Kotlin
ハイパフォーマンス
プログラミング

大前良介 (OHMAE Ryosuke)

自己紹介

- 大前良介 (OHMAE Ryosuke)
  - Github: https://github.com/ohmae
  - X(twitter): @ryo_mm2d
  - Qiita: ryo_mm2d

- ヤフー株式会社 @大阪
  - Androidアプリエンジニア
  - Yahoo!天気アプリ開発

突然の雨対策に

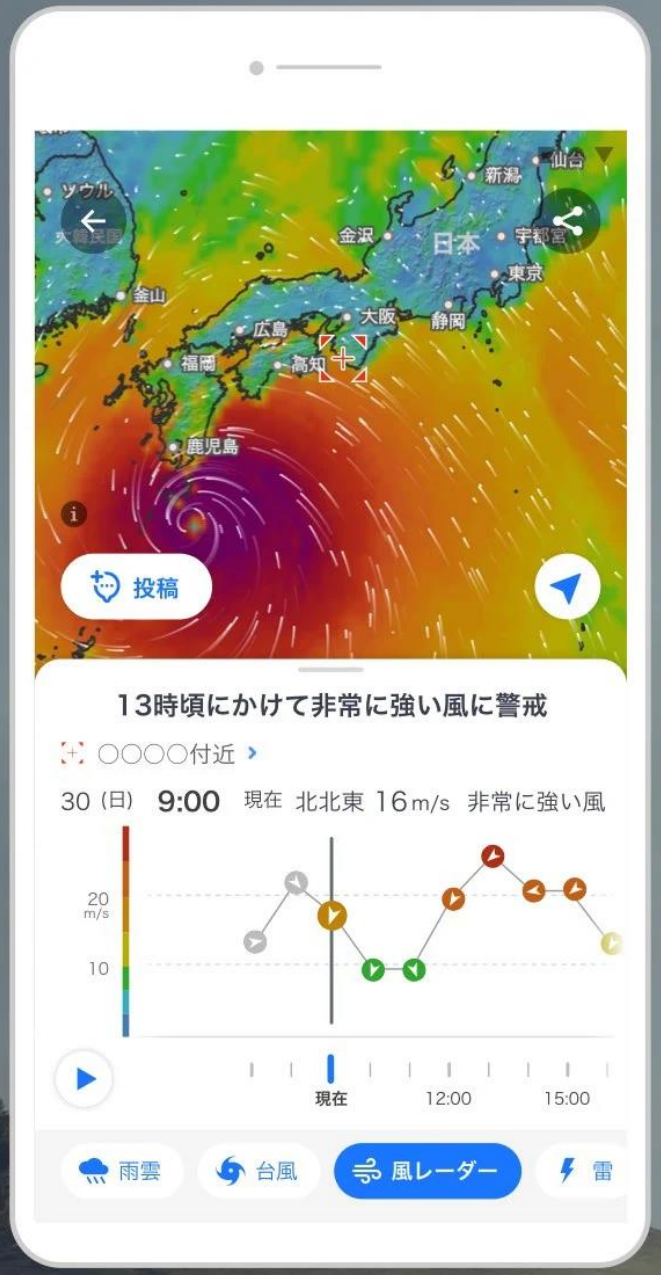
雨雲の接近がわかる天気アプリ

Yahoo!天気

雨雲レーダーに新機能追加！

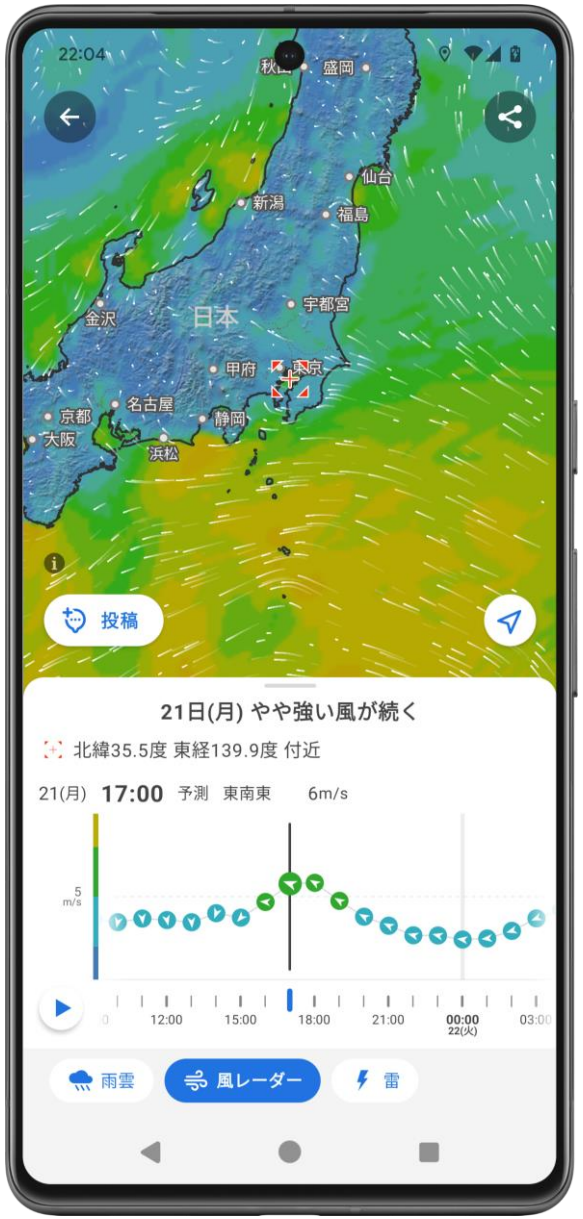
風の強さと流れが色とアニメーションで視覚的にわかる

風レーダー



風レーダー

- アニメーション
  - OpenGLによるレンダリング
  - すべてKotlinによる実装
- リファクタリング & 徹底的な最適化
  - 1stリリースから場所によっては10倍以上の高速化
  - 基本的なことの積み重ね
- DroidKaigiプロポーザル
  - 無事採択いただけました



Kotlin
ハイパフォーマンス
プログラミング

アプリの高速化のノウハウ・基礎知識

Kotlinは遅いのか？

重い処理はネイティブで実装すべき？

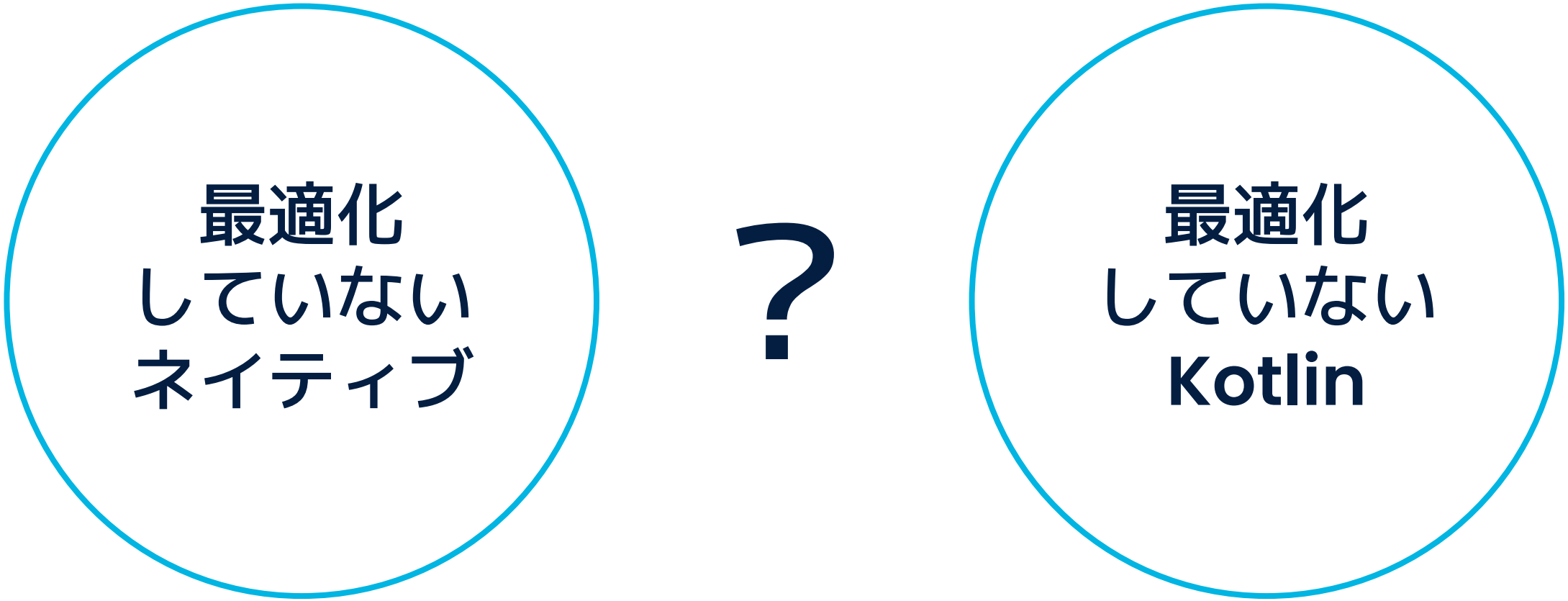
Java/Kotlin は 遅い！

C/C++で
ネイティブ実装
するべきだ！

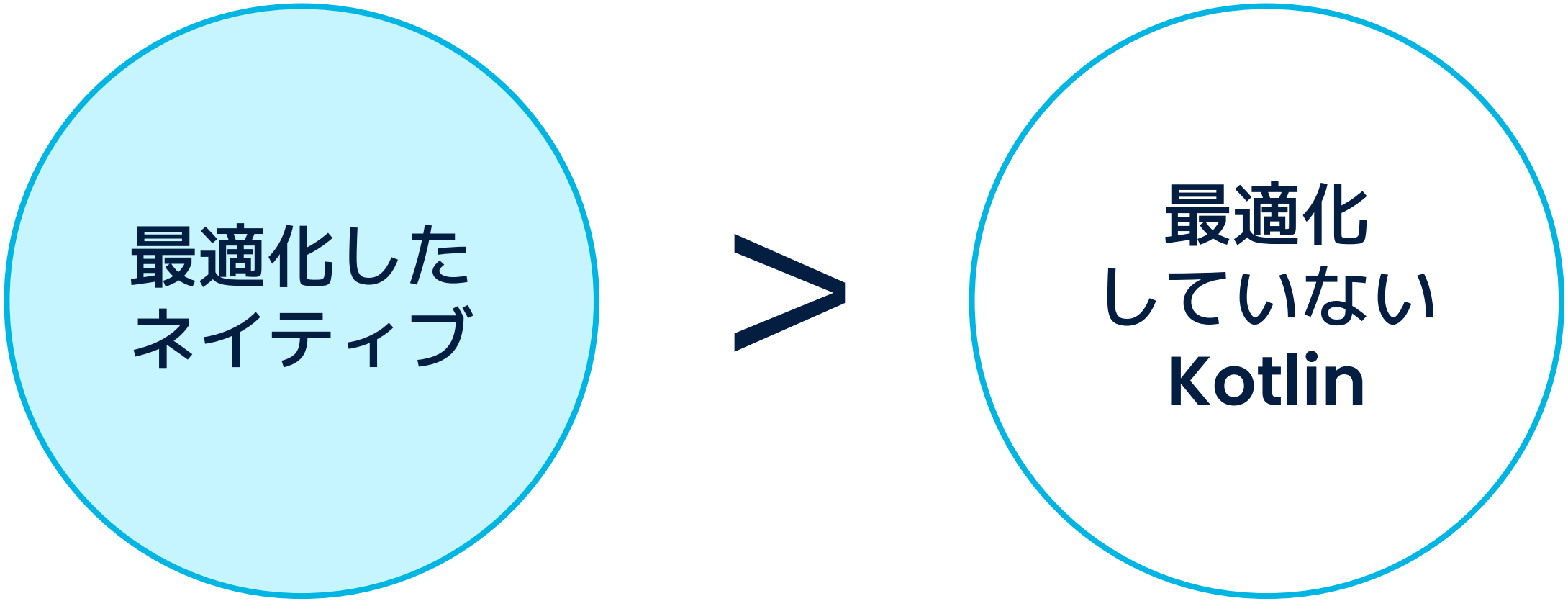
重い処理はネイティブで実装すべき？

……本当ですか？

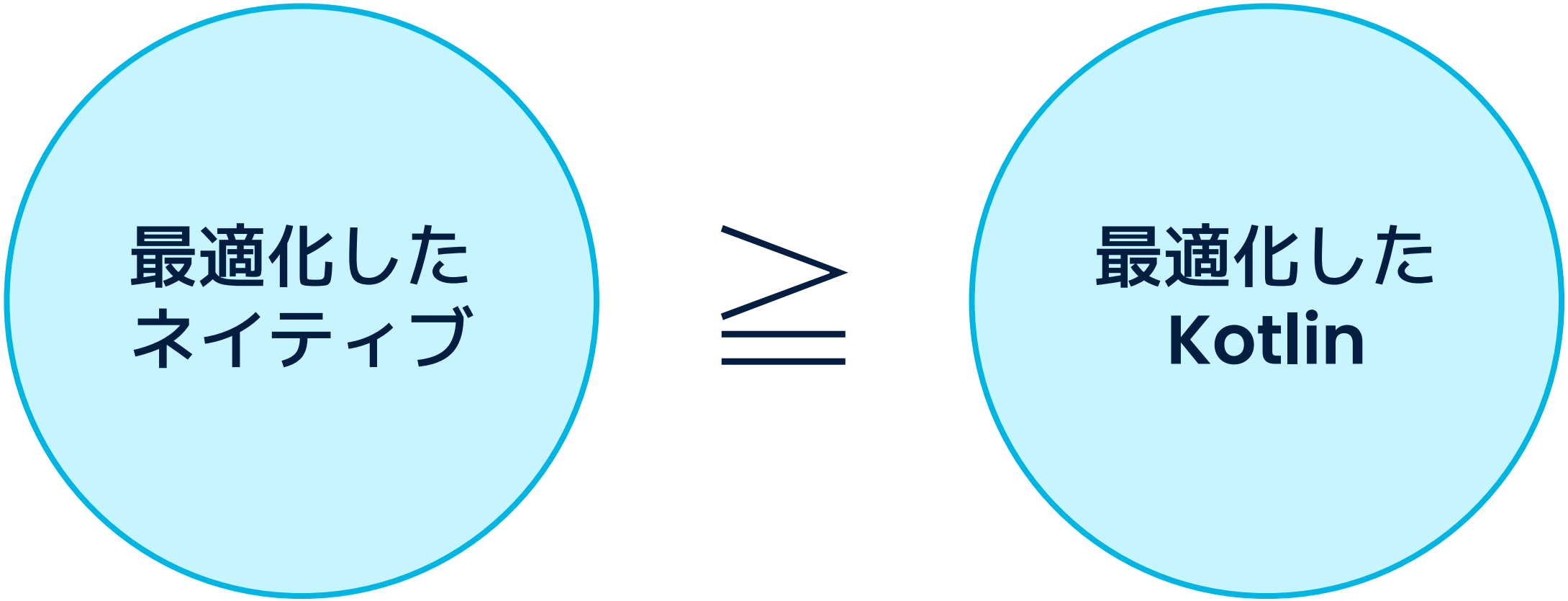
重い処理はネイティブで実装すべき？



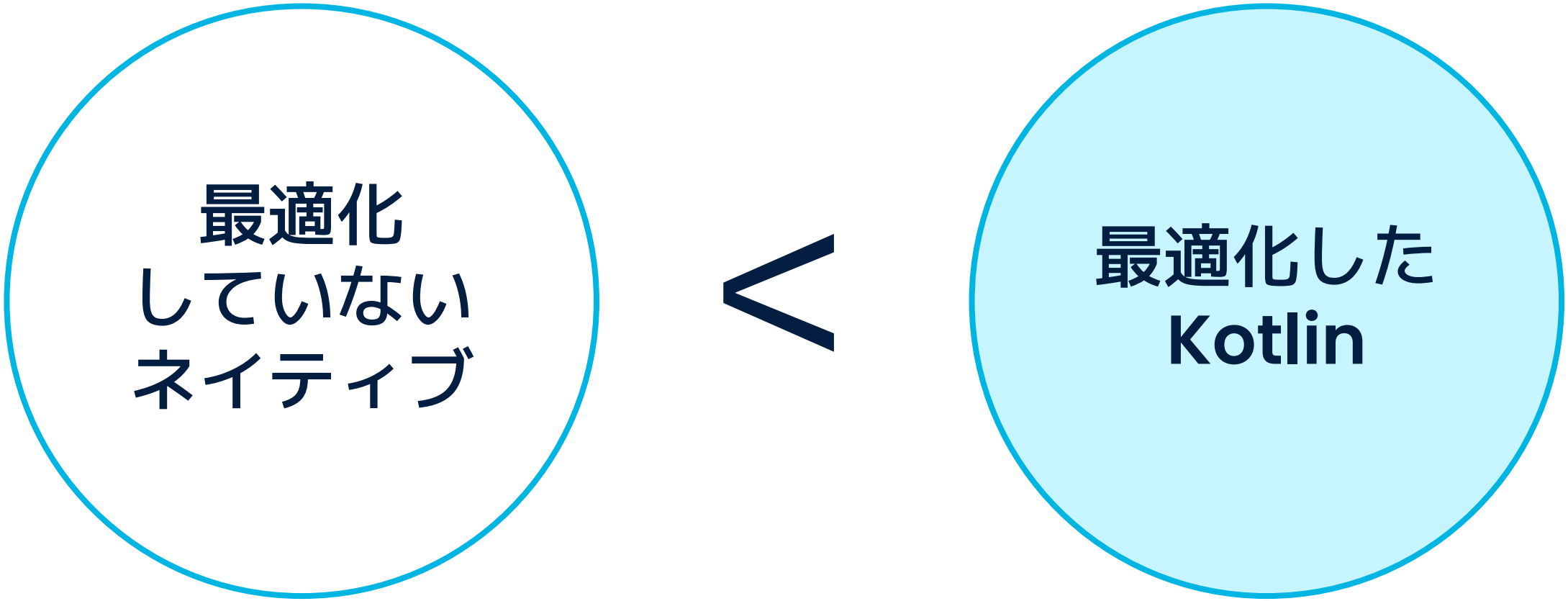
重い処理はネイティブで実装すべき？



重い処理はネイティブで実装すべき？



重い処理はネイティブで実装すべき？



重い処理はネイティブで実装すべき？

遅いロジックは
どの言語でも遅い

安易に言語スイッチしても
速くはならない

ボトルネックの
本質を見極める

JVM言語実装が遅いのは過去のこと

- Androidランタイム(ART) Android 5.0〜
  - AOT(Ahead-Of-Time)コンパイル
  - JIT(Just-In-Time)コンパイル
  - プロファイルガイド付きコンパイル

- ネイティブコードと遜色のない速度

Kotlinは十分に速い！

速くできる

重い処理はネイティブで実装すべき？

言語の変更って

開発生産性の
高い言語

だけではない

Kotlinで実装するメリットを
上回る価値がありますか？

AndroidアプリをKotlinで実装するメリット

エンジニアの
スキルセット

開発人員の
確保

将来にわたる
メンテナンス
性

ネイティブ開発のメリットを享受できる場合はもちろんある

その開発言語で
全体を開発する

既存のソフトウェア資産
を流用する

豊富な知見を持つ
メンバーが多数いる

特定のハードウェアを
使った最適化

Kotlinの特徴

- 高水準言語
  - ハードウェアを意識しなくて良い
  - ボイラープレートコード不要
  - シンプルな構文で高度な実装
  - 高い開発生産性

大きなメリット

- 実行コストが見えにくい
  - ハードウェアを意識しにくい
  - 無意識に高コストなコード
  - 高速化するには知識と経験
  - 通常は意識する必要が無い

小さなデメリット

高速化する
には
ノウハウが必要

大量のデータを高速に処理する

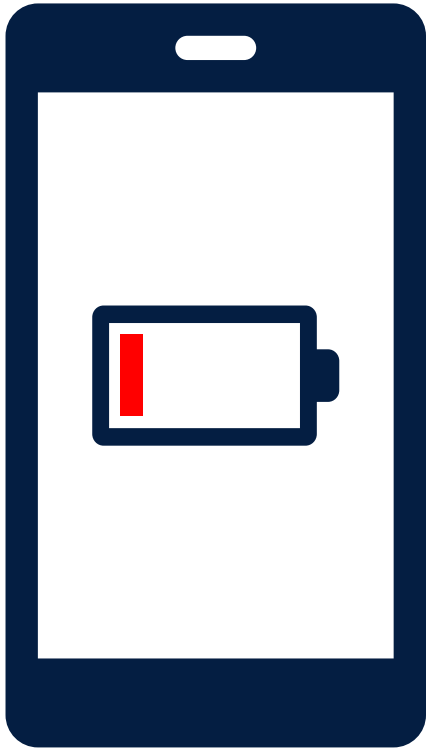
最初に考えるべきこと

やらない

何もしないのが一番速い

やらない

- Android端末のほとんどはモバイル端末
  - 計算リソースの制約
  - バッテリーの制約
  - 大量のデータを高速に処理するのに向いていない



- 他に選択肢がないか最初に考える
  - その処理はアプリ上で実行しなければならないことか？
  - 一部だけでもBEに移譲して、処理の必要最小限にする

どうしても必要？

パレートの法則 - 20対80の法則

- 処理時間の**80%**はコード全体の**20%**が占める
  - 実際にはもっと偏っているのでは？
  - 実行時間を多く使う箇所に限定して高速化

- 適用すべき個所はごく一部
  - 高速化、効率化、エンジニアにとって**魅惑のワード**
  - わずかな速度の向上よりも、**メンテナンス性を重視**

高速化を考える上で最初に理解すべきこと

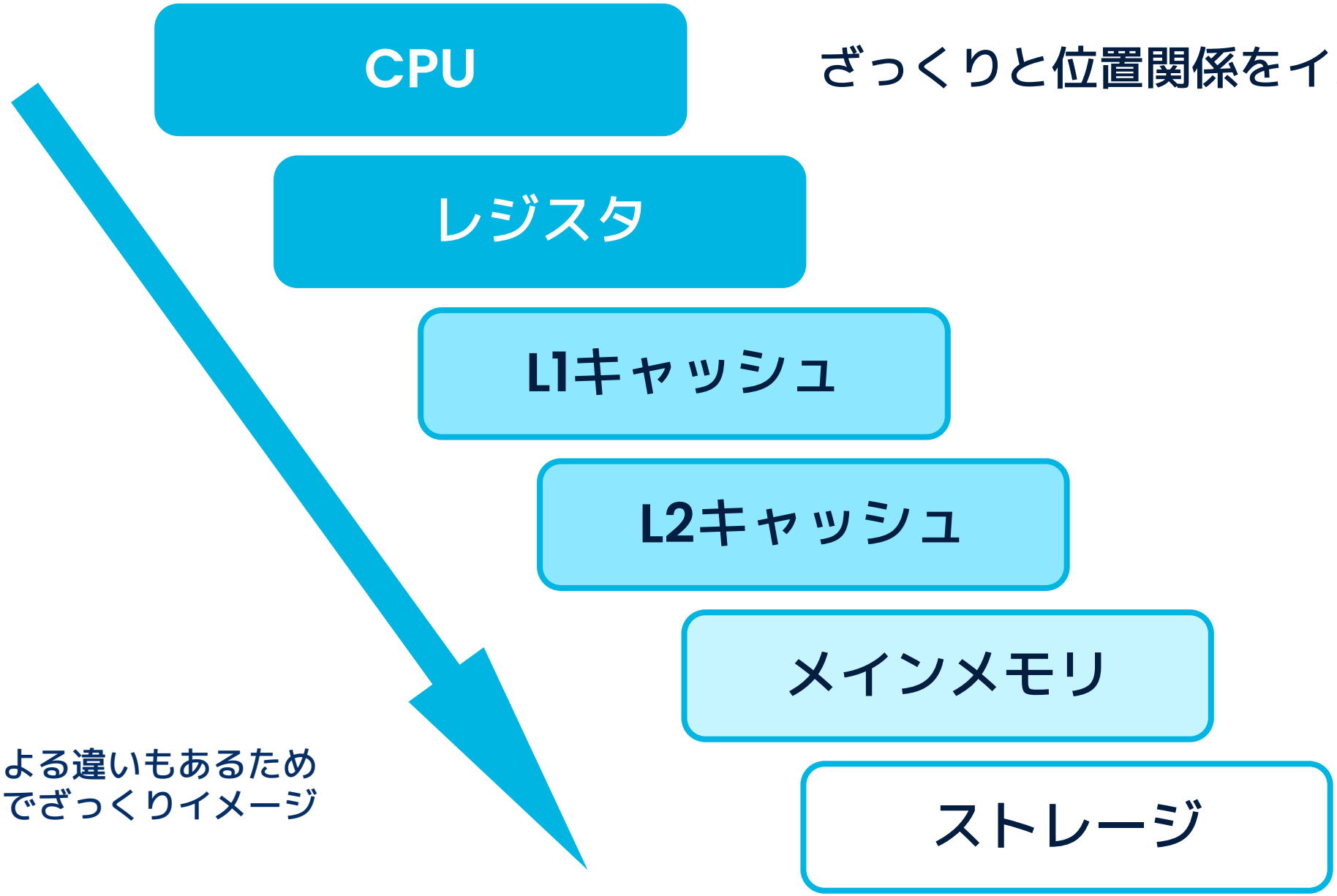
やっと本題

メモリの遅さを理解する

CPUに比べて

メモリ は 遅い

データのありかは？



ざっくりと位置関係をイメージする

※環境による違いもあるため
あくまでざっくりイメージ

メモリは遅い

- メインメモリはCPUより数百〜数千倍遅い（と考えておこう）
  - メモリアクセスの無駄を排除し最適化する
  - メモリアクセスを局所化し、キャッシュヒット率を高める

- 圧倒的に遅いので
  - 計算結果を再利用するより、毎回計算する方が速い場合もある
  - 計算量の多いアルゴリズムの方が速い場合もある

気をつけるポイント

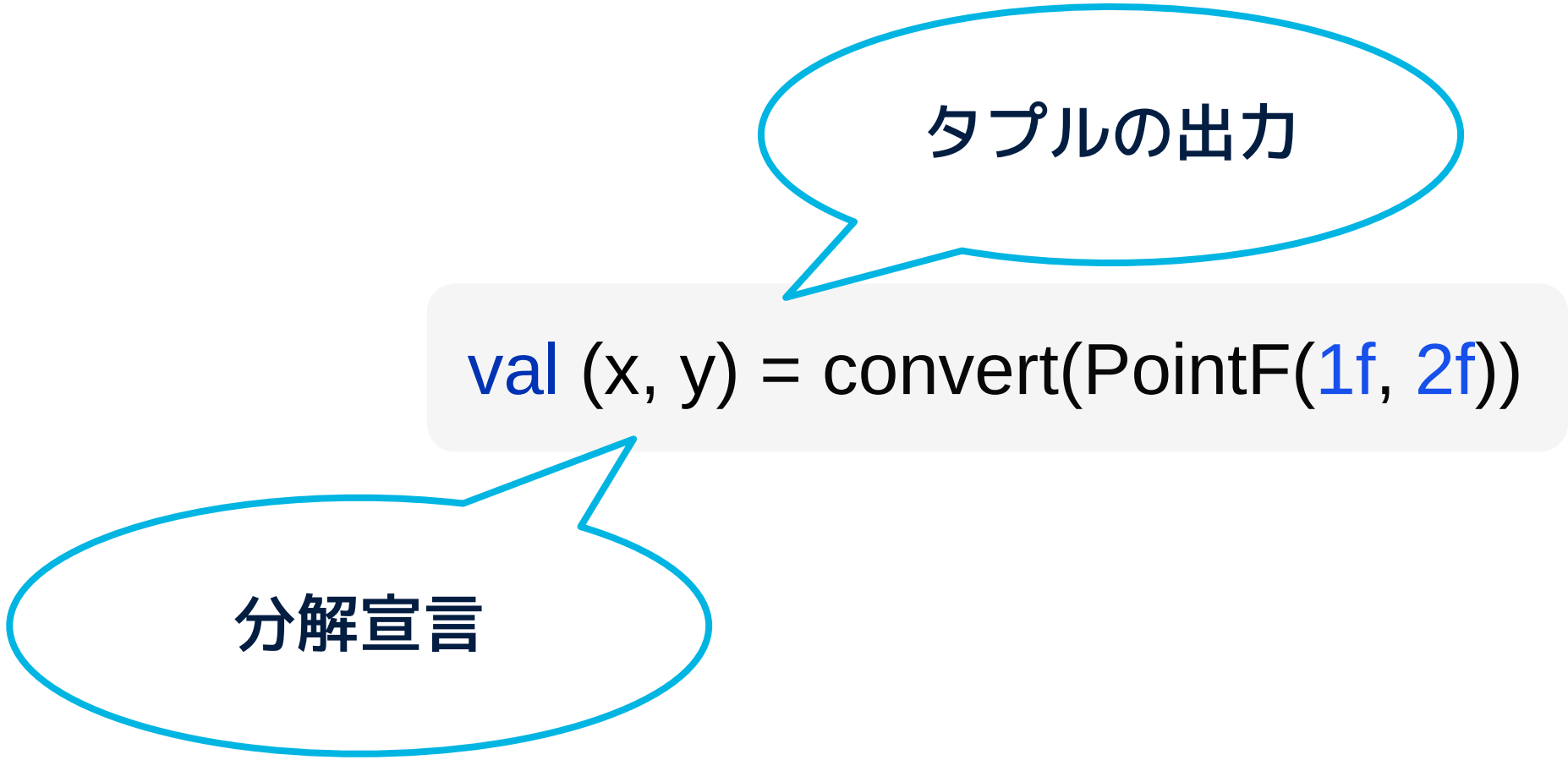
- メモリのコピー
  - データの受け渡し、詰め替え
  - 大量のデータはコピー回数を最小限に
- インスタンスの使い捨て
  - メモリを確保
  - 初期化（データの書き込み）
  - ガベージコレクションによる開放

Kotlinでは
特に注意

便利な記法に隠れたコスト

出力データクラス
タプル＋分解宣言

出力データクラス / タプル / 分解宣言（Kotlin）



出力データクラス / タプル / 分解宣言（Java Decompile）

インスタンス生成
（出力）

```
Pair var1 = this.convert(new PointF(1.0F, 2.0F));
float x = ((Number)var1.component1()).floatValue();
float y = ((Number)var1.component2()).floatValue();
```

プリミティブ
ラッパー

出力のためだけに
インスタンスを使い捨て

出力データクラス / タプル / 分解宣言

- これ自体は優れた記法
  - データの関連性や制約を適切に表現
  - 可読性・メンテナンス性に優れている
- インスタンスの使い捨てが問題になる場合もある
  - 大量・高頻度に使用される箇所では慎重に
- 対処法の一つはよくあるあの記法

書き込み先を引数で渡す

なんでこうしないの？

```
val rect = view.getGlobalVisibleRect()
```

outRectは
使い回しできる

```
val outRect = Rect()
view.getGlobalVisibleRect(outRect)
```

出力インスタンスを
使い捨てない

書き込み先を
引数で渡す

プリミティブラッパー

- Kotlinでは区別しない
  - コンパイラーが適切に使い分ける
  - 量が多くなればコストが無視できなくなる
- ラッパーを強制する使い方に注意
  - Nullable / ジェネリクス / コレクション
  - IntArrayとArray<Int>の違い

```
val a: IntArray = intArrayOf(1, 2, 3)
// int[] a = new int[]{1, 2, 3}
val b: Array<Int> = arrayOf(1, 2, 3)
// Integer[] b = new Integer[]{1, 2, 3}
```

どのぐらいの違いが出るか？

```
val list = List(100000) { PointF(Random.nextFloat(), Random.nextFloat()) }
```

要素数10万のPointFリストを入力しx,yの合計計算　@Pixel 7

11.25 ms

```
fun convert(p: PointF)
        : Pair<Float, Float> {
    return p.x * 2 to p.y * 2
}
```

1.19 ms

```
fun convert(p: PointF, op: PointF) {
    op.x = p.x * 2
    op.y = p.y * 2
}
```

便利な記法に隠れたコスト

コレクション操作

コレクション操作

```
list.map { it.first }
    .filter { it > 0 }
    .sum()
```

コレクション操作

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}
```

新しい
Collection

```
public inline fun <T, R, C : MutableCollection<in R>>
Iterable<T>.mapTo(destination: C, transform: (T) -> R): C {
    for (item in this)
        destination.add(transform(item))
    return destination
}
```

データの詰め替え

コレクション操作

```
list.map { it.first }
    .filter { it > 0 }
    .sum()
```

ステップごとに
新しいCollection

Sequenceの利用

Sequence

```
list.asSequence()
    .map { it.first }
    .filter { it > 0 }
    .sum()
```

Sequenceの利用

```
public fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
    return TransformingSequence(this, transform)
}

internal class TransformingSequence<T, R>
constructor(private val sequence: Sequence<T>, private val transformer: (T) -> R) : Sequence<R> {
    override fun iterator(): Iterator<R> = object : Iterator<R> {
        val iterator = sequence.iterator()
        override fun next(): R {
            return transformer(iterator.next())
        }

        override fun hasNext(): Boolean {
            return iterator.hasNext()
        }
    }

    internal fun <E> flatten(iterator: (R) -> Iterator<E>): Sequence<E> {
        return FlatteningSequence<T, R, E>(sequence, transformer, iterator)
    }
}
```

終端のiterateで
要素ごとに処理される

Sequenceの利用

```
list.asSequence()
    .map { it.first }
    .filter { it > 0 }
    .sum()
```

比較

詰め替えのコスト

inline

```
public inline fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    return mapTo(ArrayList<R>(collectionSizeOrDefault(10)), transform)
}
```

ArrayList生成

not inline

```
public fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R> {
    return TransformingSequence(this, transform)
}
```

Sequence生成

呼び出しのコスト

適切に使い分けを

べた書き最強説

可読性
悪い？

```
var sum = 0
list.forEach {
    if (it.first > 0) {
        sum += it.first
    }
}
```

分かる

Kotlinで書く以上
Kotlinらしい
エレガントな構文を使いたい！

どのぐらいの違いが出るか？

```
val list = List(100000) { it to it }
```

要素数10万のPairリストを入力　@Pixel 7

66.48 ms

```
list.map { it.first }
    .filter { it > 0 }
    .sum()
```

5.98 ms

```
list.asSequence()
    .map { it.first }
    .filter { it > 0 }
    .sum()
```

1.97 ms

```
var sum = 0
list.forEach {
    if (it.first > 0) {
        sum += it.first
    }
}
```

便利な記法に隠れたコスト

可変長引数 + スプレッド演算子

可変長引数 + スプレッド演算子

```
fun hoge(vararg args: String)
```

```
hoge("a", "b", "c")
```

スプレッド演算子

```
val array = arrayOf("a", "b", "c")
hoge(*array)
```

追加可能

```
val array = arrayOf("a", "b", "c")
hoge(*array, "d")
```

可変長引数 + スプレッド演算子（Java Decompile）

```
val array = arrayOf("a", "b", "c")
hoge(*array)
```

```
String[] array = new String[]{"a", "b", "c"};
hoge((String[])Arrays.copyOf(array, array.length));
```

配列のコピー

可変長引数 + スプレッド演算子（Java Decompile）

```
val array = arrayOf("a", "b", "c")
hoge(*array, "d")
```

引数の結合

```
SpreadBuilder var1 = new SpreadBuilder(2);
var1.addSpread(array);
var1.add("d");
hoge((String[])var1.toArray(new String[var1.size()]));
```

配列のコピー

配列のコピー

高コストになりやすい

どのぐらいの違いが出るか？

```
val array = FloatArray(100000) { Random.nextFloat() }
```

要素数10万のFloatArrayを入力し合計計算　@Pixel 7

45.46 ms

```
fun sum(vararg v: Float): Float
    = v.sum()
```

17.70 ms

```
fun sum(v: FloatArray): Float
    = v.sum()
```

紹介はごく一部

- すべてを把握しておく必要は無い
- 必要なときに計測し、実装を調査する
  - コストを推測できるようになっておくことが重要
- メモリは遅い
- 通常は便利な記法は積極的に利用しよう
  - 大量・高頻度に使用される箇所に限定して適用

アルゴリズムとデータ構造

思い込んでいないか

ランダウの記号 / O-記法

要素数nのときの計算量

$$an^2 + bn + c$$

$$O(n^2)$$

n が十分に大きい場合の
計算量比較に使われる

ランダウの記号 / O-記法

バブルソート

$O(n^2)$

$<$

クイックソート

$O(n \log n)$

どっちが速い？

$O(n^2)$? $O(n \log n)$

どっちが速い？

$O(n^2)$ $<$ $O(n \log n)$

とは限らない

ランダウ記法は実行速度比較の絶対指標ではない

- ランダウ記法は **十分に大きなn** の計算量の比較
  - 要素数の規模はある程度決まっている
  - 係数を無視できるほど大きな要素数ではない場合も多い
  - 全体のデータ量は大量でも、一つの計算は小さな n の場合も

- 計算量の大小だけで速さが決まるわけではない

メモリの遅さを理解する

実際の速度

- データ構造の重要性
  - 効率的なメモリアクセスができるデータ構造の方が速い
  - データ構造を変更するだけでも大きな効果がでる場合も

- ランダウ記法に惑わされない
  - 次数の大きなアルゴリズムの方が速い場合もあり得る
  - 次数が同じアルゴリズムは同じ速さではない
  - 実際のユースケースに最適なアルゴリズムを選択

本当に逆転なんて起こるの？

- IntArray.sort vs クイックソート vs バブルソート

```
fun quickSort(s: IntArray, left: Int = 0, right: Int = s.size - 1) {
    val p = s[(left + right) / 2]
    var l = left
    var r = right
    while (l <= r) {
        while (s[l] < p) l++
        while (s[r] > p) r--
        if (l <= r) {
            val tmp = s[l]
            s[l] = s[r]
            s[r] = tmp
            l++
            r--
        }
    }
    if (left < r) quickSort(s, left, r)
    if (l < right) quickSort(s, l, right)
}
```

```
fun bubbleSort(s: IntArray) {
    for (i in 0 until s.size) {
        for (j in i + 1 until s.size) {
            if (s[i] > s[j]) {
                val tmp = s[i]
                s[i] = s[j]
                s[j] = tmp
            }
        }
    }
}
```

速度比較

	IntArray.sort	クイックソート	バブルソート
要素数 10000個 x 10 @Pixel 7	4.00 ms	5.33 ms	732.14 ms
要素数 10個 x 10000 @Pixel 7	13.30 ms	10.26 ms	8.90 ms

数学的正しさ
に
こだわりすぎない

要求される正確さを理解する

数学的正しさにこだわらない

円周率 は 3

で良い場合もある

必要な正確性以上を求めない

- 必要な計算精度は？
  - Int/Long/Float/Double/BigDecimal
  - 必要の無い精度まで計算しようとしていないか？

- ロジックの正確性
  - 例：距離の計算（ユークリッド距離/マンハッタン距離）
  - 結果に影響しない、差があっても許容される場合も多い

無駄な処理を省略する

意外とたくさん見つかる無駄

その計算本当にやる必要ある？

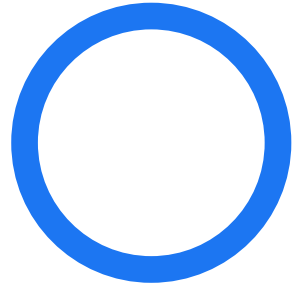
- 本質的に無駄になることをやっていないか？
  - AをBに変換してBをAに変換している！？
  - 最終的には使われない計算
  - 境界条件などを整理してみると無駄な処理がみつかるかも？
- 一度やれば良いことを繰り返しやっていないか？
  - ループ内で変化しない計算はループの外へ
  - 前処理を行うことで後段の処理の効率が上がる箇所がないか？
- 処理全体を俯瞰してみよう

無駄を省くために

- コードの可読性を大切にする
  - コードの可読性が悪いと、気づくべきことにも気づけない

高速化のため
可読性は**犠牲**にする

高速化のためにも
可読性は**大切**にする



可読性を担保する

メンテナンス性をあきらめない

高速化のためにも可読性は重要

- 高速化のために発生するいくつかの制限
  - mutableオブジェクト
  - インスタンスの使い回し・広いスコープ
  - 普段は使われない技巧的記述

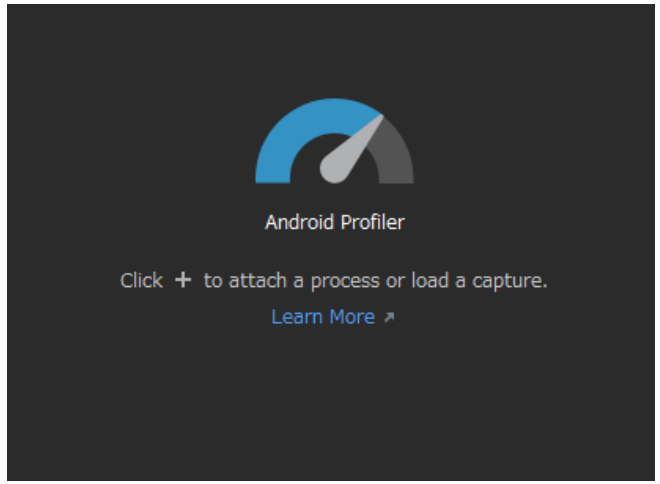
- カプセル化し影響範囲を最小限に
  - 小さなメソッドやクラスに切り出せるはず
  - 適切にカプセル化 & 理由をコメントに

計測して改善

ポイントを絞って効果的に

計測して改善

Android Profiler



printデバッグ

```
inline fun time(block: () -> Unit) {
    val start = System.nanoTime()
    block()
    val end = System.nanoTime()
    Log.d(TAG, "time: ${end - start}")
}
```

先入観にとらわれず、計測を元に改善を繰り返す

まとめ

まとめ

- Kotlinは遅くない・速くできる
- メモリは遅い、メモリアクセスの無駄を減らそう
- 高速化のためにも可読性が大切
- 最適化箇所は限定し、カプセル化する
- 計測重要：計測して事実に基づき改善

EOP

Thank you