Immutable lists

In [2]:
```
// constructor of list is `listOf(...)`
// returns an immutable list of type List<T>

var xs = listOf("a", "b", "c")
xs
```

Out[2]: [a, b, c]

In [3]:
```
// create new lists using operators
// but the original list remains unchanged.

xs + "hello"
```

Out[3]: [a, b, c, hello]

In [4]:
```
xs
```

Out[4]: [a, b, c]

In [5]:
```
// remove an element functionally
xs - "b"
```

Out[5]: [a, c]

In [6]:
```
xs
```

Out[6]: [a, b, c]

Mutable List

In [7]:
```
// `mutableListOf(...)` returns instances of `MutableList<T>`

val xs = mutableListOf("a", "b", "c")
xs
```

Out[7]: [a, b, c]

In [8]:
```
xs += "hello"
xs
```

Out[8]: [a, b, c, hello]

In [9]:
```
xs.remove("b")
xs
```

Out[9]: [a, c, hello]

Pairs

In [16]:
```
// Pair<S, T> is the constructor for pairs.

val xs = Pair("Hello", 1)
xs
```

Out[16]:
```
(Hello, 1)
```

In [17]:
```
// destructuring: bind components of pairs to symbols directly in the declarati

val (x, y) = xs
println("x = $x")
println("y = $y")
```

```
x = Hello
y = 1
```

Immutable Maps

In [31]:
```
// `mapOf(key to val, key to val, ...)` is the constructor
// of immutable maps of the type `Map<K,V>`.

val xs = mapOf<String, Int>("Jack" to 1, "Jill" to 2)
xs
```

Out[31]:
```
{Jack=1, Jill=2}
```

In [32]:
```
// `"Jack" to 1`
// is actually an object.method(arg) invocation:
// possible because
// `infix fun String.to(value: Any)`

"Jack".to(1)
```

Out[32]:
```
(Jack, 1)
```

In [33]:
```
// Check if a key exists in a map
"Jack" in xs
```

Out[33]:
```
true
```

In [34]:
```
"Joe" in xs
```

Out[34]:
```
false
```

In [35]:
```
xs["Jill"]
```

Out[35]:
```
2
```

In [36]:
```
xs["Joe"]
```

Out[36]: null

In [37]:
```
// Suppose xs:Map<K,V>
// xs.get(key: K): V?
xs.get("Jack")
```

Out[37]: 1

In [38]:
```
xs.get("Joe")
```

Out[38]: null

In [44]:
```
// get the value of "Jill", and convert the int value to float
val i = xs.get("Jill")
if(i != null) {
    println(i.toFloat())
}
```

```
2.0
```

In [47]:
```
// another way is to use the null-safe method invocation
println(xs.get("Jill")?.toFloat())
println(xs.get("Joe")?.toFloat())
```

```
2.0
null
```

In [49]:
```
// maps can treated as a list of pairs
xs.toList()
```

Out[49]: [(Jack, 1), (Jill, 2)]

In [50]:
```
// functional updates to maps
xs + ("Joe" to 2)
```

Out[50]: {Jack=1, Jill=2, Joe=2}

In [51]:
```
xs
```

Out[51]: {Jack=1, Jill=2}

Mutable maps

In [53]:
```
// the constructor is `mutableMapOf<K,V>(...)`

val xs = mutableMapOf<String, Int>()
xs
```

Out[53]: {}

In [54]:
```
xs.put("Jack", 0)
xs.put("Jill", 1)
xs
```

Out[54]: {Jack=0, Jill=1}

For-loops over collections

In [55]:
```
val xs = listOf("Jack", "Jill", "Joe")

for(x in xs) {
    println(x)
}
```

```
Jack
Jill
Joe
```

In [57]:
```
val grades = mapOf(
    "Jack" to 90,
    "Jill" to 95,
    "Joe" to 80
)

for((name, grade) in grades) {
    println("$name received a grade of $grade.")
}
```

```
Jack received a grade of 90.
Jill received a grade of 95.
Joe received a grade of 80.
```

In [69]:
```
val ranks = listOf("Jill", "Jack", "Joe")

for((i, name) in ranks.withIndex()) {
    println("$name has rank of ${i+1}." + if(i == 0) " Congrats" else "")
}
```

```
Jill has rank of 1. Congrats
Jack has rank of 2.
Joe has rank of 3.
```

Programming with higher order methods

In [70]:
```
// declare a helper function that converts day:Int to weekday:String
val dayOfWeek: (Int) -> String = {
    when(it) {
        1 -> "Monday"
        2 -> "Tuesday"
        3 -> "Wednesday"
        4 -> "Thursday"
        5 -> "Friday"
        6 -> "Saturday"
        7 -> "Sunday"
        else -> "Error"
    }
}
```

In [71]:
```
dayOfWeek(5)
```

Out[71]:
```
Friday
```

Map

In [73]:
```
listOf(1,2,3,4,5).map(dayOfWeek)
```

Out[73]:
```
[Monday, Tuesday, Wednesday, Thursday, Friday]
```

In [75]:
```
listOf(5,5,54,4,7,1,0).map(dayOfWeek)
```

Out[75]:
```
[Friday, Friday, Error, Thursday, Sunday, Monday, Error]
```

In [85]:
```
(1..20).map {
    (6..7).random()
}.map(dayOfWeek)
```

Out[85]:
```
[Saturday, Sunday, Sunday, Saturday, Saturday, Saturday, Sunday, Saturday, Sat
urday, Sunday, Sunday, Saturday, Saturday, Sunday, Sunday, Sunday, Sunday, Sat
urday, Sunday, Saturday]
```

In [87]:
```
// Map<K,V>.map((Pair<K,V>) -> T)
val grades = mapOf(
    "Jack" to 90,
    "Jill" to 95,
    "Joe" to 80,
)

grades.map {
    (name, grade) -> "$name got a grade of $grade"
}
```

Out[87]:
```
[Jack got a grade of 90, Jill got a grade of 95, Joe got a grade of 80]
```

ForEach method

In [89]:
```
grades.forEach {
    (name, grade) -> println("$name got a grade of $grade")
}
```

```
Jack got a grade of 90
Jill got a grade of 95
Joe got a grade of 80
```

Filter

In [90]:
```
// filter method takes a predicate function
// a predicate function is one that returns Boolean
```

```
(1..10).filter {
    it % 2 == 0
}
```

Out[90]: [2, 4, 6, 8, 10]

In [91]:
```
grades.filter {
    (name, grade) -> grade >= 90
}
```

Out[91]: {Jack=90, Jill=95}

In [93]:
```
// a fun example

(2..1000).filter {
    n -> (2 until n).filter { n % it == 0 }.size == 0
}
```

Out[93]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 7
3, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157,
163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 24
1, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337,
347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 43
3, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523,
541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 63
1, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733,
739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 83
9, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947,
953, 967, 971, 977, 983, 991, 997]
```

In [94]:
```
// using infix operator
infix fun Int.isFactorOf(number: Int) = (number % this == 0)
```

In [95]:
```
(2..1000).filter { n -> (2 until n).none {it isFactorOf n} }
```

Out[95]:
```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 7
3, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157,
163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 24
1, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337,
347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 43
3, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523,
541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 63
1, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733,
739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 83
9, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947,
953, 967, 971, 977, 983, 991, 997]
```

Aggregation (aka reduce)

In [99]:
```
(1..1_000_000).reduce {
    x,y -> x+y
}
```

Out[99]: 1784293664

In [105…
```
(1..10).map {
    dayOfWeek((1..7).random())
}.fold(0) {
    total, string -> total + string.length
}
```

Out[105]: 75

Scope function

https://kotlinlang.org/docs/scope-functions.html

In [107…
```
data class Address(
    var number: Int,
    var street: String,
    var city: String
)

data class Student(
    val name: String,
    var grade: Int,
    val address: Address,
)

val jack = Student(
    name="Jack",
    grade=80,
    address=Address(2000, "Simcoe Street North", "Oshawa")
)

jack
```

Out[107]: Student(name=Jack, grade=80, address=Address(number=2000, street=Simcoe Stree
t North, city=Oshawa))

In [110…
```
// Java-style

val formattedAddress = "${jack.address.number} ${jack.address.street}, ${jack.a
formattedAddress
```

Out[110]: 2000 Simcoe Street North, Oshawa

Scope function: let

- `object.let {...}`
- The caller object is the paramter to the anonymous function
- Evaluates to the return-value of the anonymous function

In [111…
```
// Kotlin style
jack.address.let {

```

```
    "${it.number} ${it.street}, ${it.city}"
}
```

Out[111]: 2000 Simcoe Street North, Oshawa

Scope function: run

- `object.run {...}`
- The caller is referred by `this` in the scope.
- Evaluates to the return-value of the anonymous function

In [113…
```
// Kotlin style
jack.address.run {
    "$number $street, $city"
}
```

Out[113]: 2000 Simcoe Street North, Oshawa

Scope function that updates the object: apply

- `object.apply {...}`
- Caller object appears as `this` inside the scope.
- Evaluates to the object

In [114…
```
jack.address
```

Out[114]: Address(number=2000, street=Simcoe Street North, city=Oshawa)

In [117…
```
// Java way
jack.address.number = 123
jack.address.street = "Bloor St."
jack.address.city = "Toronto"

jack.address
```

Out[117]: Address(number=123, street=Bloor St., city=Toronto)

In [118…
```
// Kotlin: using apply scope function
jack.address.apply {
    number = 123
    street = "Yonge Street"
    city = "New York"
}
```

Out[118]: Address(number=123, street=Yonge Street, city=New York)

In [ ]:
```

```