`IntList`

An **efficient list of integers** in *Java*

Boxing and **Unboxing**

Implicit **Boxing** and **Unboxing**

```
final Integer boxed = 1;
// Implicit boxing

int unboxed = boxed;
// Implicit unboxing
```

Boxing

```
final Integer a1 = Integer.valueOf(10_000);
final Integer a2 = Integer.valueOf(10_000);

assert a1.equals(a2); // Same value
assert a1 != a2; // Different object reference

// New instance every time!
```

Boxing (not so naive)

```java
final Integer a1 = Integer.valueOf(5);
final Integer a2 = Integer.valueOf(5);

assert a1.equals(a2); // Same value
assert a1 == a2; // Same object reference

// Preallocated for values between -128 to 127
```

Unboxing

```
final Integer boxed = Integer.valueOf(5);

final int unboxed = boxed.intValue();
// Dereferencing every time!
```

Unboxing `null` 💣

```java
final Integer boxed = null;

try {
    final int unboxed = boxed.intValue(); // BOOM!
} catch (NullPointerException e) {
    assert true : "Unboxing null should fail";
}
```

The Problem with `List<Integer>`

Boxes and Unboxes All The Time 👎

```
final ArrayList<Integer> list = new ArrayList<>();
list.add(1); // Boxing
list.add(2); // Boxing
list.add(3); // Boxing

final int item = list.get(0); // Unboxing
```

Accepts null 👎

```
final ArrayList<Integer> list = new ArrayList<>();
list.add(null);

try {
    final int item = list.get(0);
} catch (final NullPointerException e) {
    assert true : "Should fail";
}
```

Mutable by Default, **Unmodifiable** at Best 👎

```java
final List<Integer> mutableList = new ArrayList<>();
mutableList.add(1); // Mutation
mutableList.add(2); // Mutation

final List<Integer> unmodifiableList = Collections.unmodifiableList(mutableList);

try {
    unmodifiableList.add(3); // BOOM! Mutation fails at runtime
} catch (UnsupportedOperationException e) {
    assert true : "Should fail";
}
```

Cumbersome API 👎

```java
public static class Lists {
    public static List<Integer> concat(
            List<Integer> list1,
            List<Integer> list2,
            List<Integer> list3) {

        final ArrayList<Integer> buffer = new ArrayList<>(list1);
        // Defensive copy

        buffer.addAll(list2); // Does not chain
        buffer.addAll(list3); // Does not chain

        return Collections.unmodifiableList(buffer);
        // Wrap to make unmodifiable
    }
}
```

Poor Memory and CPU Efficiency 👎

- **Boxing**
  - Object creations (`new`) 👎
- **Unboxing**
  - Dereferencing 👎
- **`Integer` objects**
  - Scattered in the *heap*, defeats CPU cache 👎
  - Not memory efficient 👎

Builder Pattern against Boilerplatyness

```
final List<Integer> list1 = ListBuilder.<Integer>builder()
        .add(1)
        .add(2)
        .addAll(List.of(3, 4, 5, 6))
        .build();

final List<Integer> list2 = ListBuilder.toBuilder(List.of(1, 2, 3))
        .add(4)
        .addAll(List.of(5, 6))
        .build();
```

`ListBuilder`, some amount of work...

```java
public class ListBuilder<T> {
    private final List<T> buffer; // Working buffer

    private ListBuilder(List<T> buffer) {
        this.buffer = buffer;
    } // ...

    public static <T> ListBuilder<T> builder() {
        final ArrayList<T> buffer = new ArrayList<>();
        return new ListBuilder<>(buffer);
    }

    public static <T> ListBuilder<T> toBuilder(List<T> list) {
        final ArrayList<T> buffer = new ArrayList<>(list); // Defensive copy
        return new ListBuilder<>(buffer);
    }
}
```

`ListBuilder`, even more work…

```
public class ListBuilder<T> { // ...
    public ListBuilder<T> add(T element) {
        buffer.add(element);
        return this; // Allows chaining
    }

    public ListBuilder<T> addAll(List<T> elements) {
        buffer.addAll(elements);
        return this; // Allows chaining
    }

    public List<T> build() {
        final ArrayList<T> result = new ArrayList<>(buffer); // Defensive copy
        return Collections.unmodifiableList(result); // Wrap to make unmodifiable
    } // ...
}
```

Array to the Rescue! Not so...

Very Memory and CPU Efficient 👍

- No boxing 👍
- No unboxing 👍
- Contiguous storage in memory 👍

Only Mutable and Non-Extensible 👎

```java
final int[] array = {1, 2, 3};

array[0] = 10; // Non uniform syntax ([])

// Appending an element at the end of an array
final int[] modifiedArray = new int[array.length + 1]; // Non uniform syntax (length)
System.arraycopy(array, 0, modifiedArray, 0, array.length); // Not a method of array
modifiedArray[modifiedArray.length - 1] = 40;
```

Broken `toString` 👎

```
final int[] array = {1, 2, 3};

final String wrong = array.toString();
System.out.println(wrong);
// [I@615db445

final String result = Arrays.toString(array);
System.out.println(result);
// [1, 2, 3]
```

Broken equals 👎

```
final int[] array1 = {1, 2, 3};
final int[] array2 = {1, 2, 3};

boolean wrong = array1.equals(array2);
assert !wrong : "Broken";

final boolean result = Arrays.equals(array1, array2);
assert result : "OK";
```

Broken hashCode 👎

```java
final int[] array1 = {1, 2, 3};
final int[] array2 = {1, 2, 3};

boolean wrong = array1.hashCode() == array2.hashCode();
assert !wrong : "Broken";

final boolean result = Arrays.hashCode(array1) == Arrays.hashCode(array2);
assert result : "OK";
```

Very Poor API 👎

```
public class MoreArrays {
    public static int[] concat(
            int[] array1,
            int[] array2,
            int[] array3) {

        int[] result = new int[array1.length + array2.length + array3.length];
        System.arraycopy(array1, 0, result, 0, array1.length);
        System.arraycopy(array2, 0, result, array1.length, array2.length);
        System.arraycopy(array3, 0, result, array1.length + array2.length, array3.length);
        return result;
    }
}
```

Apache Commons, maybe not...

```java
final int[] list1 = {1, 2, 3};
final int[] array2 = {5, 6};
final int[] array3 = {7, 8, 9};

final int[] step1 = ArrayUtils.add(list1, 4); // Allocates an array and copies
final int[] step2 = ArrayUtils.addAll(step1, array2); // Allocates an array and copies
final int[] step3 = ArrayUtils.addAll(step2, array3); // Allocates an array and copies
final int[] result = step3;

// Not very efficient (redundant allocations and copies)
// Poorly legible (parenthesis nesting)
// Very underfeatured
```

`IntList`

Simple **and** efficient?

Phase 1

Experimental API

Reaching for developper experience

Experimental API

- Limited number of features
- Simplified implementation
- Not necessarily fully correct
- To assess developer experience

Immutable API for Correctness (`IntList`)

```
final IntList numbers = IntList.of(9, 5, 4);
// Immutable list of numbers

final IntList modifiedNumbers = numbers.swap(0, 2);
// A new immutable list of numbers where 1st and 3rd items have been swapped
// Initial list remains unchanged.
```

But it can be inefficient...

```
final  IntList numbers = IntList.of(6, 5, 4);

final IntList modifiedNumbers = numbers
        .swap(0, 2) // Returns a new immutable instance
        .append(7) // Returns a new immutable instance
        .prependAll(IntList.of(1, 2, 3)) // Returns a new immutable instance
        .map(i -> i * 10); // Returns a new immutable instance
```

Mutable API for Efficiency (`IntList.Builder`)

```
final IntList numbers = IntList.of(6, 5, 4);

final IntList modifiedNumbers = numbers.toBuilder() // Returns a new mutable builder
        .swap(0, 2) // Performs changes in-place on the mutable builder
        .append(7) // Performs changes in-place on the mutable builder
        .prependAll(IntList.of(1, 2, 3)) // Performs changes in-place on the mutable builder
        .map(i -> i * 10) // Performs changes in-place on the mutable builder
        .build(); // Returns a new immutable instance
```

Practice : **Exploring API**

Generate a list of numbers from 1 to 20 (`rangeClosed`),
beginning with 1 (`prepend`),
ending with 19 and 20 (`appendAll`),
and all mixed in-between (`shuffle`)

- Implement with **immutable API**
- Implement with **mutable API**

Phase 2

Preliminary Implementation

Reaching for efficiency

A **Buffer** for the `Builder`

- **Goal**
  - Perform changes in-place on the **buffer**
  - Extend buffer capacity when not enough space
  - While minimizing **moves** and **reallocations** (implying recopies) for performance
- **Several attempts**
  - **Right expansion** buffer (trailing buffer) 😐
  - **Left and right expansion** buffer (leading and trailing buffer) 😄
  - **Circular** buffer 😰

Leading and **Trailing** Buffer

```
public class Builder {
    private int[] buffer;
    private int start;
    private int end;

    public int size() { return end - start; }
    public int capacity() { return buffer.length; }
    public int leadingCapacity() { return start; }
    public int trailingCapacity() { return buffer.length - end; }
    public int freeCapacity() { return start + buffer.length - end; }

    private void ensureLeadingCapacity(int required) { /* ... */ }
    private void ensureTrailingCapacity(int required) { /* ... */ }
}
```

Implementing Methods

```
public class Builder {
    public Builder prepend(int value) {
        ensureLeadingCapacity(1);
        buffer[start - 1] = value;
        start--;
        return this;
    }

    public Builder append(int value) {
        ensureTrailingCapacity(1);
        buffer[end] = value;
        end++;
        return this;
    }
}
```

Informing the `Builder` of Future Intent

```
final IntList numbers = IntList.of(6, 5, 4);

final IntList modifiedNumbers = numbers.toBuilder(3, 1)
        // Leading capacity 3, trailing capacity 1
        .swap(0, 2)
        .append(7)
        // Trailing buffer is used, no reallocation nor move is performed
        .prependAll(IntList.of(1, 2, 3))
        // Leading buffer is used, no reallocation nor move is performed
        .map(i -> i * 10)
        .build();
```

Practice : **Exploring Optimizing API**

Generate a list of numbers from 1 to 20 (`rangeClosed`),
beginning with 1 (`prepend`),
ending with 19 and 20 (`appendAll`),
and all mixed in-between (`shuffle`)

- Implement with **optimizing mutable API** (`build` with parameters)

Phase 3

Performance Testing

Assessing efficiency

The Challenge with **Microbenchmarks**

- Easy to manipulate
- Easy to misinterpret
- Benchmarking on JVM is full of pitfalls

JMH

- ***JMH*** stands for **Java Microbenchmarking Harness**
- Accounts for JVM optimizations and other pitfalls
- Provides accurate, reproducible benchmarking results
- Supports various modes of benchmarking, including **throughput**, **average time**, and **sample time**

Setup

```
public class IntListBenchmark {
    public static class Append {
        // ...
    }

    public static void main(String[] args) throws RunnerException {
        final Options options = new OptionsBuilder()
                .include(IntListBenchmark.Append.class.getSimpleName())
                .forks(1)
                .build();

        new Runner(options).run();
    }
}
```

Benchmarks

```
public class IntListBenchmark {
    public static class Append {
        @Benchmark
        @BenchmarkMode(Mode.Throughput)
        public void list() { /* ... */ }

        @Benchmark
        @BenchmarkMode(Mode.Throughput)
        public void array() { /* ... */ }

        @Benchmark
        @BenchmarkMode(Mode.Throughput)
        public void intlist() { /* ... */ }
        }
    } // ...
}
```

Benchmarking `List<Integer>`

```
@Benchmark
@BenchmarkMode(Mode.Throughput)
public void list() {
    final List<Integer> initial = List.of(1, 2, 3);
    final List<Integer> buffer = new ArrayList<>(initial);

    for (int i = 4; i <= 10; i++) {
        buffer.add(i);
    }

    final List<Integer> result = Collections.unmodifiableList(buffer);
}
```

Benchmarking `int[]`

```
@Benchmark
@BenchmarkMode(Mode.Throughput)
public void array() {
    int[] result = new int[]{1, 2, 3};

    for (int i = 4; i <= 10; i++) {
        result = ArrayUtils.add(result, i);
    }
}
```

Benchmarking `IntList`

```java
@Benchmark
@BenchmarkMode(Mode.Throughput)
public void intlist() {
    final IntList initial = IntList.of(1, 2, 3);
    final IntList.Builder builder = initial.toBuilder(0, 10);

    for (int i = 4; i <= 10; i++) {
        builder.append(i);
    }

    final IntList result = builder.build();
}
```

Benchmark Results

Benchmark	Mode	Cnt	Score		Error	Units
IntListBenchmark.Append.list	thrpt	5	5728576,691	±	1758171,235	ops/s
IntListBenchmark.Append.array	thrpt	5	5103144,599	±	473447,116	ops/s
IntListBenchmark.Append.intlist	thrpt	5	15538682,673	±	5607228,935	ops/s

Practice : **Benchmarking APIs**

Generate a list of numbers from 1 to 20 (`rangeClosed`),
beginning with 1 (`prepend`),
ending with 19 and 20 (`appendAll`),
and all mixed in-between (`shuffle`)

- Benchmark **immutable API** implementation
- Benchmark **mutable API** implementation
- Benchmark **optimizing mutable API** implementation

Phase 4

Scaling Implementation

Reaching for simplicity

Mirrored **Immutable** and **Mutable API**

- **Mutable API**
  - Implemented in `IntList.Buffer`
  - Focus on **efficiency**
  - **Actual implementation** for methods
- **Immutable API**
  - Implemented in `IntList`
  - Focus on **ease of use**
  - Most methods perform **delegation** to corresponding method in `IntList.Buffer`

Transformation method

```
public class IntList {
    public IntList set(int index, int value) {
        return toBuilder() // Create a mutable builder from this immutable IntList
                .set(index, value) // Let the builder perform the transformation on itself
                .unsafeBuild(); // Create an immutable IntList from this mutable builder
    }

    public static class Builder {
        public Builder set(int index, int value) {
            // Actual implementation
            // This is where the transformation is actually performed by mutating this mutable builder.
            return this; // Return this builder
        }
    }
}
```

Query method

```java
public class IntList {
    public boolean contains(int value) {
        return unsafeToBuilder() // Create a mutable builder from this immutable IntList
                .contains(value); // Let the builder perform the computation and return the result
    }

    public static class Builder {
        public boolean contains(int value) {
            // Actual implementation
            // This is where the computation is actually performed
        }
    }
}
```

Phase 5

Testing Strategy

Reaching for correctness

Testing

- **Example-based testing** with ***jqwik*** (maybe ***JUnit 5*** in the future)
  - Typical test case
  - Assert conditions that should apply to the test case
- **Property testing** with ***jqwik***
  - Large number of generated test cases
  - Assert conditions (called **properties**) that should apply to all these test cases
- **Statefull property testing** with ***jqwik***
  - Property testing generalized to test subject with mutable state
  - Too deep for today
  - Still experimental

Example-Based Testing

```
@Group
class IntListTest { // ...
    @Group
    class IntListFeatures { // ...
        @Example
        void appendAll() {
            assertThat(IntList.of(10, 20, 30).appendAll(IntList.of(31, 32, 33)))
                    .isEqualTo(IntList.of(10, 20, 30, 31, 32, 33));
        }

        @Example
        void prependAll() {
            assertThat(IntList.of(10, 20, 30).prependAll(IntList.of(1, 2, 3)))
                    .isEqualTo(IntList.of(1, 2, 3, 10, 20, 30));
        }  // ...
    } // ...
}
```

Property Testing

```
@Group
class IntListTest { // ...
    @Group
    class IntListProperties { // ...
      @Property
      void appendAll_prependAll(
              @ForAll("intList") IntList as,
              @ForAll("intList") IntList bs) {

        final IntList l1 = as.appendAll(bs);
        final IntList l2 = bs.prependAll(as);
        assertThat(l1).isEqualTo(l2);
      } // ...
    } // ...
}
```

Generators for **Property Testing**

```
@Group
class IntListTest { // ...
    @Group
    class IntListProperties { // ...
    } // ...

    @Provide("intList")
    Arbitrary<IntList> intList() {
        return integer()
                .array(int[].class)
                .ofMinSize(0)
                .ofMaxSize(100)
                .map(IntList::of);
  } // ...
}
```

Contributing

Many ways to contribute

- Feedback about a feature
- Proposal for a feature
- Prototype
- Documentation
- Code

Practice : **Implementing Features**

Inspire from *Vavr* `Vector` API

- `.removeFirst(IntPredicate)` **/** `.removeLast(IntPredicate)`
- `.reject(IntPredicate)`
- `.rotateLeft(int)` **/** `.rotateRight(int)`
- `.distinct()`
- `.indexWhere(IntPredicate)` **/** `.lastIndexWhere(IntPredicate)`
- `.count(IntPredicate)` **/** `.forAll(IntPredicate)` **/** `.exists(IntPredicate)`
- `.sum()` **/** `.product()`