# Collections

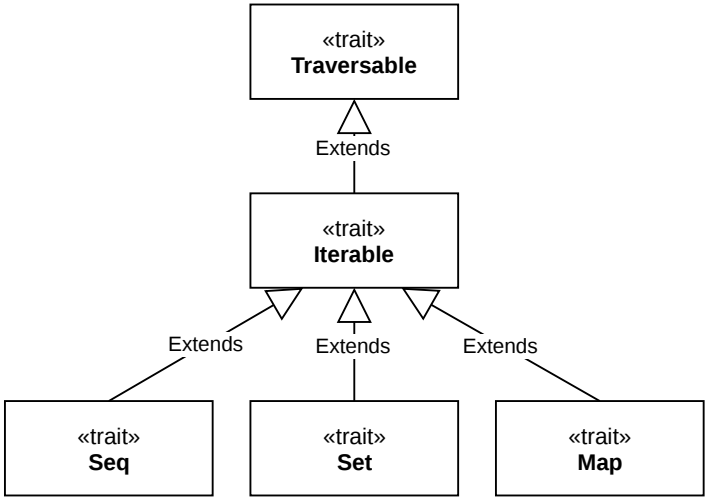
## A look at Scala's other common collections, and their performance trade-offs

# Agenda

1. Other Collections
2. Mutable vs Immutable
3. Consistent API for Collections
4. Other Sequences
5. The Mighty Vector
6. Sets
7. Maps
8. Concrete Implementations
9. Iterators, Views and Streams

# Other Collections

- Scala has a rich hierarchy of collections in addition to `List`




- Scala has three broad categories of collection:
  - `Seq` maintains order of insertion
  - `Set` maintains uniqueness but not order (though may be sorted)
  - `Map` is key -> value association, also unique by key

# Sequences (Performance)

| immutable | head | tail | apply | update | prepend | append | insert |
|---|---|---|---|---|---|---|---|
| List | C | C | L | L | C | L | - |
| Stream | C | C | L | L | C | L | - |
| Vector | eC | eC | eC | eC | eC | eC | - |
| Stack | C | C | L | L | C | L | L |
| Queue | aC | aC | L | L | C | C | - |
| Range | C | C | C | - | - | - | - |
| String | C | L | C | L | L | L | - |
| **mutable** | | | | | | | |
| ArrayBuffer | C | L | C | C | L | aC | L |
| ListBuffer | C | L | L | L | C | C | L |
| StringBuilder | C | L | C | C | L | aC | L |
| MutableList | C | L | L | L | C | C | L |
| Queue | C | L | L | L | C | C | L |
| ArraySeq | C | L | C | C | - | - | - |
| Stack | C | L | L | L | C | L | L |
| ArrayStack | C | L | C | C | aC | L | L |
| Array | C | L | C | C | - | - | - |

# Sets and Maps (Performance)

| immutable | lookup | add | remove | min |
|---|---|---|---|---|
| `HashSet/HashMap` | eC | eC | eC | L |
| `TreeSet/TreeMap` | Log | Log | Log | Log |
| `BitSet` | C | L | L | eC |
| `ListMap` | L | L | L | L |
| **mutable** | | | | |
| `HashSet/HashMap` | eC | eC | eC | L |
| `WeakHashMap` | eC | eC | eC | L |
| `BitSet` | C | aC | C | eC |
| `TreeSet` | Log | Log | Log | Log |

## Key

| Code | Description |
|---|---|
| **C** | Constant (fast) |
| **eC** | Effectively Constant |
| **aC** | Ammortized Constant |
| **Log** | Proportional to the log of the size |
| **L** | Proportional to the size |
| **-** | The operation is not supported. |

- https://docs.scala-lang.org/overviews/collections/performance-characteristics.html

# LinearSeq vs IndexedSeq

- `Seq` is further divided into two broad categories: `LinearSeq` and `IndexedSeq`
- `LinearSeq` is optimized for head-first, forward linear access
  - Default implementation in Scala is `List`
- `IndexedSeq` is optimized for random access
  - Default implementation in Scala is `Vector`
- Both `List` and `Vector` are immutable. These two form the most common choice
  - If you know you can work exclusively at the head (e.g. recursion) use `List`
  - For anything else, typically use `Vector`
- For sheer performance, particularly with primitives, you sometimes will use `Array`
  - But remember, `Array` is mutable and has no thread safety
  - Profile and prove a performance problem before using `Array`

# mutable vs immutable

- Along with the big 3 (`List`, `Vector` and `Array`) there are many other more specialized collections
- Many of these, e.g. `Set`, `Queue`, `Stack` have both mutable and immutable versions
- These are under `scala.collection.immutable` and `scala.collection.mutable` packages
- Best Practice, don't import directly from these, import the packages instead

```
import scala.collection.mutable
import scala.collection.immutable

def popImmutableQueue(q: immutable.Queue[Int]): (Int, immutable.Queue[Int]) = {
  q.dequeue
}

def popMutableQueue(q: mutable.Queue[Int]): Int = {
  q.dequeue()
}
```

# Consistent API

```
import scala.collection.immutable

val xs = List(1,2,3,4)              // List(1, 2, 3, 4)
val exs = List.empty[Int]           // List()

val v = Vector(1,2,3,4)             // Vector(1, 2, 3, 4)
val ev = Vector.empty[Int]          // Vector()

val q = immutable.Stack(1,2,3,4)    // Stack(1, 2, 3, 4)
val eq = immutable.Stack.empty[Int] // Stack()

val s = Set(1,2,3,4)
val es = Set.empty[Int]

q == xs    // true
q == v     // true
xs == v    // true
ev == es   // false
```

- Equality between `Seq` s works based on contents (**except Array** - use `.deep`)
- Consistent construction, empty, toString, etc.

# Easy Conversions

```
val arr = Array(1,2,3,4)
xs == arr
xs == arr.deep

xs.toVector             // Vector(1, 2, 3, 4)
xs.toArray              // Array(1, 2, 3, 4)
xs.toSet                // Set(1, 2, 3, 4)
xs.zipWithIndex.toMap   // Map(1 -> 0, 2 -> 1, 3 -> 2, 4 -> 3)
xs.toList                // List(1, 2, 3, 4)

xs.toQueue // compile error - no built in toQueue method

xs.to[immutable.Queue]  // Queue(1, 2, 3, 4)
```

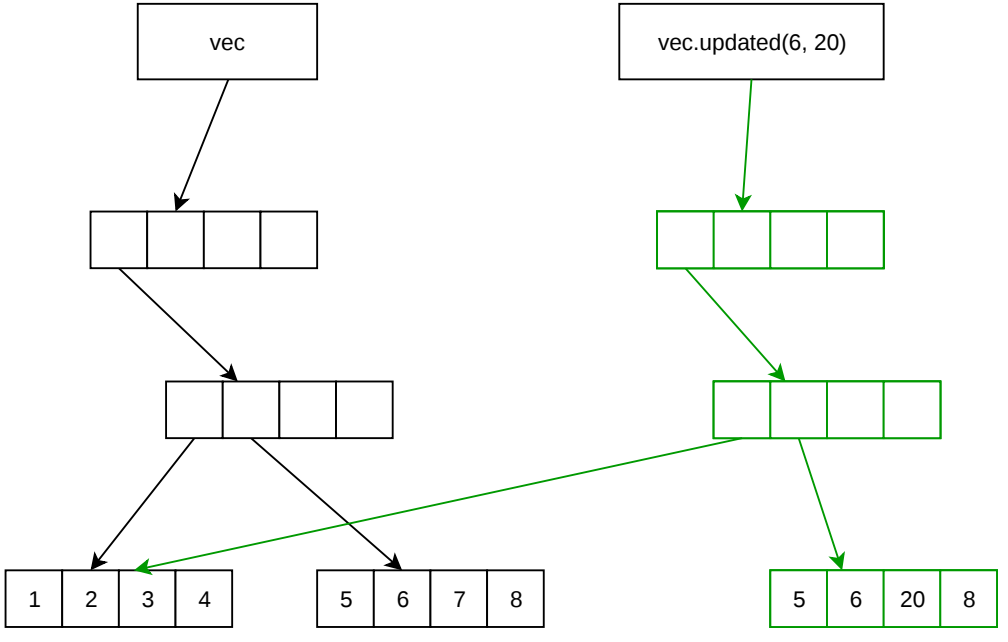
- `toList` on a `List` is a no-op (so you can call these with little-to-no overhead)

# Other Sequences (overview)

- **Queue** - FIFO - implemented as pair of Lists for immutable (one forward, one reverse)
- **Stack** - LIFO - can just use a List instead (exists for backwards compatibility)
- **Array** - Mutable and Random Access. Direct alias to Java Array. Supports primitives
- **Range** - An arithmetic progression, implemented lazily
- **Iterator**, **View** and **Stream** - Lazy sequences, Stream may be infinite. More on these later
- **Vector** - Random Access, effectively constant performance for all supported operations, clever memory re-use. We'll look at this next

# The Mighty Vector

- Example is Arity 4, the real Vector is Arity 32 making all operations `Log32` (or effectively constant)



# The Mighty Vector

- Vector is really Arity 32
- In `Int` addressable space, any size vector cell can be navigated to in at most 7 hops
- And even the largest vector will only need 7 * 32 words duplicated for a single cell update operation
- If you only work at the head of a collection, `List` is still a marginally better choice
- For anything else, you are best served going straight to `Vector` for an immutable, ordered sequence
- Also, the 32 word organization happens to be a size that the JVM manages very well

# Sets (immutable)

- Sets maintain unique identity, but not order

```
val vowels = Set('a', 'e', 'i', 'o', 'u')

vowels.contains('a')    // true
vowels.contains('t')    // false
vowels('a')             // true
vowels('t')             // false

vowels + 'y'            // Set(e, y, u, a, i, o)
vowels + 'e'            // Set(e, u, a, i, o)

val commonLetters = Set('e','t','a','o','i','n','s','r','h')

commonLetters intersect vowels  // Set(e, a, i, o)
commonLetters diff vowels       // Set(s, n, t, h, r)
vowels diff commonLetters       // Set(u)
commonLetters union vowels      // Set(e, s, n, t, u, a, i, h, r, o)

"hello to me".count(vowels) // 4
```

- `Set.apply` and `Set.contains` are equivalent
- `Set[T]` extends `T => Boolean` and can be used as a predicate

# Sorted and Mutable Sets

- Sets don't maintain insertion order, but can be sorted, e.g. `TreeSet`

```
import scala.collection.immutable
immutable.TreeSet('u', 'o', 'i', 'e', 'a')  // TreeSet(a, e, i, o, u)
```

- Mutable sets can be added to and removed from (of course)

```
import scala.collection.mutable

val vowelsMut = mutable.Set('a', 'e', 'i', 'o', 'u')

vowelsMut += 'y'         // Set(u, y, e, o, i, a)
vowelsMut('y')           // true

vowelsMut -= 'y'         // Set(u, e, o, i, a)
vowelsMut('y')           // false

vowelsMut('y') = true    // Set(u, y, e, o, i, a)
vowelsMut('y')           // true

vowelsMut('y') = false   // Set(u, e, o, i, a)
vowelsMut('y')           // false
```

# Maps

- Maps are Key -> Value associations where the keys are a Set
- Like Sets, Maps have both immutable and mutable implementations
- `Map[K, V]` extends function `K => V`

```
val numWords = Map(1 -> "one", 2 -> "two", 3 -> "three", 4 -> "four", 5 -> "five")

numWords(1)     // one   --  don't use this
numWords.get(1) // Some(one) -- use this
numWords.getOrElse(1, "?") // one -- or this

val nums = List(1,2,3,2,5)
nums.map(numWords)  // List(one, two, three, two, five)

for ((num, word) <- numWords) {
  println(s"$num -> $word")
}
// 5 -> five
// 1 -> one
// 2 -> two
// 3 -> three
// 4 -> four
```

# Sorted and Mutable Maps

- Like `Set`, there are `Maps` that maintain a sort order

```
val tm = immutable.TreeMap.empty[Int, String] ++ numWords
// Map(1 -> one, 2 -> two, 3 -> three, 4 -> four, 5 -> five)
```

- There is also a `ListMap` that does maintain insertion order, but performance is dismal
- `Maps` can be mutable too

```
val mm = mutable.Map.empty[Int, String] ++ numWords

mm -= 2           // Map(5 -> five, 4 -> four, 1 -> one, 3 -> three)
mm += 2 -> "two"  // Map(2 -> two, 5 -> five, 4 -> four, 1 -> one, 3 -> three)
```

# Maps - Key and Value Operations

```
numWords.keys      // Iterable[Int] = Set(5, 1, 2, 3, 4)
numWords.keySet    // Set[Int] = Set(5, 1, 2, 3, 4)
numWords.values    // MapLike.DefaultValuesIterable(five, one, two, three, four)

numWords.filterKeys(_ % 2 == 0)  // Map(2 -> two, 4 -> four)
numWords.mapValues(_.reverse)
// Map(5 -> evif, 1 -> eno, 2 -> owt, 3 -> eerht, 4 -> ruof)

numWords.transform { case (k, v) => s"$v($k)" }
// Map(5 -> five(5), 1 -> one(1), 2 -> two(2), 3 -> three(3), 4 -> four(4))
```

- You can also swap keys and values, but beware non-unique values

```
numWords.map(_.swap)
// Map(four -> 4, three -> 3, two -> 2, five -> 5, one -> 1)

val evens = (for (i <- 1 to 5) yield i -> (i % 2 == 0)).toMap
// Map(5 -> false, 1 -> false, 2 -> true, 3 -> false, 4 -> true)

evens.map(_.swap)
// Map(false -> 3, true -> 4)   -- oops
```

# Concrete Implementations, immutable

- `List`
- `Stream` - potentially infinite
- `Vector` - persistent immutable data structure with constant access time
- `Stack`
- `Queue`
- `Range`
- `String`
- Hash tries (`HashSet`, `HashMap`, `Set1..4`, `Map1..4`)
- `TreeSet`/`TreeMap`
- `BitSet`
- `ListMap`

# Concrete Implementations, mutable

- `ArrayBuffer`
- `ListBuffer`
- `StringBuilder`
- `Queue`
- `ArraySeq`
- `Stack`
- `ArrayStack`
- `Array`
- `HashSet` and `HashMap`
- `WeakHashMap`
- `BitSet`

# Iterators

- Lazy collection that returns a potentially different value on each `.next` call

```
val nums = List.range(1, 21)

val numsIter = nums.iterator  // get an iterator from any collection

if (numsIter.length > 0) numsIter.next()  // No such element exception!
```

- In this example, the call to `.length` exhausts the iterator
- Watch out for surprising outcomes like this
- Either stick to `.hasNext` and `.next()` or convert to another collection

# Views

- Lazy collection that stores up functions to run later on demand

```
val vec = Vector.range(0, 20)

val vecView = vec.view

def calcSquare(x: Int): Int = {
  println(s"Calculating for $x")
  x * x
}

val squaresView = vecView.map(calcSquare) // does nothing, yet

squaresView(2)  // calls calcSquare(2)
squaresView(4)  // calls calcSquare(4)
squaresView(2)  // calls calcSquare(2)

val squares = squaresView.force // forces eval of new eager collection

squares
// Vector(0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121,
//        144, 169, 196, 225, 256, 289, 324, 361)
```

# Stream

- Lazy, potentially infinite collection allowing custom implementations

```
val numsFromOne = Stream.from(1)  // infinite
// Stream[Int] = Stream(1, ?)

val firstTenNums = numsFromOne.take(10)  // stops after 10
// Stream[Int] = Stream(1, ?)

firstTenNums.toList
// List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

val factorial: Stream[BigInt] = 1 #:: factorial.zip(Stream.from(2)).
  map { case(a, b) => a * b }
// Stream[Int] = Stream(1, ?)

val firstTenFacs = factorial.take(10)
// Stream[Int] = Stream(1, ?)

firstTenFacs.toList
// List(1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800)
```

- Note that `Stream` is tricky, it memoizes. You must `drop` or `tail` to release earlier references and free up memory

# Exercises for Module 14

- Find the `Module14` class and follow the instructions to make the tests pass
- These exercises are a continuation of the problem started in Module 13