型

- Int・・・1, 25, 200
- String・・・"English", "日本語"
- Boolean・・・true, false

等

変数

```
var 変数名: 型 = 値

var foo: String = "aaa"
foo = "bbb"

// 型推論
var foo = "aaa"
```

定数

```
val 変数名: 型 = 値

val bar: Int = 12
val bar = 12
```

配列

```
var names: List<String> = listOf("田中", "山田")
var name = names[0] // nameに"田中"が入る
```

関数

```
fun 関数名(引数名: 型): 戻り値の型 {
    // TODO
    return 戻り値
}

fun add(a: Int, b: Int): Int {
    return a + b
}
```

クラス

```
class クラス名 (
    // コンストラクタ
) {
    // メソッド
}

class Animal (
    var age: Int = 0,
    val birthday: Date = Date()
) {

    fun incrementAge() {
        age += 1
    }

    fun bark(): String {
        return ""
    }

}

// インスタンスの生成
var animal1 = Animal(
    5,
    Date() // 現在時刻
)

var animal2 = Animal()

// メソッドの使用
animal1.incrementAge()
```

クラスの継承

```
class Cat: Animal() {

    override fun bark(): String {
        return "にゃー"
    }

}
```

条件分岐

```
if (a > 0) {
    // TODO
else {
    // TODO
}

if (a > 0) /* TODO*/ else /* TODO */

when (str) {
    "aaa" -> // strが"aaa"と一致する時
    "bbb" -> // strが"bbb"と一致する時
    else -> // いずれも該当しない時
}
```

null

- **null**
  変数に値が入っていない（確定していない）状態
- **Nullable**
  nullを許容する型。型名の後ろに『**?**』 が付きます。

  ```
  var num        : Int  = null // コンパイルエラー
  var nullableNum: Int? = null // OK
  ```

- **Null Safety**
  基本的に、Nullableな型の変数に直接アクセスすることは好ましくありません。
  もしその変数がnullであった時に、存在しないプロパティにアクセスすることになるからです。
  Nullableな型の変数に対してアクセスするには中身がnullであるかどうかを評価する必要があります。

  ```
  var numList: MutableList<Int>? = mutableListOf(1, 3, 5) // 要素の追加が可能なList

  numList.add(7)  // コンパイルエラー
  numList?.add(7) // numListがnullなら何もしない

  val size         = numList.size       // コンパイルエラー
  val nullableSize = numList?.size      // numListがnullならnullが代入される
  val safetySize   = numList?.size ?: 0 // numListがnullなら0が代入される

  val three       = numList[1]   // コンパイルエラー
  val safetyThree = numList!![1] // 『!!』を用いてnumListがnullでないことを明示しないといけない
                                 // （もしnumListがnullであれば実行時エラーが出ます）
  ```

```
numList?.let {                      // nullableな変数に『?.let {}』と続けると
    val safetyList = it.toList()    // {}の中でSafetyな値として扱うことができる（変数名はitになる）
}                                   // なお、その変数がnullの場合は{}内の処理は実行されない
```

スコープ関数(apply)

同一のものに関する処理を一つにまとめたいときはapply関数が便利です

```
item.title = text1
item.content = text2

// applyを使うと・・・

item.apply {
    title = text1
    content = text2
}
```

apply関数の中ではviewの持つ要素に楽にアクセスできたり、一連の処理が同一のものに関する処理だということが目で見てわかりやすくなります。
こういった関数を**スコープ関数**と言い、applyの他には**also**や**run**、Null Safetyの説明の際に少し出てきた**let**などがあります。