UNDERSTANDING COLLECTIONS, MAPS AND MORE IN DOMINO LANGUAGES

Paul Withers, HCL

in nomine patris
et filii
et spiritus sancti

Paul Withers

- Associate Director – Research, HCL
- @paulswithers
- LotusScript
- Java
- JavaScript (ES5, ES6, Node.js, TypeScript)
- VoltScript
- Rust

Agenda

- Arrays and Vectors
- Collections and Maps in Java
- Streams
- Collections Elsewhere

Arrays

- One-dimensional
- Zero-indexed
- Single type

```
int[] anArray = new int[] {1, 2, 3, 4, 5};
for (int i = 0; i < anArray.length; i++) {
    System.out.println(anArray[i]);
}

int[] anArray = new int[] {1, 2, 3, 4, 5};
for (int element : anArray) {
    System.out.println(element);
}
```

https://www.baeldung.com/java-arrays-guide

Arrays in LotusScript

- One-dimensional
- Zero-indexed *unless specified*
- Single type
- Fixed arrays
  - Maximum size of 32k for scope
  - Size allocated varies on datatype
  - Size allocated varies for 32-bit and 64-bit
- Dynamic arrays
  - Need re-dimming

Vectors

- "Old" Collection classes in Java
  - Baeldung – "*Vector* is present in the earlier versions of Java as a legacy class"
  - The only Collection class in lotus.domino
- Apparently no vector class in C, but are in C++
- Vectors are standard in Rust
- Vector in Java != Vector in Rust
  - It's just the same name and concept
  - Implementations will vary

Vectors vs ArrayList

- Vectors are synchronized, ArrayList not
  - Not the only collection type for synchronized
- Vectors double when resize, ArrayList increases by 50%
- Vectors – Iterators and Enumeration
- ArrayList – Iterators only
- Vectors slower than ArrayLists
  - Because of synchronization
- ArrayLists added in Java 1.2

https://www.baeldung.com/java-arraylist-vs-vector

Vector Enumeration

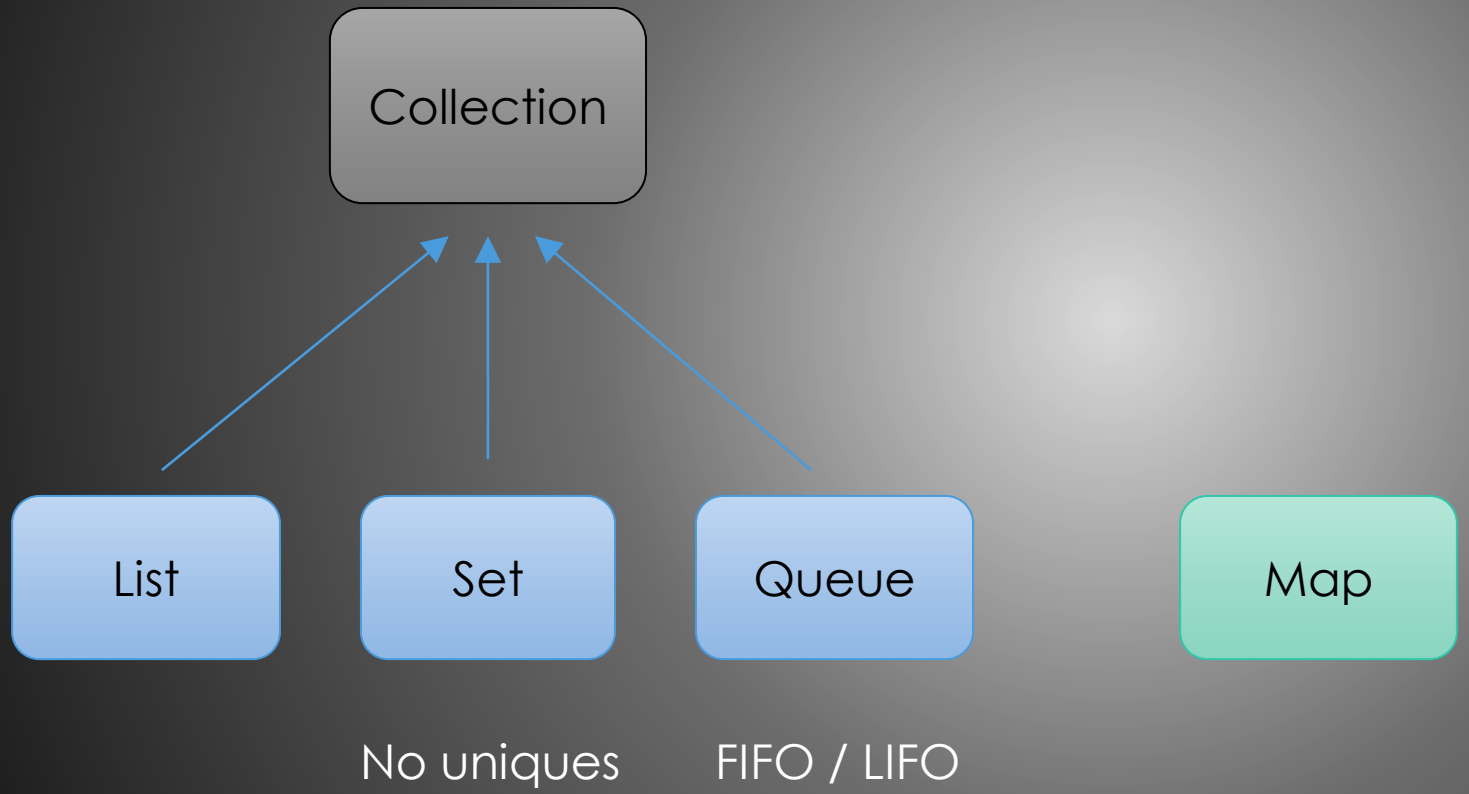
```
Iterator<String> iterator = vector.iterator();
while (iterator.hasNext()) {
    String element = iterator.next();
    // ...
}

Enumeration e = vector.elements();
while(e.hasMoreElements()) {
    String element = e.nextElement();
    // ...
}
```

Java Collections and Maps

- My blog post - https://www.intec.co.uk/mapping-java-collections/
- Source blog post - https://www.janeve.me/software-programming/which-java-collection-to-use
- PDF - https://www.janeve.me/wp-content/uploads/2018/01/Java-Collection-Matrix-Java-Collection-Matrix.pdf
- Baeldung - https://www.baeldung.com/java-choose-list-set-queue-map

Collections / Maps



ArrayList

- Typical collection type
- Insertion order
- Duplicates
- Good at adding / removing at end
- Good at getNth access

```
List<String> coll = new ArrayList<>();
coll.add(“Hello”);
coll.add(“World”);
```

HashMap

- Typical Map type
- Unordered
- Key/Value pairs
  - No duplicates

```
Map<String, String> coll = new HashMap<>();
Map.put(“Hello”, “World”);
```

Collections

- No keys
  - If you want keys, you need a Map
- Insertion order
  - Lists
- FIFO / LIFO
  - Queues
- Unique entries
  - Sets

Lists

- ArrayList
  - add(e)
  - add(index, e)
  - get(index)
- LinkedList
  - addFirst(e), addLast(e)
  - getFirst(e), getLast(e)
  - removeFirst(), removeLast()
  - pop(), push(e)
  - Better performance for adding / removing at any position
  - Worse random access

Sets

- HashSet
  - Random traversal
- LinkedHashSet
  - Insertion order
  - HashSet with LinkedList to hold position
- TreeSet
  - Sorted
  - Can’t put nulls in
- EnumSet
  - Best for sets of enum values
  - Enums = fixed set of options for parameters

Queues

- LinkedList
  - Acts as queue as well as list
- ArrayDeque
  - Deque = double-ended queue
  - Faster than LinkedList

Maps

- HashMap
  - Default choice
- LinkedHashMap
  - If insertion order is important
- TreeMap
  - Sorted – can only sort on key
- WeakHashMap
  - Keys that are not referenced become eligible for garbage collection

Concurrency

- Only required for cross-thread use, e.g. OSGi plugins
- ConcurrentSkipListSet
  - Thread-safe version of TreeSet
- ConcurrentHashMap
  - Thread-safe version of HashMap
- ConcurrentSkipListMap
  - Thread-safe version of TreeMap
- ConcurrentLinkedDeque
  - Thread-safe version of LinkedList

Java 8 Streams

- Collections and Maps can be streamed
- Three steps
  - Datasource
  - Intermediate operations (0…n)
  - Terminal operation
- Intermediate operations are lazy
- Terminal operation initiates the pipeline

Datasource

- `myList.stream()`
- `myMap.keySet().stream()`
- `myMap.entrySet().stream()`

Intermediate Operations

- `filter(Predicate<? super T> predicate)`
  - Strips entries from the stream
- `distinct()`
  - Calls `Object.equals(object)` to get uniques
- `map(Function<? super T,? extends R> mapper)`
  - Converts entries in the stream
- `flatMap(Function<? super T,? extends Stream<? extends R>> mapper)`
  - Flattens a collection of collections into a single "flat" collection
- Lambdas or double-colon syntax

Final Functions

- `forEach(Consumer<? super T> action)`
  - Loop through each performing the action
- `collect(Collector<? super T,A,R> collector)`
  - Return a Collection using a Collector
- `count()`
  - Return the number of elements
- `reduce(BinaryOperator<T> accumulator)`
  - Merges elements in the collection into a single object
- `toArray()`
  - Returns an array

Sample Stream

```
public Integer getOptsTot() {
      return opts.stream()
            .map(MessageBoxOption::getValue)
            .reduce(0, Integer::sum);
}
```

Datasource

Intermediate operation

Final function

Java 8 Streams Debugging

- peek(Consumer<? super T> action)
  - Still needs a terminal operation
  - Can also perform an action

```
Stream.of("one", "two", "three", "four")
   .filter(e -> e.length() > 3)
   .peek(e -> System.out.println("Filtered value: " + e))
   .map(String::toUpperCase)
   .peek(e -> System.out.println("Mapped value: " + e))
   .collect(Collectors.toList());
```

https://www.baeldung.com/java-streams-peek-api

Java 8 Streams Debugging

- Call separate function
- Add breakpoints accordingly

- Or multi-line lambda
  - Requires curly braces around lambda function

JavaScript

- Arrays
- Objects
- ES6
  - Set
  - Map

ES6 Set

- Unique values only
- add()
- delete()
- has()
- forEach() – callback for each element
- values() – iterator with all values in the set

ES6 Map

- Remembers insertion order
- Not restricted to specific datatype
- set()
- get()
- delete()
- has()
- forEach() – callback for each key/value pair in the map
- entries() – iterator with [key, value] in the map

Rust Collections

- std::collections module
- Sequences
  - Vec
  - VecDeque
  - LinkedList
- Maps
  - HashMap
  - BTreeMap
- Sets
  - HashSet
  - BTreeSet

LotusScript Lists

- Like Java maps
- Very performant
- Key can only be a string
- Remember to Erase the List at the end of processing

LotusScript

- Devin Olson
  - Lists and Collections
- Andre Guirard
  - Large arrays
  - Queue data structure
  - Stack data structure

VoltScript

- VoltScript Collections
  - Pair
  - Collection
    - Like LinkedList / HashSet / TreeSet
    - Insertion order by default
    - Can be ordered with custom comparator
    - Can be forced unique
    - Can iterate, get by position, get and remove first / last
  - Map
    - Like LinkedHashMap / TreeMap
    - Similar options to Collection
- Simplicity and flexibility vs optimal performance

VoltScript

- CollectionFilter / MapFilter
  - Allows custom filtering
  - Returns filtered collection
    - Allows streaming
- CollectionTransformer / MapTransformer
  - Allows conversion of content
- Map.collectKeys and .collectValues
  - Converts keys / values to a Collection

Links

- https://www.baeldung.com/java-choose-list-set-queue-map
- https://www.janeve.me/software-programming/which-java-collection-to-use
- https://www.baeldung.com/java-8-new-features
- Big O notation https://www.baeldung.com/java-algorithm-complexity
- Java lambdas https://www.baeldung.com/java-8-functional-interfaces
- Lambda best practices https://www.baeldung.com/java-8-lambda-expressions-tips#short
- Java Streams https://www.baeldung.com/java-streams