Kotlin – Efficient collection processing

Liza Baby

Agenda

- **Iterable** vs **Sequence** Processing
  - **What is Sequence**
  - **Benefit of Sequence**
  - **When to use Sequence**
- **Inline Function**

Collections

- **Collections**
  - **Eg : List, Set, Map**
  - **Iterable**
    - **All collections are inherited from Iterable Interface and implements the Iterator function**

Iterable

```
interface Iterable<out T> {
    operator fun iterator(): Iterator<T>
}
```

Sequence

```
interface Sequence<out T> {
    operator fun iterator(): Iterator<T>
}
```

- Allows same set of functions performed on Iterable

Iterable

- **Multi step processing are executed eagerly**
- Intermediate function returns new collections
  - result of each operation is stored in a new collection
- process the whole collection on every step.
- Faster for smaller collections and few steps
- use Inline functions

Sequence

- **Multi step processing are executed lazily**
- Intermediate functions are not performed on the spot
  - All the operations are stored and evaluated at the end of terminal operation
- make whole processing for a single element, then for another etc.
- Faster for larger collections and multiple steps
- Intermediate functions are not inline, so has memory overhead

Iterable

- Multi step processing are executed eagerly
- **Intermediate function returns new collections**
  - result of each operation is stored in a new collection
- process the whole collection on every step.
- Faster for smaller collections and few steps
- use Inline functions

Sequence

- Multi step processing are executed lazily
- **Intermediate functions are not performed on the spot**
  - **All the operations are stored and evaluated at the end of terminal operation**
- make whole processing for a single element, then for another etc.
- Faster for larger collections and multiple steps
- Intermediate functions are not inline, so has memory overhead

Iterable

- Multi step processing are executed eagerly
- Intermediate function returns new collections
  - result of each operation is stored in a new collection
- **process the whole collection on every step.**
- Faster for smaller collections and few steps
- use Inline functions

Sequence

- Multi step processing are executed lazily
- Intermediate functions are not performed on the spot
  - All the operations are stored and evaluated at the end of terminal operation
- **make whole processing for a single element, then for another etc.**
- Faster for larger collections and multiple steps
- Intermediate functions are not inline, so has memory overhead

Iterable

- Multi step processing are executed eagerly
- Intermediate function returns new collections
  - result of each operation is stored in a new collection
- process the whole collection on every step.
- **Faster for smaller collections and few steps**
- use Inline functions

Sequence

- Multi step processing are executed lazily
- Intermediate functions are not performed on the spot
  - All the operations are stored and evaluated at the end of terminal operation
- make whole processing for a single element, then for another etc.
- **Faster for larger collections and multiple steps**
- Intermediate functions are not inline, so has memory overhead

Iterable

- Multi step processing are executed eagerly
- Intermediate function returns new collections
  - result of each operation is stored in a new collection
- process the whole collection on every step.
- Faster for smaller collections and few steps
- **use Inline functions**

Sequence

- Multi step processing are executed lazily
- Intermediate functions are not performed on the spot
  - All the operations are stored and evaluated at the end of terminal operation
- make whole processing for a single element, then for another etc.
- Faster for larger collections and multiple steps
- **Intermediate functions are not inline, so has memory overhead**

Inline functions for higher Order functions

- Using higher order functions imposes certain runtime penalties.
- Memory allocations (both for function objects and classes) and virtual calls introduce runtime overhead.
- overhead can be eliminated by inlining functions having the lambda expressions.

```
inline fun <T> Iterable<T>.filterInline(destination: ArrayList<T>, predicate: (T) -> Boolean) {
    for (element in this) if (predicate(element)) destination.add(element)
}
```

Keep in mind – Inline Functions

- Use inline functions where we need to use higher order functions to reduce memory overhead.
  - Don’t use for bigger functions as the code gets bigger during compilation
  - Use for generic functions. Don’t use class level variables and functions inside inline functions