Lambda Ausdrücke,

funktionale Interfaces und Methodenreferenzen

Lambda Expressions

```
(a,b,c,..) -> { body }
```

Parameterlist

- If there is exactly one parameter, we can omit the parenthesis
- If the body is a single expression, we can omit return and the curly braces
- Parameters must not hide local variables

Examples

- `dec = (Integer e) -> { return e - 1; }`
- `square = (e) -> e * e;`
- `inc = e -> e + 1;`
-
```
div = (n1,n2) -> {
  if (n2 != 0)
   return (n1/n2);
  else
   throw new IllegalArgumentException("...");
};
```

Using Lambda Expressions

- Can be passed to methods if a functional interface is expected
- Single method interface (+ default methods)
- Predefined interfaces: java.util.function

java.util.function

- **Consumer** a → void, **BiConsumer** (a,b) → void
- **Function** a → b, **BiFunction** (a,b) → c
- **Supplier** ( ) → a
- **Predicate** a → boolean **BiPredicate** (a,b) → boolean
- **UnaryOperator** a → a, **BinaryOperator** (a,a) → a
- Specialized interfaces for primitives (double, int, long)

Happy little accidents

Bild: The joy of painting

Evaluating Lambda Expressions

Method name depends on the functional interface

- `someFunction.apply(3);`
- `someConsumer.accept("foo");`
- `somePredicate.test("bar");`
- `someSupplier.get();`
- `someRunnable.run();`
- `...`

Anonymous inner classes?

- We know how to pass functions around almost since the dawn of Java

- So, are Lambda Expressions just syntactic sugar?

```java
new Thread(new Runnable() {
  public void run() {
    // do some stuff
  }
}).start();
```

```java
new Thread(() -> {/* do some stuff */}).start();
```

Favor lambda expressions over anonymous inner classes

Don't overuse lambdas

```
n -> {
  int sum = 0;
  for (int i = 1; i < n; i++) {
    if (n % i == 0) {
      sum += i;
    }
  }
  return n == sum;
};
```

Lambda expressions are great for simple small functions

You can have names!

```
private boolean perfect(int n) {
  int sum = 0;
  for (int i = 1; i < n; i++) {
    if (n % i == 0) {
      sum += i;
    }
  }
  return n == sum;
}
```

```
n -> perfect(n);
```

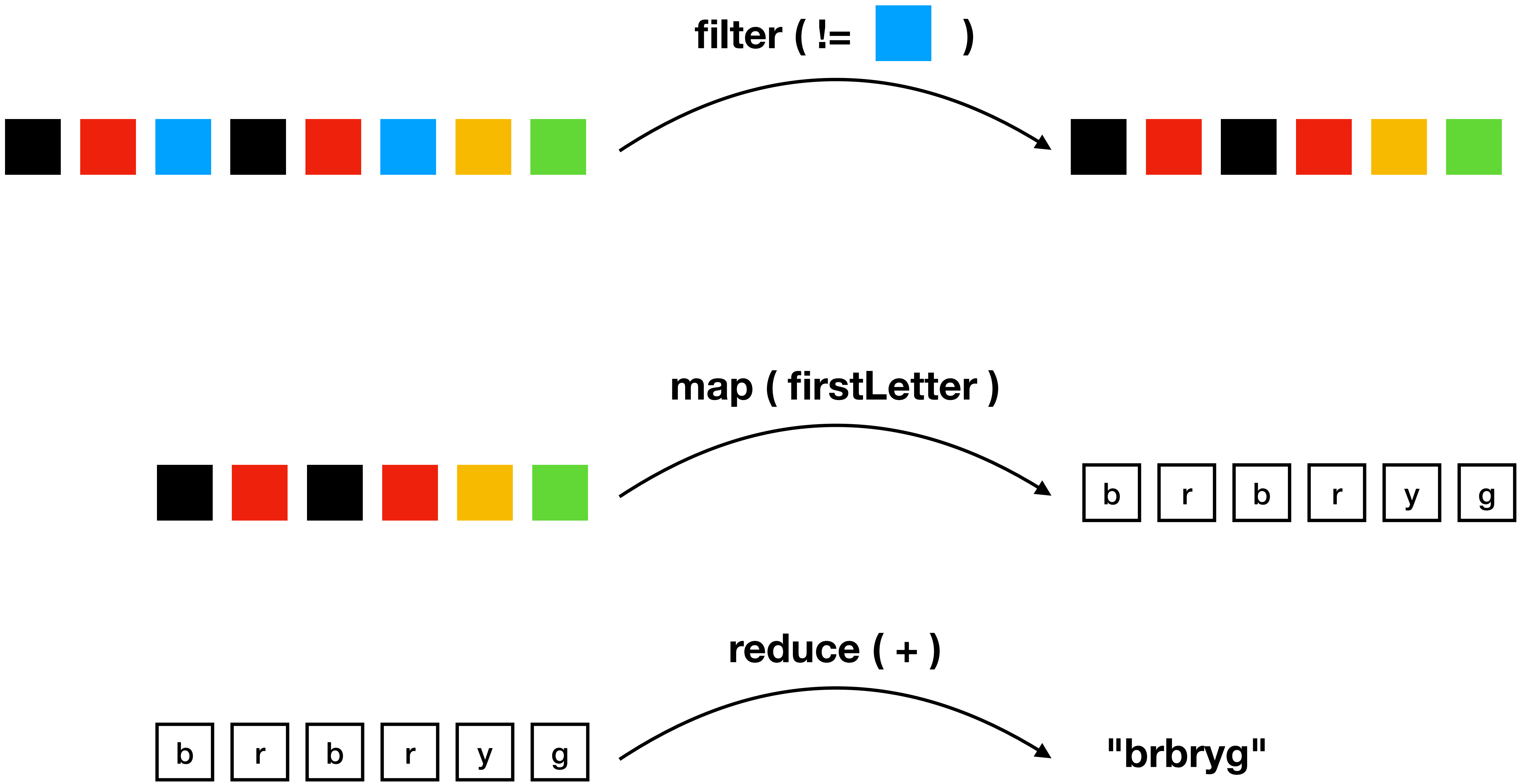
Method References

- `x -> MyClass.staticMethod(x)` ⇒ `MyClass::theMethod`
- `x-> obj.instanceMethod(x)` ⇒ `obj::instanceMethod`
- `(a,b) -> a.instanceMethod(b)` ⇒ `MyClass::instanceMethod`
- `x -> new MyClass(x)` ⇒ `MyClass::new`

Extract code into methods

Higher Order Functions

The usual suspects ...



DIY HOF

Function as a parameter

```
static <A, B> Collection<B> map(Function<A, B> f, Collection<A> c) {
  Collection<B> result = new ArrayList<>();
  for (A a : c) result.add(f.apply(a));
  return result;
}
```

Function as a return value

```
static <A> Supplier<A> constantly(A v) {
  return () -> v;
}
```

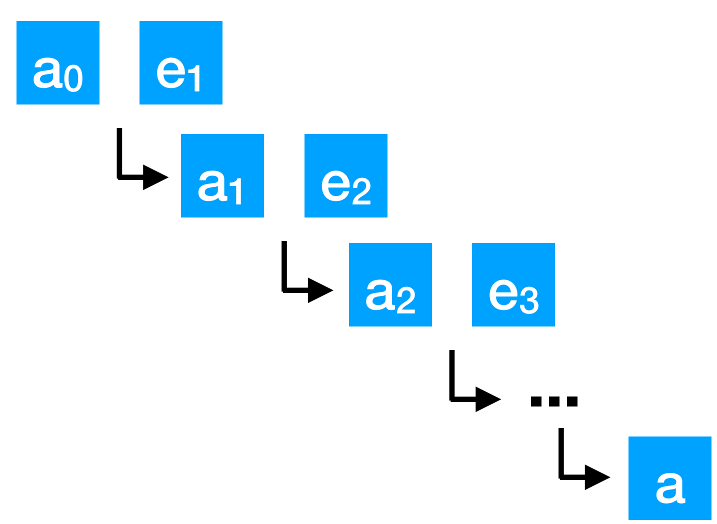
Function as a parameter and a return value

```
static <A,B,C> Function<A, C> compose(Function<B,C> g, Function<A,B> f) {
  return a -> g.apply(f.apply(a));
}
```

reduce

reduce function
(a,e) → a

```java
static <A, E> A reduce(A init, BiFunction<A, E, A> rf, Collection<E> c) {
  A a = init;
  for (E e : c) a = rf.apply(a,e);
  return a;
}
```



Streams

Processing Collections

- Before Java 8: Loops
- Since Java 8: Streams

What's wrong with the good old for loop that I used for 20 years?

Given a collection ns of numbers:

- Find the sum
- of the squares
- of the first k numbers
- of the even numbers
- greater than 5
- If there are less than k matching numbers take all of them

ns = [10,~~11~~,12,~~17~~,14,~~2~~,~~3~~,~~4~~,~~5~~,20,~~27~~,~~12~~], k = 4

100 144 196 400 → 840

ns = [10,~~11~~,12,~~17~~,14,~~2~~,~~3~~,~~4~~,~~5~~,20,~~27~~,12], k = 8

100 144 196 400 144 → 984

Inspired by Venkat Subramaniam's talk given at US Devoxx

Solution A

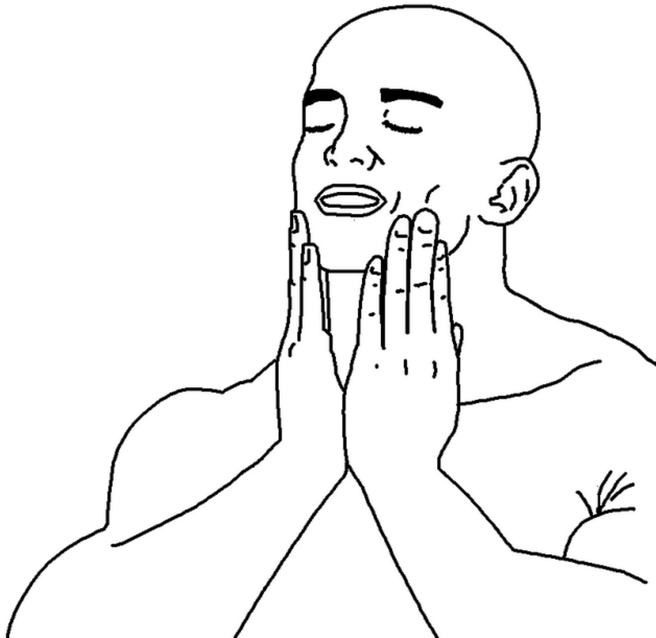
```java
public static int compute(List<Integer> ns, int k) {
    int result = 0;
    int found = 0;
    for (int i = 0; (i <  ns.size() && found < k); i++) {
        int v = ns.get(i);
        if (v > 5 && v % 2 == 0) {
            result += v * v;
            found++;
        }
    }
    return result;
}
```

Is this code correct?

Solution B

```
public static int compute(List<Integer> ns, int k) {
  return ns.stream()
           .filter(x -> x % 2 == 0)
           .filter(x -> x > 5)
           .map(x -> x * x)
           .limit(k)
           .reduce(0, (x,y)->x+y);
}
```

How about this code?



So ... what is the problem with for loops?

http://www.sideshowcollectors.com/forums/marvel-action-figures/176882-spectulation-hot-toys-mms-spider-homecoming-vulture-collectible-figure-20.html

It's simpler!

- Reasoning about imperative code is hard!
  - Brain has to simulate the computer
  - Iteration logic mixed* with domain logic
- Reasoning about functional code is easier
  - Single pass
  - Iteration logic is abstracted away

*** or as a Clojure programmer would say: complected**

Streams

- Generate a stream (e.g. from a collection)
- Apply some intermediate operations (returning a stream)
  Examples: map, filter, distinct, sorted, ...
- Apply a terminal Operation (reduces the stream)
- Note
  - Operations are combined into a single operation
  - Evaluation is driven by the terminal operation - without a terminal operation it is a no op (lazy evaluation)

Streams

```
.filter(x -> x % 2 == 0)
.filter(x -> x > 5)
.map(x -> x * x)
.limit(k)
```


```
ns                                          .reduce(0,(x,y)->x+y)
```

Streams

- Stream Creation
- Intermediate Operations (transform, filter, ...)
- Terminal Operations

Performance

```
public static int compute(List<Integer> ns, int k) {
  return ns.stream()
          .filter(x -> x % 2 == 0)   unboxing
          .filter(x -> x > 5)        unboxing
          .map(x -> x * x)           unboxing + boxing
          .limit(k)
          .reduce(0, (x,y)->x+y);    unboxing + boxing
}
```

A lot of unboxing and boxing is going on

Faster: Use specialized primitive streams

Performance

```
public static int compute(List<Integer> ns, int k) {
  return ns.stream()
          .filter(x -> x % 2 == 0)
          .filter(x -> x > 5)
          .mapToInt(x -> x * x)
          .limit(k)
          .sum();
}
```

IntStream instead of Stream<Integer>

Specialized reduce methods

collect

- reduce - immutable reduction
  - Initial value of Type A
  - reducing function BiFunction<A,E,A>
  - joining function: BinaryOperator<A>
- collect - mutable reduction
  - Supplier<A> for initial value
  - BiConsumer<A,E>
  - BiConsumer<A,A>

Collectors

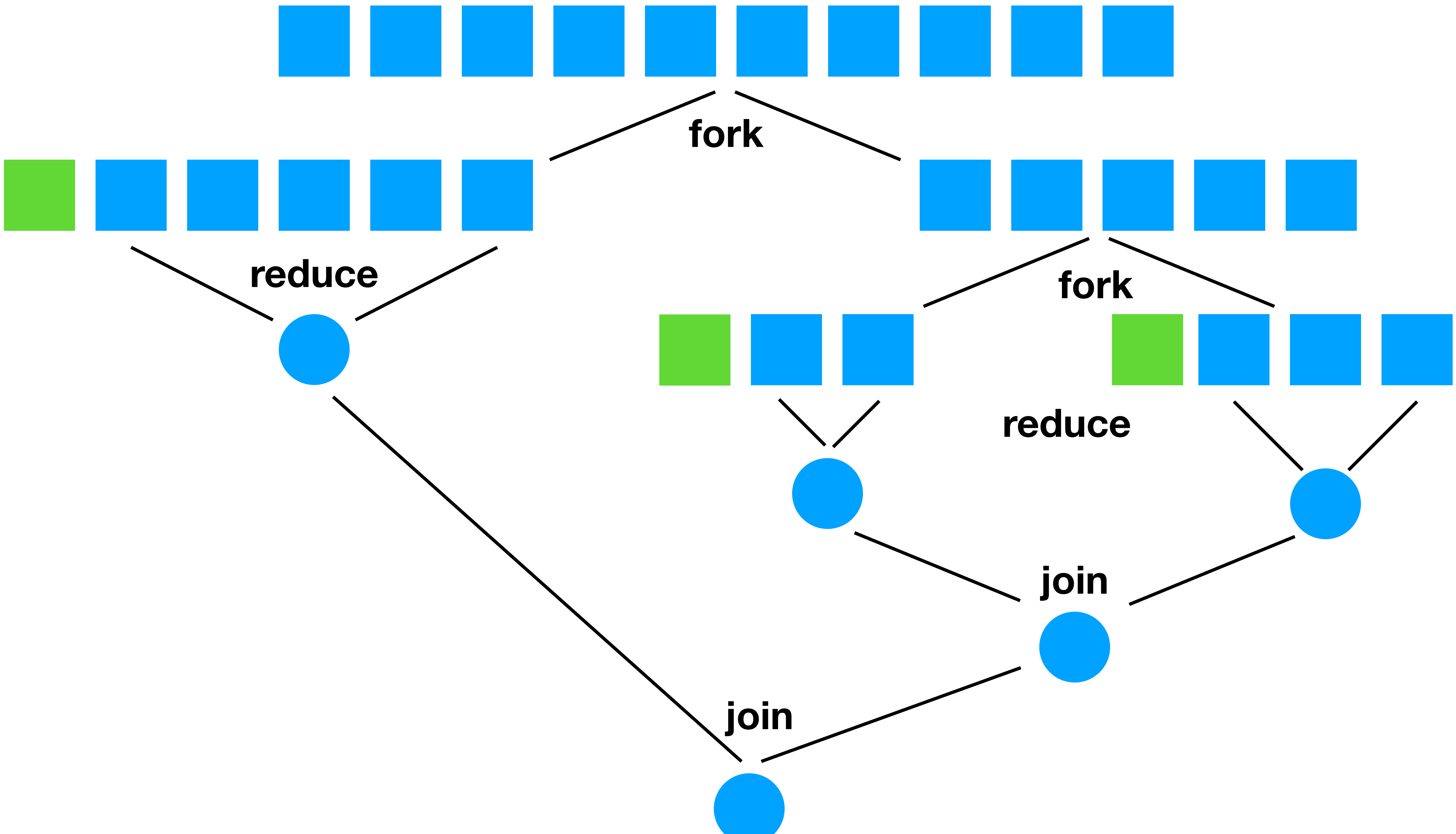
- collect can take a java.util.stream.Collector (encapsulates the functions)
- see java.util.stream.Collectors for predefined Collectors
  - toList() / toSet()
  - joining(...)
  - groupingBy(...)
  - ...

```
.parallel()
```

Parallel Streams (Mutable Reduction)

- Supplier<A> for initial value
- BiConsumer<A,E> accumulator (reducing function)
- BiConsumer<A,A> combiner (joining function)

Parallel stream



Uses fork join framework (Java 7)

Constraints

- **Not everything can be done in parallel!**
- Reduce (accumulator) function must be associative
- Initial value must be the (left) identity for the reduce function
- Combine function must be associative

Not everything is faster!

Always test with parallel streams