infix 呼び出し記法

Kotlin には、「infix 記法 (infix notation)」 と呼ばれる、特別なメソッドコールの構文が用意されています。 「infix 呼び出し (infix call)」や「中置記法」などと呼ばれることもあります。

例えば、次のように `Map` オブジェクトを生成するときに使用する `to` 関数などが infix 呼び出しできるように定義されています。

```
val map = mapOf("one" to 1, "two" to 2, "three" to 3)
println(map["one"])  //=> 1
```

infix とは「接中辞」のことで、上記の例では、`to` がちょうど間に挟まる形になるのでこう呼ばれています。 この `to` 関数は、ジェネリックな関数として次のように定義されています。

```
infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

つまり、`to` 関数は `Pair` オブジェクトを生成するためのユーティリティ関数です。

```
val pair = "one" to 1  //=> Pair("one", 1)
```

この `to` 関数は本来は下記のように、ドットや括弧を付けたメソッド呼び出しの形で呼び出さなければならないはずですが、関数の先頭に `infix` を付けて定義してあると、ドットや括弧を省略した「infix 記法」で呼び出せるようになります。

```
val pair1 = "one".to(1)  // 通常のメソッド呼び出し記法
val pair2 = "one" to 1   // infix 呼び出し記法
```

ちなみに、`Map` オブジェクトを生成する `mapOf()` 関数は次のように定義されており、任意の数の `Pair` オブジェクトをパラメータとして受け取るようになっています。

```
public fun <K, V> mapOf(vararg pairs: Pair<K, V>): Map<K, V>
```

最初のコード例では、これに渡す `Pair` 引数を生成するために、infix 関数である `to` 関数を利用しているということです。

infix 関数を定義するときの制約

`infix` キーワードは、クラスのメソッドや、拡張関数を定義するときに付加することができますが、infix キーワードを付けられる関数には下記のような制約があります。

- 1 つのパラメータしか受け取れない
- デフォルト引数 (`=`) は扱えない
- 可変長引数 (`vararg`) は扱えない

まぁ、1 レシーバー 1 引数の形になる記述方法なので、当たり前といえば当たり前ですね。

infix 関数の例

`CharSequence` クラスには `matches()` という infix 関数が定義されており、文字列が指定した正規表現に一致するかどうかを調べることができます。

```
infix fun CharSequence.matches(regex: Regex): Boolean
```

```
val text = "2020-01-16"
val pattern = """\d{4}-\d{2}-\d{2}""".toRegex()
println(text matches pattern)  //=> true
```

少しだけ英語風に読めるようになりますが、これくらいだったら次のような通常の呼び出し方でも充分読みやすいので、まぁどちらでもいいかなという感じですね。

```
text.matches(pattern)
```

Kotlin でもっとも infix 呼び出しが活用されているのは、前述の `to` 関数による `Map` 要素の初期化や、次のような `for` ループでの数値処理かと思います。

```
for (i in 1 until 5)          //=> for (i in 1.until(5))
for (i in 5 downTo 1)         //=> for (i in 5.downTo(1))
for (i in 1 until 5 step 2)   //=> for (i in 1.until(5).step(2))
for (i in 5 downTo 1 step 2)  //=> for (i in 5.downTo(1).step(2))
```

これは、前者の infix 呼び出しの形式の方が明らかに読みやすいです。

参考（分解宣言）

上記で説明した `to` 関数は、2 つの値から `Pair` オブジェクトを作成するのに便利でした。 Kotlin には、これとは逆に、`Pair` オブジェクトから 2 つの値を別々の変数に取り出す構文（分解宣言）が用意されています。

- **分解宣言 (destructuring declarations) による Pair 要素や Triple 要素の分解**