

# フロントエンドエンジニアの友人と “型”で話がすれ違った原因

YUMEMI.grow 合同 LT 会 in 横浜

@Kaito-Dogi



# 自己紹介

- ❑ どぎー
- ❑ 株式会社ゆめみ
- ❑ Android エンジニア
- ❑ React Native に挑戦中 🔥



@Kaito\_Dogi



@Kaito-Dogi



フロントエンドエンジニアの友人と  
開発したとき💭

**普段 Kotlin を扱っている私が  
TypeScript を学んでみた**

「“型”の解釈違くない？😓」

**どちらも静的型付け言語**

**“型”の概念は共通しているはず…？**

型システムが異なっていた💡



**Kotlin** 🙋 **公称型**

**TypeScript** 🙋 **構造的部分型**

今回の目的

# 公称型と構造的部分型の 違いを学ぶ

# 公称型

(Kotlin など)

- ❑ 置換できない
- ❑ 名前で区別される
- ❑ 厳密さ・安全性

# 構造的部分型

(TypeScript など)

- ❑ 同じ構造で置換できる
- ❑ 構造で区別される
- ❑ 柔軟性・拡張性

## 公称型

(Kotlin など)

- ❑ **置換できない**
- ❑ 名前で区別される
- ❑ 厳密さ・安全性

## 構造的部分型

(TypeScript など)

- ❑ **同じ構造で置換できる**
- ❑ 構造で区別される
- ❑ 柔軟性・拡張性

## 公称型 (Kotlin) の場合

名前は異なる

```
class Dog(  
    val name: String,  
) {  
    fun move() {  
        // わんわんは動く  
    }  
}
```

```
class Cat(  
    val name: String,  
) {  
    fun move() {  
        // にゃんにゃんも動く  
    }  
}
```

構造は同じ

## 公称型 (Kotlin) の場合

```
val tama: Dog = Cat("tama")  
// => Type mismatch: inferred type is  
//      Cat but Dog was expected
```

左辺の変数と右辺のインスタンスの型が異なるのでエラー

## 公称型 (Kotlin) の場合

```
val tama: Dog = Cat("tama")  
// => Type mismatch: inferred type is  
//      Cat but Dog was expected
```

左辺の変数と右辺のインスタンスの型が異なるのでエラー



名前で型を区別するため

# 構造的部分型 (TypeScript) の場合

名前は異なる

```
class Dog {  
  name: string;  
  
  constructor(name: string) {  
    this.name = name;  
  }  
  
  move() {  
    // わんわんは動く  
  }  
}
```

構造は同じ

```
class Cat {  
  name: string;  
  
  constructor(name: string) {  
    this.name = name;  
  }  
  
  move() {  
    // にゃんにゃんも動く  
  }  
}
```



## 構造的部分型 (TypeScript) の場合

```
const dog: Dog = new Cat("tama");  
console.log(dog.name);  
// => "tama"
```

左辺の変数と右辺のインスタスの型が異なるが代入可能

# 構造的部分型 (TypeScript) の場合

```
const dog: Dog = new Cat("tama");  
console.log(dog.name);  
// => "tama"
```

左辺の変数と右辺のインスタスの型が異なるが代入可能



構造で型を区別するため

**UI を実装しながら理解していこう**

# 今回実装する UI

- ❑ 3種類のアイテムをリスト表示
  - ❑ ListItem(横長)
  - ❑ LargeListItem(大きい正方形)
  - ❑ MultiColumnListItem(複数カラム)
- ❑ 各アイテムの個数制限なし
- ❑ アイテムの順番は好きに入れ替え可



# Jetpack Compose (Kotlin) での実装

# LazyColumn

```
import androidx.compose.foundation.lazy.items

@Composable
fun MessageList(messages: List<Message>) {
    LazyColumn {
        items(messages) { message ->
            MessageRow(message)
        }
    }
}
```

items に渡せる型は1種類だが  
3種類のリストアイテムを渡したい

## sealed class

継承できる class を限定

```
sealed class HomeScreenUiModel {  
    // 横長のリストアイテム  
    data class ListItemUiModel(  
        val text: String,  
    ) : HomeScreenUiModel()  
  
    // 大きい正方形のリストアイテム  
    data class LargeListItemUiModel ...  
  
    // 複数カラムのリストアイテム  
    data class MultiColumnListItemUiModel ...  
}
```

継承することで  
HomeScreenUiModel として  
扱える

## HomeScreenUiModel の リストとして渡す

```
@Composable
fun HomeScreen(uiModels: List<HomeScreenUiModel>) {
    LazyColumn {
        items(uiModels) { uiModel ->
            when (uiModel) {
                is HomeScreenUiModel.ListItemUiModel ->
                    ListItem(text = uiModel.text)
                is HomeScreenUiModel.LargeListItemUiModel ->
                    LargeListItem(text = uiModel.text)
                is HomeScreenUiModel.MultiColumnListItemUiModel ->
                    MultiColumnListItem(texts = uiModel.texts)
            }
        }
    }
}
```

型の名前で条件分岐



# React Native (TypeScript) での実装

# FlatList

```
const App = () => {  
  return (  
    <SafeAreaView style={styles.container}>  
      <FlatList  
        data={DATA}  
        renderItem={({ item }) => <Item title={item.title} />}  
        keyExtractor={item => item.id}  
      />  
    </SafeAreaView>  
  );  
};
```

data に渡せる型は1種類だが  
3種類のリストアイテムを渡したい

出典: <https://reactnative.dev/docs/flatlist>

# ユニオン型

```
// 横長のリストアイテム
type ListItemUiModel = {
  type: "item";
  text: string;
};

// 大きい正方形のリストアイテム
type LargeListItemUiModel = ...

// 複数カラムのリストアイテム
type MultiColumnListItemUiModel = ...

type HomeScreenUiModel =
  | ListItemUiModel
  | LargeListItemUiModel
  | MultiColumnListItemUiModel;
```

構造を表す文字列を  
持たせておく

連結した型のうち  
「どれか」を表す

```
const HomeScreen: React.FC<Props> = ({ uiModels }) => {  
  <FlatList<HomeScreenUiModel>  
    data={uiModels}  
    renderItem={({ item }) => <HomeScreenUiModelBinder item={item.item} />}  
  /> };
```

HomeScreenUiModel の  
リストとして渡す

```
const HomeScreenUiModelBinder: React.FC<Props> = ({ item }) => {  
  switch (item.type) {  
    case "item":  
      return <ListItem text={item.text} />;  
    case "large":  
      return <LargeListItem text={item.text} />;  
    case "multi-column":  
      return <MultiColumnListItem texts={item.texts} />;  
  } };
```

型の構造で  
条件分岐

# まとめ

- ❑ 同じ静的型付け言語でも異なる型システムがある
  - ❑ 公称型: **置換**できない
  - ❑ 構造的部分型: 同じ構造で**置換**できる
- ❑ “型”定義とは
  - ❑ Android エンジニアの私: 型の**名前**を定義すること
  - ❑ フロントエンドエンジニアの友人: 型の**構造**を定義すること

サンプルリポジトリはこちら 

<https://github.com/Kaito-Dogi/type-systems>

# 参考記事

- ❑ 構造的部分型 (structural subtyping) | TypeScript 入門『サバイバル TypeScript』  
<https://typescriptbook.jp/reference/values-types-variables/structural-subtyping>
- ❑ Lists and grids | Compose | Android Developers  
<https://developer.android.com/jetpack/compose/lists>
- ❑ Sealed classes and interfaces | Kotlin Documentation  
<https://kotlinlang.org/docs/sealed-classes.html>
- ❑ FlatList · React Native  
<https://reactnative.dev/docs/flatlist>
- ❑ TypeScript: Handbook – Unions and Intersection  
<https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html>
- ❑ ユニオン型 (union type) | TypeScript入門『サバイバル TypeScript』  
<https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html>
- ❑ Keishin Yokomaku, (2023), Kotlin で Either したい  
<https://speakerdeck.com/keithyokoma/either-in-kotlin>

ありがとうございました🙌🙌