O'REILLY®

Java Collections, Generics & Streams - May 2020

About Me - Career

- New Relic, Principal Engineer
- jClarity, Co-founder
  - Sold to Microsoft
- Deutsche Bank
  - Chief Architect (Listed Derivatives)
- Morgan Stanley
  - Google IPO
- Sporting Bet
  - Chief Architect

About Me - Community

- Java Champion
- JavaOne Rock Star Speaker
- Java Community Process Executive Committee
- London Java Community
  - Organising Team
  - Co-founder, AdoptOpenJDK

Installing Java

- We will use AdoptOpenJDK - Java 11

  https://adoptopenjdk.net/

- We will use “OpenJDK 11 (LTS)” and the HotSpot JVM

- We will also use IntelliJ IDEA - Community Edition

- For more on the history of Java - free book:
  - “Java: The Legend” (O’Reilly 2015)

    https://learning.oreilly.com/library/view/java-the-legend/9781492048299/cover.html

Java Type System

- Review: Object Orientation and Inheritance
- Review: Access Control & Interfaces
- Working with Interfaces
- Lambda Expressions
- Working with Lambdas

Review: Object Orientation

- Object orientation focuses on data (state)
  - Functionality is associated with the data elements

- Objects have three primary properties
  - Attributes (fields)
  - Behaviour (methods)
  - Identity (reference identity)

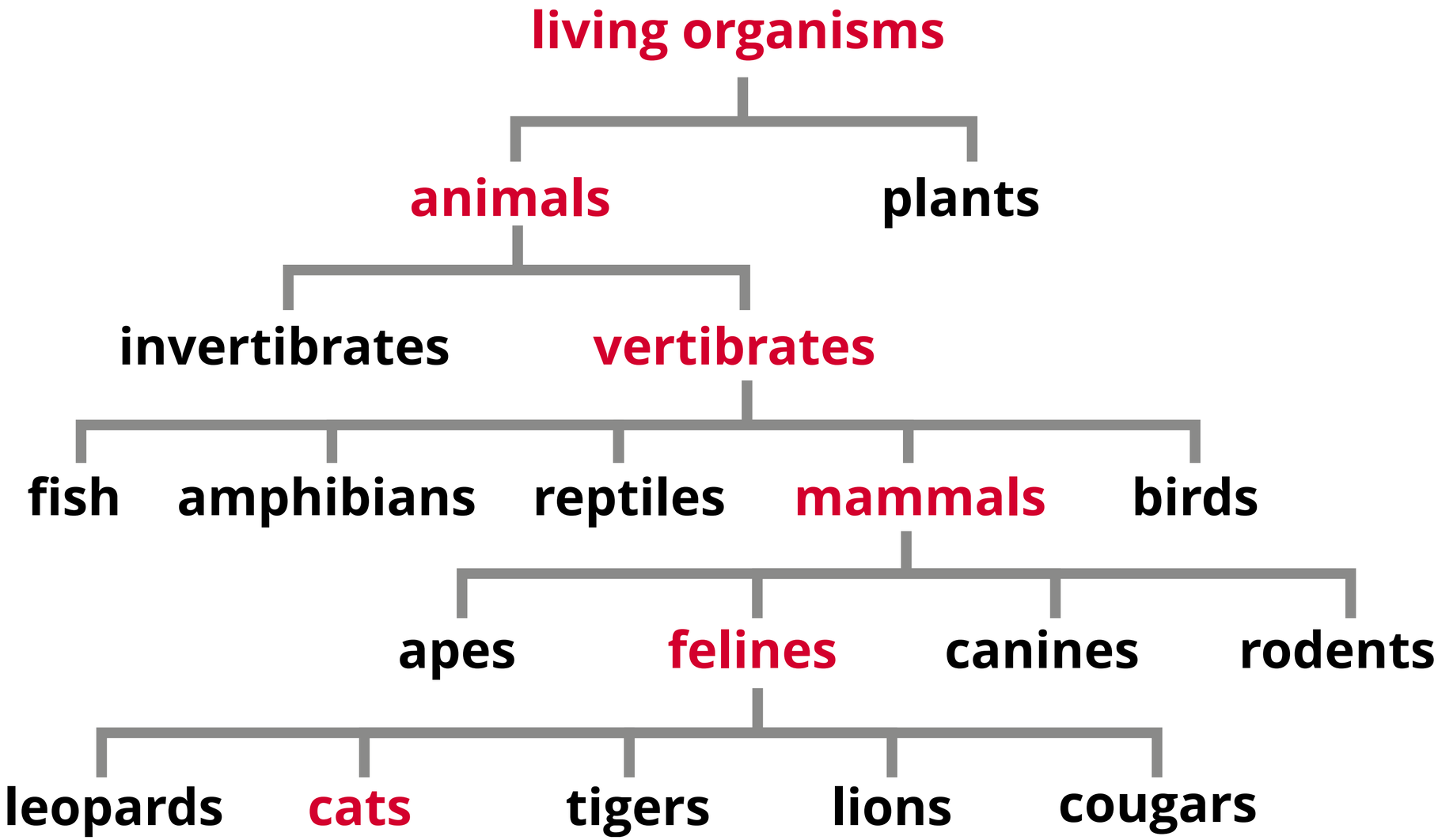
Review: Object Orientation

- Objects of a given type belong to a class
  - Blueprint for building objects
- Classes may be related by inheritance
  - Define one class in terms of another
  - Brings many advantages
- Key concept is “(Liskov) Substitutability”

Inheritance

- Every Java class has exactly one parent class
  - Except **`java.lang.Object`**
- Specified in class declaration by the `extends` keyword
- If a class doesn't specify a parent
  - The parent is taken to be **`Object`**
- We say the class inherits from the parent class
- This means that classes form an inheritance hierarchy
  - With **`Object`** at the top

Inheritance Hierarchy – Biology



Wrapper Classes

- Primitive types are not classes
- Primitive variables are not objects
  - do not need to be instantiated
  - do not have methods associated with them

`Byte`	`Float`
`Short`	`Double`
`Integer`	`Boolean`
`Long`	`Character`

Wrapper Classes

- Primitive types are not classes
- Primitive variables are not objects
  - do not need to be instantiated
  - do not have methods associated with them
- Wrappers give primitives object behaviour
  - used when primitives cannot be used (eg. in collection classes)
  - useful location for relevant methods (numeric, character)
  - conversion to and from wrapper objects automatic

`Byte`	`Float`
`Short`	`Double`
`Integer`	`Boolean`
`Long`	`Character`

```
int n = Integer.parseInt("123");

while (n < Integer.MAX_VALUE)
    n = n + 1;
```

Interfaces

- Interfaces represent only an API
  - Provides a description of a type
  - Methods that classes which implement that API must supply

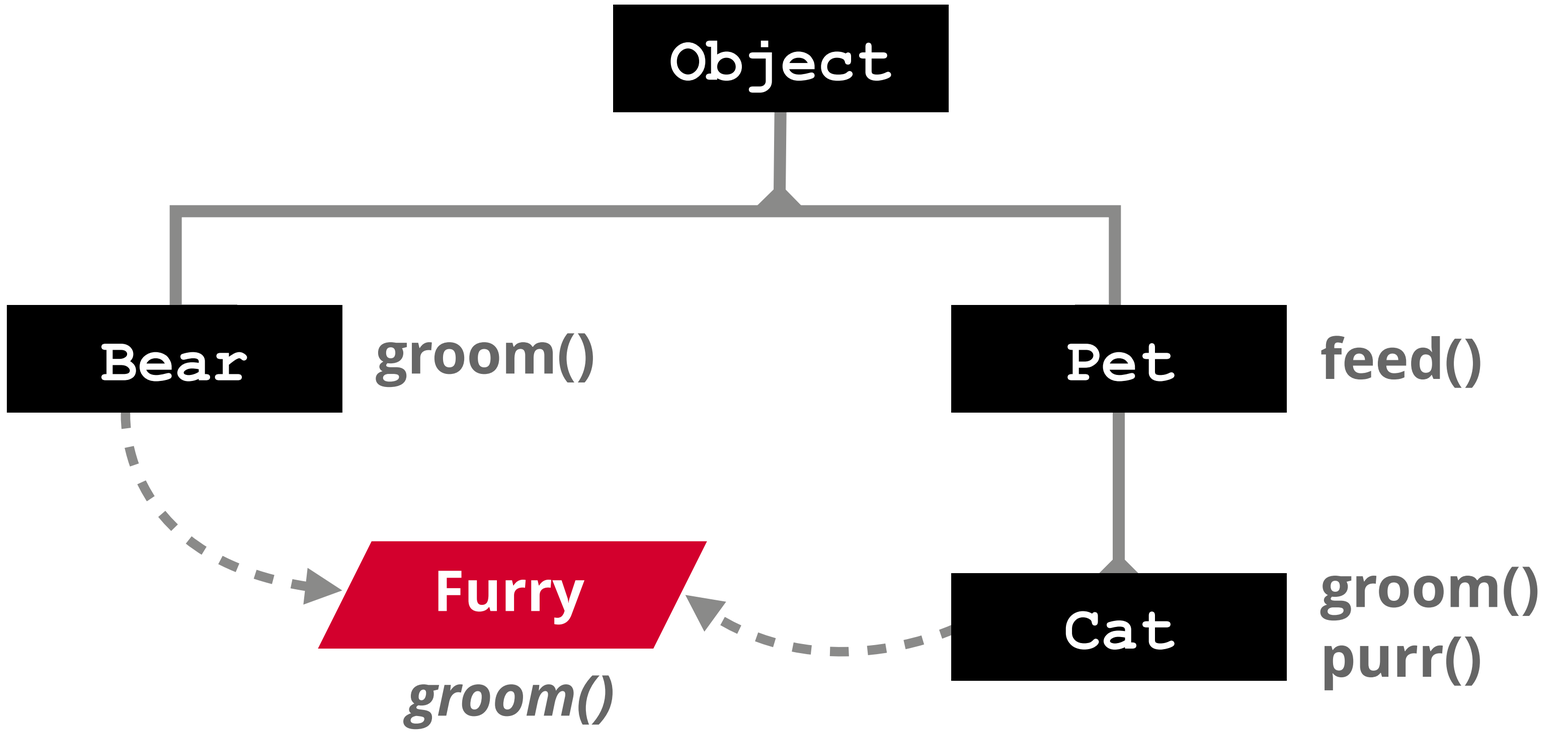
Interfaces

- Interfaces represent only an API
  - Provides a description of a type
  - Methods that classes which implement that API must supply
- Interface does not usually provide any implementation
  - Methods are considered mandatory
  - Interfaces **must** implement these methods

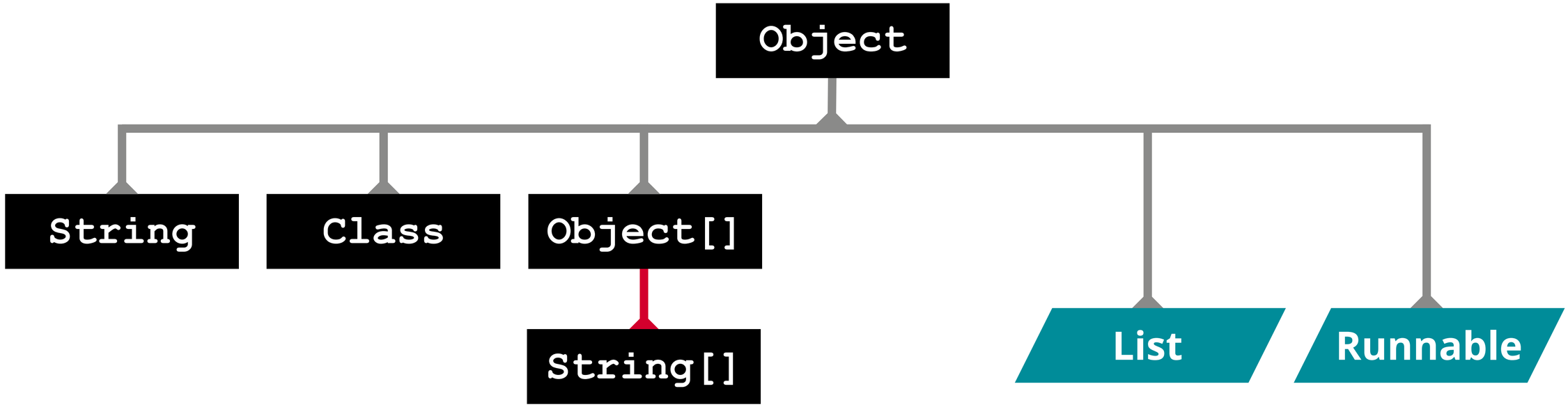
Interfaces

- Interfaces represent only an API
  - Provides a description of a type
  - Methods that classes which implement that API must supply
- Interface does not usually provide any implementation
  - Methods are considered mandatory
  - Interfaces **must** implement these methods
- Some methods may be optional
  - Called **`default`** methods
  - The interface must supply a default implementation

Class Hierarchy



Inheritance Hierarchy – Java SDK



Lambdas - Motivation

- More expressive programming
- Better libraries
- Concise code
- Improved programming safety
- Data parallelism

FP in Java from first principles

- How do we model functions?
- How do we combine functions?
- What other concerns are there?

Lambdas - Motivation

- Treat code (logic) as though it were data (a value)
- Write the bit of code as a “function literal”
- Then assign it to a variable
- Pass it around like any other value

FP in Java from first principles

IDE

Type Inference

- Lambda is auto-converted to the appropriate type
- Single Abstract Method (SAM) type
- Correct static typing
- Name of method doesn’t matter
- Uses an API called Method Handles…

JVM Function Objects

- In Java, Functions Are Objects
- In the simplest case:
  - No fields
  - One extra method (apart from those of Object)
- These are the Single Abstract Method (SAM) types
  - Target types for Java lambdas
- This concept has been used in Java since 1.1 (as inner classes)

Method References

- A way to talk about a method as a literal
  - A bit like a class literal
- Equivalent to a lambda expression
- Uses C++ style ‘::’ syntax

Method Reference Example

Type	Method Reference	Lambda
Static	`System::getProperty`	`k -> System.getProperty(k)`
Bound	`System.out::println`	`s -> System.out.println(s)`
Unbound	`Trade::getPrice`	`t -> t.getPrice()`
Constructor	`Trade::new`	`price -> new Trade(price)`

Introducing Java Generics

- Element Containers in Java
- Demo: Array Covariance
- Demo: Introducing Generics
- Writing simple generic types
- Language Stability & Evolution

What is an Element Container?

- A group of elements
  - related in some way
  - usually objects

What is an Element Container?

- A group of elements
  - related in some way
  - usually objects
- Can be manipulated as a single object
  - stored in a variable
  - passed as argument to method
  - returned as a method result

What is an Element Container?

- A group of elements
  - related in some way
  - usually objects
- Can be manipulated as a single object
  - stored in a variable
  - passed as argument to method
  - returned as a method result
- Three main families in Java
  - arrays
  - collections
  - “transparent containers” (functional) - Java Streams

Arrays

- The simplest form of container
- Arrays are fixed-size, indexed containers
  - elements can be primitive or reference types

Arrays

- The simplest form of container
- Arrays are fixed-size, indexed containers
  - elements can be primitive or reference types
- Various syntactic shortcuts for handling arrays

```
int[] values = new int[5];
assertEquals(values.length, 5);
```

- Arrays are objects
  - must be created with new before use
  - new operator requires type and number of array elements

Arrays

- The simplest form of container
- Arrays are fixed-size, indexed containers
  - elements can be primitive or reference types
- Various syntactic shortcuts for handling arrays

```
int[] values = new int[5];
assertEquals(values.length, 5);
```

- Arrays are objects
  - must be created with new before use
  - new operator requires type and number of array elements
  - size of an array can be obtained from the length attribute
  - array elements are initialised to “zero bits”
    - primitives to zero (or equivalent, e.g. false, 0.0), objects to null

Array Covariance

DEMO

Introducing Generics

DEMO

Java Generics

- Use compiler to eliminate dangerous code
  - That’s the point of static typing
- Create a homogeneous aggregate type
- Formed of two parts
  - Container type
  - Payload type (Object types only)
- Container defines a family of types
  - One for each payload

```
interface Box<T> {
  void box(T t);
  T unbox();
}
```

Java Generics

- The syntax <T> is called a type parameter
- Another name for a generic type is a parameterized type
- T is a placeholder for a type, not a type
  - Can’t make any assumptions about T when we use it in a definition
  - Only basic consistency

Language Stability

- Java language evolved modestly over time
  - Generics (Java 5)
  - Lambdas (Java 8)
- Stability popular with enterprises and larger shops
  - Clear roadmaps
  - Signal Java investment protected over time

A bit of history - Java approach

- Backwards compatibility is a major concern
- Java 1.4 .class MUST work & interoperate in Java 5
  - Without modification
  - Without recompilation

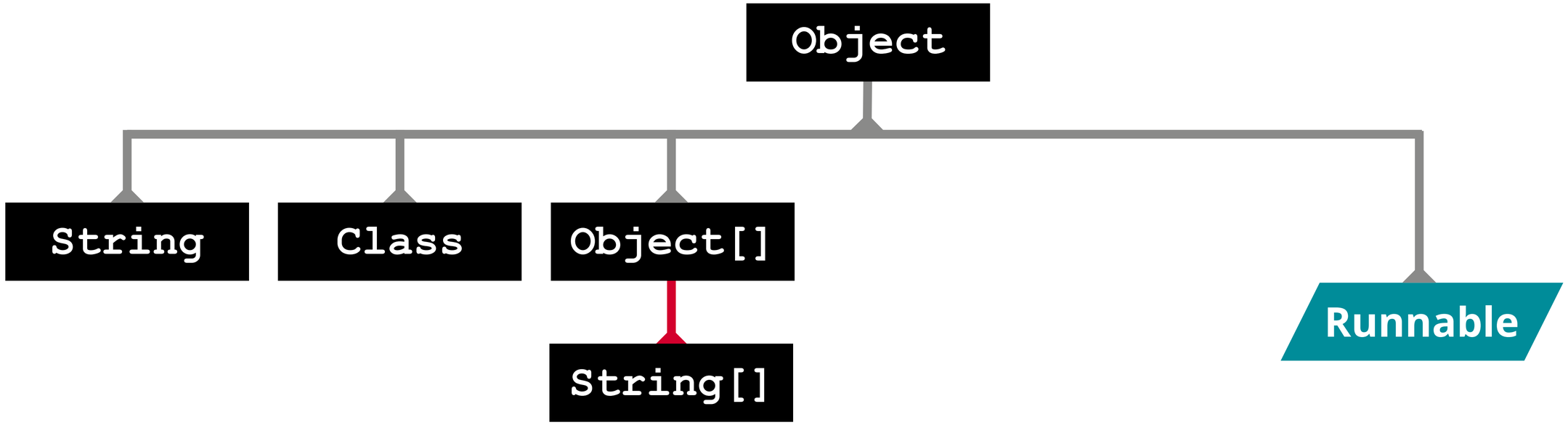
Type Erasure

- Java achieves backwards compatibility by type erasure
- Generic type parameters are only visible at compile time
- They are stripped out by javac
  - Some traces remain which can be seen via reflection

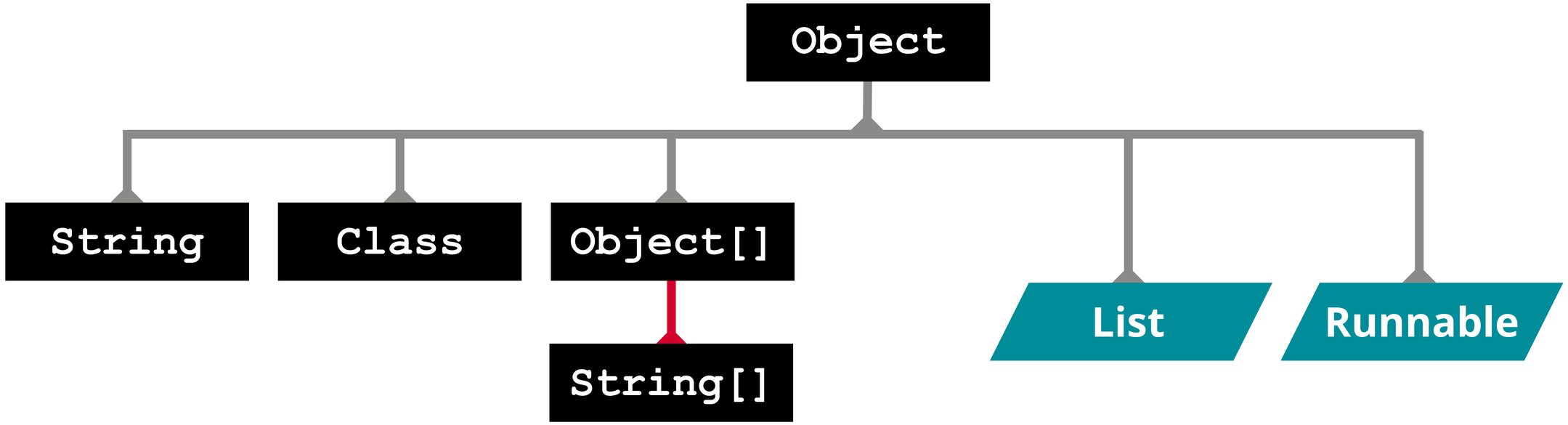
Generics

- Example of consistency & strong compatibility
- Wanted parameterised types
  - Didn’t want to sacrifice compatibility
  - Type erasure as a consequence
- Consequence
  - Libraries and language had to evolve in step
  - Required careful engineering & planning

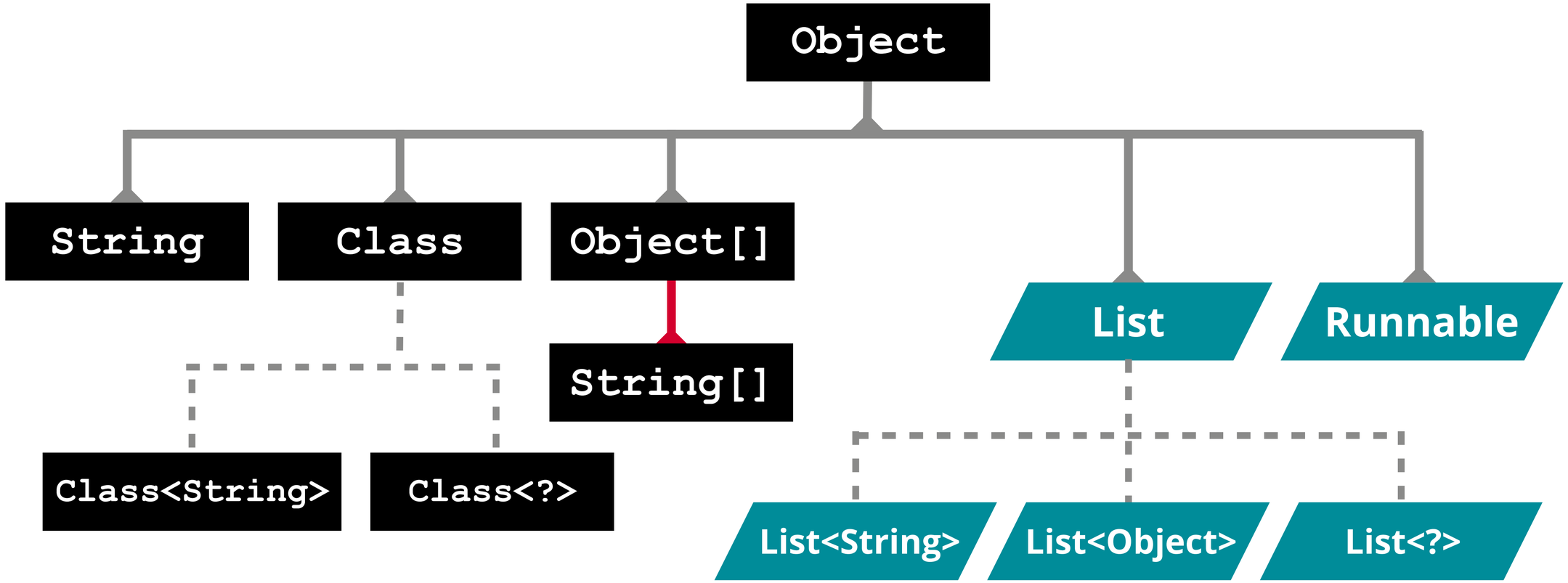
Language evolution (1.0)



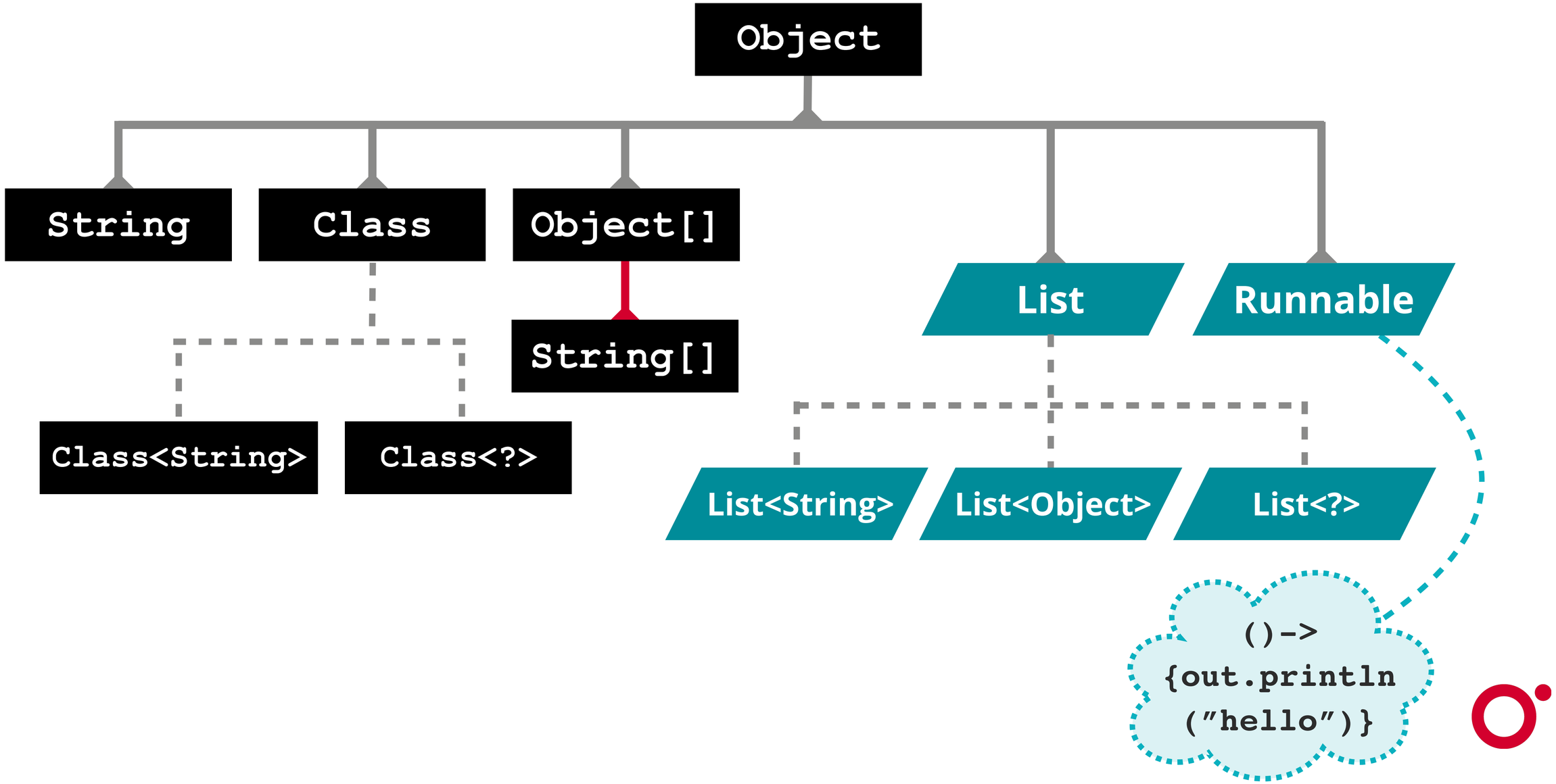
Language evolution (1.2)



Language evolution (5)



Language evolution (8)



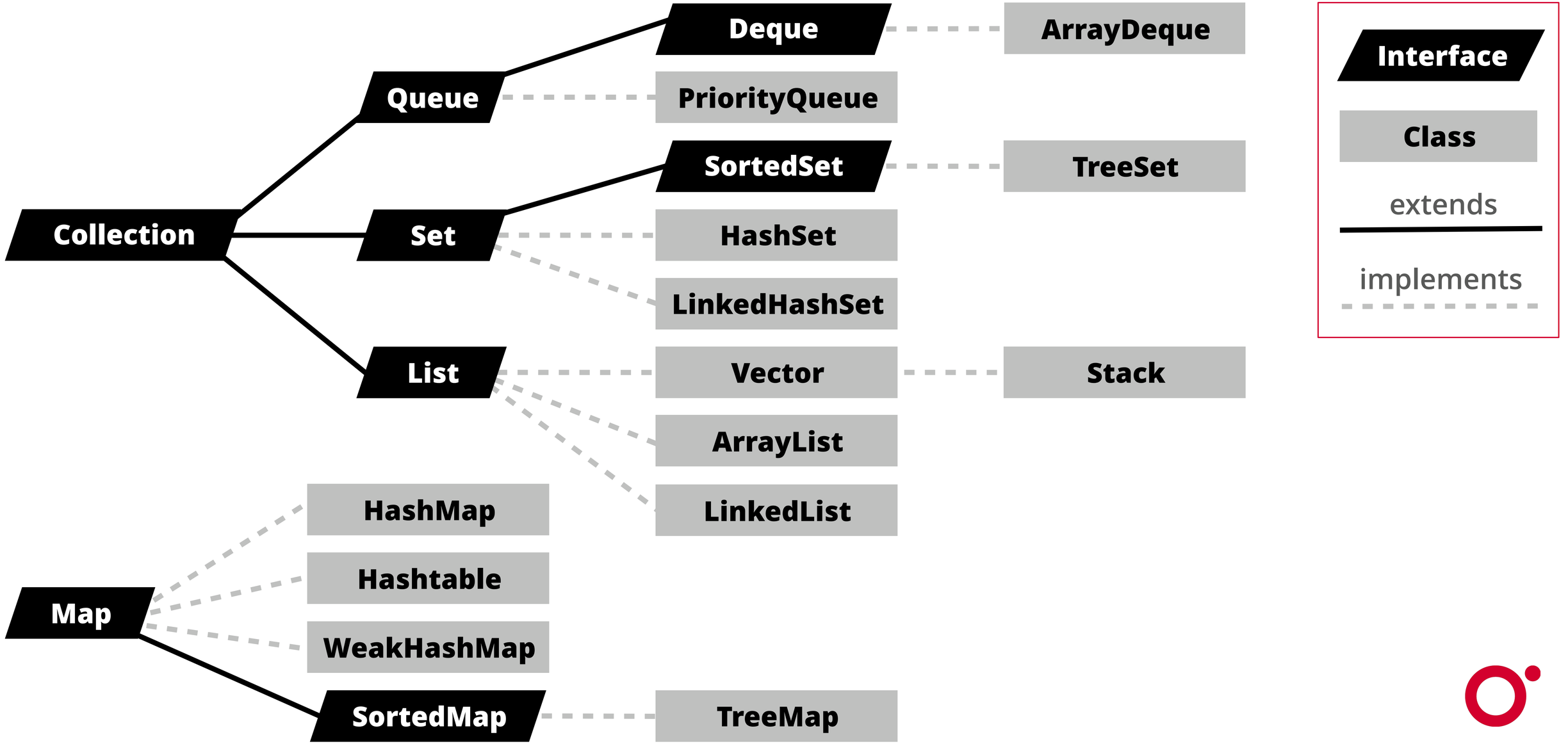
Introduction to Java Collections

- Introduction
- Demo: Introducing Java's List
- Demo: Introducing Java's Map
- Java's Map types & Implementation
- Demo: Introducing Java's Set
- Demo: Java Collections helper methods

Java Collections

- Introduced in Java 2
- State of the Art (for 20 years ago)
- Imperative APIs
- Large, Rich APIs - fully featured
- Based around mutation (e.g. Iterator)

Java Collections



Java’s List

- Linear data type
- Object-oriented replacement for arrays
- No syntax / language level support
- Example of an Iterable type
  - Produces an Iterator to traverse the List

Foreach Loops

- Java's Iterable types can be used in a special for loop
- Used when we don't need the loop index
- Any Iterable (e.g. List or Set) or Java arrays

```
for(String word : c) {
    System.out.println(word);
}
```

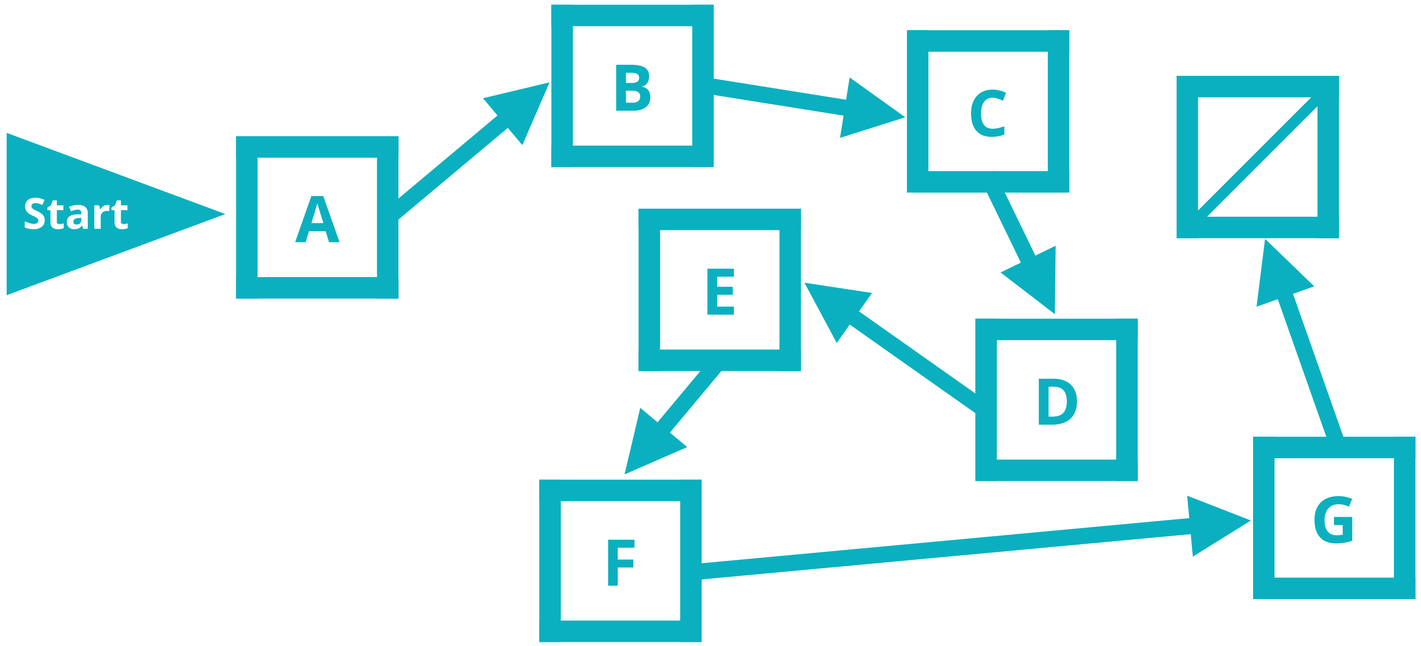
```
for(Iterator<String> i = c.iterator(); i.hasNext();) {
    System.out.println(i.next());
}
```

Iterator and Iterable

```
public interface Iterator<E> {
      boolean hasNext();
      E next();
      void remove();
}
```

```
public interface Iterable<E> {
      Iterator<E> iterator();
}
```

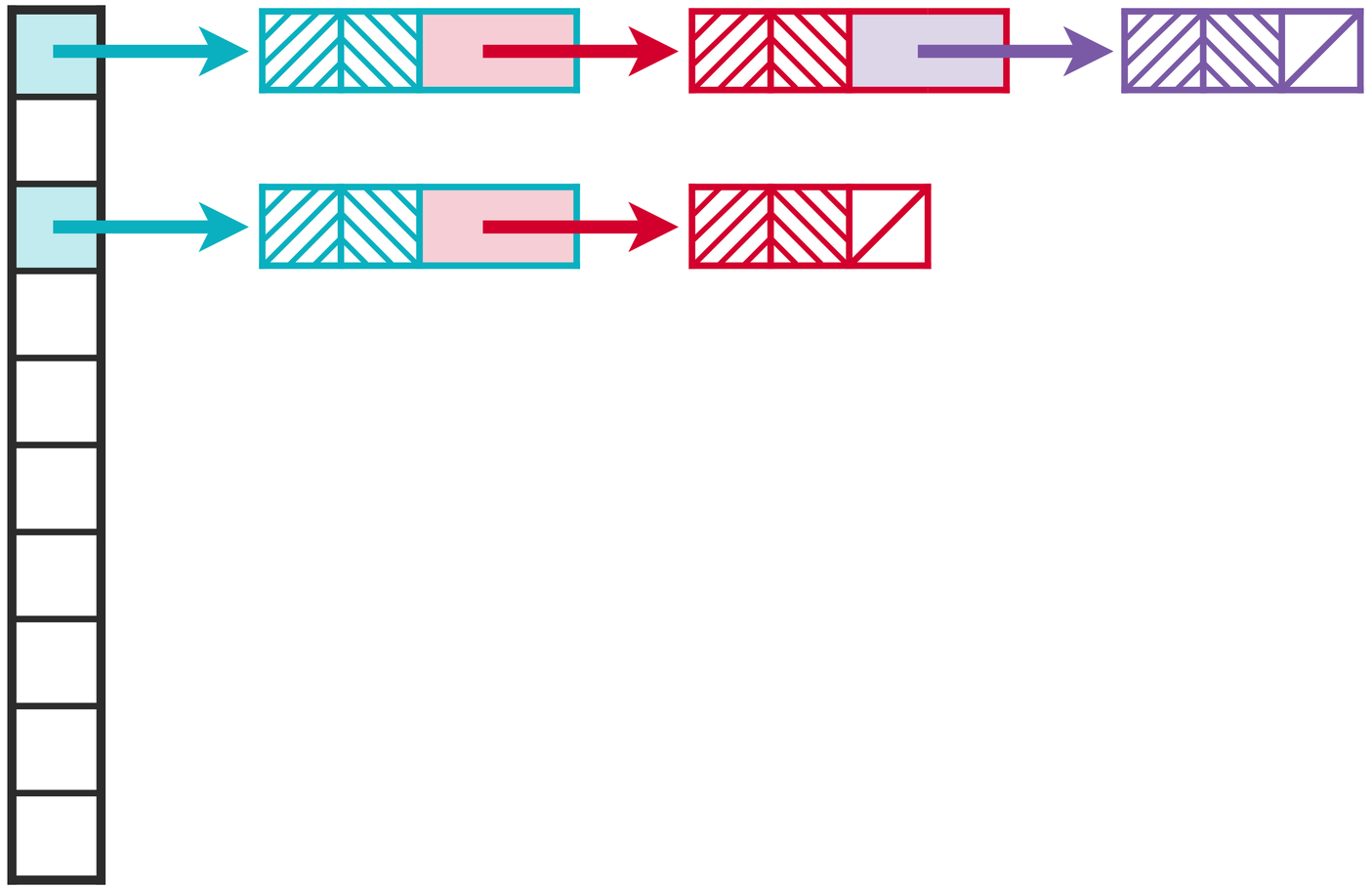
List Implementations



Java’s Map Types

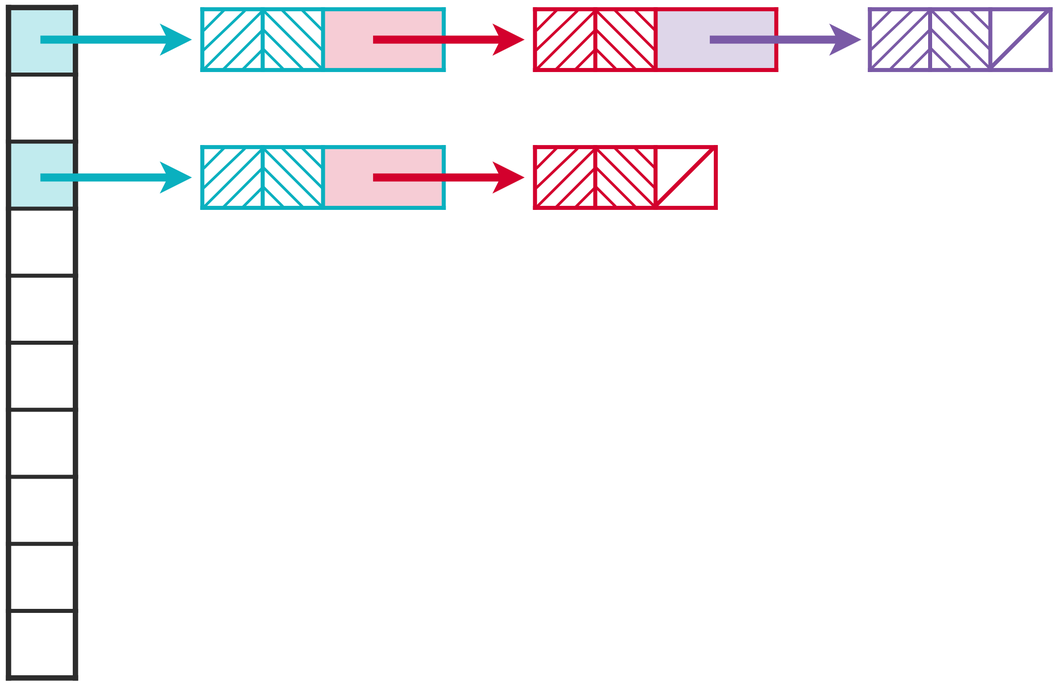
- Map - aka dictionaries or associative arrays
- Provide a mapping between keys and values
- Keys and values can be distinct types
- Keys must NOT be mutable types
- Main Java impl is HashMap

Hashing - Basic Theory



HashMap

- Hash function maps a data object to a bucket index
- Classical case: index points into an array of linked lists
- How Java implements **`HashMap`**
- Two additional wrinkles for Java
  - Hash function is supplied by the key objects (and it might be bad)
  - As **`HashMap`** grows, the bucket may become too small



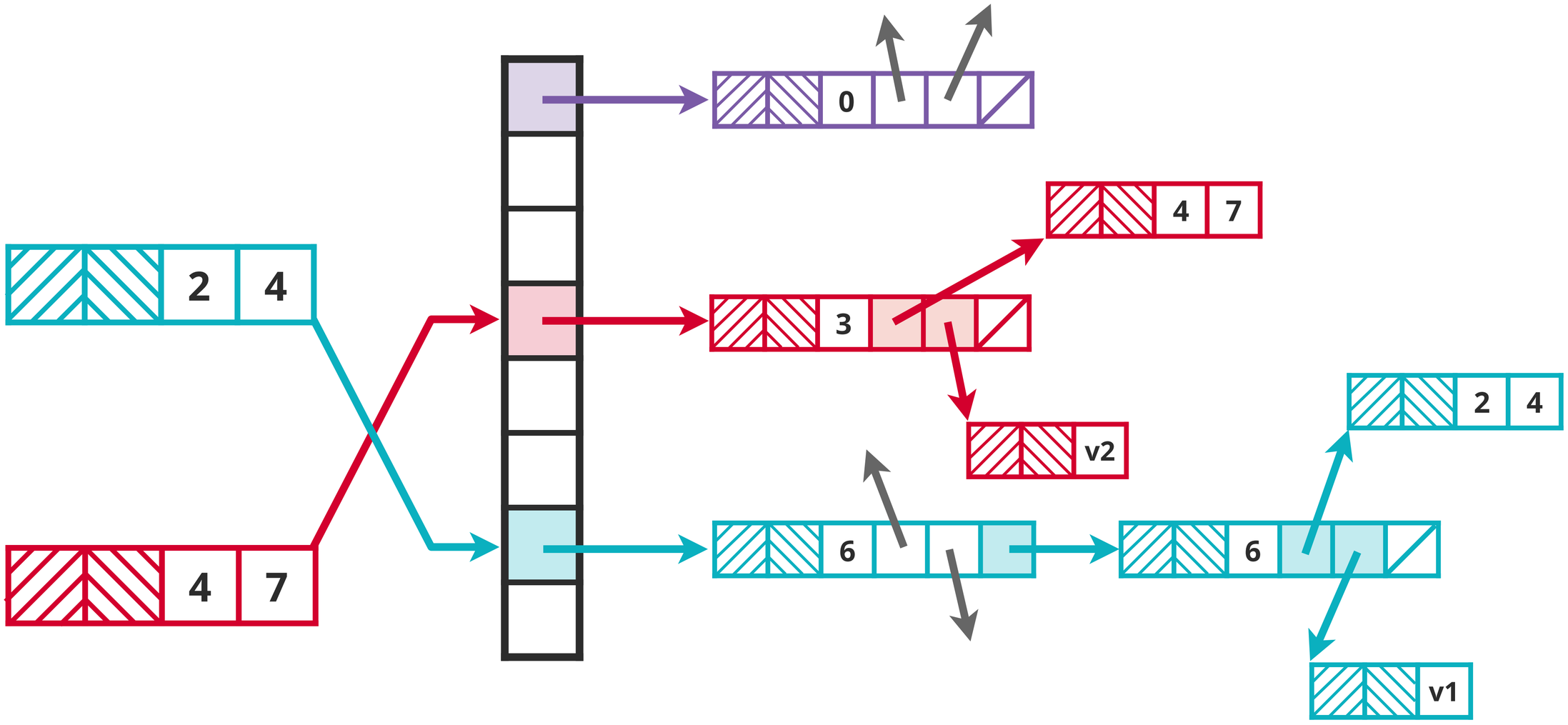
HashMap Internals

```
public Object get(Object key) {
    // SIMPLIFY: Null keys are not supported
    if (key == null) return null;

    int hash = key.hashCode();
    int i = indexFor(hash, table.length);
    for (Entry e = table[i]; e != null;
         e = e.next) {
        Object k;
        if (e.hash == hash &&
                ((k = e.key) == key || key.equals(k)))
            return e.value;
    }

    return null;
}

private int indexFor(int h, int length) {
    return h & (length-1);
}
```

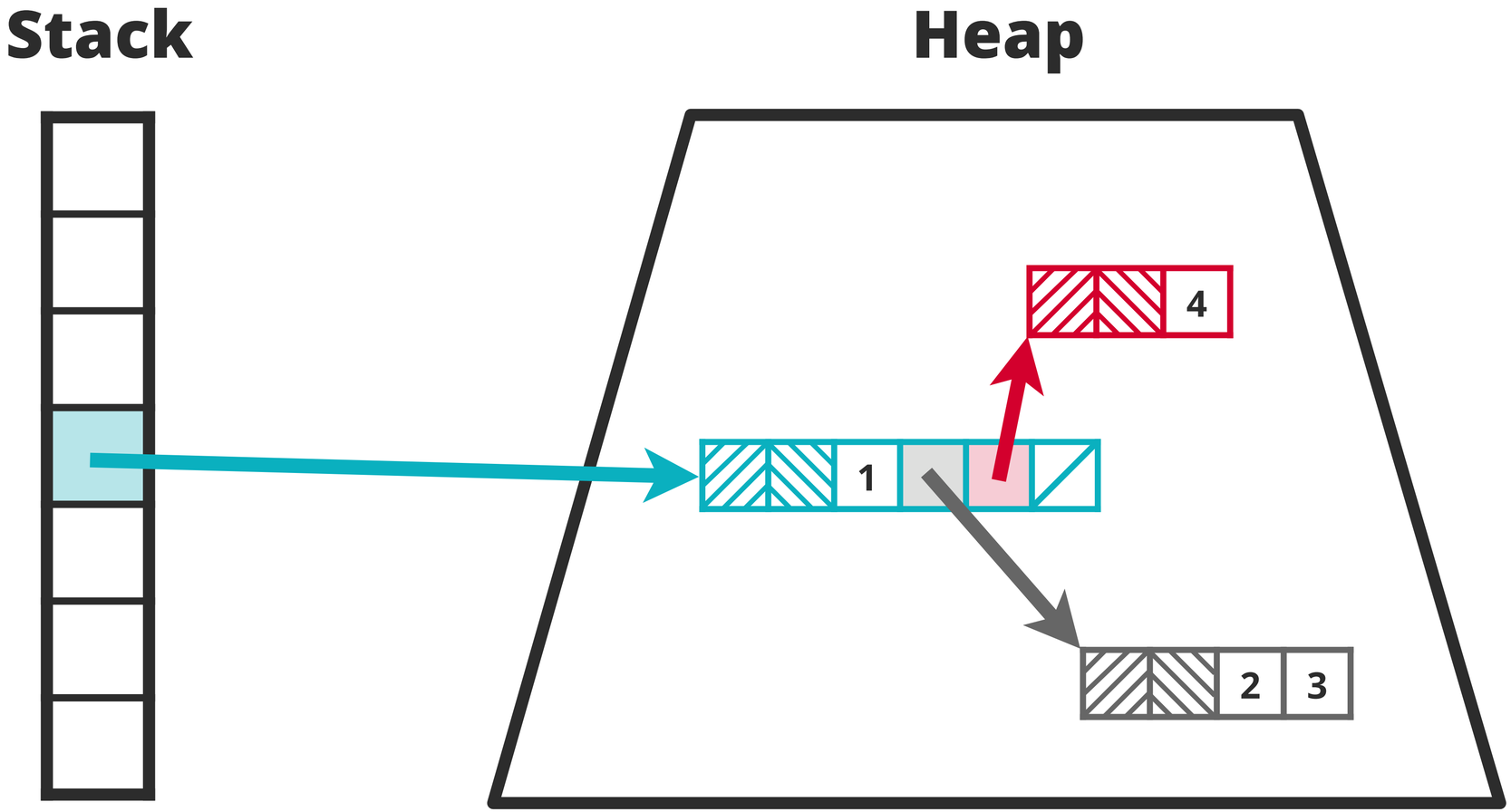
```
// This is a linked list node
private static class Entry {
    final int hash;
    final Object key;
    Object value;
    Entry next;

    Entry(int h, Object k,
          Object v, Entry n) {
        hash = h;
        key = k;
        value = v;
        next = n;
    }
}
```

Simplified version of the code in **jdk/src/share/classes/java/util/HashMap.java**

`HashMap` at Runtime



`HashMap` at Runtime



The `equals()` & `hashcode()` Contract

- `get()` shows `equals()` and `hashCode()` tied together:

```
if (e.hash == hash &&
    ((k = e.key) == key || key.equals(k))) {
    /* ... matched the object ... */
    return e.value;
}
```

- The contract is:
  - **If** two objects are equal (as determined by `equals()`)
  - Then **MUST** have equal hash values (as given by `hashCode()`)

- If you just override `equals()`, you are probably violating this contract.
  - Your object design is bad, and you should feel bad

Hashing transient & derived fields

- Mostly, IDE can auto-generate **`equals()`** & **`hashCode()`**
- But be careful
  - If you add fields, it's easy to forget to update the **`equals()`** & **`hashCode()`**
  - This can cause subtle / hard to find bugs
- Some fields on objects are transient or derived
  - These should not be be used in your **`equals()`** or **`hashCode()`**
  - Make sure these fields really are transient or derived

Java’s Set Types

- Set is an unordered linear type
- It is an Iterable type
  - If elements change, iteration order may change
- Duplicates are not allowed
- Usual Java implementation: HashSet

Collections Interfaces Are Heavyweight

- Java's Collection Interfaces are designed for reuse
- Many methods on each interface
- Implementors have to provide a lot of methods
- Collection types turn up in the interface declarations
  - E.g. keySet() method on HashMap
- Java collections have an expectation of mutability
  - E.g. Iterator

Java Generics Continued

- Demo: Generic Methods
- Type Invariance & The Unknown Type
- Demo: Covariance and Contravariance
- PECS

Generic Methods

DEMO

Wildcards

- What is the difference between `<T>` and `<?>` ?

Wildcards

- What is the difference between `<T>` and `<?>` ?
  - `T` is a Type Parameter, `?` is the “unknown type”

Wildcards

- What is the difference between `<T>` and `<?>` ?
  - `T` is a Type Parameter, `?` is the “unknown type”
  - `List<T>` is NOT a type that can be used as type for assignment
    - A so-called “lvalable” type

Wildcards

- What is the difference between `<T>` and `<?>` ?
  - `T` is a Type Parameter, `?` is the “unknown type”
  - `List<T>` is NOT a type that can be used as type for assignment
    - A so-called “lvalable” type
  - `List<?>` IS a type which can be manipulated & is lvalable
    - It’s a list of an unknown (but consistent) type

Wildcards

- What does this mean:

```
public boolean addAll(Collection<? extends E> c)
```

Wildcards

- What does this mean:

```
public boolean addAll(Collection<? extends E> c)
```

- We see these signatures throughout the Collections
  - There's a principle running through this
  - What is it?

Restate the Problem

- Is `List<Cat>` a subtype of `List<Pet>` ?
  - Assuming `Cat` extends `Pet`
  - Let’s have a look…

Type Variance

- Is `List<Cat>` a subtype of `List<Pet>` ?
  - Assuming `Cat` extends `Pet`

- As we've seen, by default, Java's answer is No
- However, we can use `List` in such a way that this can be Yes

Type Variance

- Is `List<Cat>` a subtype of `List<Pet>` ?
  - Assuming `Cat` extends `Pet`

- As we've seen, by default, Java's answer is No
- However, we can use `List` in such a way that this can be Yes
- `List<Cat>` IS a subtype of `List<? extends Pet>`
- This is called Type Covariance

Example: Introducing the PECS Principle

- Still confused about <? extends T> … ?
  - Let’s do an example
- Recall the definition of Box:

```
interface Box<T> {
  void box(T t);
  T unbox();
}
```

- Note it has 2 operations on it

Wildcards and PECS

DEMO

PECS

- Producer `extends`
- Consumer `super`
- From “Effective Java” by J. Bloch

Java Streams

- External and Internal Iteration
- Demo: Default methods
- Exercise: Writing an interface with a default method
- Demo: Introducing Java Streams
- Functional Collections & Streams
- Advanced streams (Optional)

External Iteration

- Original Java Collections
  - Client code has control
  - Exposes internal implementation
  - Designed around iteration
  - Deal with individual elements
  - Boilerplate

Internal Iteration

- New Java 8 Streams
  - Collection has control
  - Hide internal implementation
  - “What, not how”
    - Lambda defines the “what”
  - Deal with collection as a whole
  - Reduce boilerplate

Default Methods and OO in Java 8

- Binary compatibility
- **`default`** methods
- **`invokedynamic`** and implementing lambdas

Binary Interface Compatibility

- Java always maintains backwards binary compatibility
  - But new methods added to the Collections API in 8
- Why is this a problem?
- How do we solve this?

Default Methods

- Pair Lambdas with another new language change

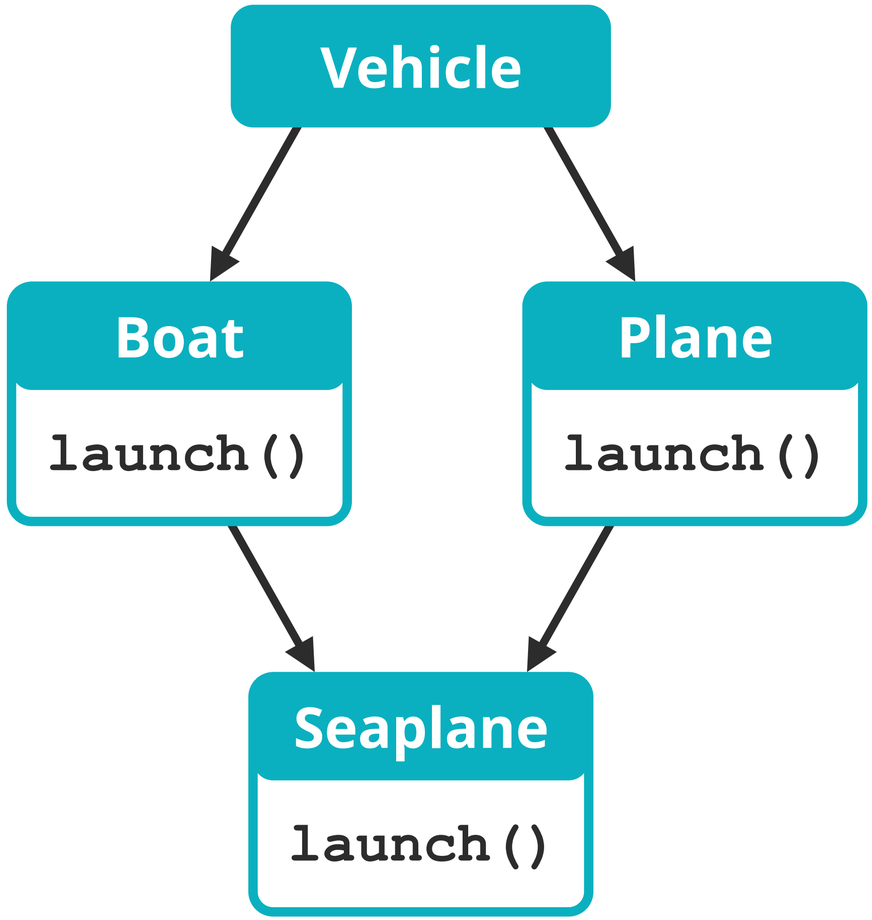
```
public interface List<E> extends Iterable<E>, Collection<E> {
    //...
    default void sort(Comparator<? super E> c) {
        Collections. sort(this, c);
    }
}
```

- Add a **`default`** method to interfaces
  - Interfaces can now define methods with bodies
  - **`default`** method must only use the interface contract

The ~~devil~~ `default` is in the details

- Different Inheritance Model
  - Any object overrides the default method
  - Normally its the most specific override
- Multiple Implementation Inheritance
  - Isn’t this bad? (remember C++)

Multiple Inheritance



Java 8's Solution

- Only inherit methods, not state
  - "Stateless Trait"
- Now can choose: abstract classes or interfaces
- Is this Traits or Interface Evolution?

Implementing Lambdas

- Despite similarities lambdas
  - Are not inner classes in disguise
  - Use **`invokedynamic`** under the hood
- The implementation has changed between versions
  - Which was the point of using **`invokedynamic`**
  - New strategies can be plugged in without recompilation

`invokedynamic`

- **`invokedynamic`** is a key new JVM feature
  - First new bytecode since Java 1.0
  - Joins **`invokevirtual`**, **`invokestatic`**, **`invokeinterface`** and **`invokespecial`**
- Relaxes a key part of the static typing system
  - Method names / signatures will not need to be known at compile time
  - User code can determine dispatch at runtime (uses **`MethodHandle`**)
- Aims to be as fast as regular method dispatch - **`invokevirtual`**
- No Java syntax for handling it in Java yet

Implementing Default Methods

- At classloading time
  - Implementations are checked
  - If they have an impl of a default method, this is used
  - If not, then the interface file is consulted
    - A stub is generated (uses **`invokedynamic`**)
    - When the method is called, the default impl is patched in

Functional Collections and Streams

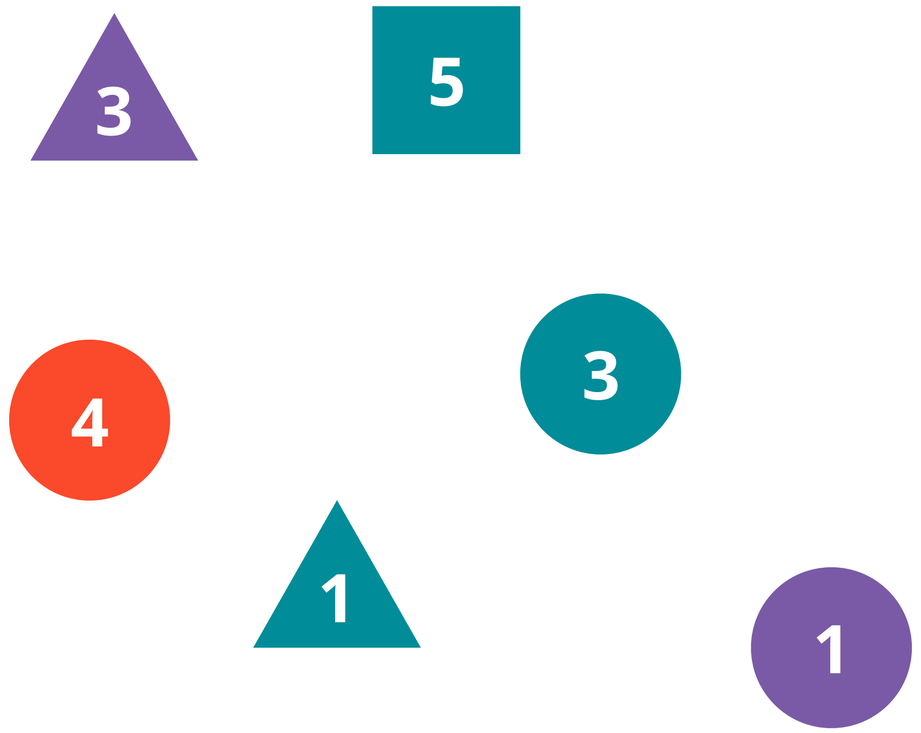
- The Use Case for Streams
- Some Functional Operations
- Weaknesses of the Streams approach

“Slightly Functional” Programming

```
public class Widget {
     Shape s;
     Colour c;
     int value;
}
```

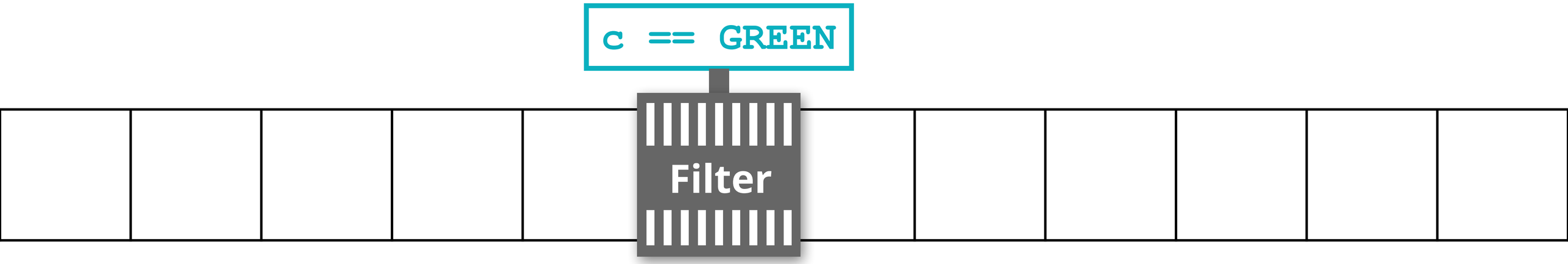
“Slightly Functional” Programming

```
public class Widget {
     Shape s;
     Colour c;
     int value;
}
```

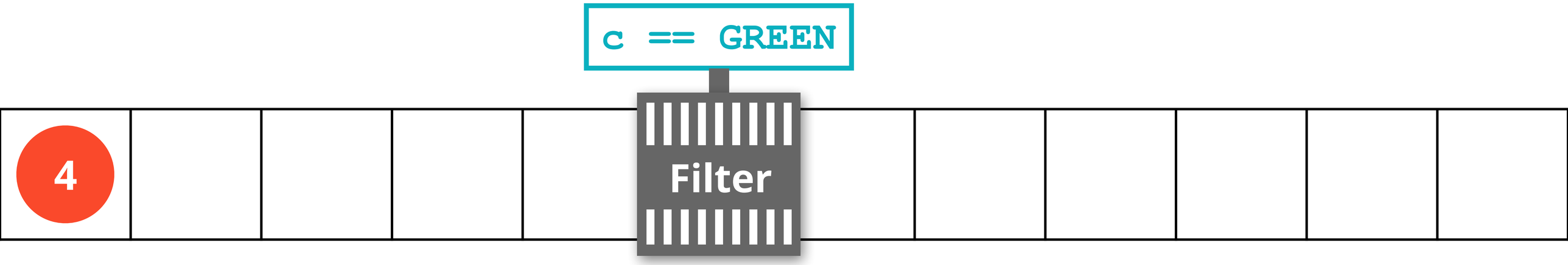


“Slightly Functional” Programming

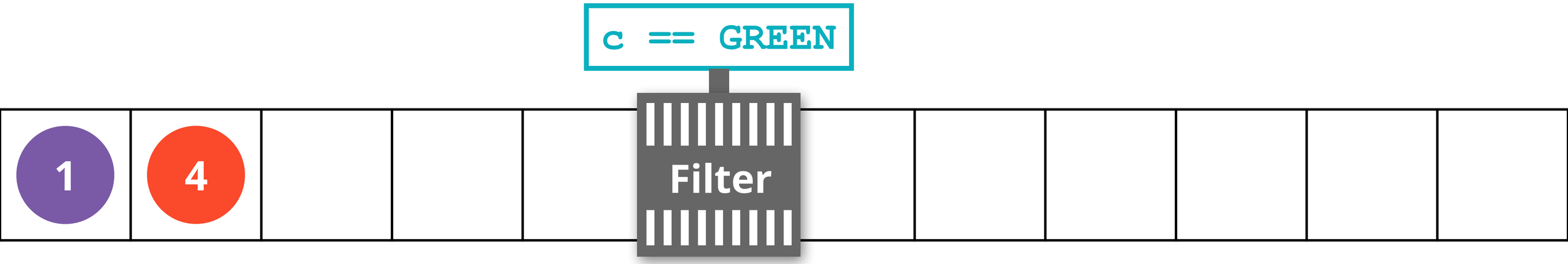
Filter-Map-Reduce



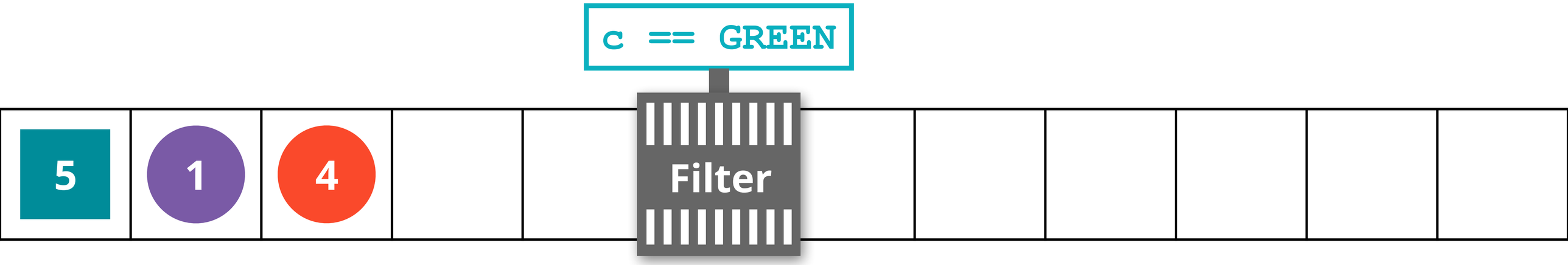
Filter-Map-Reduce



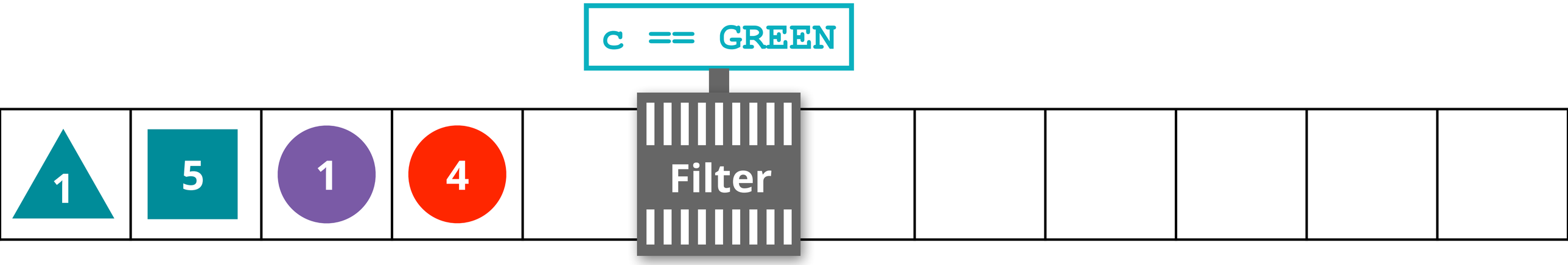
Filter-Map-Reduce



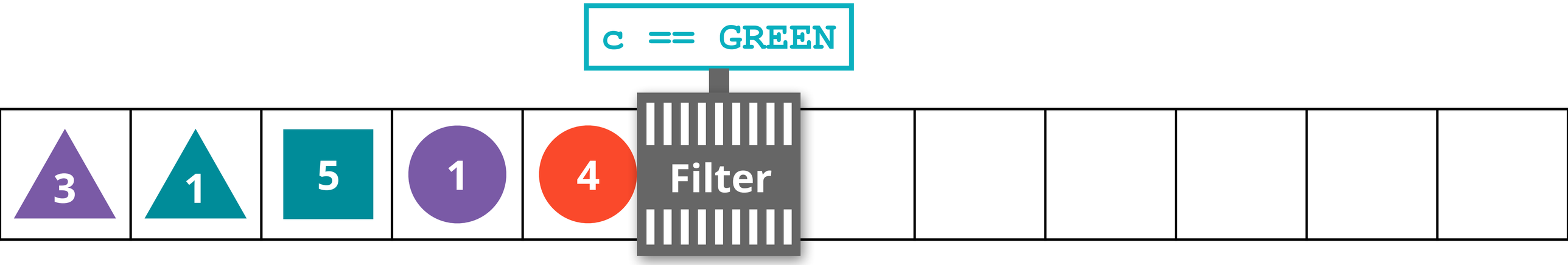
Filter-Map-Reduce



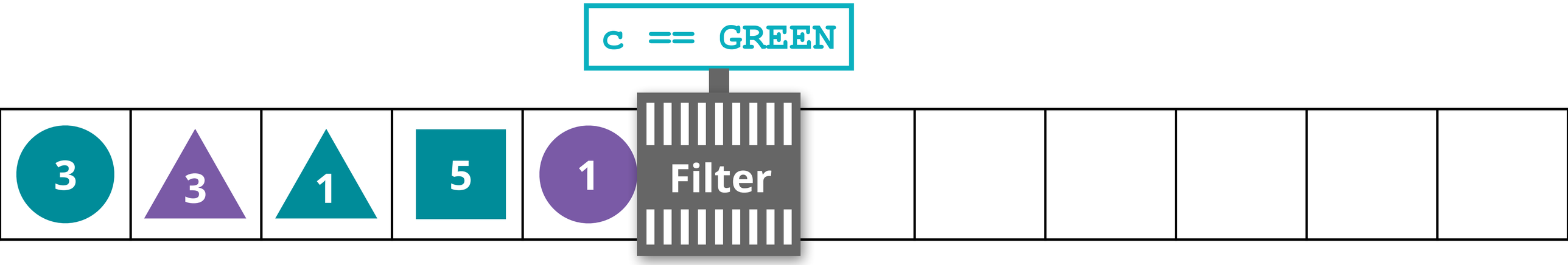
Filter-Map-Reduce



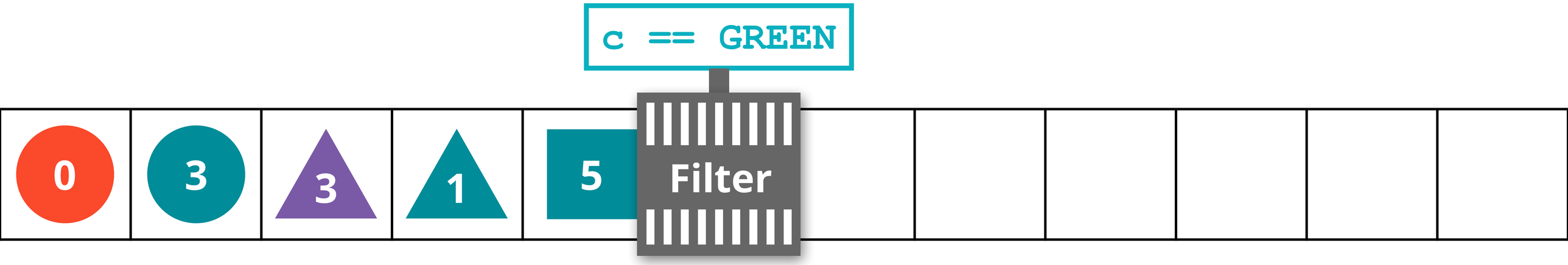
Filter-Map-Reduce



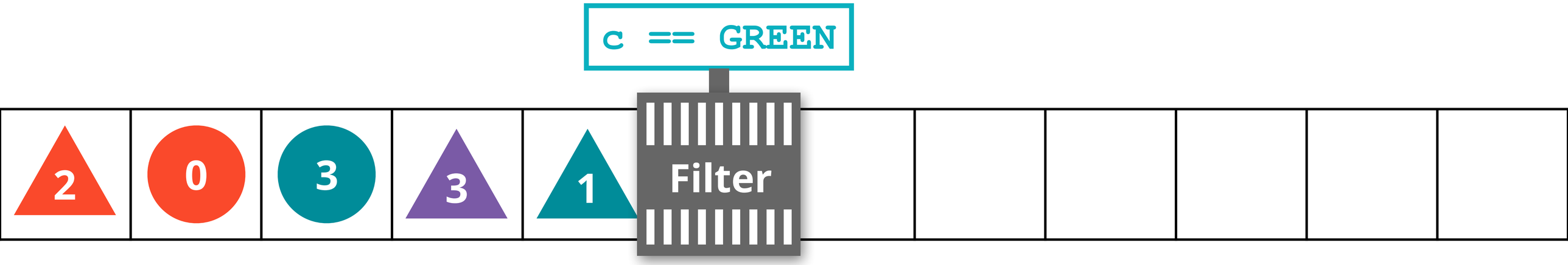
Filter-Map-Reduce



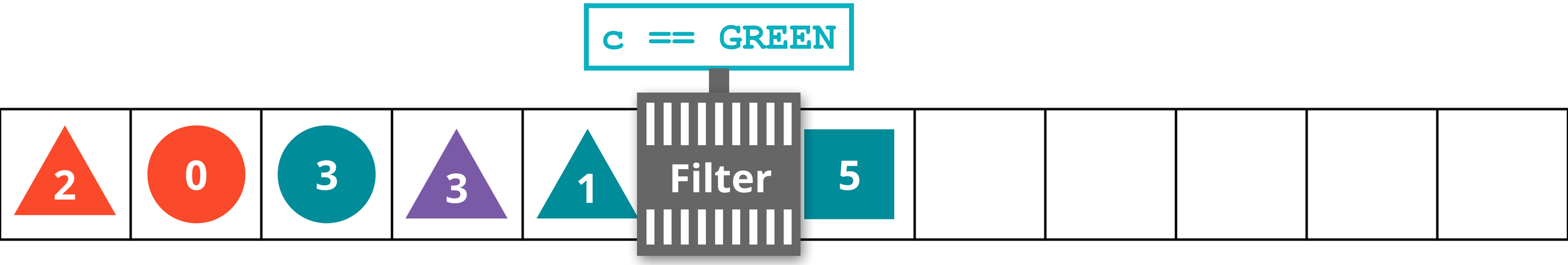
Filter-Map-Reduce



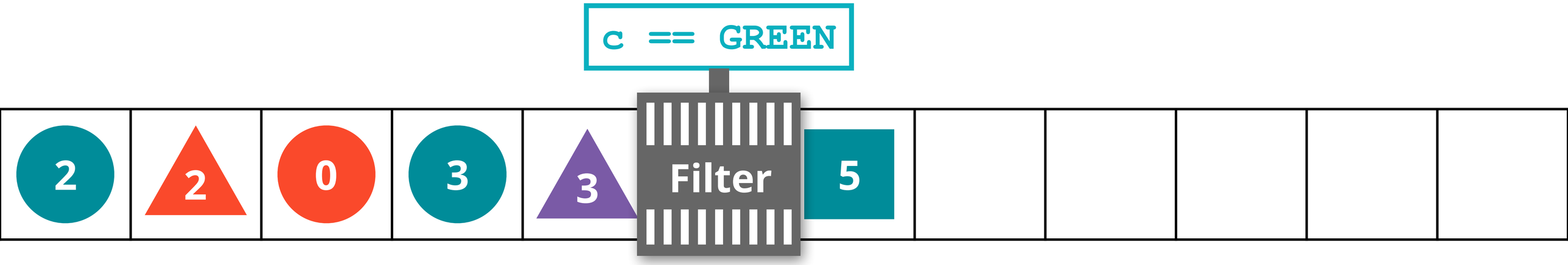
Filter-Map-Reduce



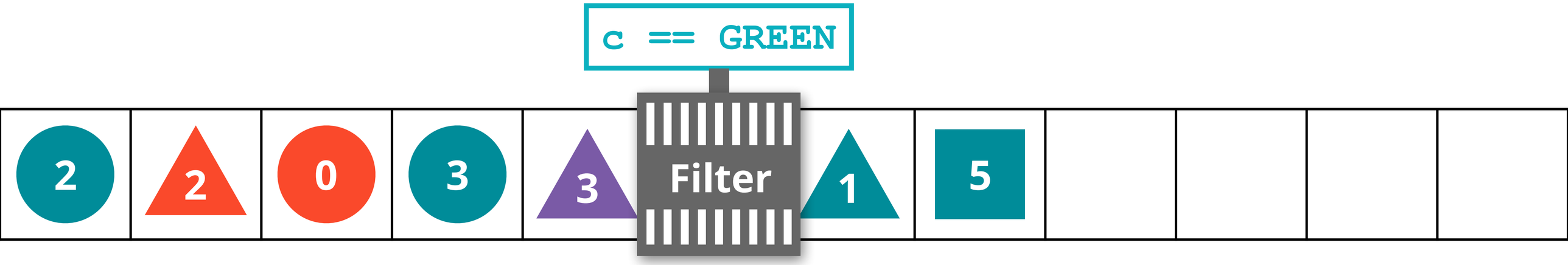
Filter-Map-Reduce



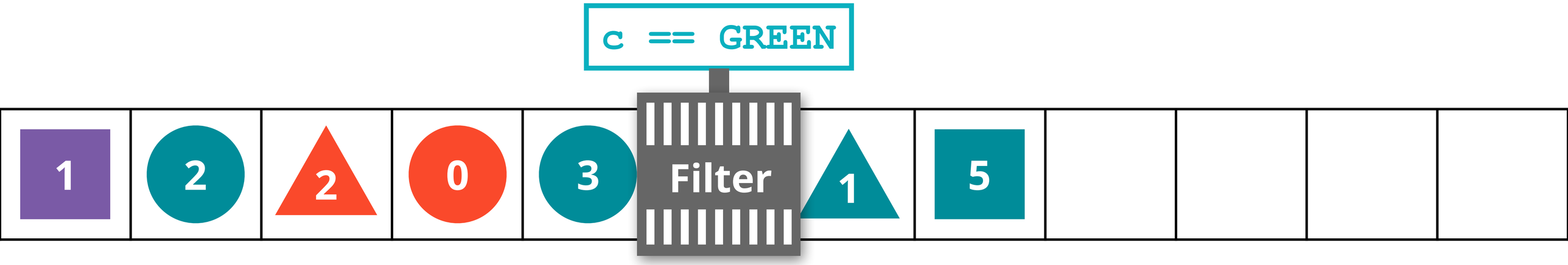
Filter-Map-Reduce



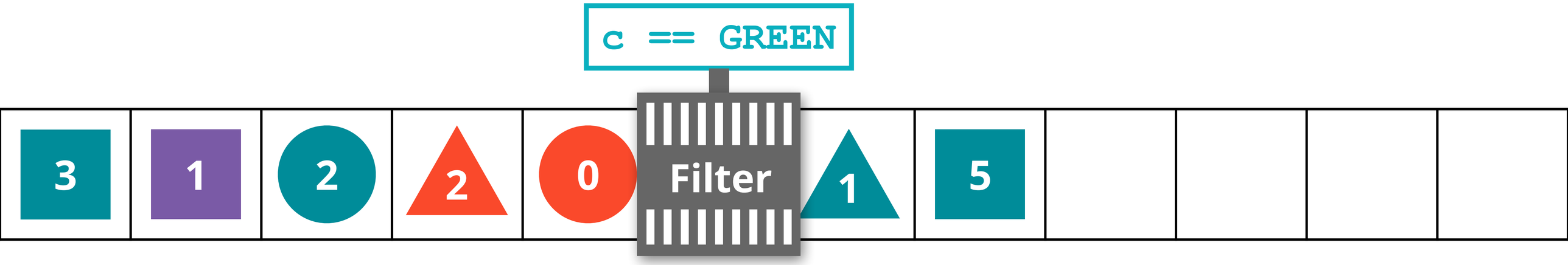
Filter-Map-Reduce



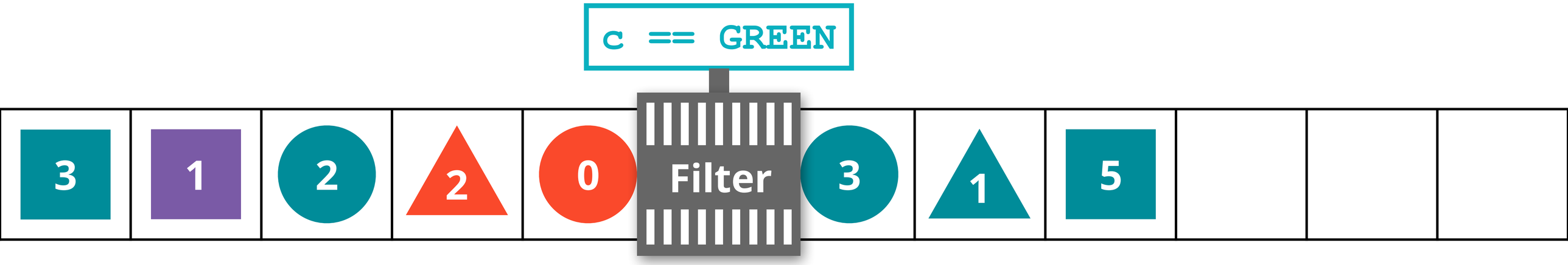
Filter-Map-Reduce



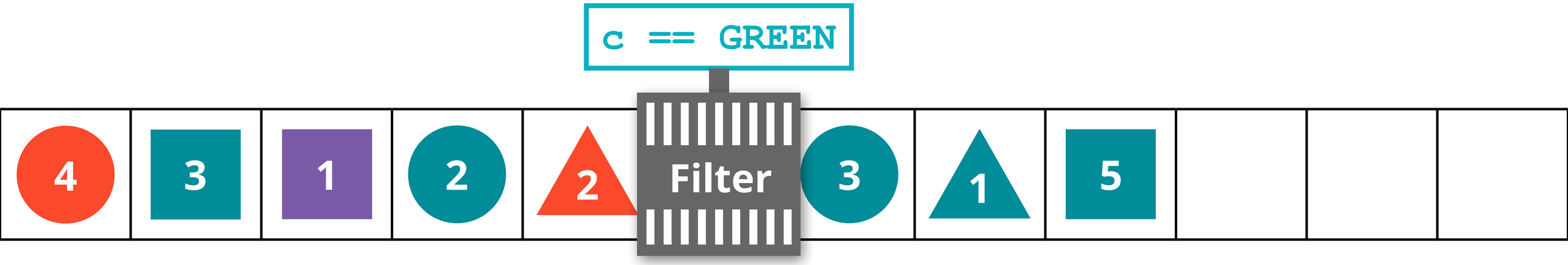
Filter-Map-Reduce



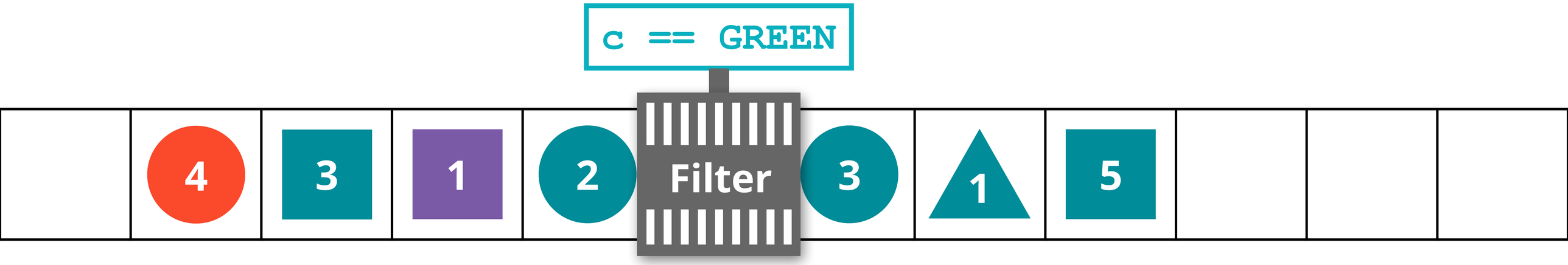
Filter-Map-Reduce



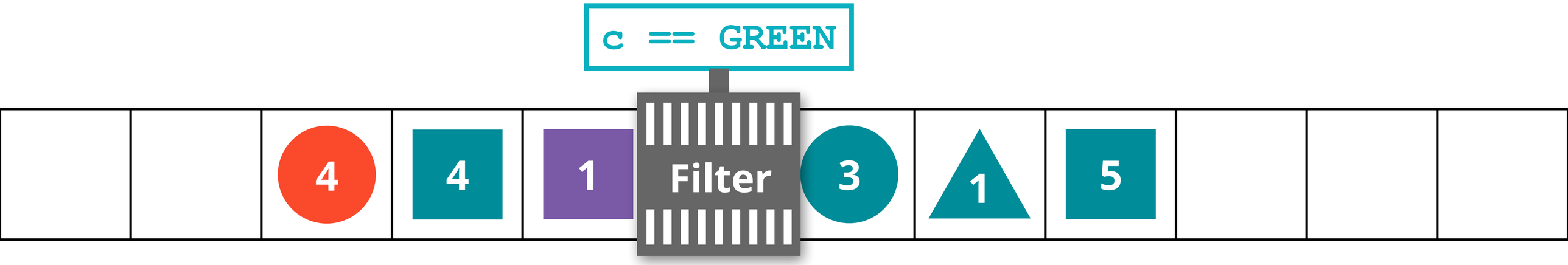
Filter-Map-Reduce



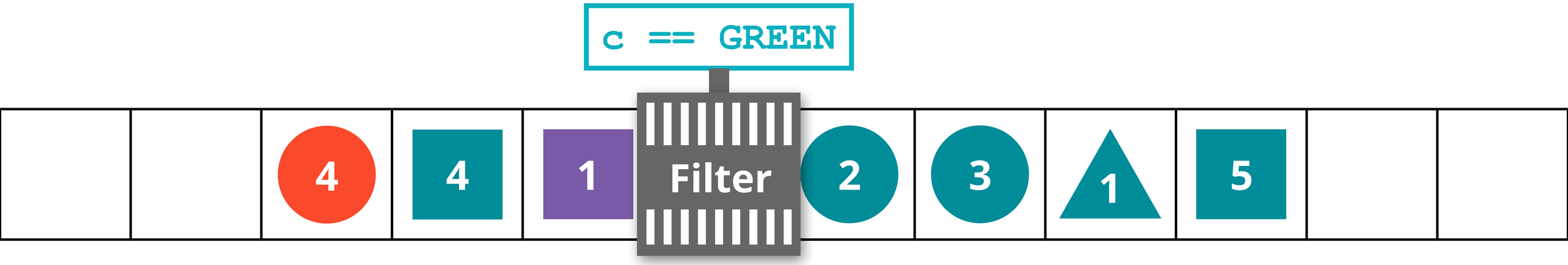
Filter-Map-Reduce



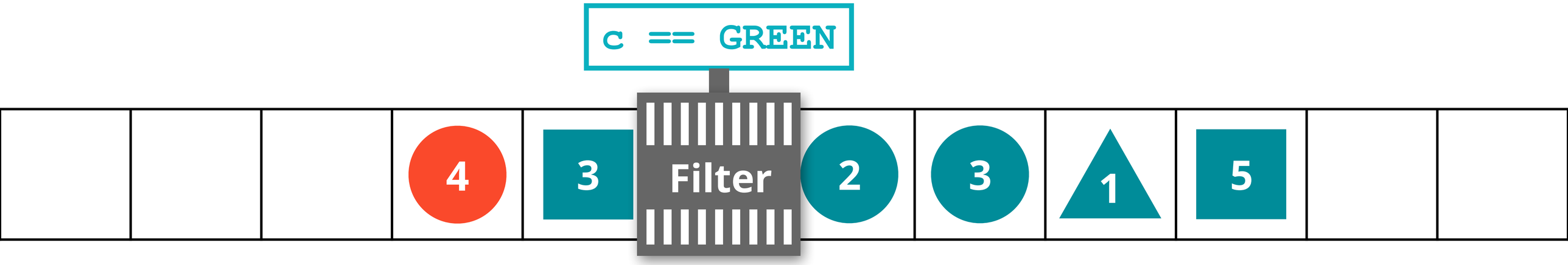
Filter-Map-Reduce



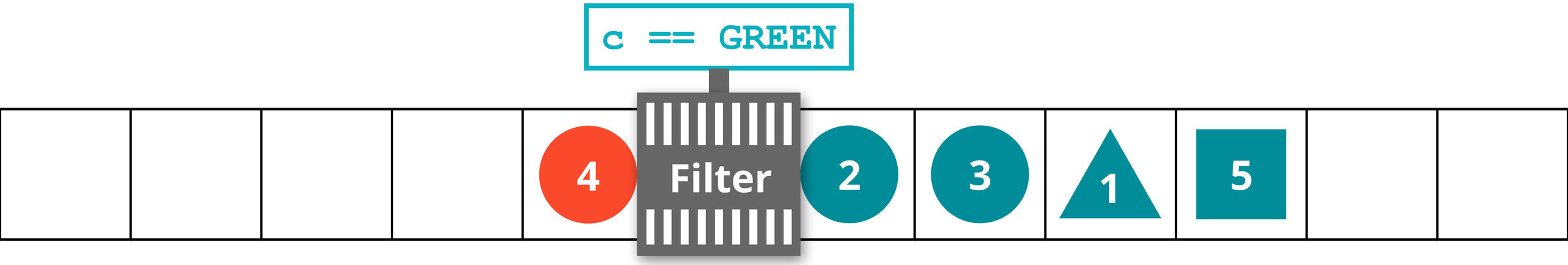
Filter-Map-Reduce



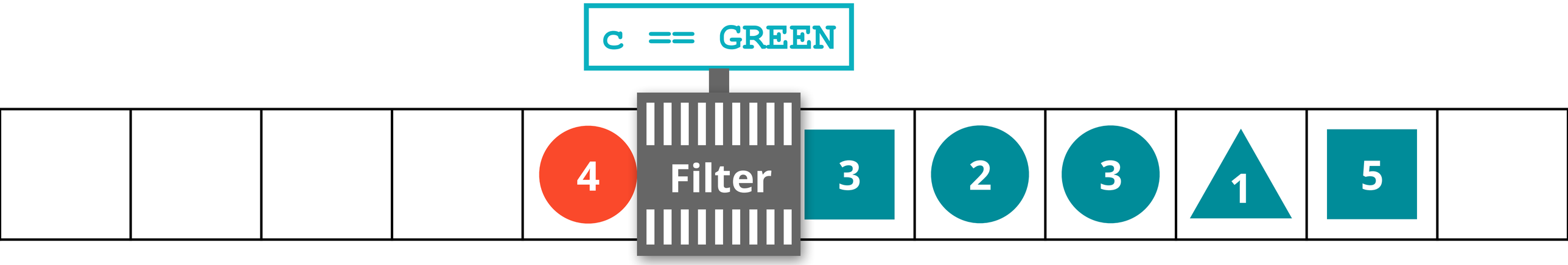
Filter-Map-Reduce



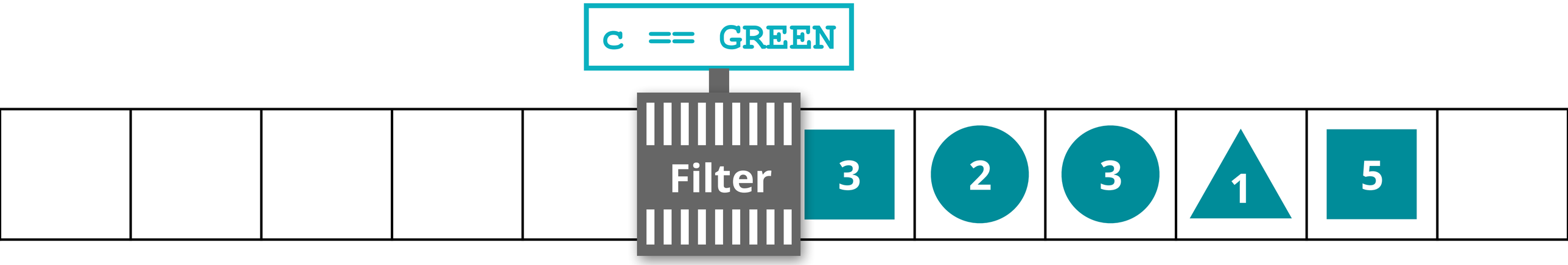
Filter-Map-Reduce



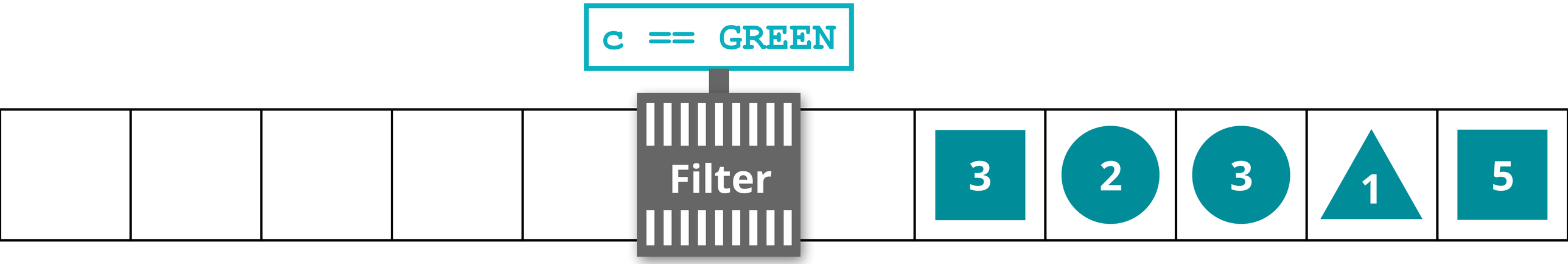
Filter-Map-Reduce



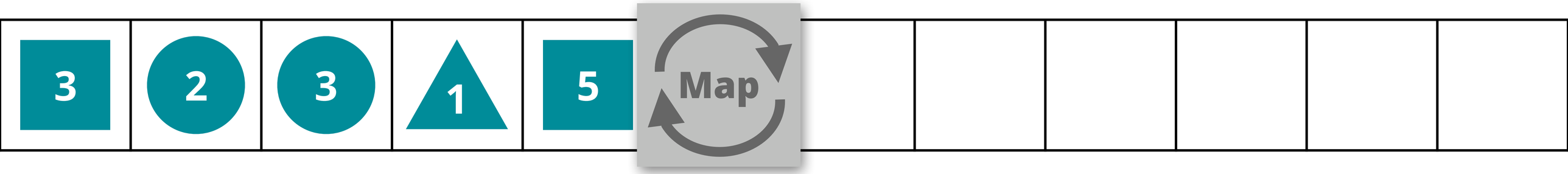
Filter-Map-Reduce



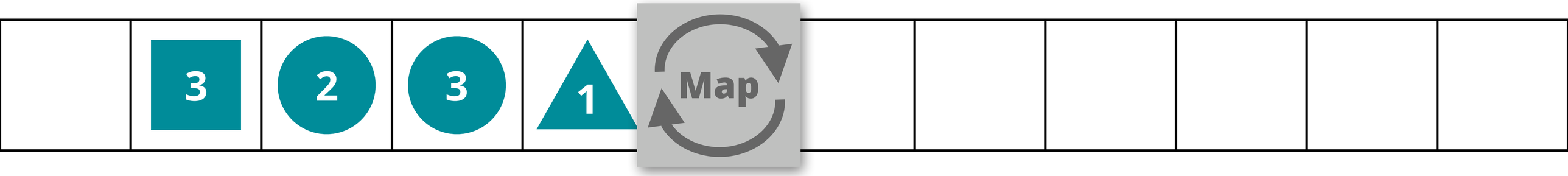
Filter-Map-Reduce



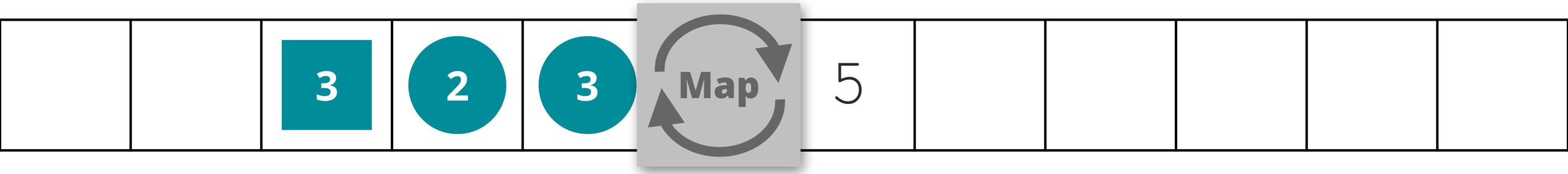
Filter-Map-Reduce



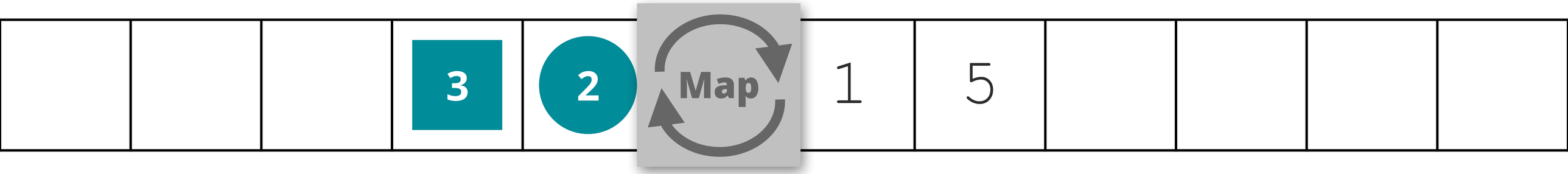
Filter-Map-Reduce



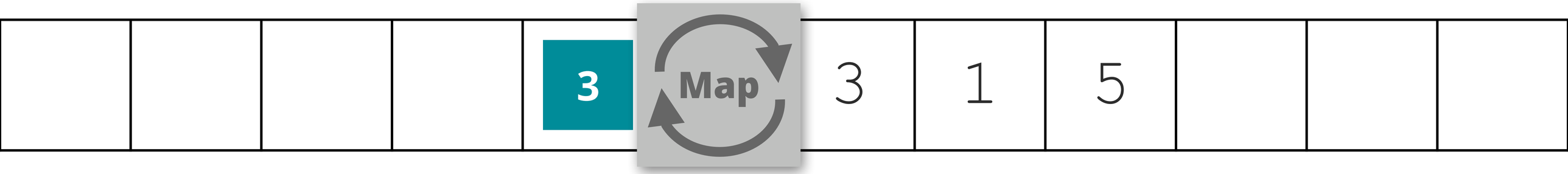
Filter-Map-Reduce



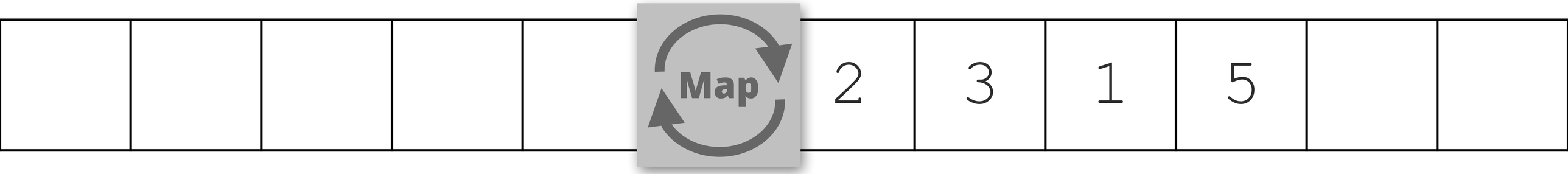
Filter-Map-Reduce



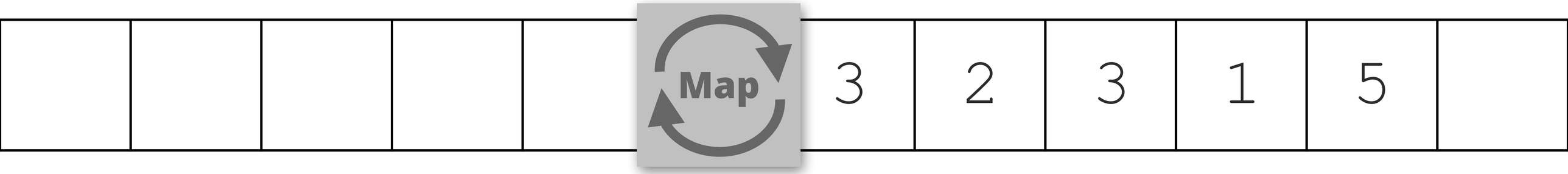
Filter-Map-Reduce



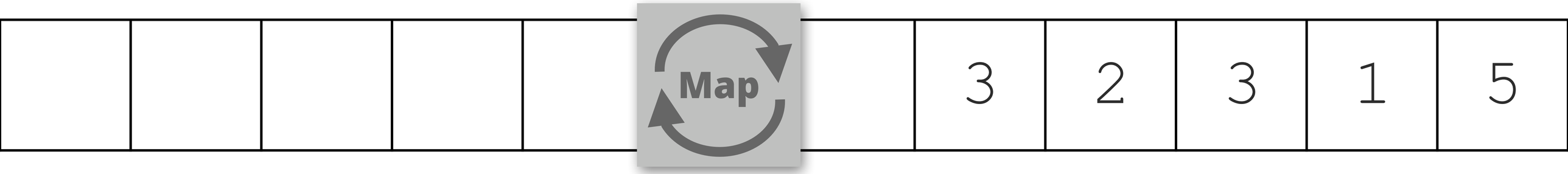
Filter-Map-Reduce



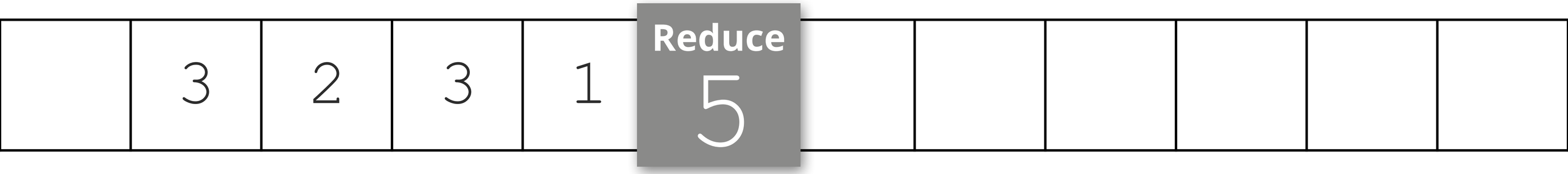
Filter-Map-Reduce



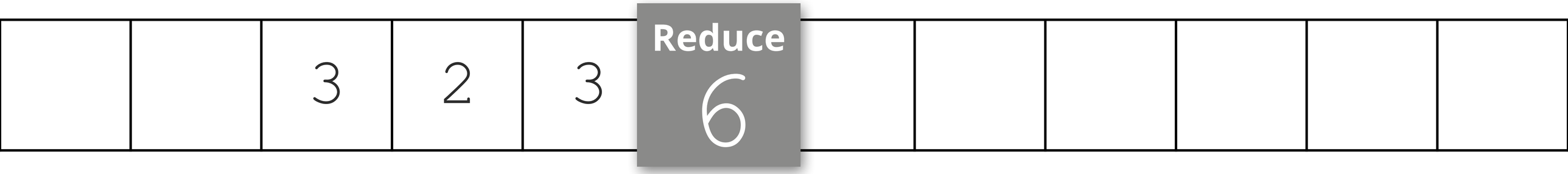
Filter-Map-Reduce



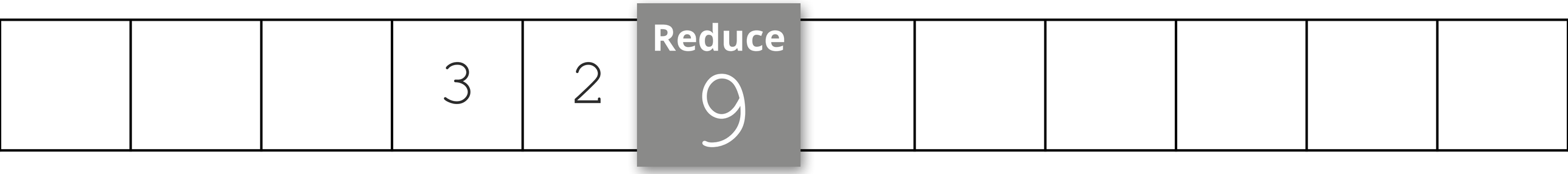
Filter-Map-Reduce



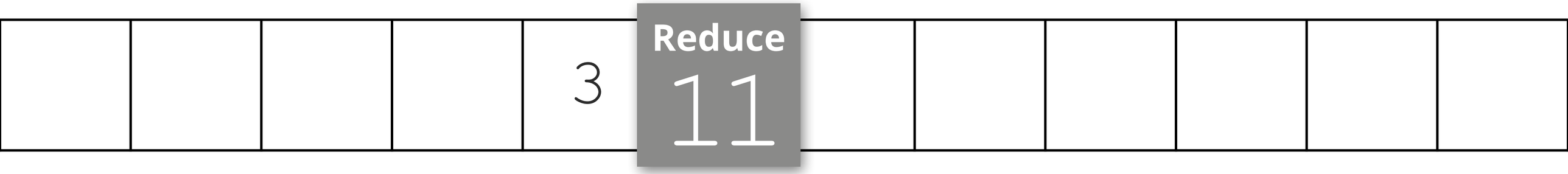
Filter-Map-Reduce



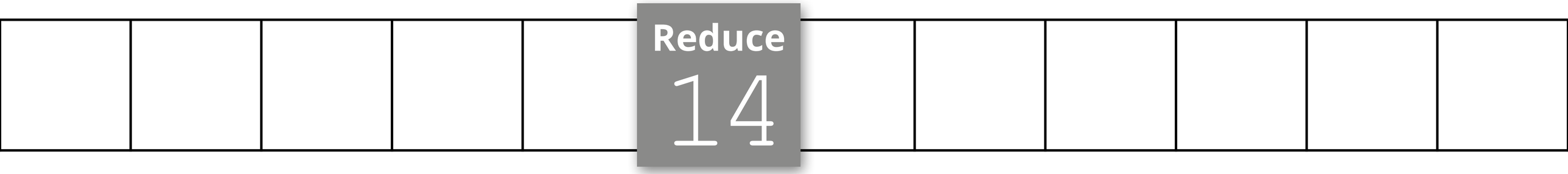
Filter-Map-Reduce



Filter-Map-Reduce



Filter-Map-Reduce



Filter-Map-Reduce

14

Streams

DEMO

Fluent Interfaces

- Fluent interfaces
  - Use method chaining
  - Return the same type from each operation
  - Have a terminating operation (which often returns void)

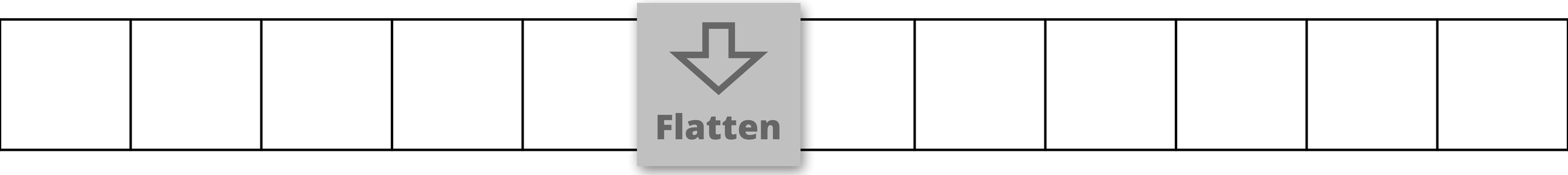
- New Collections methods are fluent interfaces
  - Methods on `Stream` typically return another `Stream`
  - Build pipelines of operations
  - `collect()` methods terminate the pipeline

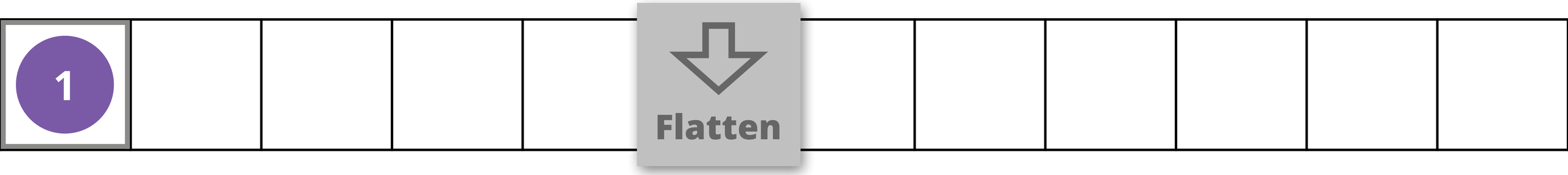
Weaknesses of Streams

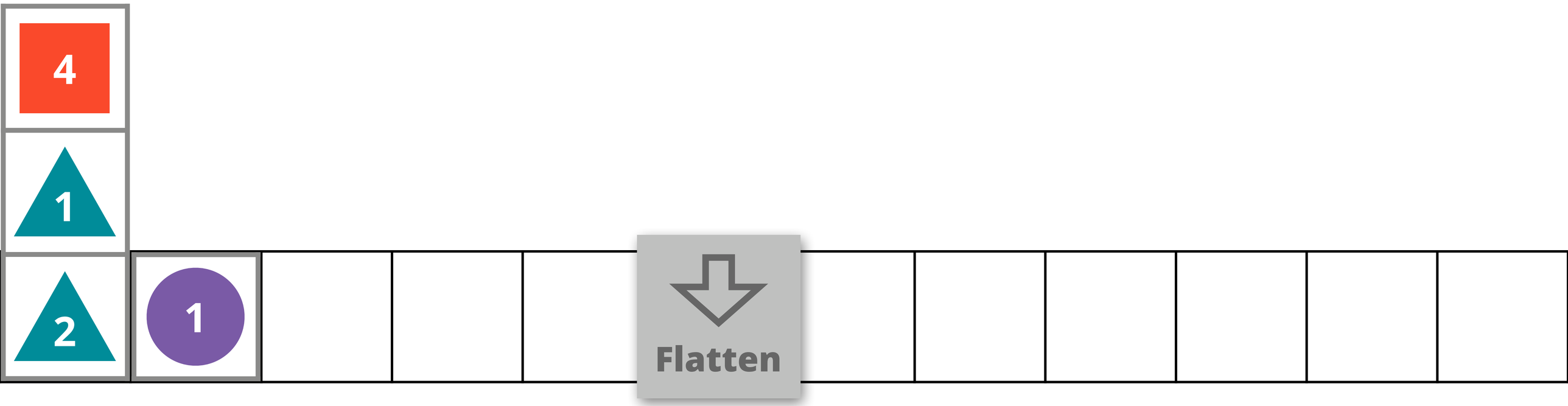
- Collections are first-class
  - Embedded in signatures throughout the JDK
  - Streams are not
- Java has no type-level “Traversable” concept
- Collections are not “functional containers”
- Streams are - to a limited degree

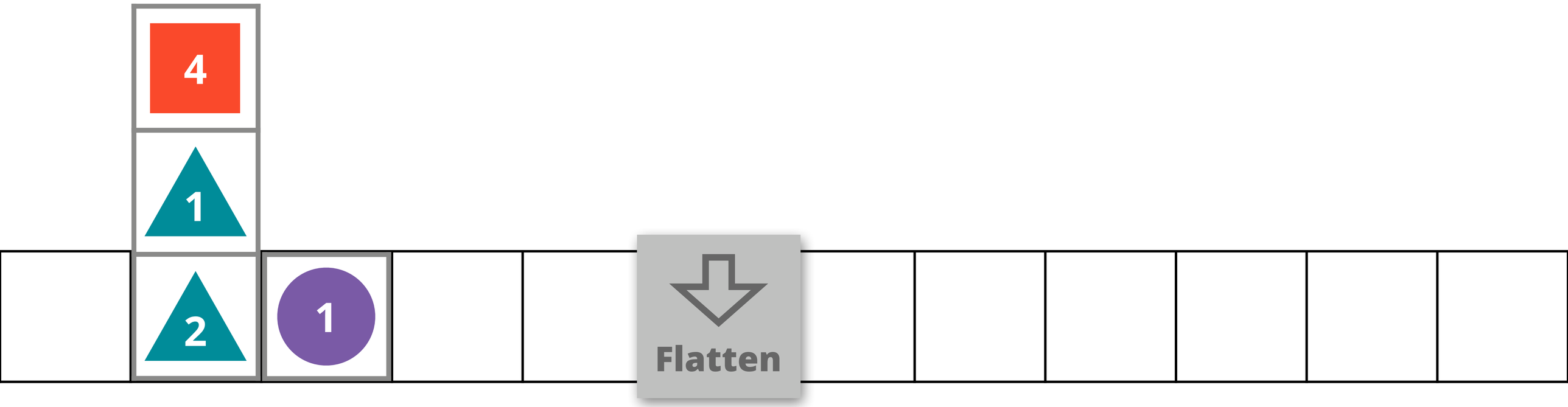
Advanced Streams

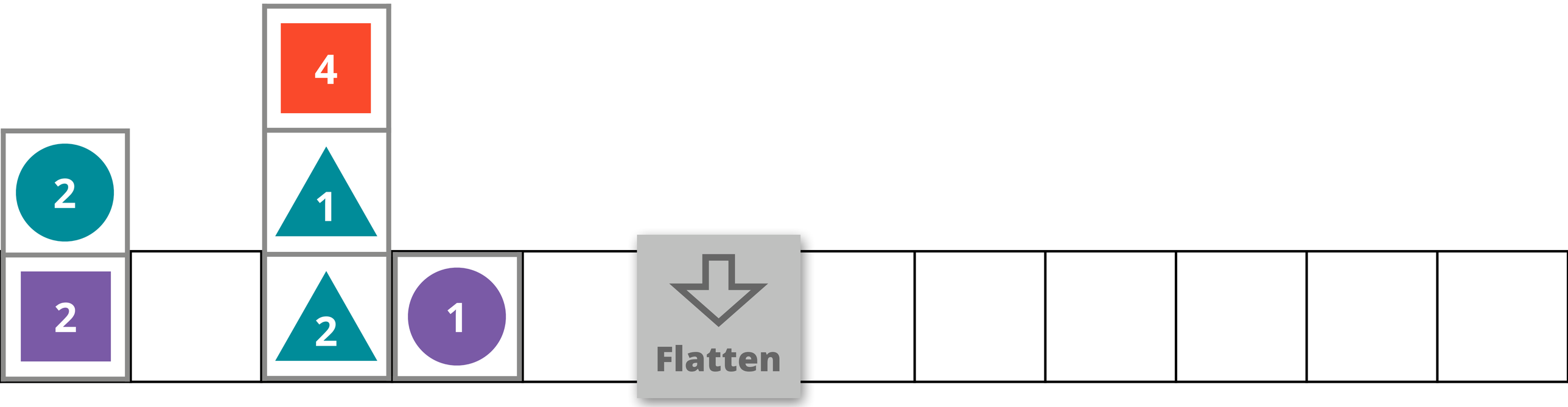
- Flattening
- Demo: Primitive Streams
- Demo: Laziness

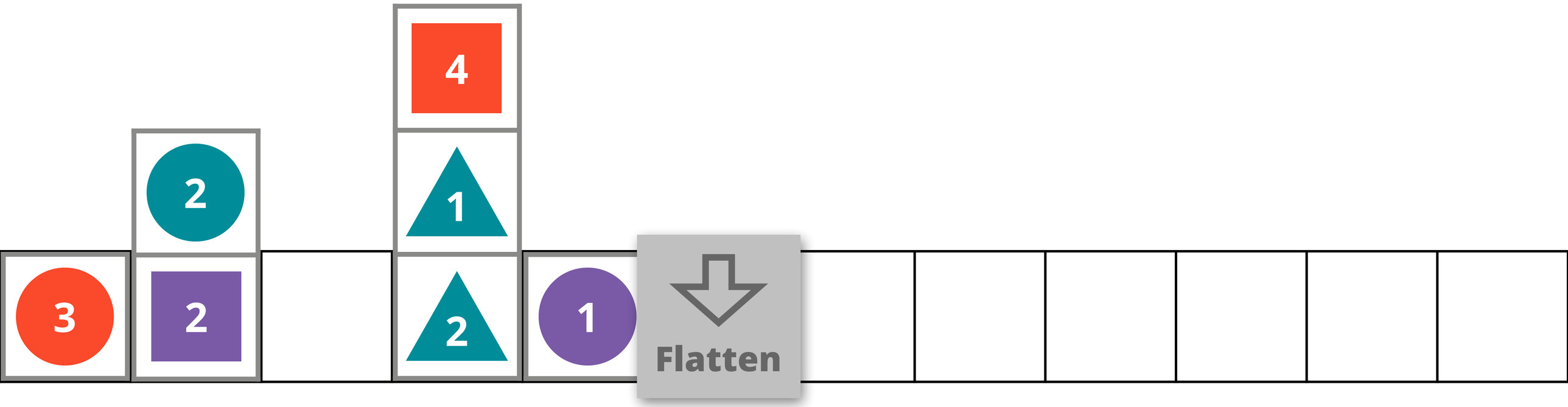

Flatten

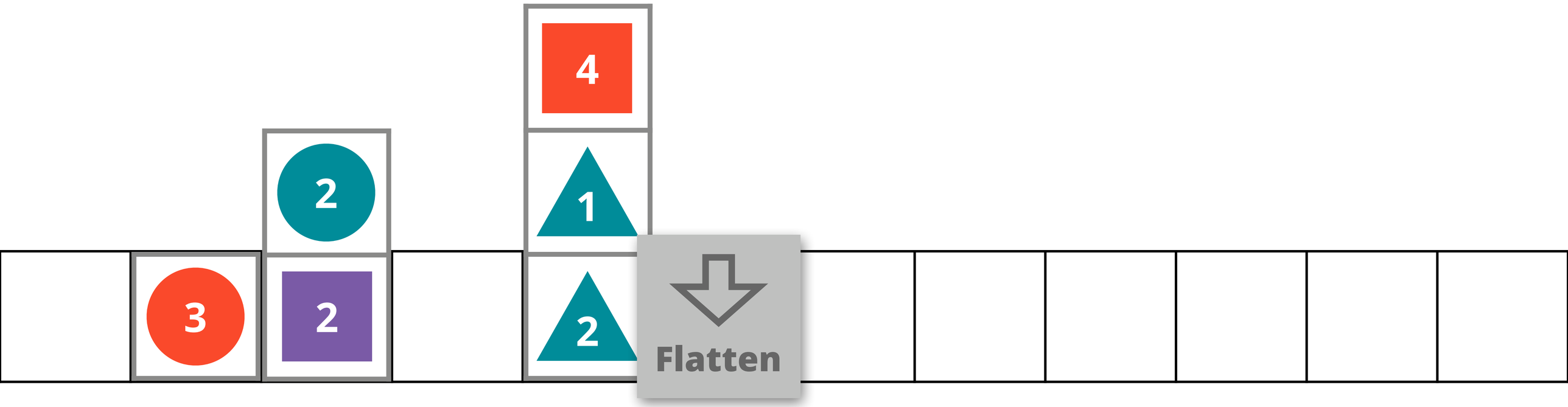

1
Flatten

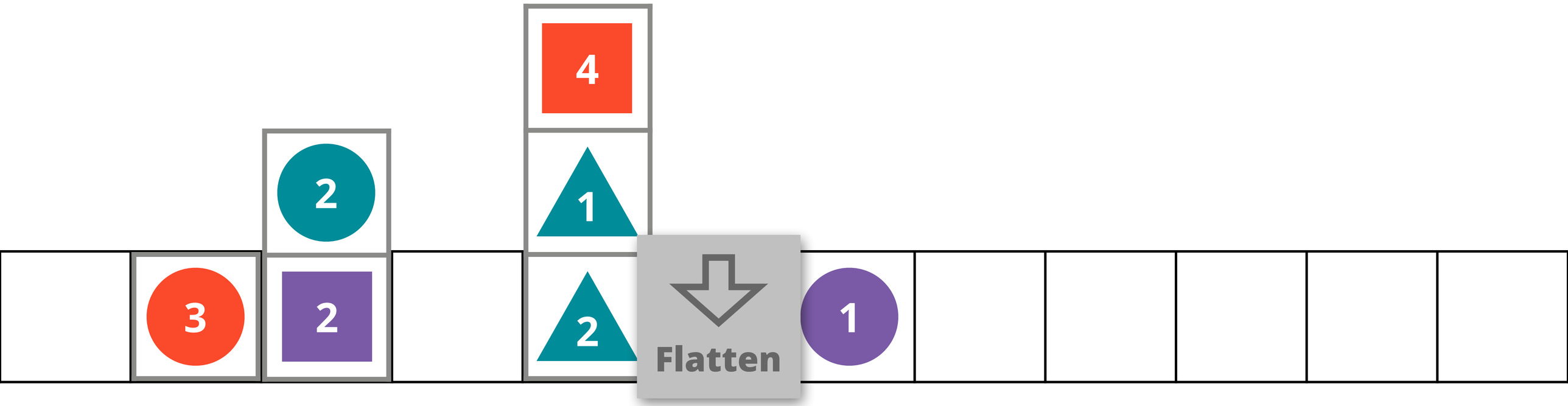

4
1
2
1
Flatten

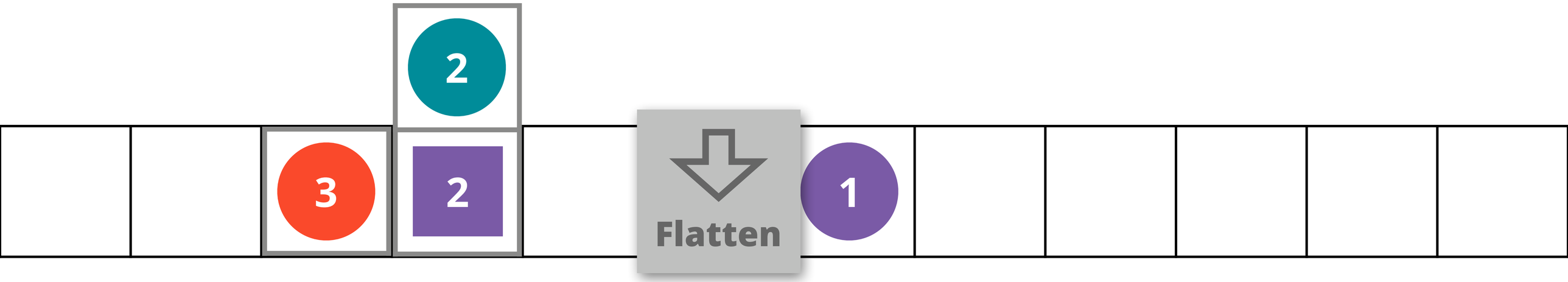

4
1
2
1
Flatten

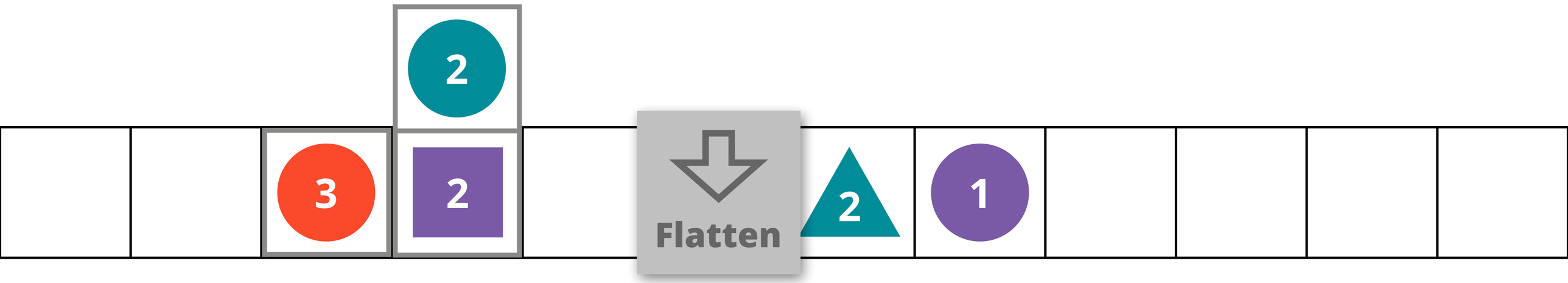

4
2
1
2
2
1
Flatten

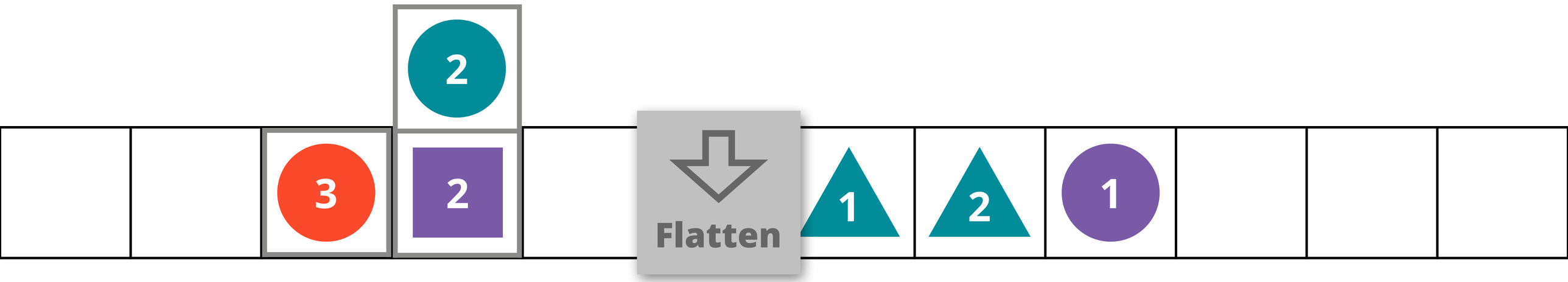

4
2
1
3
2
2
1
Flatten

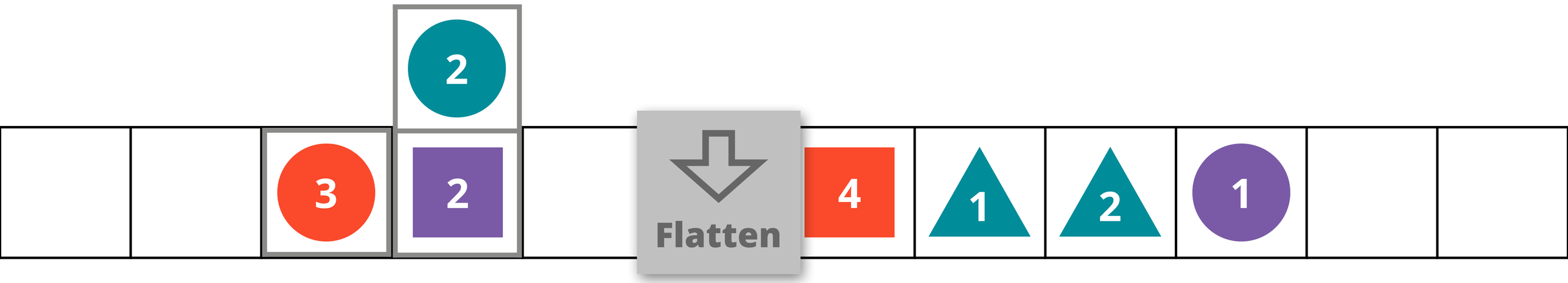

4
2
1
3
2
2
Flatten

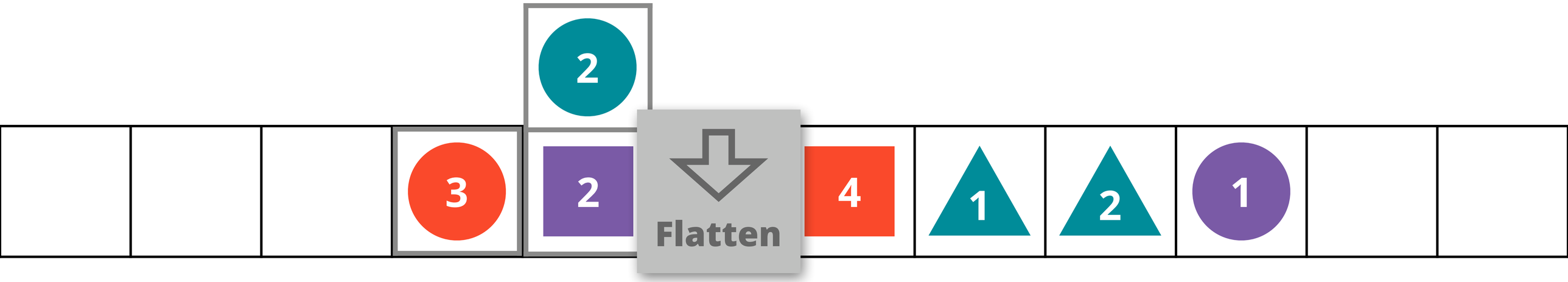

4
2
1
3
2
2
Flatten
1


2
3
2
Flatten
1

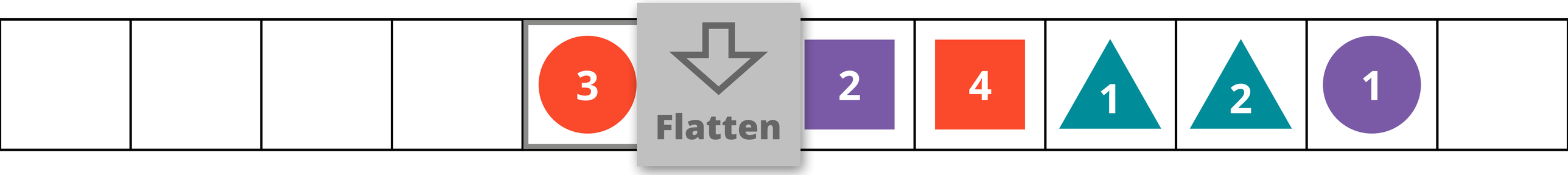

2
3
2
Flatten
2
1

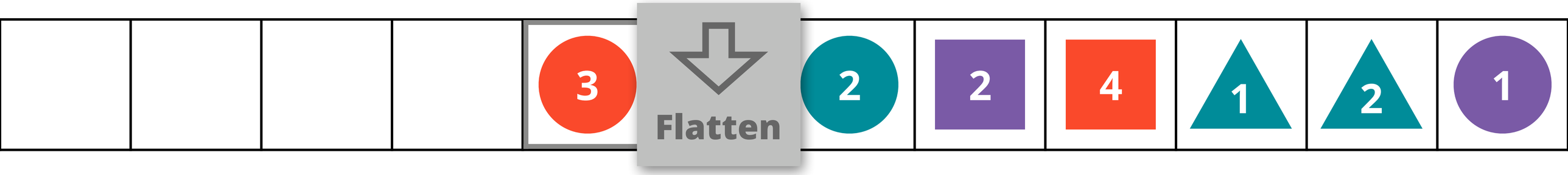

2
3
2
Flatten
1
2
1

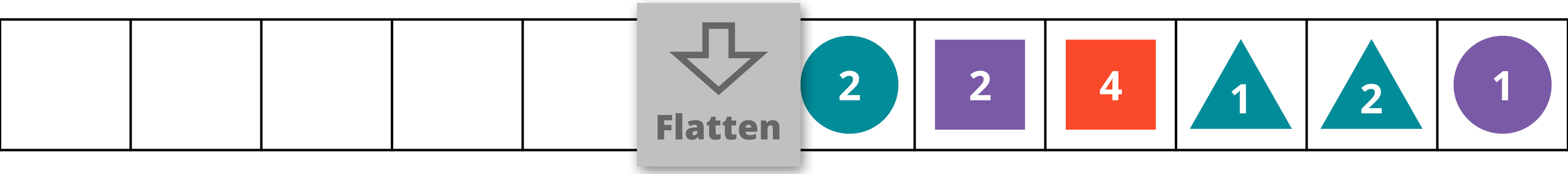

2
3
2
Flatten
4
1
2
1

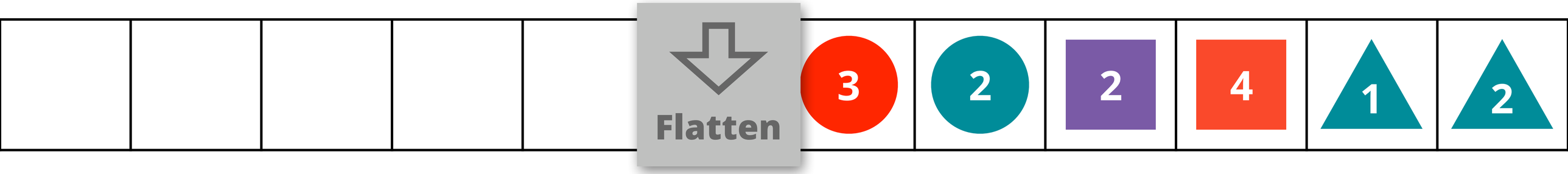

2
3
2
Flatten
4
1
2
1

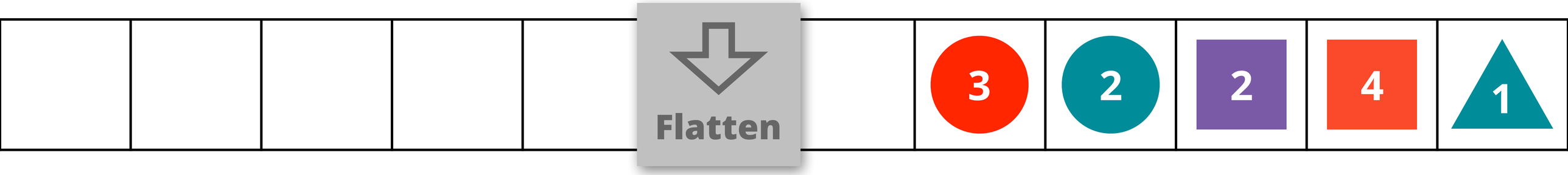

3
Flatten
4
1
2
1

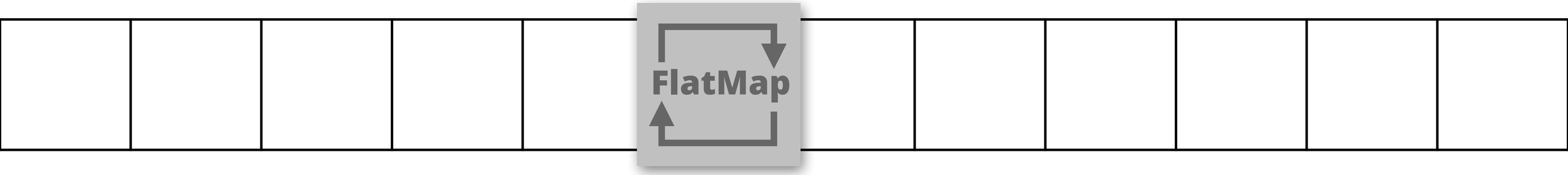

3
Flatten
2
4
1
2
1

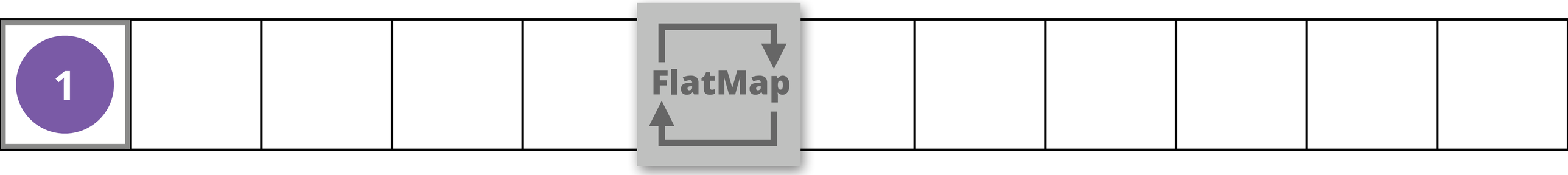

3
Flatten
2
2
4
1
2
1

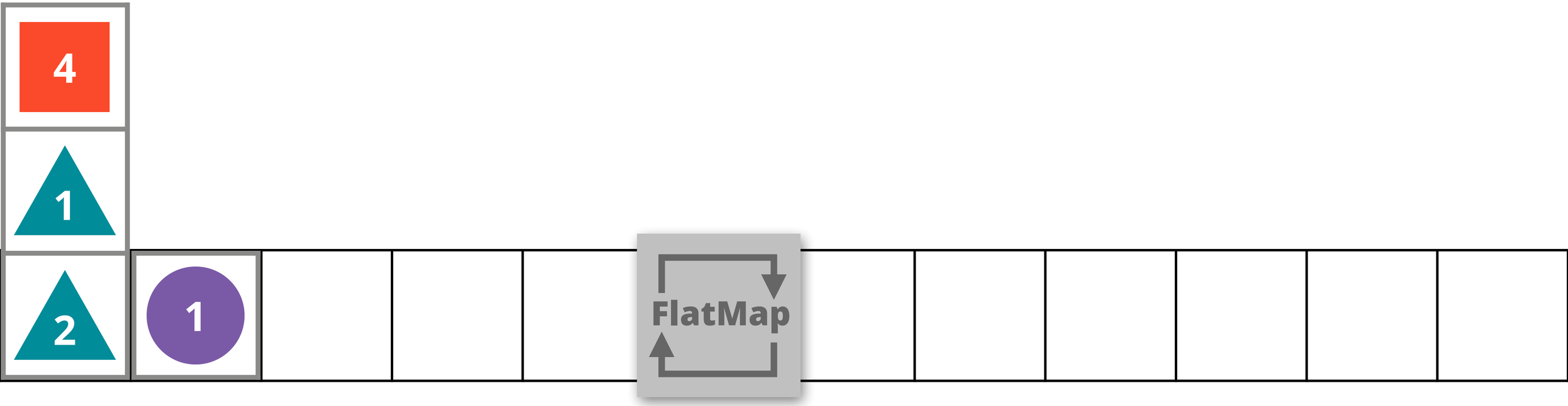

Flatten
2
2
4
1
2
1

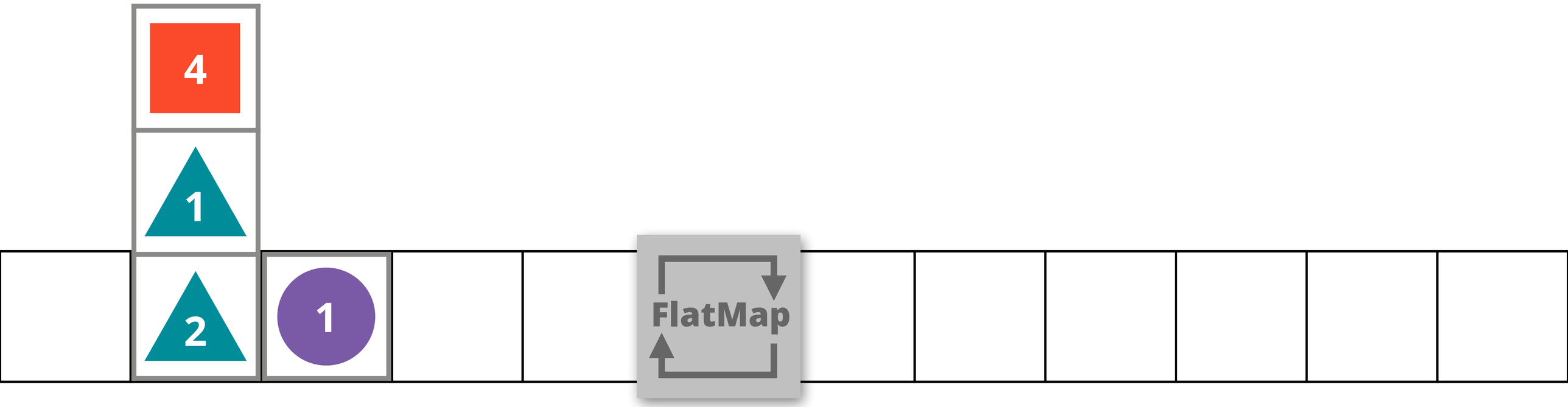

Flatten
3
2
2
4
1
2

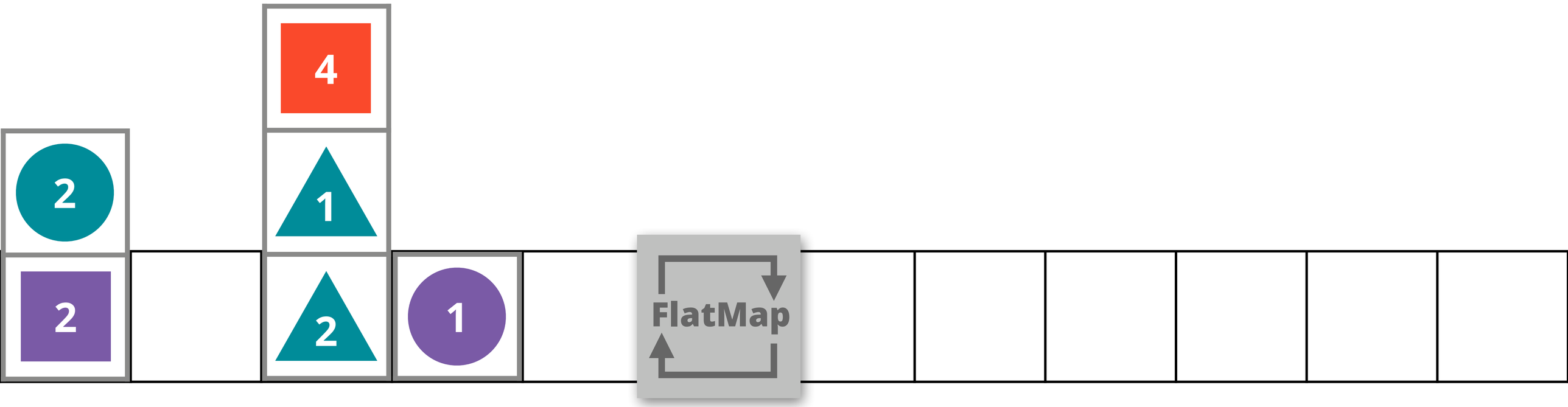

Flatten
3
2
2
4
1

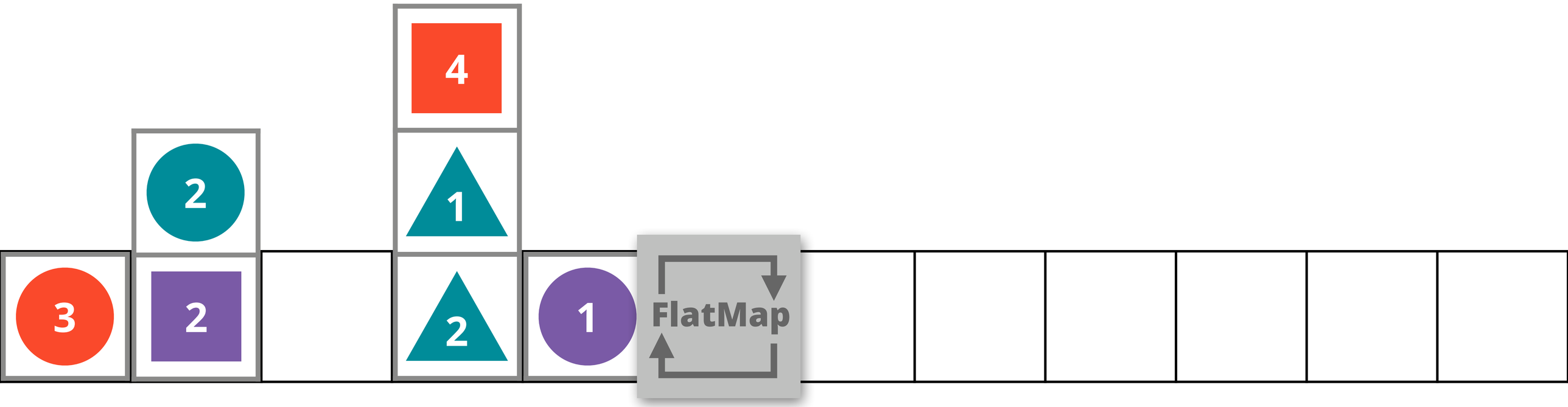

FlatMap

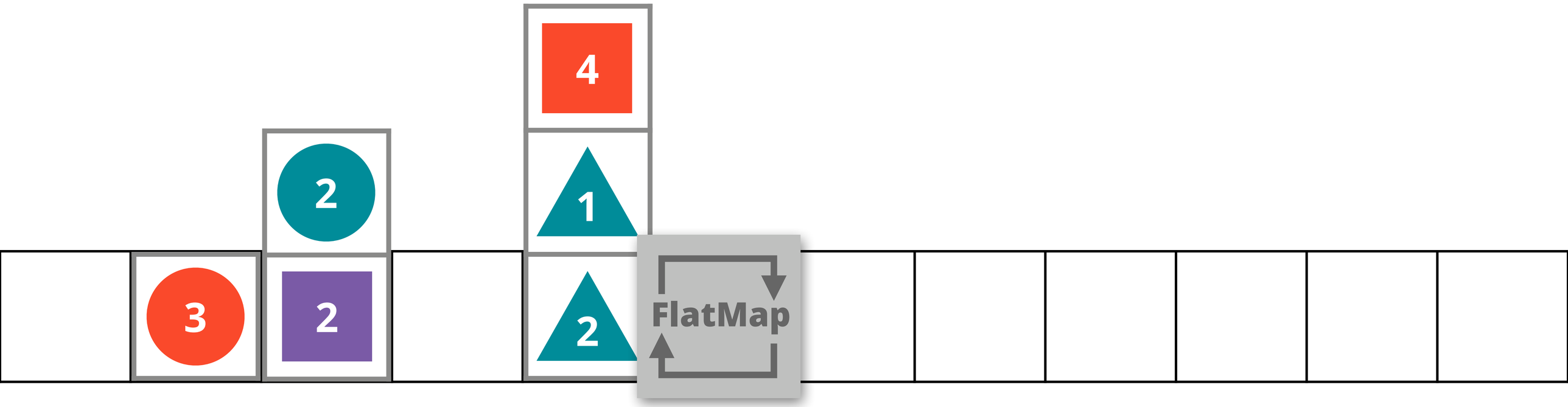

1
FlatMap

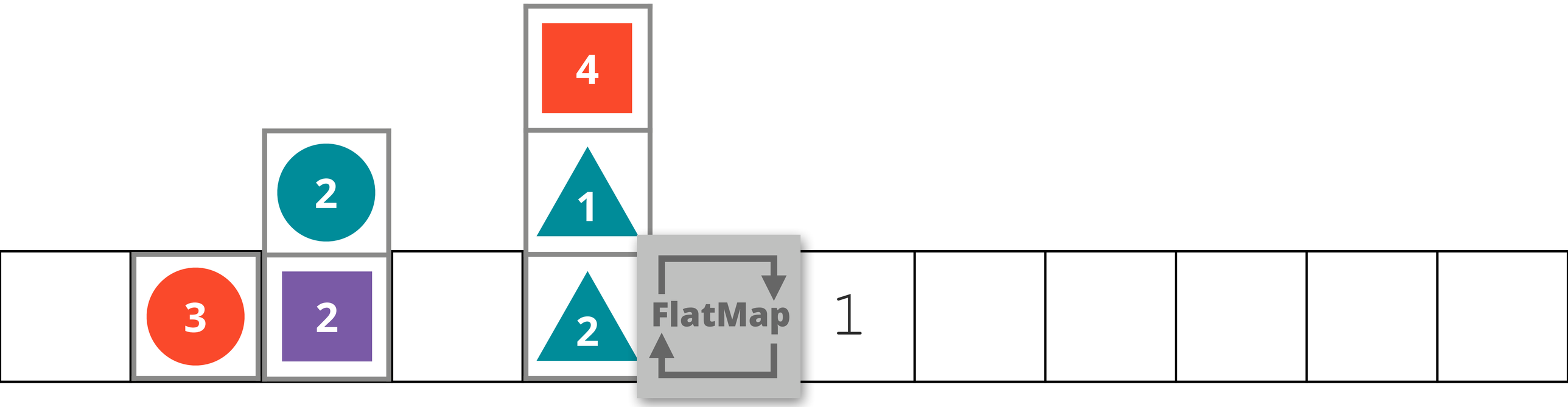

4
1
2
1
FlatMap

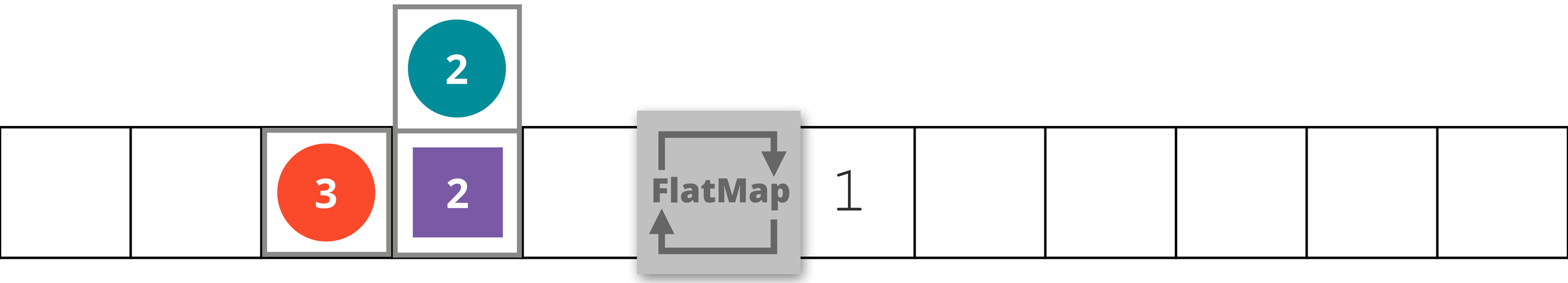

4
1
2
1
FlatMap

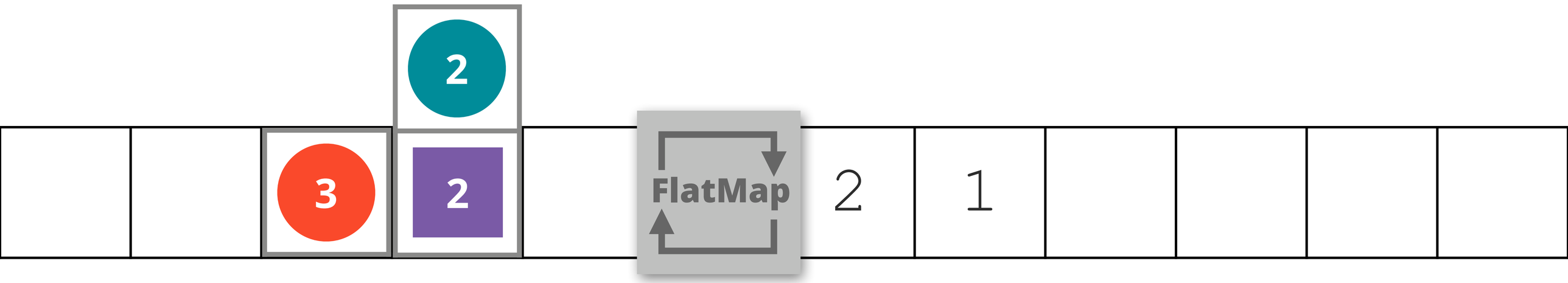

4
2
1
2
2
1
FlatMap

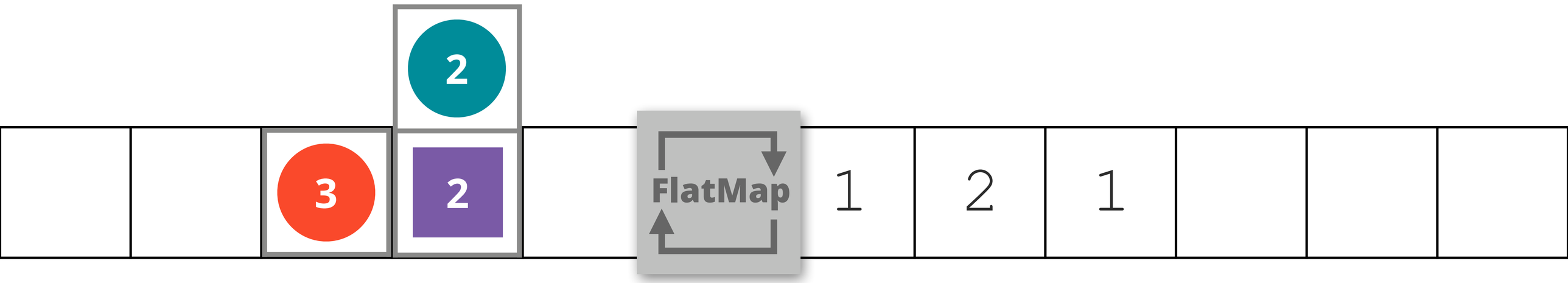

4
2
1
3
2
2
1
FlatMap

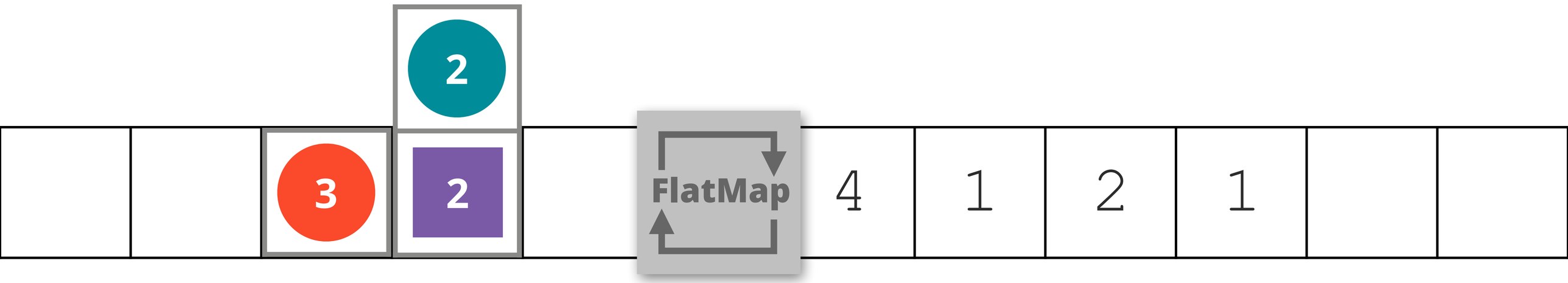

4
2
1
3
2
2
FlatMap

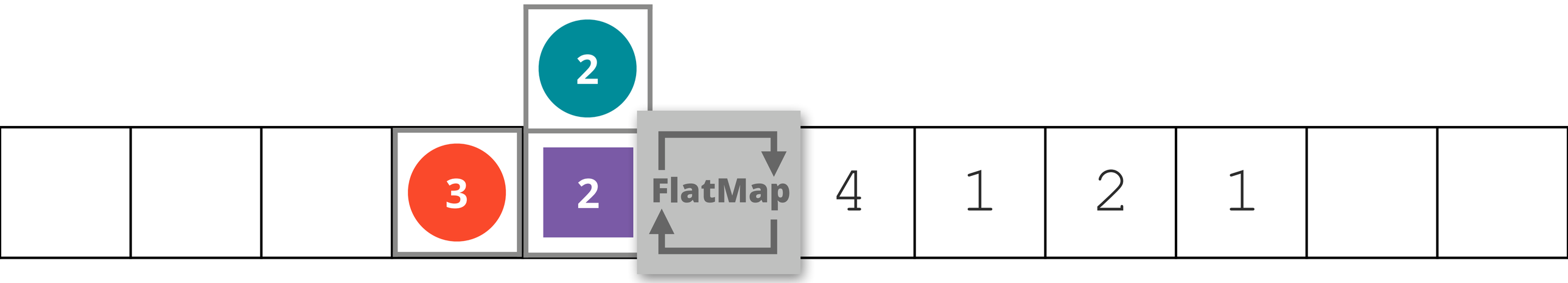

4
2
1
3
2
2
FlatMap
1

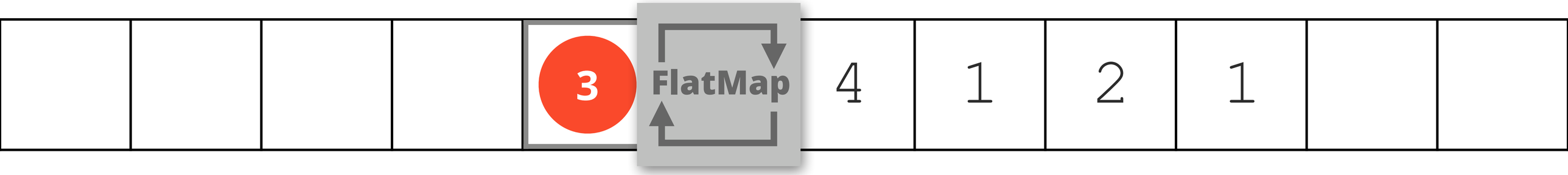

2
3
2
FlatMap
1

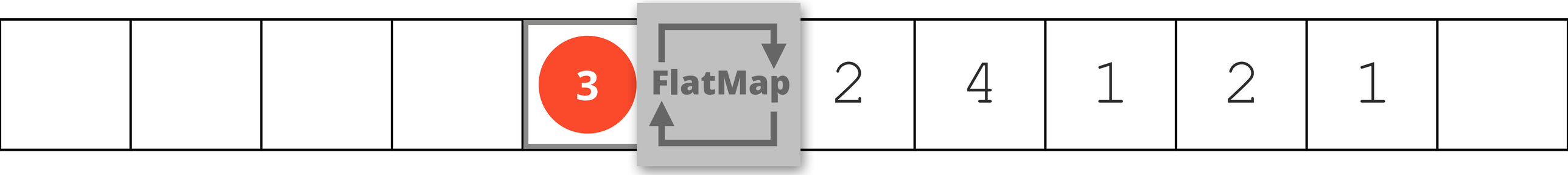

2
3
2
FlatMap
2
1

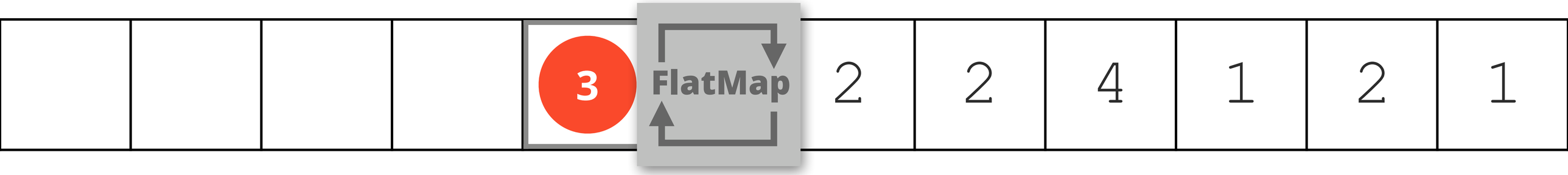

2
3
2
FlatMap
1
2
1

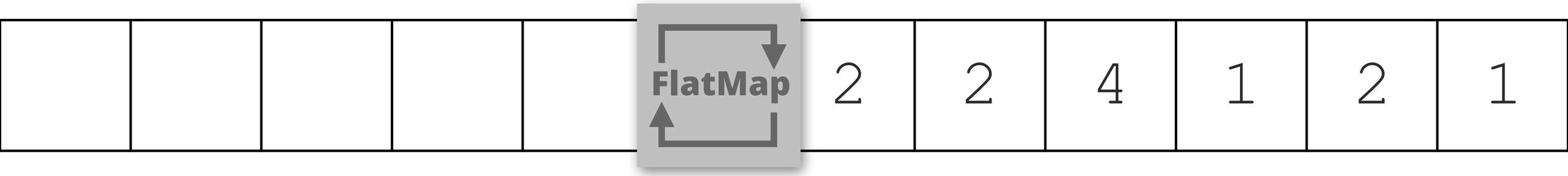

2
3
2
FlatMap
4
1
2
1

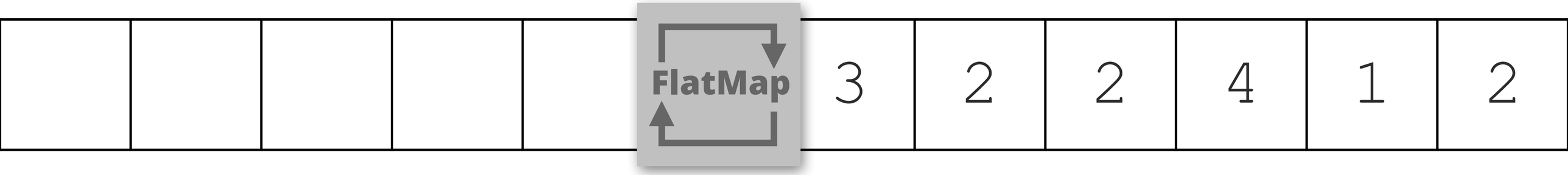

2
3
2
FlatMap
4
1
2
1

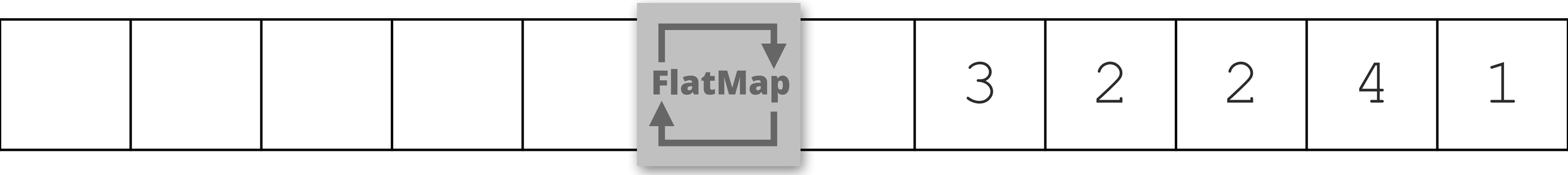

3
FlatMap
4
1
2
1


3
FlatMap
2
4
1
2
1


3
FlatMap
2
2
4
1
2
1


FlatMap
2
2
4
1
2
1


FlatMap
3
2
2
4
1
2


FlatMap
3
2
2
4
1

Advanced Streams

DEMO

Java's Type System

- Static
- Nominal
- Object / Imperative
- Type erased
- Modestly type-inferred
- Slightly Functional

Static

- Type system enforced by Java language (& javac)
- The JVM has a philosophy of:
  - Trust but verify
  - Go Classfile or go home
  - Single entry point (checkpoint) for code
  - Helps enforce runtime type system
- Every generation rediscovers static typing
  - And why we invented it in the first place

Title Text

"**Strong typing:** A type system that I like and feel comfortable with.

Weak typing: A type system that worries me, or makes me feel uncomfortable.."

– Curtis Poe

Nominal

- Java’s types are Nominal (name-based)
- Types only compatible if explicit inheritance exists
  - Either class inheritance or interface inheritance
- No way of talking about a type except by name
  - At least, for lvalues
- No duck typing

Object / Imperative

- Java has a simple OO model
- Not pure OO
  - Simplifies working with primitives
  - Makes it easier to teach (in theory)
  - Historically, aided transition for C/C++ devs

Type Erased

- Java has parametrised types
  - Unbounded type parameters exist only at compile time
  - Generation of bytecode removes information
- Exists for backwards compatibility
  - “Raw types”
- Often complained about
  - Also often misunderstood

Modestly Type Inferred

- Started with Java 7 type diamond syntax
  - `List<String> l = new List<>();`
- Compare to Scala:
  - `val x = new HashMap();`

A Secret

- Java has always had type inference

Modestly Type Inferred

- Java has always had type inference

```
class Foo {
      public Foo bar() { return this; }
      public String toString() { return "Foo!"; }
}
System.out.println(new Foo().bar().bar().bar());
```

- Java can infer the type at each bar() call

Modestly Type Inferred

- What about this?

```
public class Test {
  public static void main(String[] args) {
     (new Object() {
      public void bar() {
          System.out.println("bar!");
      }
     }).bar();
  }
}
```

Slightly Functional (8)

- Java 8 isn’t (that) functional
  - Implements key idioms (map, filter etc)
- Java Collections are very concrete
  - Require Stream abstraction
- Type erasure affects higher-order function type safety
- Existence of void complicates matters
- No “monadic” approach
- But do these limitations really matter to devs?

A Short Java Reading List

- **Effective Java (3rd Ed)** - J. Bloch
- **Java in a Nutshell (7th Ed)** - B. Evans & D. Flanagan
- **Core Java** - C. Hortsmann
- **Java 8 Lambdas** - R. Warburton
- **Modern Java in Action** - R. Urma et al.
- **Oracle Java Magazine** (free subscription)