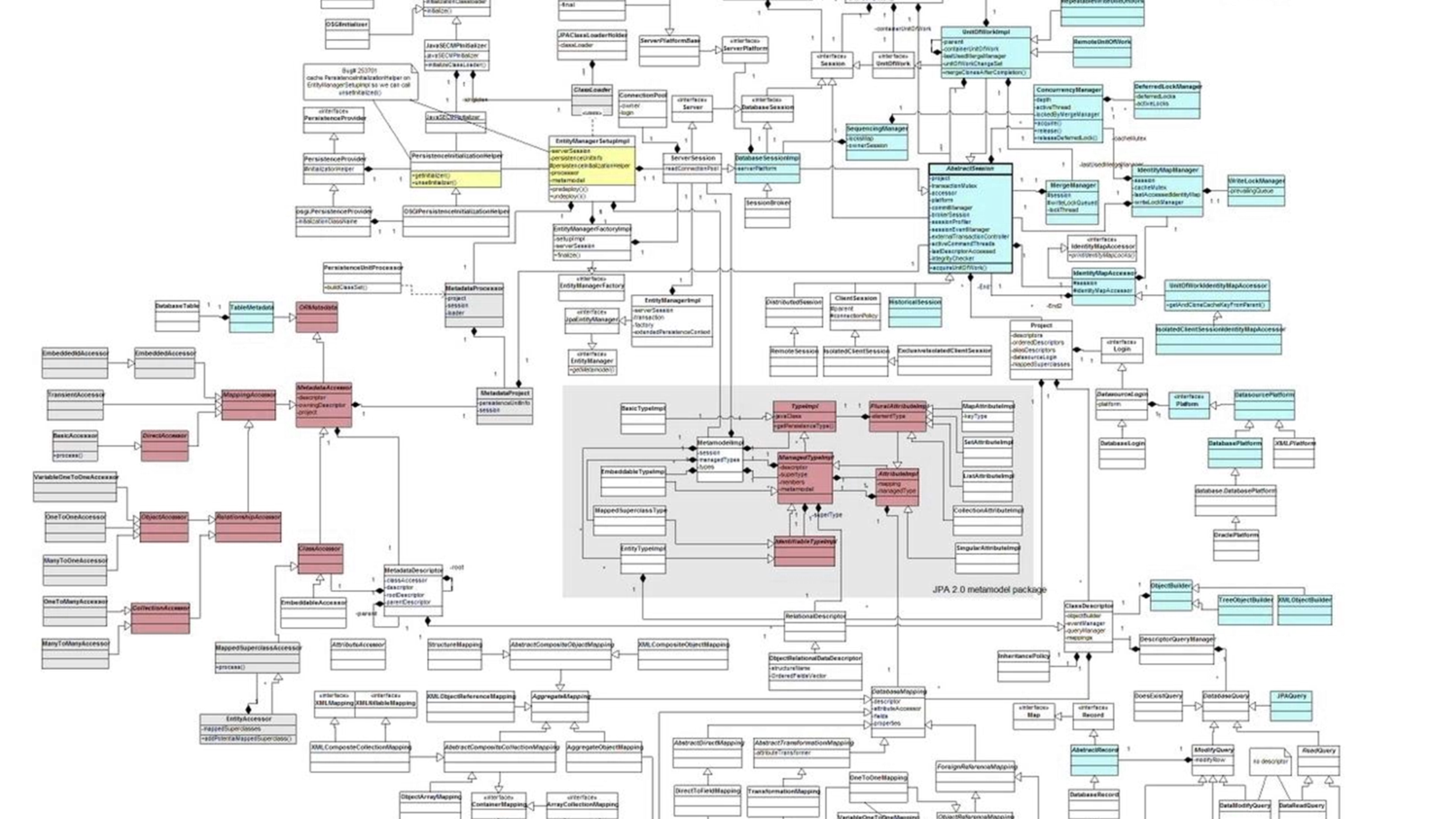

Simpler Java
Reducing complexity in Java code
Stefan von Stein

zacco

O
X O
X
X
X



OSGiInitializer
JavaSECMPInitializer
PersistenceProvider
PersistenceInitializationHelper
OSGiPersistenceInitializationHelper
ClassLoader
JPAClassLoaderHolder
ConnectionPool
EntityManagerSetupImpl
EntityManagerFactoryImpl
EntityManagerFactory
EntityManagerImpl
JpaEntityManager
EntityManager
ServerPlatformBase
ServerPlatform
Server
ServerSession
DatabaseSession
DatabaseSessionImpl
SessionBroker
Session
UnitOfWork
UnitOfWorkImpl
RemoteUnitOfWork
ConcurrencyManager
DeferredLockManager
SequencingManager
AbstractSession
MergeManager
IdentityMapManager
WriteLockManager
IdentityMapAccessor
UnitOfWorkIdentityMapAccessor
IsolatedClientSessionIdentityMapAccessor
DistributedSession
ClientSession
HistoricalSession
RemoteSession
IsolatedClientSession
ExclusiveIsolatedClientSession
Project
Login
DatasourceLogin
DatabaseLogin
Platform
DatasourcePlatform
DatabasePlatform
XMLPlatform
OraclePlatform
PersistenceUnitProcessor
MetadataProcessor
DatabaseTable
TableMetadata
ORMetadata
EmbeddedIdAccessor
EmbeddedAccessor
TransientAccessor
MappingAccessor
MetadataAccessor
MetadataProject
BasicAccessor
DirectAccessor
VariableOneToOneAccessor
OneToOneAccessor
ObjectAccessor
RelationshipAccessor
ManyToOneAccessor
ClassAccessor
MetadataDescriptor
OneToManyAccessor
CollectionAccessor
EmbeddableAccessor
ManyToManyAccessor
MappedSuperclassAccessor
EntityAccessor
BasicTypeImpl
TypeImpl
PluralAttributeImpl
MapAttributeImpl
SetAttributeImpl
ListAttributeImpl
CollectionAttributeImpl
SingularAttributeImpl
MetamodelImpl
ManagedTypeImpl
AttributeImpl
EmbeddableTypeImpl
MappedSuperclassTypeImpl
IdentifiableTypeImpl
EntityTypeImpl
JPA 2.0 metamodel package
ObjectBuilder
TreeObjectBuilder
XMLObjectBuilder
ClassDescriptor
RelationalDescriptor
ObjectRelationalDataDescriptor
DescriptorQueryManager
InheritancePolicy
AttributeAccessor
StructureMapping
AbstractCompositeObjectMapping
XMLCompositeObjectMapping
XMLMapping
XMLNillableMapping
XMLObjectReferenceMapping
AggregateMapping
DatabaseMapping
XMLCompositeCollectionMapping
AbstractCompositeCollectionMapping
AggregateObjectMapping
ObjectArrayMapping
ContainerMapping
ArrayCollectionMapping
AbstractDirectMapping
AbstractTransformationMapping
DirectToFieldMapping
TransformationMapping
OneToOneMapping
ForeignReferenceMapping
ObjectReferenceMapping
Map
Record
AbstractRecord
DatabaseRecord
DoesExistQuery
DatabaseQuery
JPAQuery
ModifyQuery
ReadQuery
DataModifyQuery
DataReadQuery

Variables

Variables are the moving parts

- Can be changed by any referring it.

Hidden variables are more complex

- someone can change them
- according to additional rules

Local variables are fine

- scope may change
- during addition of functionality, refactoring, by others

Complexity

Complexity undermines understanding

Entangled things need to be considered together

Incidental complexity

About avoiding variables

without variables

Functional programming

- without variables
- composing functions

Implies

- no if statements
- no loops

if statements

```
String answer ="Unknown";
if(value==3)
  answer="Yes";
else
  answer="No";
process(answer);
```

```
if(value==3)
  process("Yes");
else
  process("No");
```

temporaries or code duplication

if statements

```
final String answer =
  value==3 ? "Yes" : "No";
process(answer);
```

```
final String answer =
  if(value==3; "Yes"; "No");
process(answer);
```

But, how about a better `if`

ifelse the expression

```
final String answer = ifelse(value==3, ()->"Yes", this::sayNo);
```

Should return a value

Should evaluate conditionally

Lazy evaluation

```
<T> T ifelse(boolean test,
             Supplier<T> then,
             Supplier<T> otherwise) {
  return test ? then.get() : otherwise.get();
}
```

ifelse the expression

Without unnecessary variables

Represents a value

Still clear when nested

```
return ifelse(null==question,
  () -> "What? Question missing",
  () -> ifelse(isDifficult(question),
    () -> question.toUpperCase(),
    () -> concat("Don’t worry: ", question)));
```

```
final String answer = ifelse(value==3, ()->"Yes", this::sayNo);
```

ifelse the expression

Without unnecessary variables

Represents a value

Still clear when nested

Removes the {} from your lambdas

```
return ifelse(null==question,
  () -> "What? Question missing",
  () -> {
    if (isDifficult(question))
      return question.toUpperCase();
    return concat("Don't worry: ", question);
  });
```

```
final String answer = ifelse(value==3, ()->"Yes", this::sayNo);
```

Revealing intent

`if` is a Swiss Army Knife

Have to be read more than twice

- else clause?
- validity of temporaries?
- execution paths?
- nesting?
- category?

Category?

Can be categorized

- **`when`** - without else clause
- **`mapOr`** - nullcheck
- **`unless`** - else is exceptional
- **`doWhen`** - void for side effects

Intent at head

```
process( when(value==3,
              ()->"YES"));
```

Category?

Can be categorized

- **`when`** - without else clause
- **`mapOr`** - nullcheck
- **`unless`** - else is exceptional
- **`doWhen`** - void for side effects

Intent at head

```
process( mapOr(name,
               String::toUpperCase,
               ()->"Anonymous"));
```

Category?

Can be categorized

- **`when`** - without else clause
- **`mapOr`** - nullcheck
- **`unless`** - else is exceptional
- **`doWhen`** - void for side effects

Intent at head

```
process( unless(()->"YES",
                value==3,
                ()->"NO"));
```

Category?

Can be categorized

- **`when`** - without else clause
- **`mapOr`** - nullcheck
- **`unless`** - else is exceptional
- **`doWhen`** - void for side effects

Intent at head

```
doWhen(value==3,
        ()->println("YES"));
```

Category?

or with type hygiene

```
<T,V> V either(T t,
               Predicate<T> p,
               Function<T,V> then,
               Function<T,V> otherwise){
  return ifelse(p.test(t),
                () -> truth.apply(t),
                () -> otherwise.apply(t));
}
```

```
String s= either(value,
                 this::is3,
                 v->"III",
                 Integer::toBinaryString);
```

Category?

or with type hygiene

```
<T,V> V either(T t,
               Predicate<? super T> p,
               Function<? super T,? extends V> then,
               Function<? super T,? extends V> otherwise){
  return ifelse(p.test(t),
                () -> truth.apply(t),
                () -> otherwise.apply(t));
}
```

```
String s= either(value,
                 this::is3,
                 v->"III",
                 Integer::toBinaryString);
```

without variables

Functional programming

- without variables
- composing functions

Implies

- no if statements
- no loops

loops

```
<T> int count(T t, Iterable<?> haystack) {
  int hits=0;
  for(Object e:haystack){
    if (Objects.equals(t, e))
      hits=hits+1;
  }
  return hits;
}
```

Imposes variables

loops

```
<T> int count(T t, Iterable<?> haystack) {
  int hits=0;
  Iterator<?> i=haystack.iterator();
  while(i.hasNext()){
    if (Objects.equals(t, i.next()))
      hits=hits+1;
  }
  return hits;
}
```

Imposes variables

- next() mutates

loops

```
<T> int count(final T t, final Iterable<?> haystack) {
  final int hits=0;
  final Iterator<?> i=haystack.iterator();
  return count(t, hits, i);
}

<T> int count(final T t, final int hits, final Iterator<?> i) {
  if(i.hasNext())
    return count(t,
                 hits + (Objects.equals(t, i.next()) ? 1 : 0),
                 i);
  return hits;
}
```

Recursion turns variables into values

loops

```
<T> int count(final T t, final Iterable<?> haystack) {
  final int hits=0;
  final Iterator<?> i=haystack.iterator();
  return count(t, hits, i);
}

<T> int count(final T t, final int hits, final Iterator<?> i) {
  return ifelse(i.hasNext(),
    ()-> count(t,
               increasedWhenEqual(hits, t, i.next()),
               i),
    ()-> hits);
}
```

Stack and Iterator

Recursive calls with
Higher Order or Trampolines

Trampolines

Continuation Passing Style

A compiler construct?

Add a continuation to the returned value

```
Continuation<V>{
  Continuation<V> next;
  V value;
}
```

Trampolines

```
<T> int count(T t, Iterable<?> haystack) {
  return trampoline(count(t, 0, haystack.iterator()));
}

<T> Supplier<Continuation<Integer>> count(T t, int hits, Iterator<?> i) {
  return () -> ifelse(i.hasNext(),
    () -> recur(count(t, increasedWhenEqual(hits, t, i.next()), i)),
    () -> done(hits));
}
```

```
<T> int count(T t, Iterable<?> haystack) {
  return count(t, 0, haystack.iterator());
}

<T> int count(T t, int hits, Iterator<?> i) {
  return ifelse(i.hasNext(),
    () -> count(t, increasedWhenEqual(hits, t, i.next()), i),
    () -> hits);
}
```

Will not eat stack

Trampoline implementation

```
class Continuation<V> {
  final Supplier<Continuation<V>> fun;
  final V value;
  Continuation(Supplier<Continuation<V>> fun, V value)...
}

<V> Continuation<V> recur(Supplier<Continuation<V>> f) {
  return new Continuation(f, null);
}

<V> Continuation<V> done(V v) {
  return new Continuation(null, v);
}

<V> V trampoline(Supplier<Continuation<V>> f) {
  Continuation<V> r = new Continuation<>(null, null);
  while(f!=null) {
    r = f.get();
    f = r.fun;
  }
  return r.value;
}
```

```
<T> int count(T t, Iterable<?> haystack) {
  return trampoline(count(t, 0, haystack.it
}

<T> Supplier<Continuation<Integer>> count(T
  return () -> ifelse(i.hasNext(),
    () -> recur(count(t, increasedWhenEqual
    () -> done(hits));
}
```

Trampolines

Optimized Tail calls

A bit more crazy signature

`Supplier<Continuation<T>>`

Reduces variables

```
<T> int count(T t, Iterable<?> haystack) {
  return trampoline(count(t, 0, haystack.iterator()));
}

<T> Supplier<Continuation<Integer>> count(T t, int hits, Iterator<?> i) {
  return () -> ifelse(i.hasNext(),
    () -> recur(count(t, increasedWhenEqual(hits, t, i.next()), i)),
    () -> done(hits));
}
```

Getting rid of the Iterator

- Higher Order
  - Conversely, with lazy evaluation
- Other data structures
  - Immutable data structures

Higher Order Functions

Functions that take functions

and defer evaluation.

Exercised for composition and polymorphism

Command Pattern

May implement internal iteration

map, filter, fold

Turns variables into values

```
Iterable<?> a = map( v->v+1, asList(1,2,3));
process(a.iterator());
=>[2, 3, 4]
```

Strict Map

```
<T,V> Iterable<V> map(Function<T,V> f, Iterable<T> i ){
  List<V> r= new ArrayList<>();
  i.forEach(t -> r.add(f.apply(t)));
  return r;
}
```

```
Iterable<?> a = map( v->v+1, asList(1,2,3));
process(a.iterator());
=>[2, 3, 4]
```

Strict Map

```
<T,V> Iterable<V> map(Function<T,V> f, Iterable<T> i ){
  List<V> r= new ArrayList<>();
  i.forEach(t -> r.add(f.apply(t)));
  return r;
}
```

```
Iterable a = take(3, map( v->v+2, range()));
process(a.iterator());
=> OutOfMemory
```

```
<T> Iterable<T> take(int num, Iterable<? extends T> ts);
Iterable<Integer> range();
```

Lazy Map

```
<T,V> Iterable<V> map(Function<T,V> f,
                      Iterable<T> i){
   return ()->
     new Iterator<V>() {
       Iterator<T> in = i.iterator();

       @Override
       public boolean hasNext() {
         return in.hasNext();
       }

       @Override
       public V next() {
         return f.apply(in.next());
       }
     };
  }
```

```
Iterable a = take(3, map( v->v+2, range()));
process(a.iterator());
=> [2, 3, 4]
```

Quickly becomes complex

next() both mutates and delivers

Streams

Higher order, internal iteration on sequential data

Composable as data isn't realized between components

Lazy evaluation

```
Stream.iterate(0, v -> v+1).map(v -> v+2).limit(3).collect(toList());
=>[2,3,4]
```

Stream difficulties

Is object oriented

- Difficult to add new functionality

Transformation has to be stateless

- Difficult ordering semantics

Are Stream functions trusty?

```
Stream<Integer> incEach(Stream<Integer> s){
  return s.parallel().map(e->e+1);
}
```

Stream difficulties

Laziness and ordering is good

Iterators are ordered

- simple
- and well known

Internal iteration is a simple thing

Complex lazy Iterators

next() both mutates and delivers

```
interface Iterator<T>{
  boolean hasNext();
  T next();
}
```

Compose Iterable functions

Single function model

We can use trampoline

Iterable transformers

Trampoline functions:

lazy : Creates lazy Iterable from continuation supplier, similar to trampoline

seq : Recurs with intermediate value, retrieved by a lazy Iterator

```
Iterable<String> i=
  take(3,
     map( v->v+1,
        range(0))));
```

```
<T, V> Iterable<V> map(Function<T, V> f,
                             Iterable<T> in){
  return () -> {
    Iterator<T> i=in.iterator();
    if( i.hasNext())
      return lazy(mapI(f, i)).iterator();
    return emptyIterator();
  };
}

<T, V> Supplier<Continuation<V>>
mapI(Function<T,V> f, Iterator<T> i) {
  return () ->
    either(f.apply(i.next()),
           v->i.hasNext(),
           v->seq(mapI(f, i), v),
           v->done(v));
}
```

Iterable transformers

Trampoline functions:

lazy : Creates lazy Iterable from continuation supplier, similar to trampoline

seq : Recurs with intermediate value, retrieved by a lazy Iterator

```
Iterable<String> i=
  take(3,
     map( v->v+1,
        range(0)))));
```

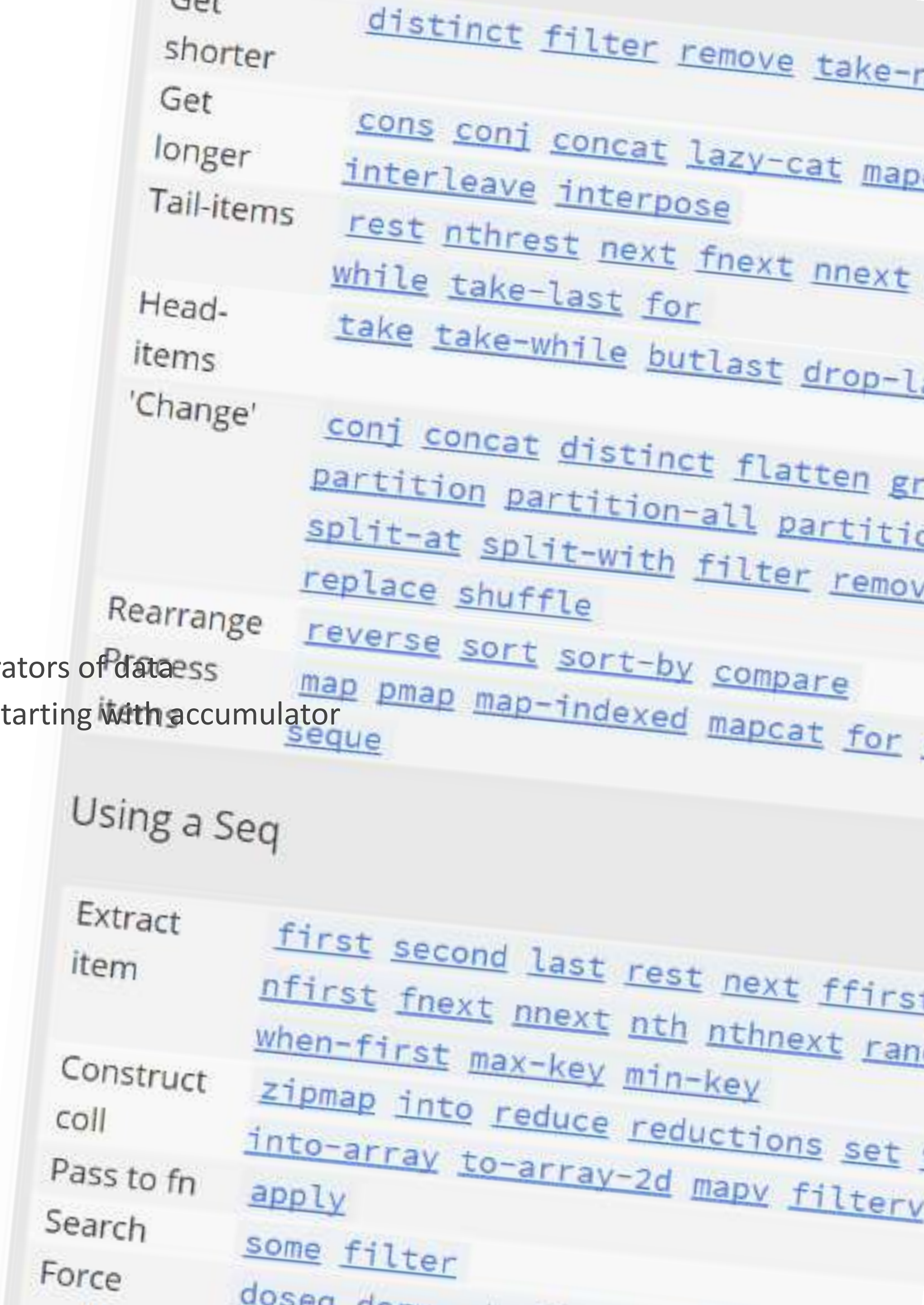
```
<T> Iterable<T> take(int num,
                     Iterable<T> in) {
  return () ->
    either(in.iterator(),
      i-> i.hasNext() && num>0,
      i-> lazy(takeI(num, i)).iterator(),
      i-> emptyIterator());
}
```

```
<T> Supplier<Continuation<T>>
takeI(int num, Iterator<T> i) {
  return () ->
    either(i.next(),
      v-> i.hasNext() && num>0,
      v-> seq(takeI(num-1, i), v),
      v-> done(v));
}
```

Iterable transformers

rest is the iterable with all but the first element
next is rest but never empty
with is the iterable with an element prepended
withLast is the iterable with an element appended
drop is the iterable with a number of elements dropped
dropLast has all elements except the last
dropWhile is the iterable of with all first matching a predicate removed
flatMap is the iterable of concatenations of the result of applying map to every element in iterators of data
reductions is the iterable of producing intermediate values of the reduction of elements by f, starting with accumulator
takeNth is the iterable of every nth element
partition is the iterable of num length iterables
partitionBy is the iterable of iterables grouped by application
splitAt is the tuple of iterables split at i, where both are lazy
repeatedly is the iterable of continuous supplier retrieval
repeat is the iterable of continuous value
range is the iterable of increasing integers
cycle is the iterable of repeated iterable

Easier than expected using trampolines
Laziness makes them composable



Getting rid of the Iterator

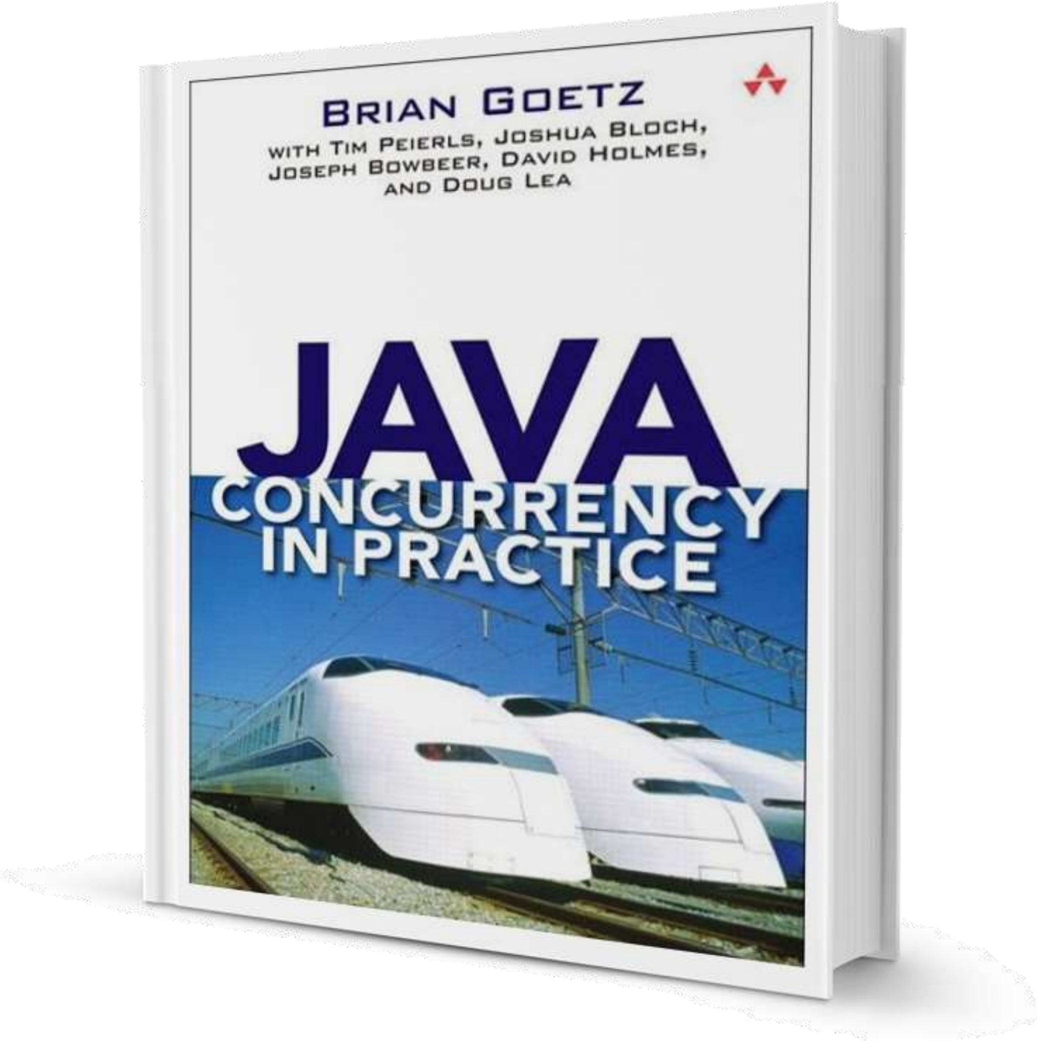
Higher Order

- Conversely, with lazy evaluation

Other data structures

- Immutable data structures

```
<T> int count(T t, Iterable<?> haystack){
  return fold((a,v)-> a+1, 0,
     filter(v->Objects.equals(t,v), haystack));
}
```


Brian Goetz
with Tim Peierls, Joshua Bloch,
Joseph Bowbeer, David Holmes,
and Doug Lea
JAVA
CONCURRENCY
IN PRACTICE

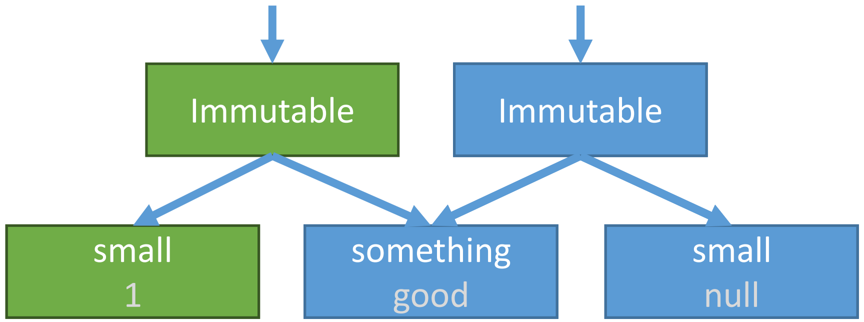
Anatomy of an Immutable

Final primitive or...

```
a = immutable().withSomething("good");
b = a.withSmall(1);
c = mapOr(a.small,
          (size)->a.withSmall(size-1),
          ()->immutable().withSmall(0));
```

a is always only good

Updated from leaf to root




```
class Immutable {
    public final String something;
    public final Integer small;

    public Immutable(String something, Integer small) {
        this.something = something;
        this.small = small;
    }

    public static Immutable immutable(){
       return new Immutable(null, null);
    }

    public Immutable withSomething(String something) {
        return new Immutable(something, small);
    }

    public Immutable withSmall(int small) {
        return new Immutable(something, small);
    }
}
```

Persistent Data Structures

Immutable data structures

- Vector, Map, Set...

Updates are pure functions

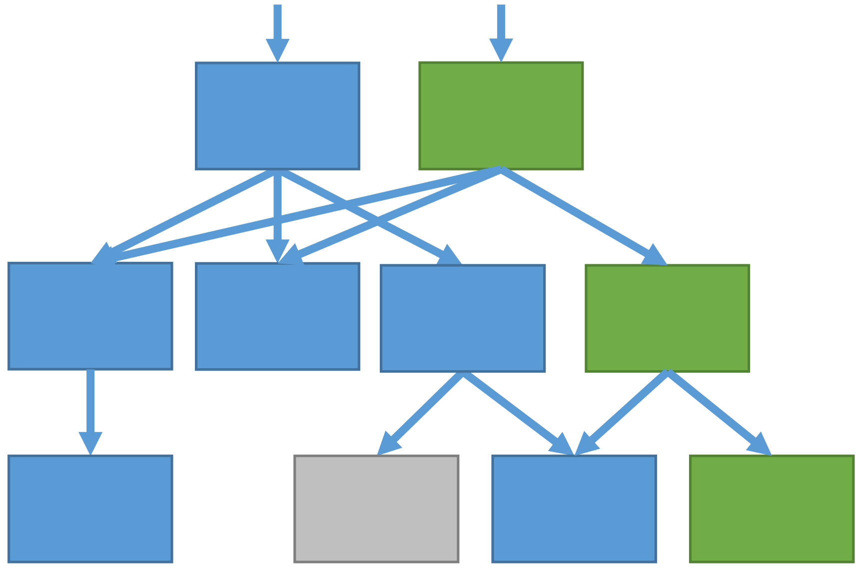
- returning modified versions
- shared structures

Maintained algorithmic complexity

Maps are usually trees

Sets are maps, without values

Vectors has highly branching nodes



Difficulty creating Immutables?

Persistent Data Structures

Default in functional languages

Values can be preserved

Separation of identity and state

Identity with values over time

Atomic state

Preserved throughout a transaction

AtomicReference is transactions of one

Optimistic state transformation

```
class AtomicReference<V>{
  boolean compareAndSet(V expect, V update) ...
  V updateAndGet(UnaryOperator<V> updateFunction) ...
```

Program state is one thing

Not many

Iterators redundant

```
<T> int count(T t, PersistentList<?> haystack) {
  return trampoline(count(t, 0, haystack));
}

<T> Supplier<Continuation<Integer>> count(T t, int hits, PersistentList<?> i) {
  return () ->
    ifelse(i.isEmpty(),
      () -> done(hits),
      () -> recur(count(t,
                        increasedWhenEqual(hits, t, i.get()),
                        i.without())));
}
```

haystack is always haystack

Iterators redundant

```
<T> int count(T t, PersistentVector<?> haystack) {
  return trampoline(count(t, 0, haystack));
}

<T> Supplier<Continuation<Integer>> count(T t, int hits, PersistentVector<?> i){
  return () ->
    ifelse(i.isEmpty(),
      () -> done(hits),
      () -> recur(count(t,
                        increasedWhenEqual(hits, t, i.get(0)),
                        i.subList(1, i.size()))));
}
```

haystack is always haystack

https://github.com/stefanvstein/stonehorse.candy
https://github.com/stefanvstein/stonehorse.grit
Search "Candy" or "Grit" on Maven central

stefan.von.stein@zacco.com