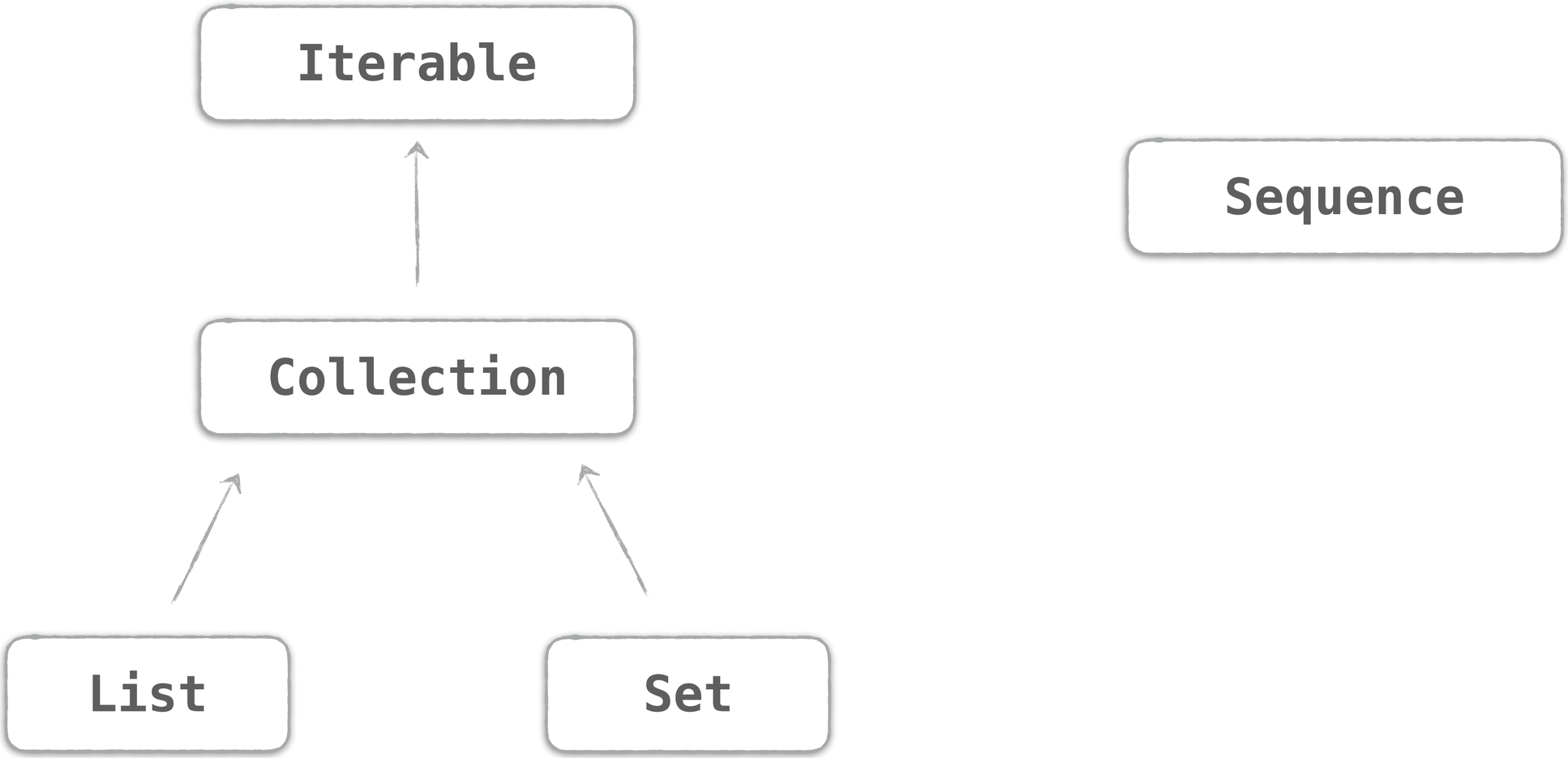
# Sequences

# Sequence

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

# Collections vs Sequences



# Extensions on sequences match extensions on collections

Intermediate operations

```
fun <T> Sequence<T>.filter(predicate: (T) -> Boolean): Sequence<T>
fun <T, R> Sequence<T>.map(transform: (T) -> R): Sequence<R>
```

Terminal operations

```
inline fun <T> Sequence<T>.any(predicate: (T) -> Boolean): Boolean
inline fun <T> Sequence<T>.find(predicate: (T) -> Boolean): T?
...
```

# Generating a sequence

```
generateSequence { Random.nextInt() }
```

```
val seq = generateSequence {
    Random.nextInt(5).takeIf { it > 0 }
}
println(seq.toList())
```

```
sample output: [4, 4, 3, 2, 3, 2]
```

# Reading input

```
val input = generateSequence {
    readLine().takeIf { it != "exit" }
}
println(input.toList())
```

```
>>> a
>>> b
>>> exit
[a, b]
```

# Generating an infinite sequence

```
val numbers = generateSequence(0) { it + 1 }
numbers.take(5).toList()                    [0, 1, 2, 3, 4]


// to prevent integer overflow:
val numbers = generateSequence(BigInteger.ZERO) {
    it + BigInteger.ONE
}
```

# How many times the phrase `"Generating element…"` will be printed?

```
val numbers = generateSequence(3) {
    n ->
    println("Generating element…")
    (n + 1).takeIf { it < 7 }
}
println(numbers.first())
```

1. `0`
2. `1`
3. `4`

# How many times the phrase "Generating element…" will be printed?

```
val numbers = generateSequence(3) {
    n ->
    println("Generating element…")
    (n + 1).takeIf { it < 7 }
}
println(numbers.first())    // 3
```

1. 0
2. 1
3. 4

# How many times the phrase "Generating element…" will be printed?

```
val numbers = generateSequence(3) {
    n ->
    println("Generating element…")
    (n + 1).takeIf { it < 7 }
}
println(numbers.first())    // 3

println(numbers.toList())
```

```
Generating element...
Generating element...
Generating element...
Generating element...
[3, 4, 5, 6]
```

# yield

# yield

```
val numbers = buildSequence {
    var x = 0
    while (true) {
        yield(x++)
    }
}
numbers.take(5).toList()  // [0, 1, 2, 3, 4]
```

# yield

```
buildSequence {
    yield(value)
    yieldAll(list)
    yieldAll(sequence)
}
```

# How many times the phrases starting with `yield` will be printed?

```
fun mySequence() = buildSequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}

println(mySequence()
        .map { it * it }
        .filter { it > 10 }
        .take(1))
```

# How many times the phrases starting with `yield` will be printed?

```
fun mySequence() = buildSequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}

println(mySequence()
        .map { it * it }
        .filter { it > 10 }
        .take(1))
```

0

# Intermediate operation

```
/**
 * Returns a sequence containing first [n] elements.
*/
fun <T> Sequence<T>.take(n: Int): Sequence<T>
```

```
println(mySequence()
        .map { it * it }
        .filter { it > 10 }
        .take(1))
```

no elements are yielded until the terminal operation is called

*kotlin.sequences.TakeSequence@f6f4d33*

# Building sequence in a lazy manner

```
fun mySequence() = buildSequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}

println(mySequence()
        .map { it * it }
        .filter { it > 10 }
        .first())
```

```
yield one element
yield a range
16
```

# Building sequence in a lazy manner

```
fun mySequence() = buildSequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}

println(mySequence()
        .map { it * it }
        .filter { it > 10 }
        .first())
```

```
yield one element
1
1
-
```

# Building sequence in a lazy manner

```
fun mySequence() = buildSequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}

println(mySequence()
        .map { it * it }
        .filter { it > 10 }
        .first())
```

```
yield one element
yield a range
1  3
1  9
-  -
```

# Building sequence in a lazy manner

```
fun mySequence() = buildSequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}
```

```
yield one element
yield a range
```

```
println(mySequence()
        .map { it * it }
        .filter { it > 10 }
        .first())
```

```
1  3  4
1  9  16
-  -  16
```

# Building sequence in a lazy manner

```
fun mySequence() = buildSequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}
```

```
yield one element
yield a range
16
```

```
println(mySequence()
        .map { it * it }
        .filter { it > 20 }
        .first())
```

```
1  3  4  …
1  9  16
–  –  16
16
```

# Building sequence in a lazy manner

```
fun mySequence() = buildSequence {
    println("yield one element")
    yield(1)
    println("yield a range")
    yieldAll(3..5)
    println("yield a list")
    yieldAll(listOf(7, 9))
}
```

won't be called

```
println(mySequence()
        .map { it * it }
        .filter { it > 20 }
        .first())
```

```
1   3   4   …
1   9   16
–   –   16
16
```

# Implement the function that builds a sequence of Fibonacci numbers

```
fun fibonacci(): Sequence<Int> = buildSequence {
    TODO()
}
```