# 表現を広げる関数型プログラミング

## TypeScript Functional Programming

kii310

# 高階関数

JS/TSは高階関数を扱える!

引数に関数を渡したり、関数を返す
関数が作れる！

# 高階関数

高階関数はコールバック関数のこと

```
const callBack = [0, 1, 2].map((num) => num + 1);
```

# 部分適用とカリー化

**高階関数は部分適用とカリー化を**
**可能にする！**

# 部分適用とカリー化

```
1 type Add = (x: number, y: number) => number;
2 const add: Add = (x, y) => x + y;
3 console.log(add(2, 3)); // 5
4
```

この関数をもとにカリー化を行う

# 部分適用とカリー化

```
 5 type AddCurried = (x: number) => (y: number) => number;
 6 const addCurried: AddCurried = (x) => {
 7   return (y) => x + y;
 8 };
 9 const five = addCurried(2)(3);
10 console.log(five); // 5
```

カリー化とは、複数の引数を取る関数を
一つの引数をとり、関数を返す関数にすること

# 部分適用とカリー化

```
12 const add5 = addCurried(5);
13 console.log(add5(5)); // 10
```

部分適用は、カリー化した関数を一部分に適用すること

# 部分適用とカリー化

応用

　カリー化は一つずつ引数を受け取る

　→再帰と組み合わせれば、引数をいくつ受け取るか分からないものを扱える

```ts
type AddLoopAccum = (accum: number) => AddLoop;
type AddLoop = (x: number) => AddMethod;
type AddMethod = {
  stack: AddLoop;
  push: () => number;
};
const addLoopAccum: AddLoopAccum = (accum) => {
  const addLoop: AddLoop = (x) => {
    return {
      stack: addLoopAccum(accum + x),
      push: () => accum + x,
    };
  };
  return addLoop;
};

const addLoop = addLoopAccum(0);
const result = addLoop(5).stack(5).stack(5).push();
console.log(result); // 15
```

# 部分適用とカリー化

**カリー化は再帰処理と相性が良い！**
**map関数と組み合わせることもできる！**
**関数でクラスを再現できる！**