResolution: tscのモジュールの名前解決方法

sourceMap: ソースマップファイル(コンパイル前後の対応関係 JSON)

outDir: コンパイル後の出力先パス

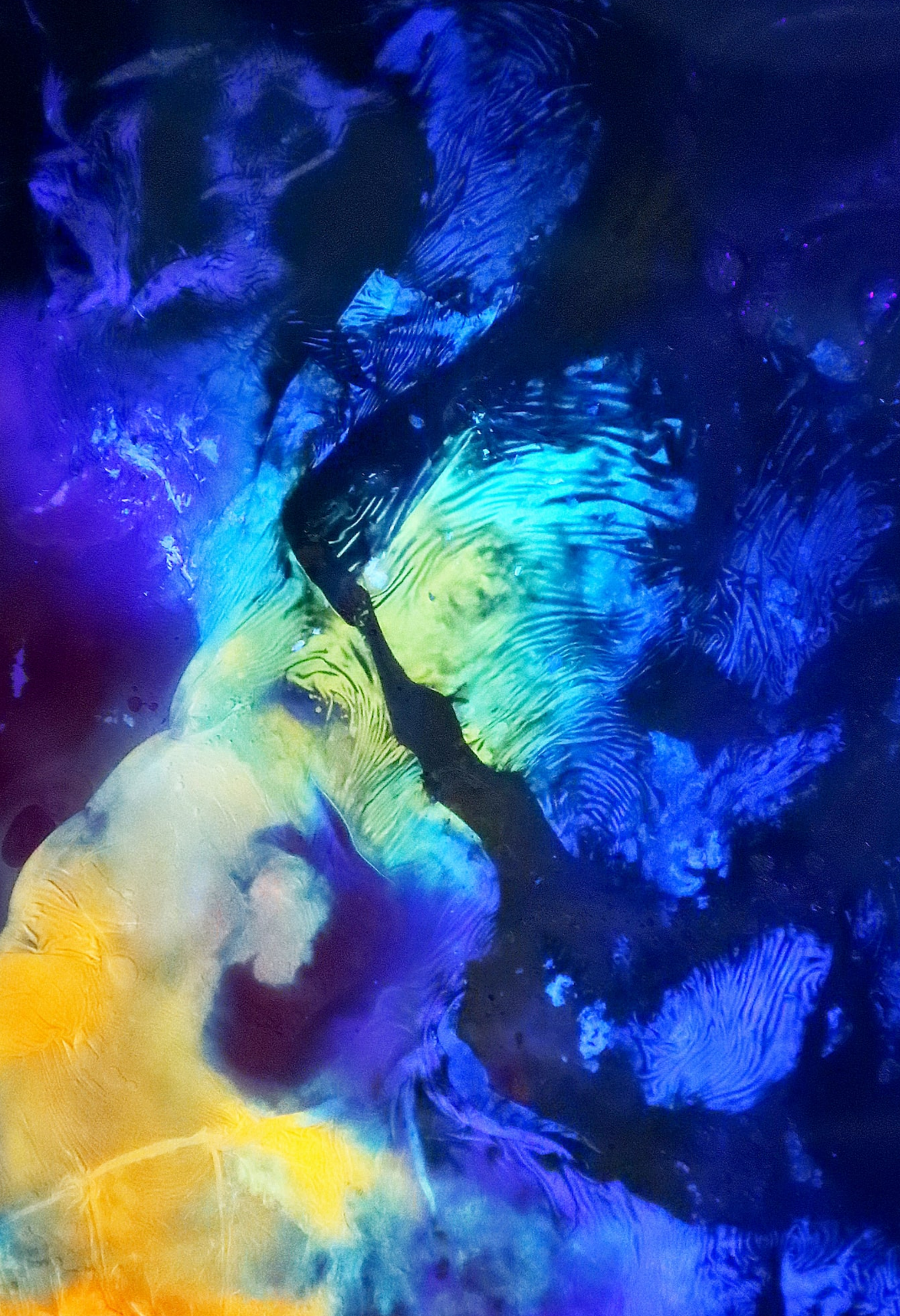
baseUrl: ベースのディレクトリ

include: コンパイル対象のパス

exclude: コンパイル対象外のパス

参考 <https://qiita.com/crml1206/items/8fbfbec0b40968bfc42>






# TypeScript オブジェクト等



# オブジェクトに型をつける



```
const objectTest:{  
  name: string,  
  age: number  
} = {  
  name: '堂安',  
  age: 30  
}
```

他に type や interfaceがある

# typeとinterface

	type (型エイリアス)	interface
書き方	<pre>type MemberType = {   name: string,   age: number }</pre>	<pre>interface MemberInterface {   name: string,   age: number }</pre>
動作	<pre>const memberT: MemberType = {   name: "三苦",   age: 30 };</pre>	<pre>const memberI: MemberInterface = {   name: "前田",   age: 30 };</pre>
定義できる型の種類	オリジナルの型をつくれる	型の宣言 (型に名前をつけられる)
	他の型も参照できる type Color = "白"   "黒" など作れる	オブジェクト、クラス、関数
	別名で作成後 & で組み合わせる事ができる <pre>type MemberHobby = {   hobby: string }</pre> <pre>type MemberProfile = MemberType &amp; MemberHobby</pre> <pre>const memberInfo: MemberProfile = {   name: 'ああ', age: 1, hobby: 'サッカー' }</pre>	<pre>interface MemberHobby { hobby: string }</pre>  extendsを使って拡張もできる <pre>interface ProfileInterface extends MemberInterface, HobbyInterface {}</pre>  <pre>const memberInfol: ProfileInterface = {   name: '前田',   age: 25, hobby: 'サッカー' }</pre>



# 配列内にオブジェクト

```
type ObjectInArray1 = { id : number, name : string, hobby: string }[]
```

```
type ObjectInArray2 = { [key: string] : string | number }[]
```

```
const members : ObjectInArray1 = [  
  { id: 1, name: '浅野', hobby: 'サッカー' },  
  { id: 2, name: '伊東', hobby: 'サッカー'}, ]
```

```
for(const member of members){  
  console.log(`id: ${member.id}, name: ${member.name}`)  
}
```

# 読取専用 readonly と as const

	readonly	as const (アサーション(主張))
用途	プロパティ毎に読み取り専用にする	配列・オブジェクトを まとめて読み取り専用にする
サンプル	<pre>type MemberTypeR = {   readonly name: string,   age: number }  const memberR: MemberTypeR = {   name: "三苦", // 初期値は設定できる   age: 30 };  // 書き換えしようとするときエラー memberR.name = '流川'</pre>	<pre>const MemberAC = {   name: 'あああ',   age: 30 } as const  // 書き換えしようとするときエラー memberAC.name = '山田'  // 配列 const arrayAC = ['aaa', 'bbb'] as const arrayAC.push('ccc') // エラー</pre>



# enum (列挙型)

複数の定数を1つにまとめる機能

```
enum SIZEEnum {  
    'Small', 'Medium', 'Large'  
}
```

SIZEEnum.Small // small と表示

複数の問題が指摘されている

SIZEEnum[5] // エラーがでない

トランスパイルすると即時関数になる (function(){})(())

->別の手段で代替する



# typeof と keyof

typeof JSの機能 型を判別 ObjectならObjectと表示

keyof typescriptの機能

オブジェクトのプロパティ名(key)を型として返す

プロパティが2つ以上あるとunion型で返す

```
const SIZE = { SMALL : '小', MEDIUM : '中', LARGE : '大' } as const;
```

```
type SizeType = keyof typeof SIZE; // keyユニオン型で
```

```
const sizeCheck = (size: KeyOfSize) => console.log(size) // 引数にtype型
```

```
sizeCheck('XL') // エラー
```

```
type ValueOfSize = typeof SIZE[keyof typeof SIZE] // valueユニオン型
```