Is Functional Programming Worth it?

Correctness, Performance & Complexity

Hello

John McClean architect @Oath

Maintainer of cyclops-react

2nd visit to St. Petersburg

Agenda

- Correctness
- Collections types
  - mutable
  - immutable
  - persistent
- Choosing between collection types
- Conversions
- Is Functional Programming Worth It?

Correctness?

Informally

An algorithm is correct if it is error free

Errors

- Compile time errors
- Runtime errors
- Logic errors

Functional Programming

- Compile time errors
- ~~Runtime errors~~
- Logic errors

Why do FP?

Why do FP?

Compile time correctness

Collection Types

mutable collections

Correctness, performance, complexity?

Mutable Data Structures

Definition	Correctness	Performance	Complexity
Collections that can be mutated in place			

```java
public class MutableArrayList<E> {

    private int size;
    private E[] elementData;



    public E set(int index, E e) {
        E oldValue = elementData[index];
        elementData[index] = e;
        return oldValue;
    }


}
```

Mutable Data Structures

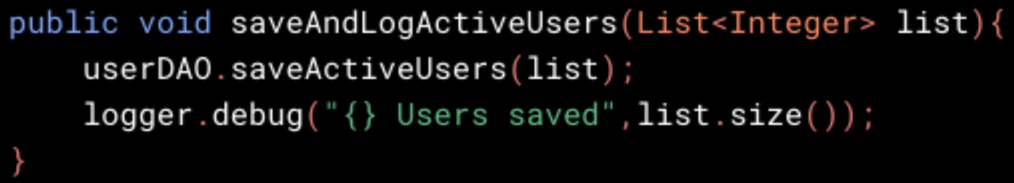
Definition	Correctness	Performance	Complexity
Collections that can be mutated in place			✅ Generally Simple

Time Complexity

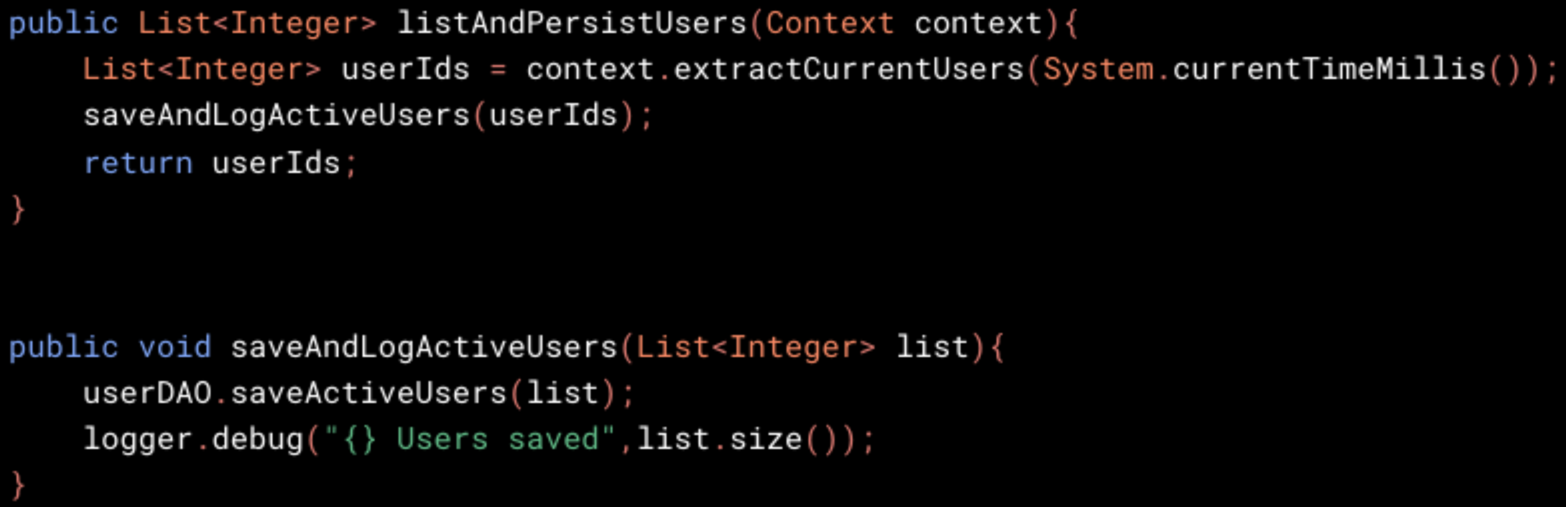
Type / Op.	**get**	**update**	**append**	**delete**
Mutable	O(1)	O(1)	O(1) / O(n)	O(1) / O(n)

Mutable Data Structures

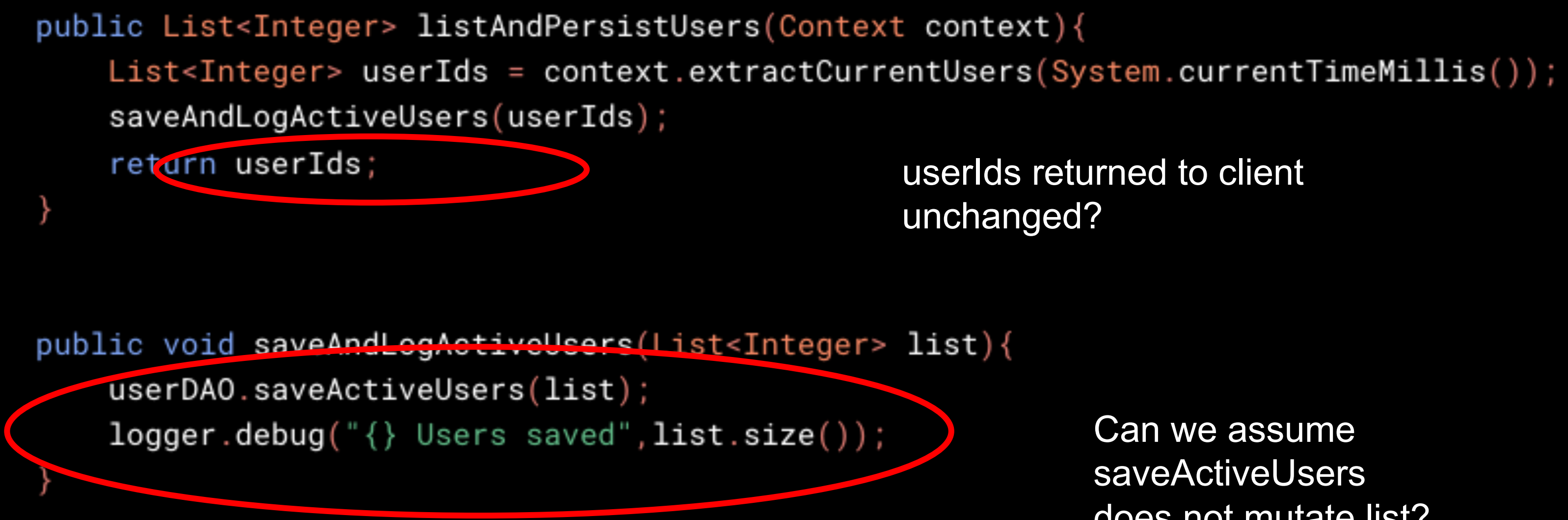
Definition	Correctness	Performance	Complexity
Collections that can be mutated in place		✅ Generally good	✅ Generally Simple



```java
public void saveAndLogActiveUsers(List<Integer> list){
    userDAO.saveActiveUsers(list);
    logger.debug("{} Users saved",list.size());
}
```



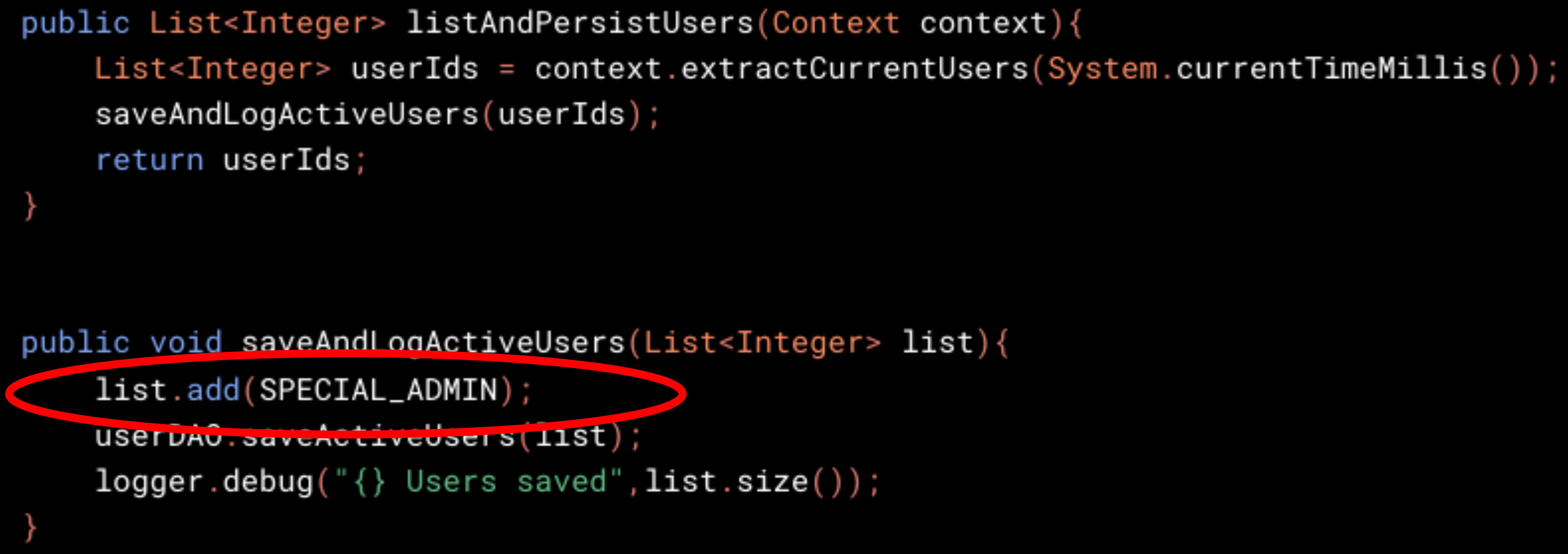
```
public List<Integer> listAndPersistUsers(Context context){
    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());
    saveAndLogActiveUsers(userIds);
    return userIds;
}


public void saveAndLogActiveUsers(List<Integer> list){
    userDAO.saveActiveUsers(list);
    logger.debug("{} Users saved",list.size());
}
```



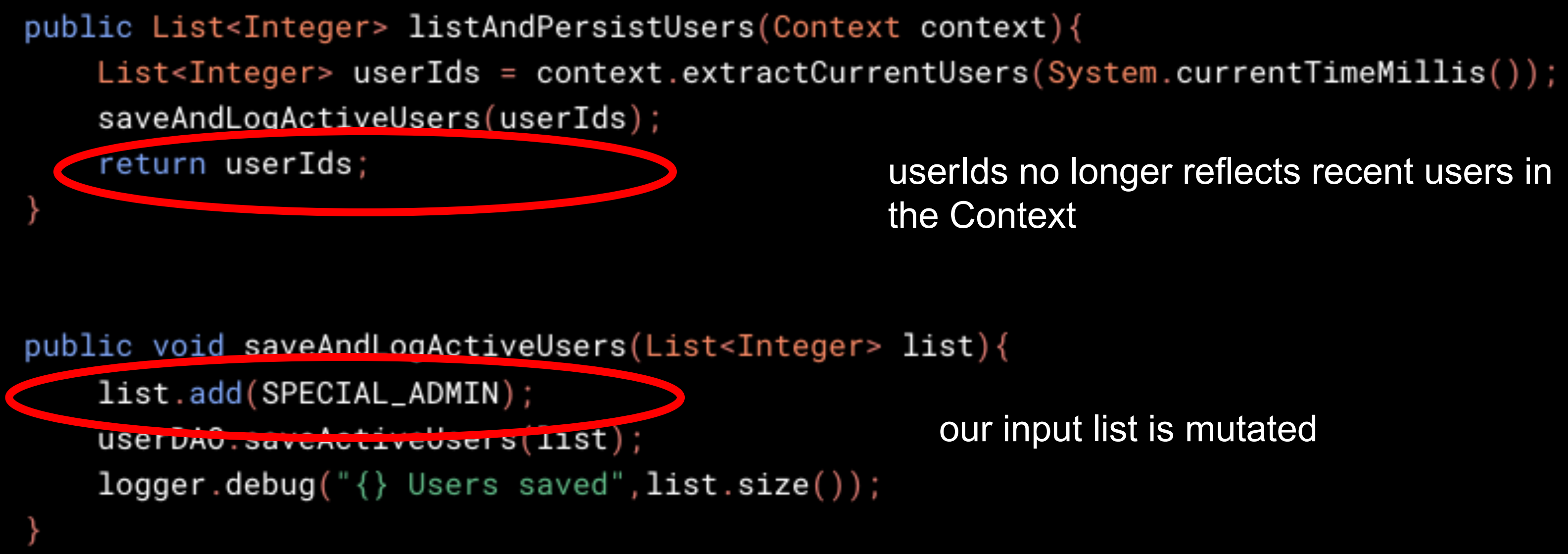
```
public List<Integer> listAndPersistUsers(Context context){
    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());
    saveAndLogActiveUsers(userIds);
    return userIds;
}


public void saveAndLogActiveUsers(List<Integer> list){
    userDAO.saveActiveUsers(list);
    logger.debug("{} Users saved",list.size());
}
```

userIds returned to client
unchanged?

Can we assume
saveActiveUsers
does not mutate list?



```
public List<Integer> listAndPersistUsers(Context context){
    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());
    saveAndLogActiveUsers(userIds);
    return userIds;
}


public void saveAndLogActiveUsers(List<Integer> list){
    list.add(SPECIAL_ADMIN);
    userDAO.saveActiveUsers(list);
    logger.debug("{} Users saved",list.size());
}
```



```
public List<Integer> listAndPersistUsers(Context context){
    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());
    saveAndLogActiveUsers(userIds);
    return userIds;
}


public void saveAndLogActiveUsers(List<Integer> list){
    list.add(SPECIAL_ADMIN);
    userDAO.saveActiveUsers(list);
    logger.debug("{} Users saved",list.size());
}
```

userIds no longer reflects recent users in the Context

our input list is mutated

Mutable Data Structures

Definition	Correctness	Performance	Complexity
Collections that can be mutated in place	❌ Can’t rely on them being unchanged	✅ Generally good	✅ Generally Simple

Mutable Data Structures

Definition	**Correctness**	**Performance**	**Complexity** (of implementation)	**Complexity** (of client code)
Collections that can be mutated in place	❌ Can’t rely on them being unchanged	✅ Generally good.	✅ Generally Simple	❌ Difficult to reason about

immutable collections

Correctness, performance, complexity?

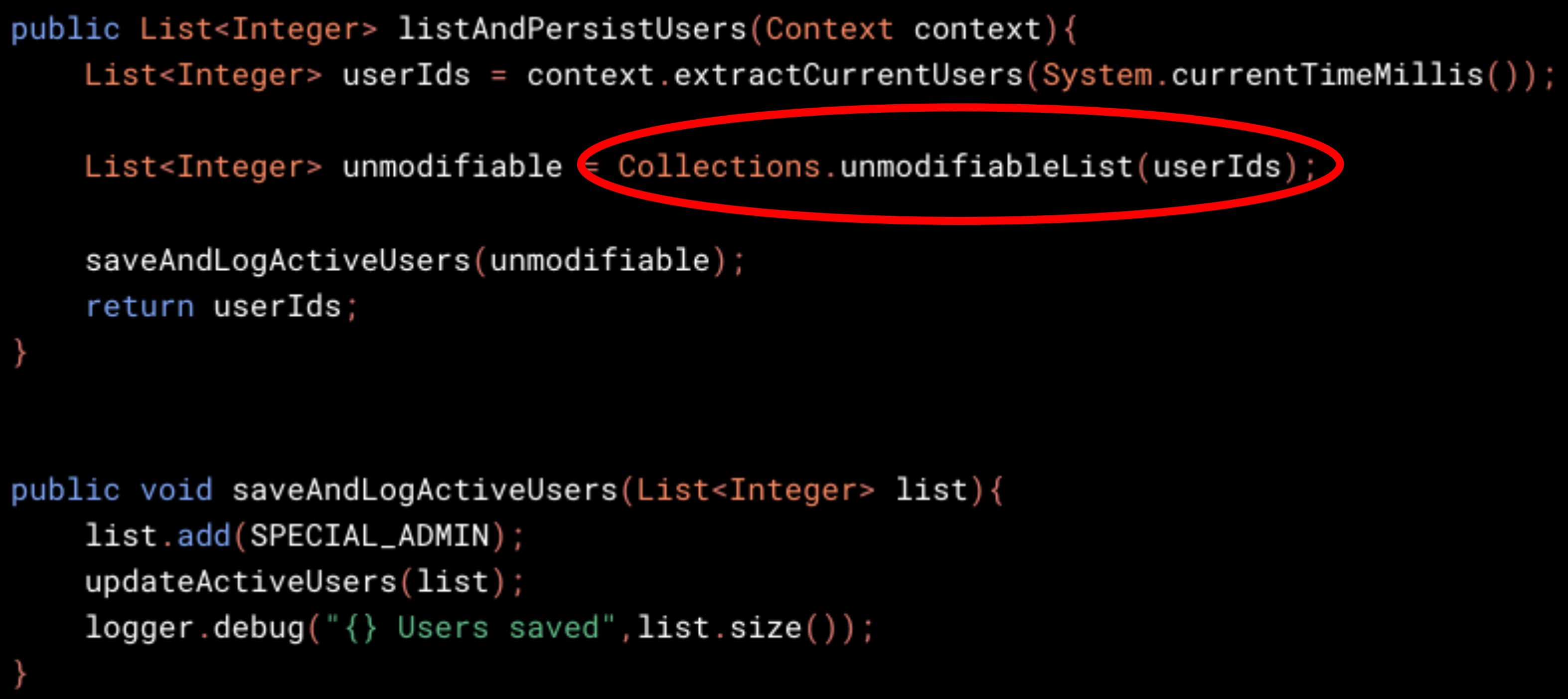
JDK Unmodifiable

‘Unmodifiable’ Data Structures

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Collections that expose an unmodifiable view					

‘Unmodifiable’ Data Structures

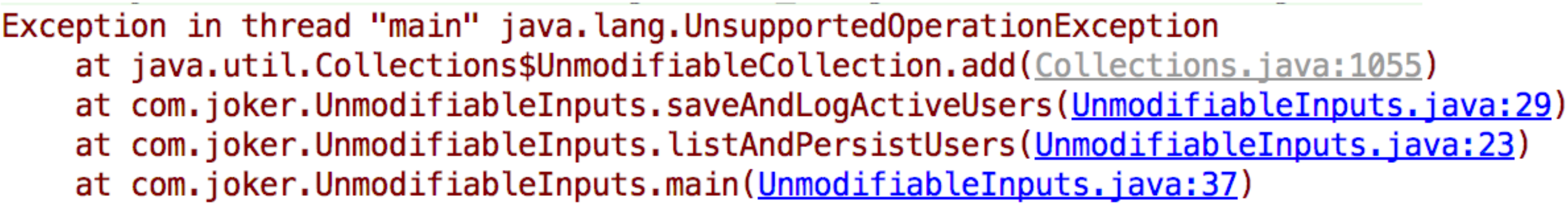
Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Collections that expose an unmodifiable view		✅ Good		✅ Simple	



```
public List<Integer> listAndPersistUsers(Context context){
    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());

    List<Integer> unmodifiable = Collections.unmodifiableList(userIds);

    saveAndLogActiveUsers(unmodifiable);
    return userIds;
}


public void saveAndLogActiveUsers(List<Integer> list){
    list.add(SPECIAL_ADMIN);
    updateActiveUsers(list);
    logger.debug("{} Users saved",list.size());
}
```

```java
public List<Integer> listAndPersistUsers(Context context){
    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());

    List<Integer> unmodifiable = Collections.unmodifiableList(userIds);

    saveAndLogActiveUsers(unmodifiable);
    return userIds;
}


public void saveAndLogActiveUsers(List<Integer> list){
    list.add(SPECIAL_ADMIN);
    updateActiveUsers(list);
    logger.debug("{} Users saved",list.size());
}
```

UnsupportedOperationException



```
Exception in thread "main" java.lang.UnsupportedOperationException
	at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
	at com.joker.UnmodifiableInputs.saveAndLogActiveUsers(UnmodifiableInputs.java:29)
	at com.joker.UnmodifiableInputs.listAndPersistUsers(UnmodifiableInputs.java:23)
	at com.joker.UnmodifiableInputs.main(UnmodifiableInputs.java:37)
```

‘Unmodifiable’ Data Structures

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Collections that expose an unmodifiable view	❌ API may throw Exceptions	✅ Good		✅ Simple	❌ Difficult to reason effectively about

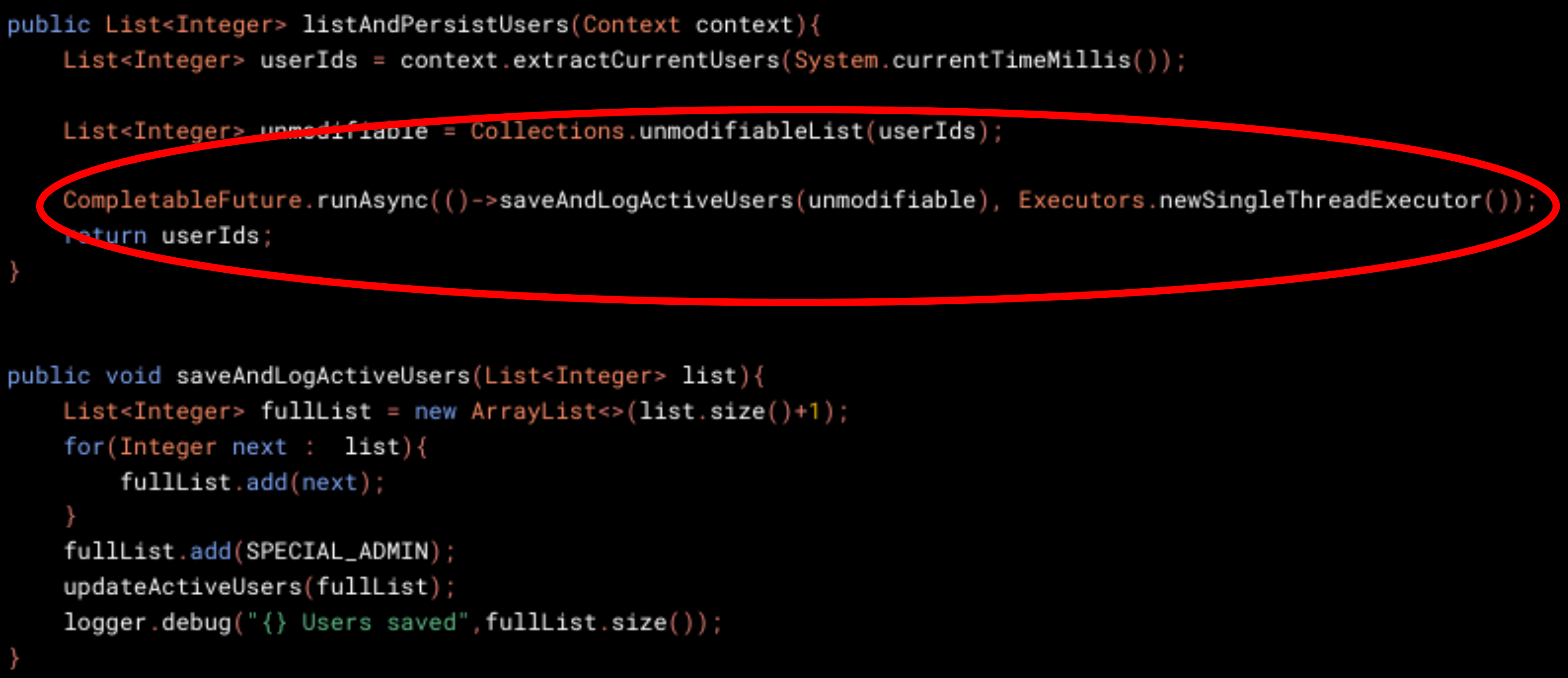
```
public List<Integer> listAndPersistUsers(Context context){
    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());

    List<Integer> unmodifiable = Collections.unmodifiableList(userIds);

    saveAndLogActiveUsers(unmodifiable);
    return userIds;
}


public void saveAndLogActiveUsers(List<Integer> list){
    List<Integer> fullList = new ArrayList<>(list.size()+1);
    for(Integer next :  list){
        fullList.add(next);
    }
    fullList.add(SPECIAL_ADMIN);
    updateActiveUsers(fullList);
    logger.debug("{} Users saved",fullList.size());
}
```

Time Complexity

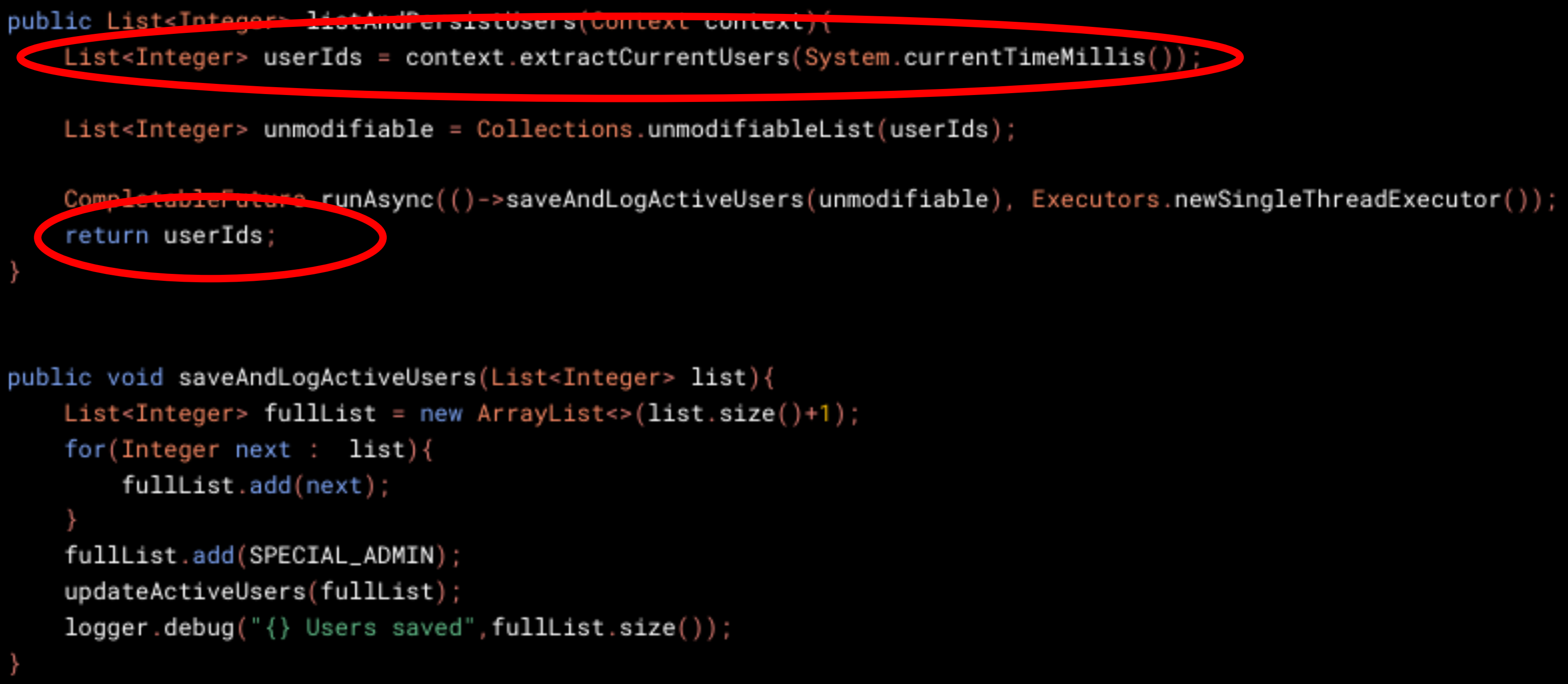
Type / Op.	**get**	**update**	**append**	**delete**
Unmodifiable	O(1)	O(n)	O(n)	O(n)

‘Unmodifiable’ Data Structures

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Collections that expose an unmodifiable view	❌ API may throw Exceptions	✅ Good	❌ Requires copy-on-write	✅ Simple	❌ Difficult to reason effectively about



```
public List<Integer> listAndPersistUsers(Context context){
    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());

    List<Integer> unmodifiable = Collections.unmodifiableList(userIds);

    CompletableFuture.runAsync(()->saveAndLogActiveUsers(unmodifiable), Executors.newSingleThreadExecutor());
    return userIds;
}


public void saveAndLogActiveUsers(List<Integer> list){
    List<Integer> fullList = new ArrayList<>(list.size()+1);
    for(Integer next :  list){
        fullList.add(next);
    }
    fullList.add(SPECIAL_ADMIN);
    updateActiveUsers(fullList);
    logger.debug("{} Users saved",fullList.size());
}
```



```java
public List<Integer> listAndPersistUsers(Context context){
    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());

    List<Integer> unmodifiable = Collections.unmodifiableList(userIds);

    CompletableFuture.runAsync(()->saveAndLogActiveUsers(unmodifiable), Executors.newSingleThreadExecutor());
    return userIds;
}


public void saveAndLogActiveUsers(List<Integer> list){
    List<Integer> fullList = new ArrayList<>(list.size()+1);
    for(Integer next :  list){
        fullList.add(next);
    }
    fullList.add(SPECIAL_ADMIN);
    updateActiveUsers(fullList);
    logger.debug("{} Users saved",fullList.size());
}
```

```
public void raceCondition(Context context){

    List<Integer> mutable = listAndPersistUsers(context);
    mutable.remove(0);//remove first

}
```

‘Unmodifiable’ Data Structures

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Collections that expose an unmodifiable view	❌ API may throw Exceptions ❌ Original collection can be mutated	✅ Good	❌ Requires copy-on-write	✅ Simple	❌ Difficult to reason effectively about

Guava Immutable

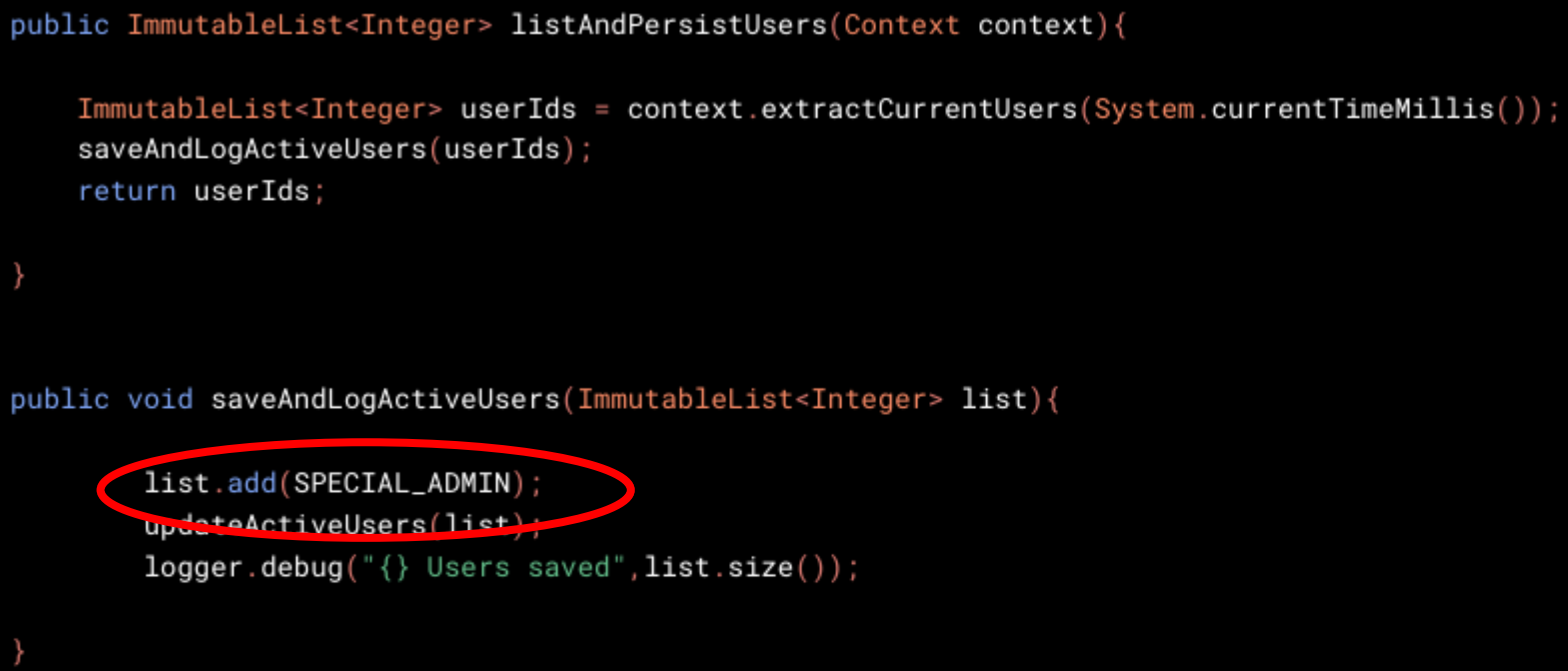
Removes the reference to a mutable collection

‘Unmodifiable’ Data Structures

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Collections that expose an unmodifiable view	❌ API may throw Exceptions ❌ Original collection can be mutated	✅ Good	❌ Requires copy-on-write	✅ Simple	❌ Difficult to reason effectively about

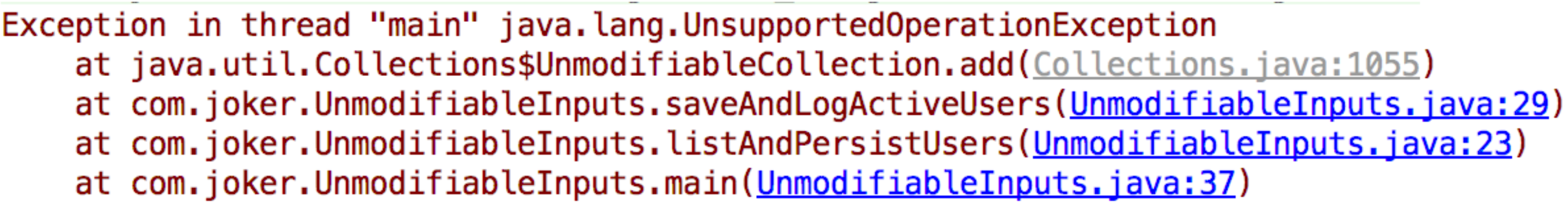
Immutable Data Structures (Guava)

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Collections that can't be changed	❌ API may throw Exceptions	✅ Good	❌ Requires copy-on-write	✅ Simple	❌ Still difficult to reason effectively about



```
public ImmutableList<Integer> listAndPersistUsers(Context context){

    ImmutableList<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());
    saveAndLogActiveUsers(userIds);
    return userIds;

}


public void saveAndLogActiveUsers(ImmutableList<Integer> list){

      list.add(SPECIAL_ADMIN);
      updateActiveUsers(list);
      logger.debug("{} Users saved",list.size());

}
```

UnsupportedOperationException



```
Exception in thread "main" java.lang.UnsupportedOperationException
	at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
	at com.joker.UnmodifiableInputs.saveAndLogActiveUsers(UnmodifiableInputs.java:29)
	at com.joker.UnmodifiableInputs.listAndPersistUsers(UnmodifiableInputs.java:23)
	at com.joker.UnmodifiableInputs.main(UnmodifiableInputs.java:37)
```

```
public void saveAndLogActiveUsers(ImmutableList<Integer> list){
    ImmutableList<Integer> usersAndAdmin = ImmutableList.<Integer>builder()
                                                        .addAll(list)
                                                        .add(SPECIAL_ADMIN)
                                                        .build();

    updateActiveUsers(usersAndAdmin);
    logger.debug("{} Users saved",list.size());
}
```

Time Complexity

Type / Op.	**get**	**update**	**append**	**delete**
Immutable	O(1)	O(n)	O(n)	O(n)

Immutable Data Structures (Guava)

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Collections that can't be changed	❌ API may throw Exceptions	✅ Good	❌ Requires copy-on-write	✅ Simple	❌ Still difficult to reason effectively about

persistent collections

Correctness, performance, complexity?

Cyclops Persistent

Persistent Data Structures (Cyclops)

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Immutable shared memory collections					

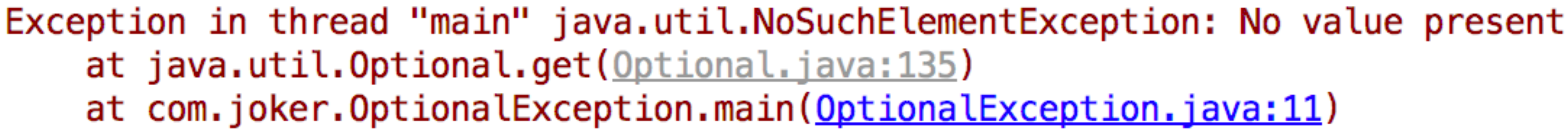
Immutable Data Structures (Guava)

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Collections that can't be changed	❌ API may throw Exceptions	✅ Good	❌ Requires copy-on-write	✅ Simple	❌ Still difficult to reason effectively about

Make Illegal States Unrepresentable

```java
Optional<Integer> opt = Optional.empty();

opt.get();
```

NoSuchElementException



```
Exception in thread "main" java.util.NoSuchElementException: No value present
	at java.util.Optional.get(Optional.java:135)
	at com.joker.OptionalException.main(OptionalException.java:11)
```

```
Option<Integer> opt = Option.none();

opt.orElse(-1);
```

```
class Vector<E> implements List<E> {

    public boolean add(E element){
       throw new UnsupportedOperationException("Add not supported");
    }

}
```

```
class Vector<E> implements List<E> {

    public boolean add(E element){
        throw new UnsupportedOperationException("Add not supported");
    }

}
```

```java
public Vector<Integer> listAndPersistUsers(PersistentContext context){

    Vector<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());
    saveAndLogActiveUsers(userIds);
    return userIds;

}


public void saveAndLogActiveUsers(Vector<Integer> list){

    Vector<Integer> fullList = list.plus(SPECIAL_ADMIN);
    updateActiveUsers(fullList);
    logger.debug("{} Users saved",list.size());

}
```

```
Vector<Integer> list = Vector.of(1,2,3);

Option<Integer> valueAt = list.get(-1);
```

Persistent Data Structures (Cyclops)

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Immutable shared memory collections	✅ API *should* make illegal states unrepresentable				✅ Keeps it simple

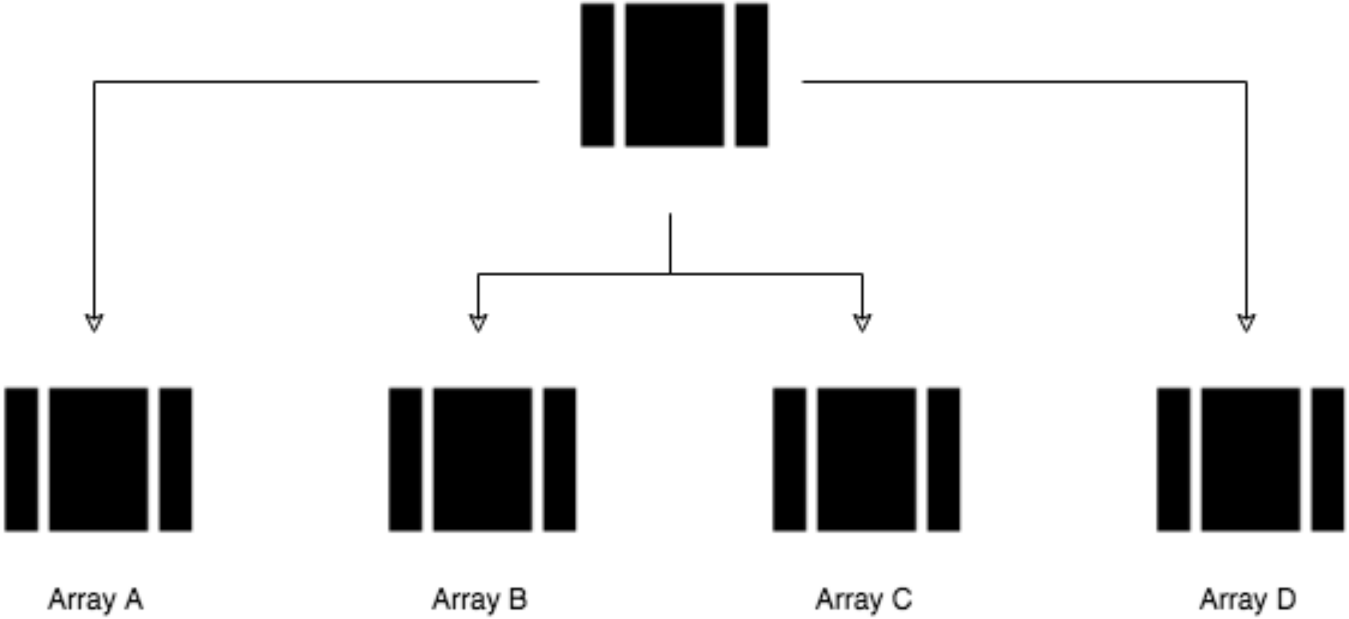
Performance?

Daniel Spiewak

Extreme Functional Cleverness

https://www.youtube.com/watch?v=pNhBQJN44YQ

Bitmapped vector trie

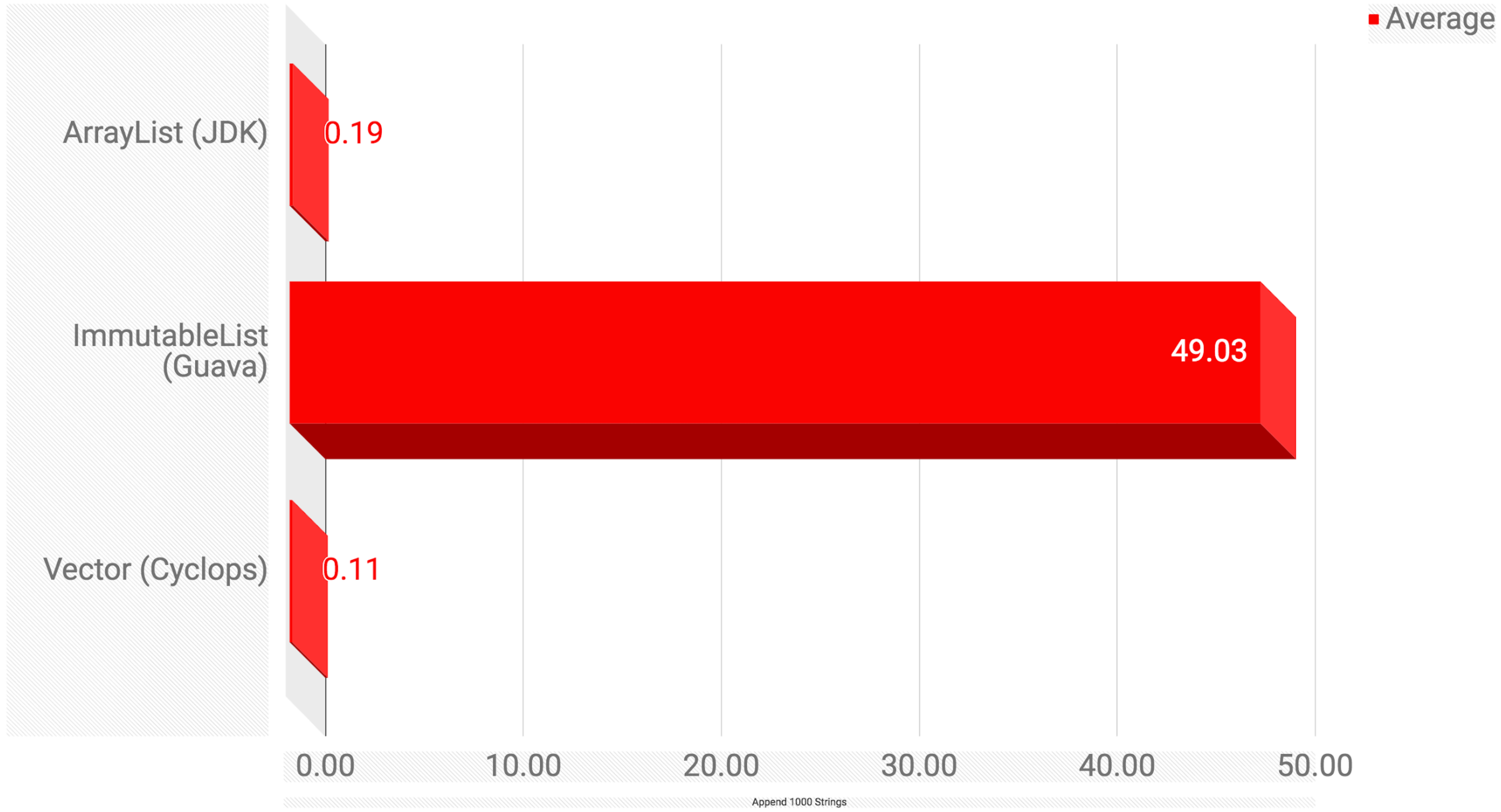


Time Complexity

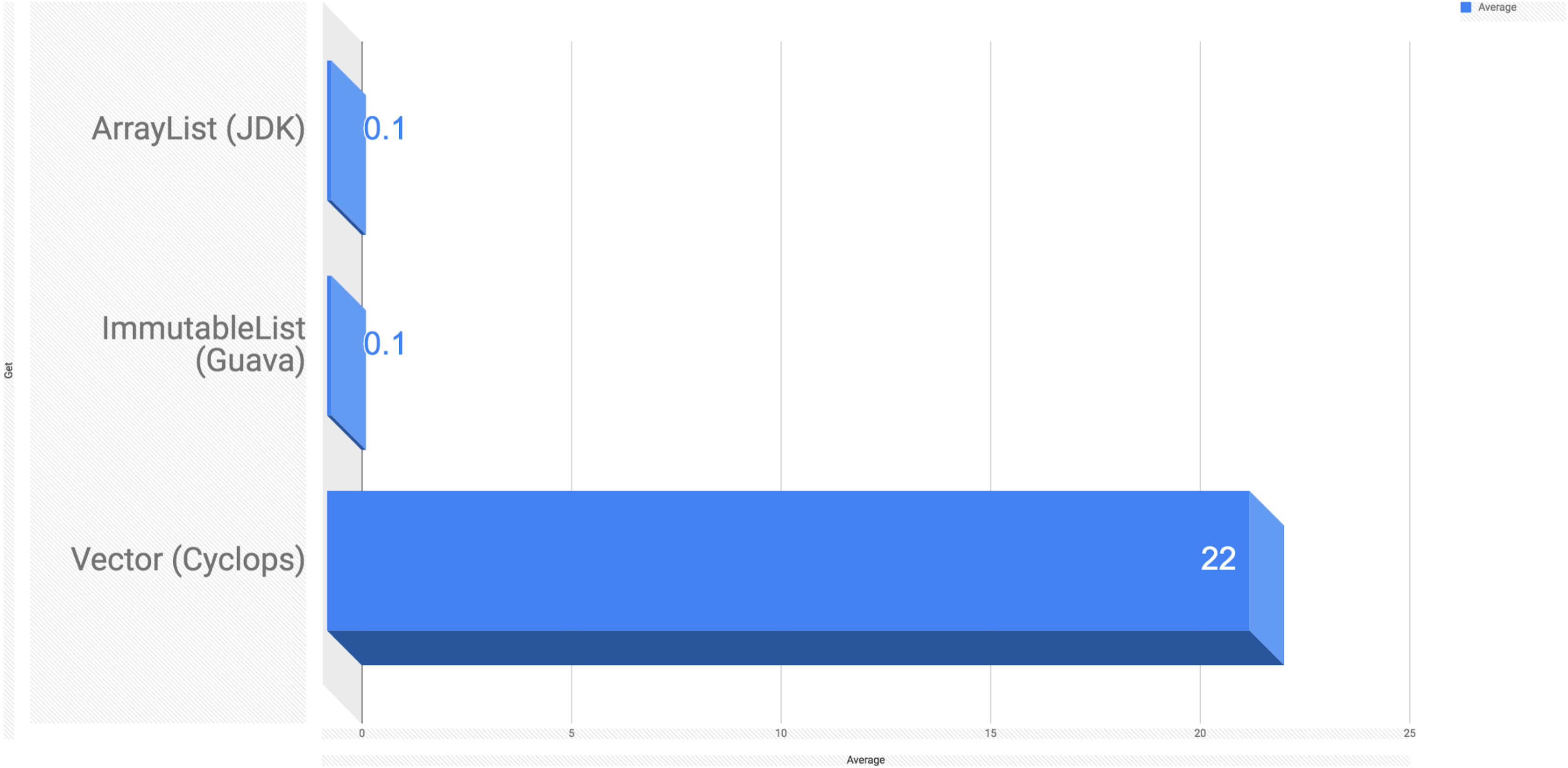
Type / Op.	**get**	**update**	**append**	**delete**
Persistent	O(log(n))	O(log(n))	O(log(n))	O(log(n))

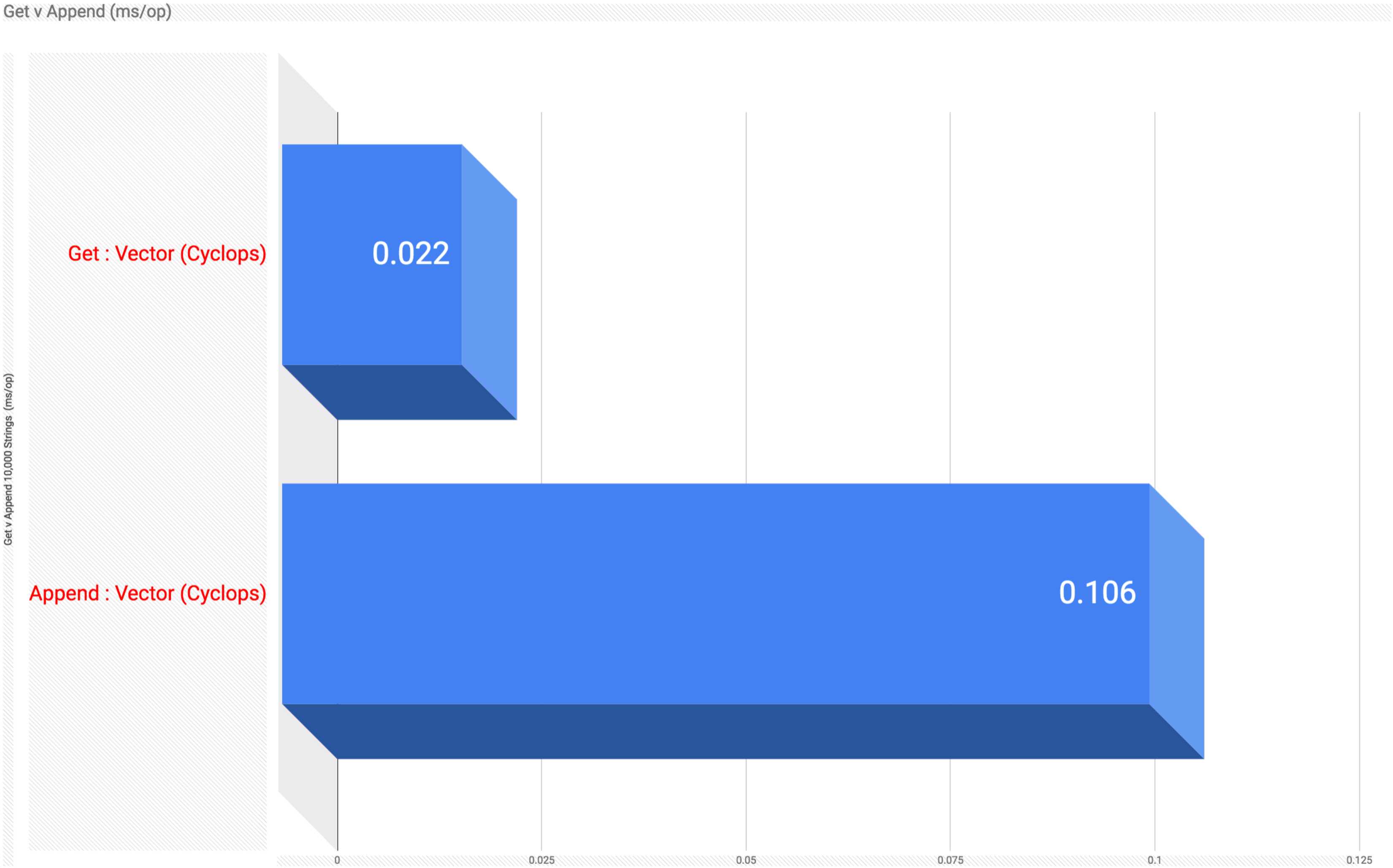
Some performance benchmarks

Append 10,000 String (ms/op) : lower is better



Get 10,000 Strings (µs / op)

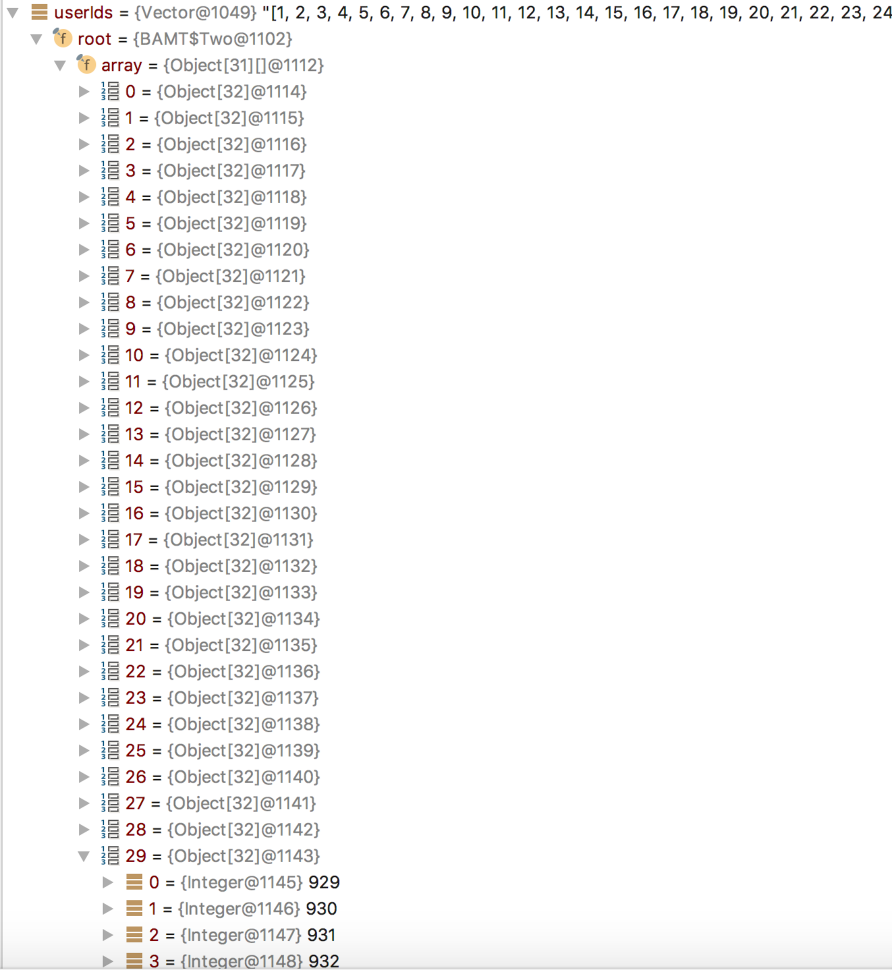



Get v Append (ms/op)
Get v Append 10,000 Strings (ms/op)
Get : Vector (Cyclops)
0.022
Append : Vector (Cyclops)
0.106
0
0.025
0.05
0.075
0.1
0.125

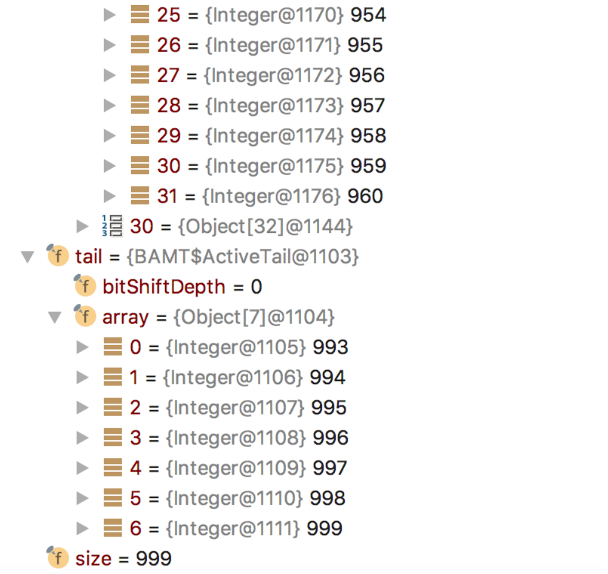
Persistent Data Structures (Cyclops)

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Immutable shared memory collections	✅ API *should* make illegal states unrepresentable	✅ Reasonable in most cases	✅ Reasonable in most cases		✅ Keeps it simple

Debugging



```
▼ userIds = {Vector@1049} "[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24
  ▼ root = {BAMT$Two@1102}
    ▼ array = {Object[31][]@1112}
      ▶ 0 = {Object[32]@1114}
      ▶ 1 = {Object[32]@1115}
      ▶ 2 = {Object[32]@1116}
      ▶ 3 = {Object[32]@1117}
      ▶ 4 = {Object[32]@1118}
      ▶ 5 = {Object[32]@1119}
      ▶ 6 = {Object[32]@1120}
      ▶ 7 = {Object[32]@1121}
      ▶ 8 = {Object[32]@1122}
      ▶ 9 = {Object[32]@1123}
      ▶ 10 = {Object[32]@1124}
      ▶ 11 = {Object[32]@1125}
      ▶ 12 = {Object[32]@1126}
      ▶ 13 = {Object[32]@1127}
      ▶ 14 = {Object[32]@1128}
      ▶ 15 = {Object[32]@1129}
      ▶ 16 = {Object[32]@1130}
      ▶ 17 = {Object[32]@1131}
      ▶ 18 = {Object[32]@1132}
      ▶ 19 = {Object[32]@1133}
      ▶ 20 = {Object[32]@1134}
      ▶ 21 = {Object[32]@1135}
      ▶ 22 = {Object[32]@1136}
      ▶ 23 = {Object[32]@1137}
      ▶ 24 = {Object[32]@1138}
      ▶ 25 = {Object[32]@1139}
      ▶ 26 = {Object[32]@1140}
      ▶ 27 = {Object[32]@1141}
      ▶ 28 = {Object[32]@1142}
      ▼ 29 = {Object[32]@1143}
        ▶ 0 = {Integer@1145} 929
        ▶ 1 = {Integer@1146} 930
        ▶ 2 = {Integer@1147} 931
        ▶ 3 = {Integer@1148} 932
```



```
        ▶ 25 = {Integer@1170} 954
        ▶ 26 = {Integer@1171} 955
        ▶ 27 = {Integer@1172} 956
        ▶ 28 = {Integer@1173} 957
        ▶ 29 = {Integer@1174} 958
        ▶ 30 = {Integer@1175} 959
        ▶ 31 = {Integer@1176} 960
      ▶ 30 = {Object[32]@1144}
  ▼ tail = {BAMT$ActiveTail@1103}
      bitShiftDepth = 0
    ▼ array = {Object[7]@1104}
      ▶ 0 = {Integer@1105} 993
      ▶ 1 = {Integer@1106} 994
      ▶ 2 = {Integer@1107} 995
      ▶ 3 = {Integer@1108} 996
      ▶ 4 = {Integer@1109} 997
      ▶ 5 = {Integer@1110} 998
      ▶ 6 = {Integer@1111} 999
    size = 999
```

Persistent Data Structures (Cyclops)

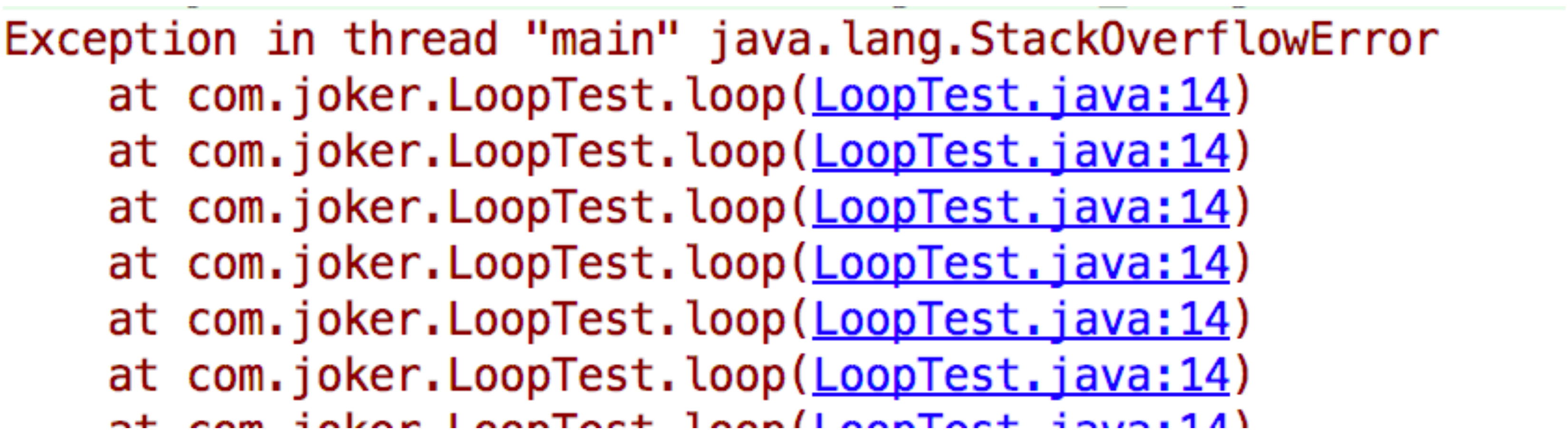
Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Immutable shared memory collections	✅ API *should* make illegal states unrepresentable	✅ Reasonable in most cases	✅ Reasonable in most cases	❌ May be more complex	✅ Keeps it simple

A more complex example

Eval

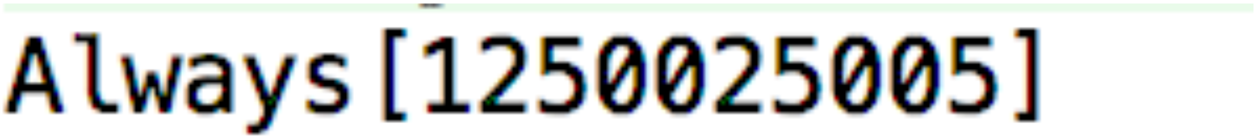
```
int loop(int times, int sum){
    if(times==0)
        return sum;
    else
        return loop(times-1,sum+times);
}



loop(50000,5);
```

StackOverflowError



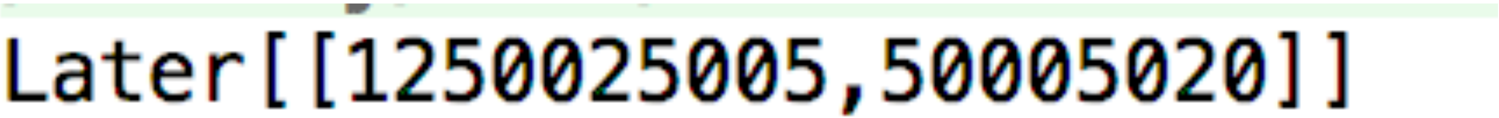
```
Exception in thread "main" java.lang.StackOverflowError
	at com.joker.LoopTest.loop(LoopTest.java:14)
	at com.joker.LoopTest.loop(LoopTest.java:14)
	at com.joker.LoopTest.loop(LoopTest.java:14)
	at com.joker.LoopTest.loop(LoopTest.java:14)
	at com.joker.LoopTest.loop(LoopTest.java:14)
	at com.joker.LoopTest.loop(LoopTest.java:14)
	at com.joker.LoopTest.loop(LoopTest.java:14)
```

```
Eval<Integer> loop(int times,Eval<Integer> sum){
    if(times==0)
        return sum;
    else
        return sum.flatMap(s->loop(times-1,Eval.now(s+times)));
}


 loop(50000,Eval.now(5));
```



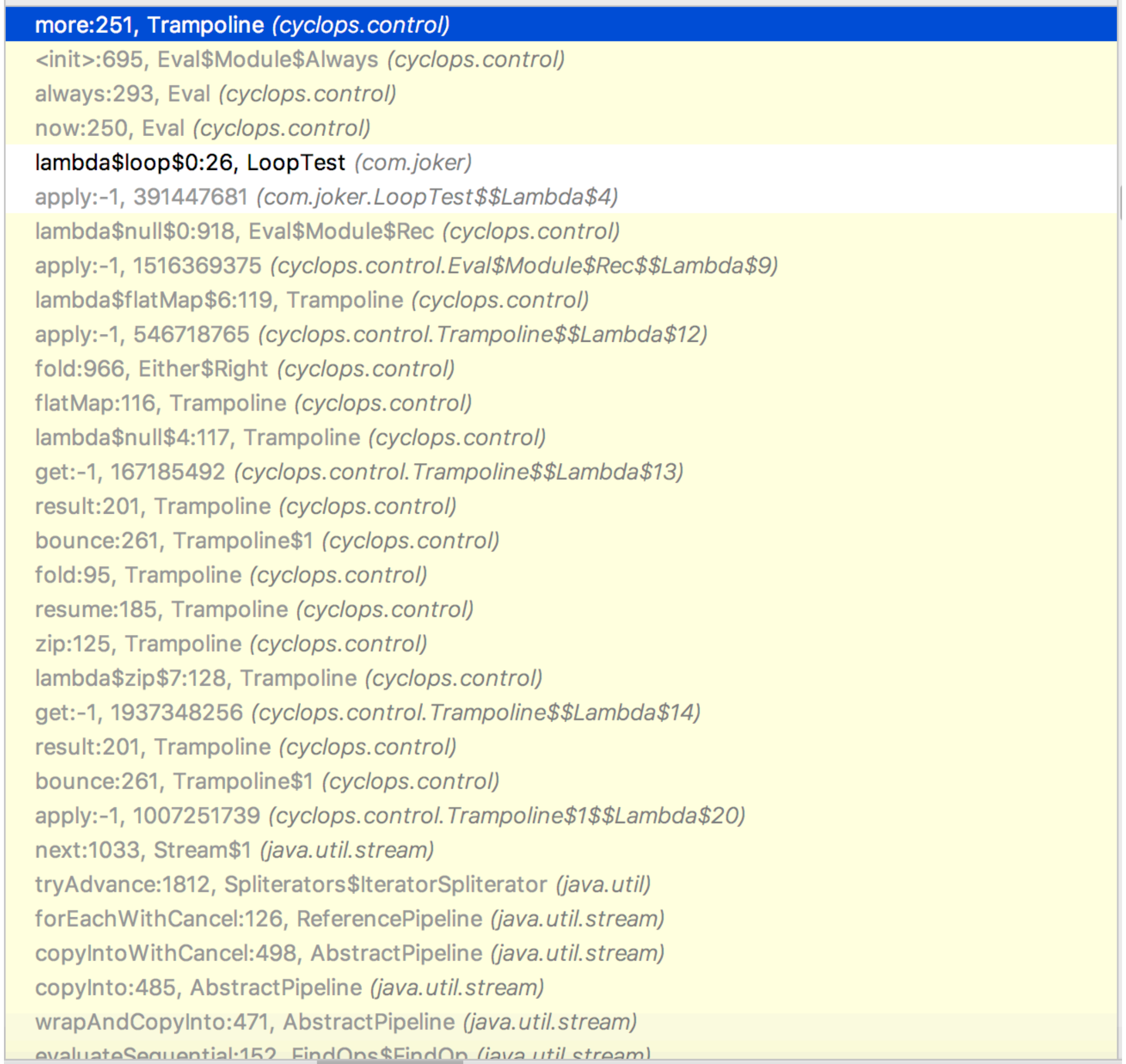
```
Always[12500025005]
```

```
loop(50000,Eval.now(5)).zip(loop(10000,Eval.now(20)),Tuple::tuple);
```



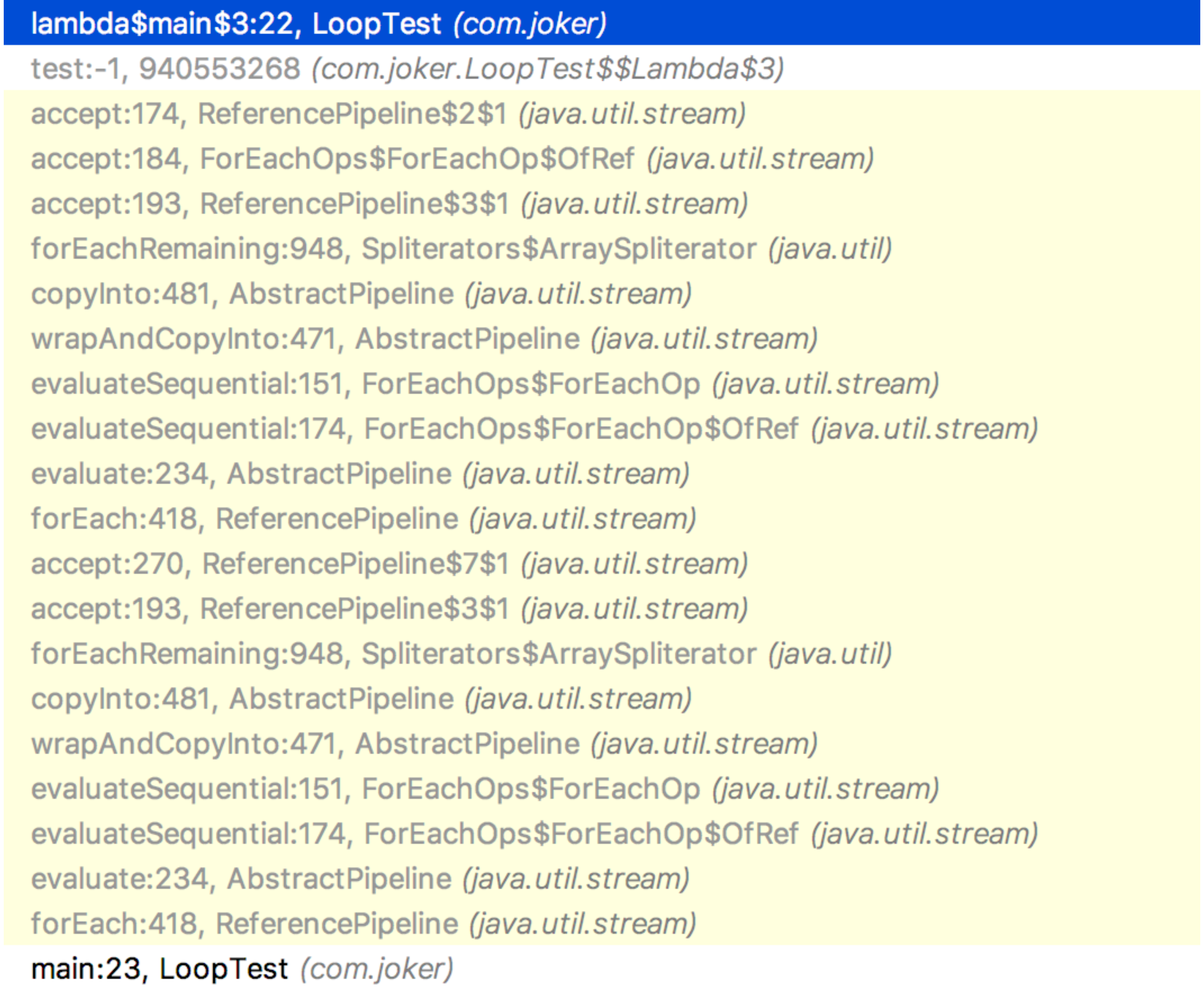
```
Later[[12500025005,50005020]]
```

Eval Stacktrace



```
more:251, Trampoline (cyclops.control)
<init>:695, Eval$Module$Always (cyclops.control)
always:293, Eval (cyclops.control)
now:250, Eval (cyclops.control)
lambda$loop$0:26, LoopTest (com.joker)
apply:-1, 391447681 (com.joker.LoopTest$$Lambda$4)
lambda$null$0:918, Eval$Module$Rec (cyclops.control)
apply:-1, 1516369375 (cyclops.control.Eval$Module$Rec$$Lambda$9)
lambda$flatMap$6:119, Trampoline (cyclops.control)
apply:-1, 546718765 (cyclops.control.Trampoline$$Lambda$12)
fold:966, Either$Right (cyclops.control)
flatMap:116, Trampoline (cyclops.control)
lambda$null$4:117, Trampoline (cyclops.control)
get:-1, 167185492 (cyclops.control.Trampoline$$Lambda$13)
result:201, Trampoline (cyclops.control)
bounce:261, Trampoline$1 (cyclops.control)
fold:95, Trampoline (cyclops.control)
resume:185, Trampoline (cyclops.control)
zip:125, Trampoline (cyclops.control)
lambda$zip$7:128, Trampoline (cyclops.control)
get:-1, 1937348256 (cyclops.control.Trampoline$$Lambda$14)
result:201, Trampoline (cyclops.control)
bounce:261, Trampoline$1 (cyclops.control)
apply:-1, 1007251739 (cyclops.control.Trampoline$1$$Lambda$20)
next:1033, Stream$1 (java.util.stream)
tryAdvance:1812, Spliterators$IteratorSpliterator (java.util)
forEachWithCancel:126, ReferencePipeline (java.util.stream)
copyIntoWithCancel:498, AbstractPipeline (java.util.stream)
copyInto:485, AbstractPipeline (java.util.stream)
wrapAndCopyInto:471, AbstractPipeline (java.util.stream)
evaluateSequential:152, FindOps$FindOp (java.util.stream)
```

Stream Stacktrace



```
lambda$main$3:22, LoopTest (com.joker)
test:-1, 940553268 (com.joker.LoopTest$$Lambda$3)
accept:174, ReferencePipeline$2$1 (java.util.stream)
accept:184, ForEachOps$ForEachOp$OfRef (java.util.stream)
accept:193, ReferencePipeline$3$1 (java.util.stream)
forEachRemaining:948, Spliterators$ArraySpliterator (java.util)
copyInto:481, AbstractPipeline (java.util.stream)
wrapAndCopyInto:471, AbstractPipeline (java.util.stream)
evaluateSequential:151, ForEachOps$ForEachOp (java.util.stream)
evaluateSequential:174, ForEachOps$ForEachOp$OfRef (java.util.stream)
evaluate:234, AbstractPipeline (java.util.stream)
forEach:418, ReferencePipeline (java.util.stream)
accept:270, ReferencePipeline$7$1 (java.util.stream)
accept:193, ReferencePipeline$3$1 (java.util.stream)
forEachRemaining:948, Spliterators$ArraySpliterator (java.util)
copyInto:481, AbstractPipeline (java.util.stream)
wrapAndCopyInto:471, AbstractPipeline (java.util.stream)
evaluateSequential:151, ForEachOps$ForEachOp (java.util.stream)
evaluateSequential:174, ForEachOps$ForEachOp$OfRef (java.util.stream)
evaluate:234, AbstractPipeline (java.util.stream)
forEach:418, ReferencePipeline (java.util.stream)
main:23, LoopTest (com.joker)
```

Persistent Data Structures (Cyclops)

Definition	**Correctness**	**Performance** (access)	**Performance** (append / update)	**Complexity** (of implementation)	**Complexity** (client code)
Immutable shared memory collections	✅ API *should* make illegal states unrepresentable	✅ Reasonable in most cases	✅ Reasonable in most cases	❌ May be more complex	✅ Keeps it simple

Choosing Collections

know your goals

Know your goals

if(correctness > performance)

Guava style Immutable	Cyclops style persistent
❌ API throws Exceptions	✅ Makes illegal states unrepresentable

else if(writes > reads)

Guava style Immutable	Cyclops style persistent
❌ Generally poor performance	✅ Reasonable performance

else if(reads >> writes)

Guava style Immutable	**Cyclops style persistent**
✅ Good performance	❌ Reasonable performance (but slower)

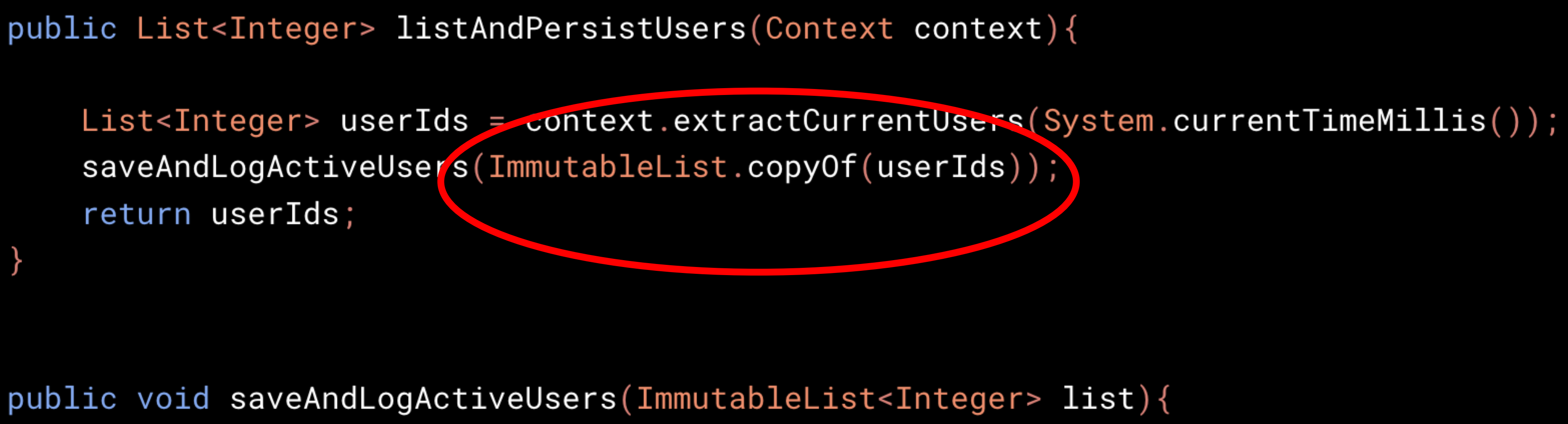
Danger Conversions!

Mutable to Immutable

Performance hit

```java
public List<Integer> listAndPersistUsers(Context context){

    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());
    saveAndLogActiveUsers(Vector.fromIterable(userIds));
    return userIds;
}


public void saveAndLogActiveUsers(Vector<Integer> list){
```

Performance hit?



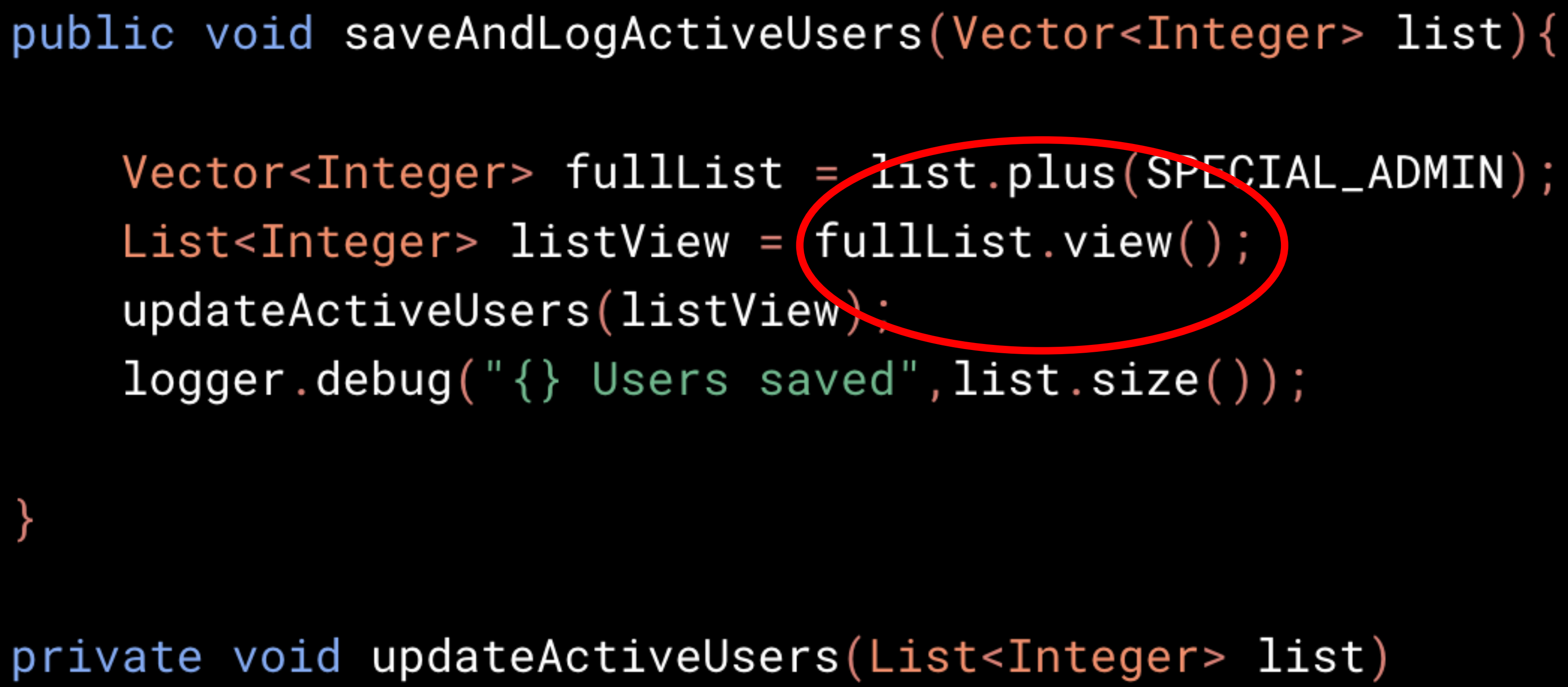
```
public List<Integer> listAndPersistUsers(Context context){

    List<Integer> userIds = context.extractCurrentUsers(System.currentTimeMillis());
    saveAndLogActiveUsers(ImmutableList.copyOf(userIds));
    return userIds;
}


public void saveAndLogActiveUsers(ImmutableList<Integer> list){
```

Immutable to Mutable

Safely

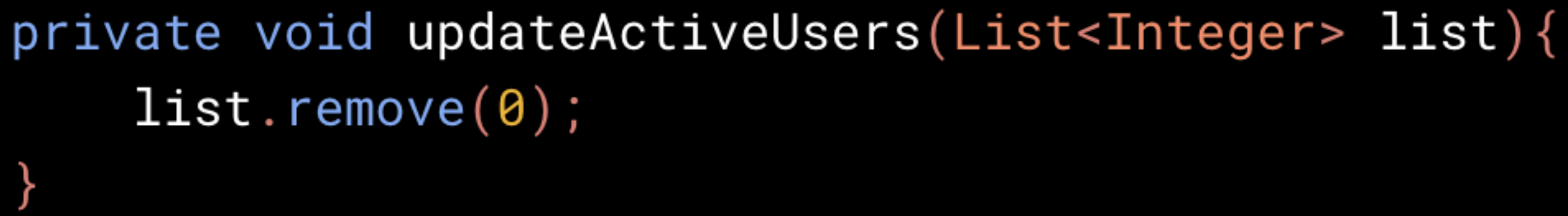
```
public void saveAndLogActiveUsers(Vector<Integer> list){
    Vector<Integer> fullList = list.plus(SPECIAL_ADMIN);

    List<Integer> converted = fullList.toList();
    updateActiveUsers(converted);

    logger.debug("{} Users saved",list.size());
}

private void updateActiveUsers(List<Integer> list)
```

Views



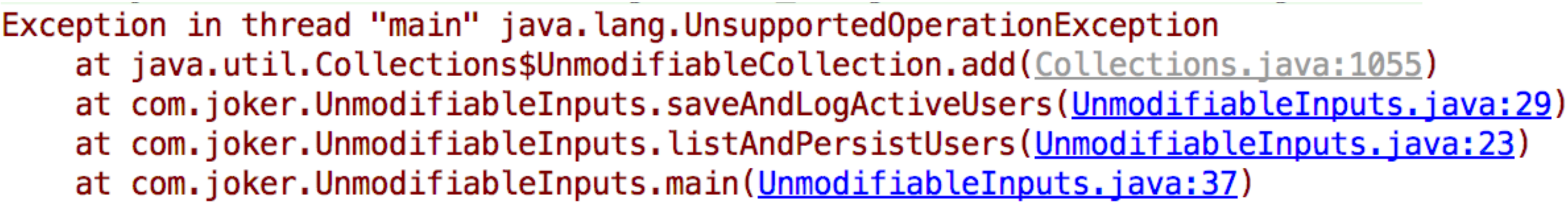
```
public void saveAndLogActiveUsers(Vector<Integer> list){

    Vector<Integer> fullList = list.plus(SPECIAL_ADMIN);
    List<Integer> listView = fullList.view();
    updateActiveUsers(listView);
    logger.debug("{} Users saved",list.size());

}

private void updateActiveUsers(List<Integer> list)
```

UnsupportedOperationException



```
private void updateActiveUsers(List<Integer> list){
    list.remove(0);
}
```

UnsupportedOperationException

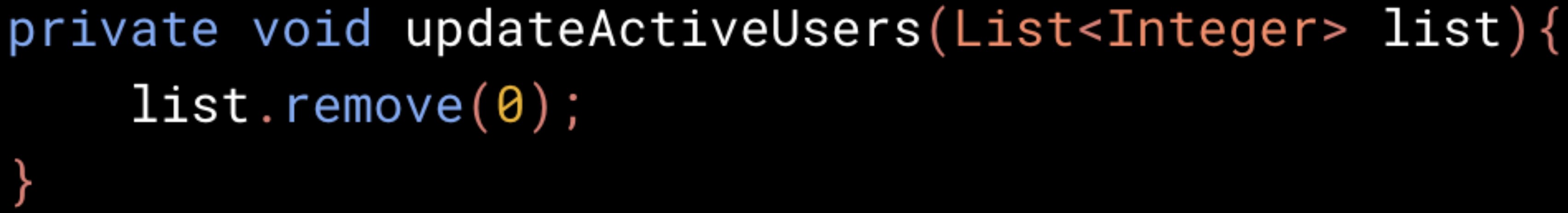


```
Exception in thread "main" java.lang.UnsupportedOperationException
	at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
	at com.joker.UnmodifiableInputs.saveAndLogActiveUsers(UnmodifiableInputs.java:29)
	at com.joker.UnmodifiableInputs.listAndPersistUsers(UnmodifiableInputs.java:23)
	at com.joker.UnmodifiableInputs.main(UnmodifiableInputs.java:37)
```

Lists

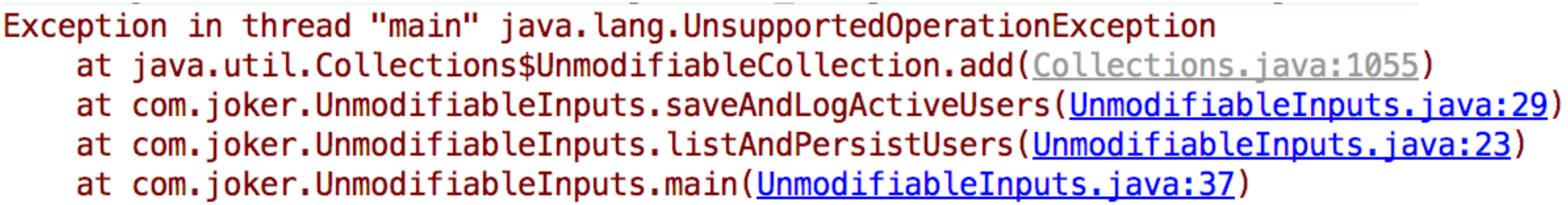
```
public void saveAndLogActiveUsers(ImmutableList<Integer> list){

    ImmutableList<Integer> fullList = ImmutableList.<Integer>builder()
                                                   .addAll(list)
                                                   .add(SPECIAL_ADMIN)
                                                   .build();
    updateActiveUsers(fullList);
    logger.debug("{} Users saved",list.size());

}

private void updateActiveUsers(List<Integer> list)
```

UnsupportedOperationException



```java
private void updateActiveUsers(List<Integer> list){
    list.remove(0);
}
```

UnsupportedOperationException



```
Exception in thread "main" java.lang.UnsupportedOperationException
	at java.util.Collections$UnmodifiableCollection.add(Collections.java:1055)
	at com.joker.UnmodifiableInputs.saveAndLogActiveUsers(UnmodifiableInputs.java:29)
	at com.joker.UnmodifiableInputs.listAndPersistUsers(UnmodifiableInputs.java:23)
	at com.joker.UnmodifiableInputs.main(UnmodifiableInputs.java:37)
```

Conversions

Mutable to immutable involves a performance hit

Using views involves compromising correctness

We can avoid this by copying (performance hit)

Is Functional Programming Worth It?

YES WHEN

You..

know your goals

limit complexity appropriately

make sure Java doesn’t undermine you!

Thank you!

cyclops-react.io
github.com/aol/cyclops-react

Oath: