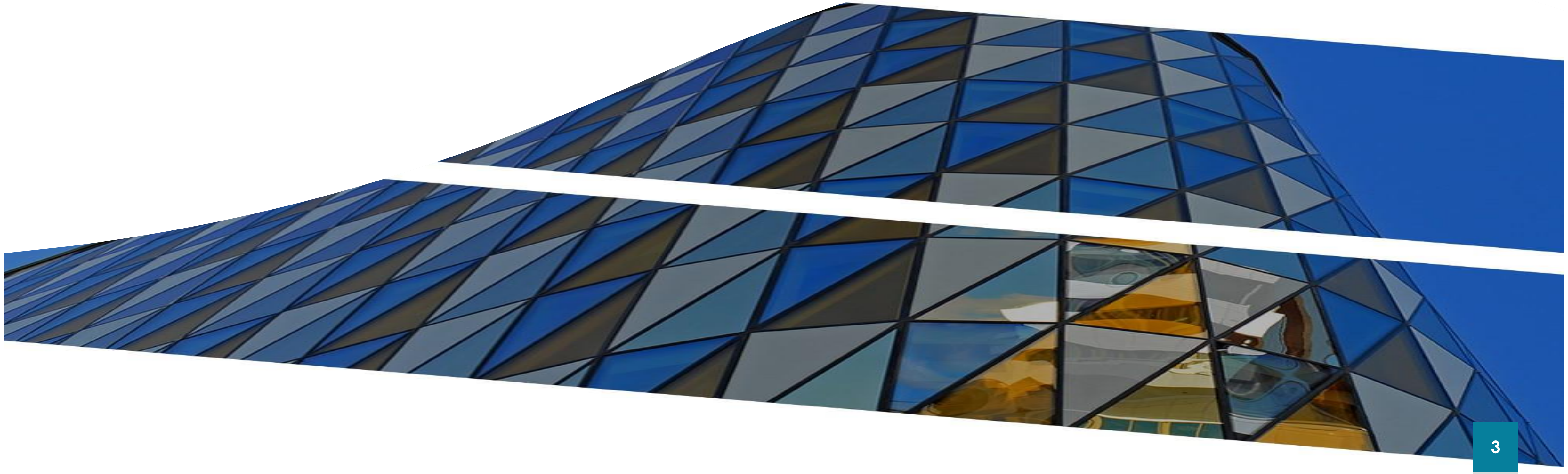
Java Stream API

Where to Find The Code and Materials?

https://github.com/iproduct/course-stream-api-2022

Java 8 Stream API

Practical Exercises – Functional Programming Koans



Agenda for This Session

- Fundamentals
- Functional interfaces
- Method references
- Constructor references

Новости в Java™ 8+

- Ламбда изрази и поточно програмиране – пакети **java.util.function** и **java.util.stream**)
- Референции към методи
- Методи по подразбиране и статични методи в интерфейси – множествено наследяване на поведение в Java 8
- Функционално програмиране в Java 8 с използване на монади (напр. Optional, Stream) – предимства, начин на реализация, основни езикови идиоми, примери

Функционални интерфейси в Java™ 8

- Функционален интерфейс = интерфейс с един абстрактен метод SAM (Single Abstract Method) – @FunctionalInterface

- Примери за функционални интерфейси в Java 8:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
public interface Runnable {
    public void run();
}
public interface Callable<V> {
    V call() throws Exception;
}
```

Ламбда изрази – пакет java.util.function

Примери:

```
(int x, int y) -> x + y

() -> 42

(a, b) -> a * a + b * b;

(String s) -> { System.out.println(s); }

book -> book.getAuthor().fullName()

voter -> voter.getAge() >= legalAgeOfVoting

(person1, person2) -> person1.getAge() - person2.getAge()

(song1, song2) -> song1.getArtist().compareTo(song2.getArtist())
```

Правила за форматирне на ламбда изрази

- **Ламбда изразите (функциите)** могат да имат произволен брой **параметри**, които се ограждат в скоби, разделят се със запетаи и могат да имат или не деклариран тип (ако нямат - типът им се извежда от **контекста на използване = target typing**). Ако са само с един параметър, то скобите не са задължителни.

- **Тялото на ламбда изразите** се състои от произволен езикови конструкции (statements), разделени с ; и заградени във фигурни скоби. Ако имаме само една езикова конструкция – израз то използването на фигурни скоби не е необходимо – в този случай стойността на израза автоматично се връща като стойност на функцията.

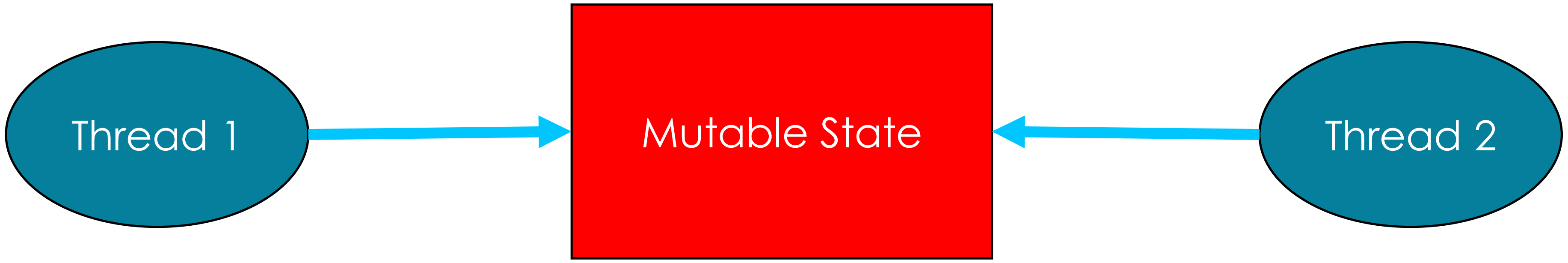
Пакет java.util.function

- **Predicate<T>** – предикат = булев израз представящ свойство на обекта подавано като аргумент
- **Function<A,R>**: функция която приема като аргумент **A** и го трансформира в резултат **R** (метод **apply()**)
- **Supplier<T>** – с помощта на **get()** метод всеки път връща инстанция (обект) – фабрика за обекти
- **Consumer<T>** – приема аргумент (метод **accept()**) и изпълнява действие върху него
- **UnaryOperator<T>** – оператор с един аргумент **T -> T**
- **BinaryOperator<T>** – бинарен оператор **(T, T) -> T**

Data Streams Programming

Problem with OOP: Mutable State

- The object methods are supposed to mutate the object’s internal state
- When there is state sharing:



- Bottlenecks (Contention), Deadlocks, Complexity in State Access Management (mutual exclusion between threads)

OOP vs. Functional Composition

- **OOP** – imperative, hard to chieve concurrency, less-reusable abstractions (how many times you have created User class in your career?)
- **FP** – declarative, always safe concurrency (pure functions), coarse grained abstractions, code reuse via functional composition, Composable abstractions: Stream, Optional, etc.

Functional Programming

FP is a type of programming paradigm which has several features:

- Purity:
  - Function reads all inputs from its input arguments.
  - Function exports all outputs to its return values.
  - The function always evaluates the same result value given the same argument value(s).
  - Evaluation of the result does not cause any semantically observable side effect or output, such as mutation of mutable objects or output to I/O devices.
- Immutability – state of objects cannot be modified after it is created, but wait, how can we program without modifying state ?
- First-Class & High-Order Function
- TCO, Closure, Curry…

First Class Functions

Capability of programming language to:

- pass functions as arguments to other functions
- return functions as the values from other functions
- assign functions to variables
- store functions in data structures

To be concise, function is just like all other values like integer, float, double, etc..

Higher Order Functions

Function that does at least one of the following:

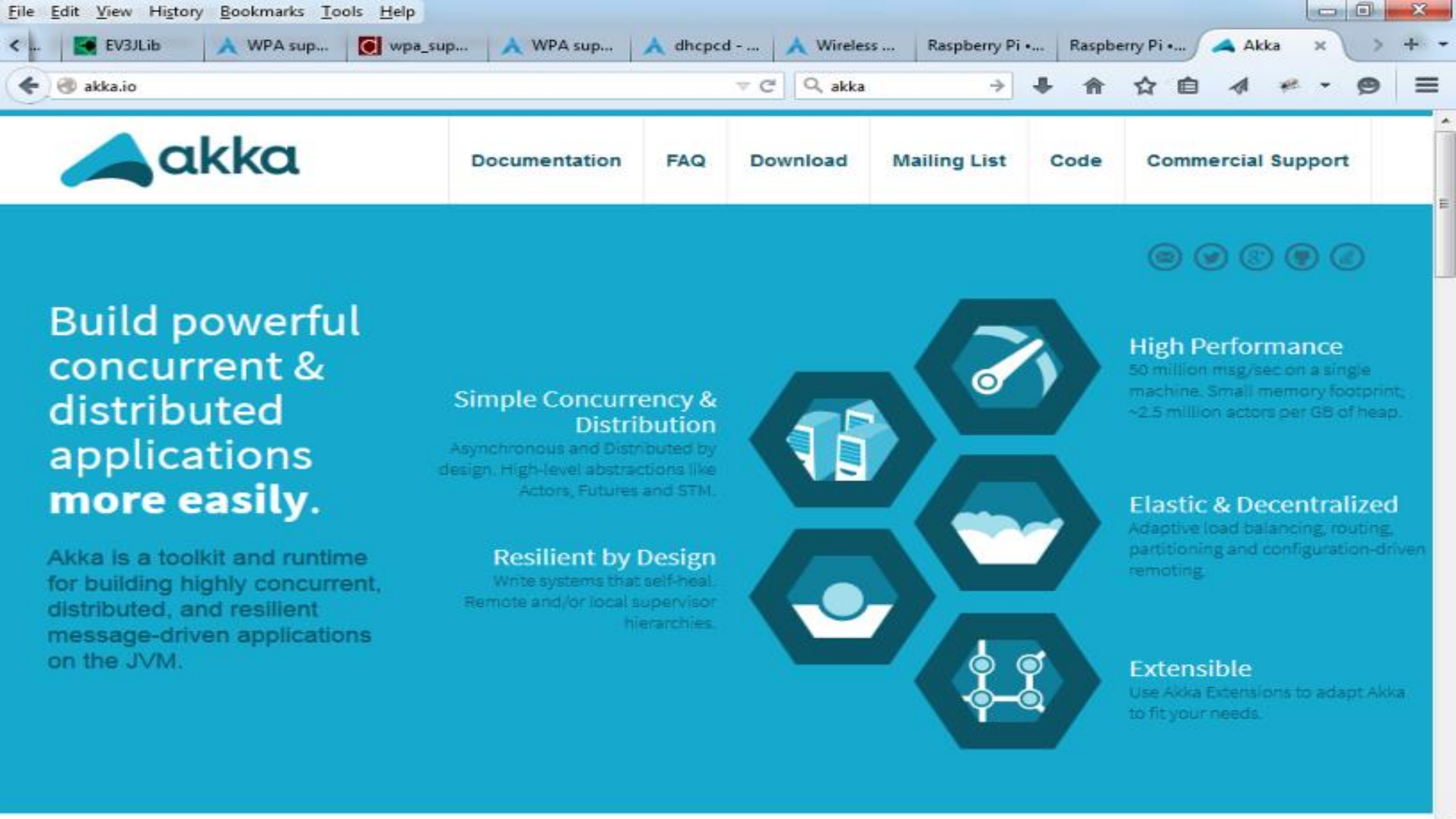
- takes one or more functions as arguments
- returns a function as its result
- Examples:

```
var lines = Files.lines(path).map(line -> line.toUpperCase());
var numbers = IntStream.iterate(1, x -> x + 1).boxed();
var results = zip(numbers, lines, (Integer n, String line) -> n + ": " + line);
results.forEach(System.out::println);
```

What can FP offer to distributed computing ?

- No side-effects and immutable variables – FP facilitates code distribution over several CPU and eases concurrent programming
- Functions are better building components than objects:
  - Functions can be combined, sent remotely
  - Functions can be applied locally on distributed data sets (e.g. parallel stream, using Fork-Join pool underneath).
- In order to do the splitting of the work between multiple threads (forking) the Java Streams use:

  spliterator = split iterator
- The results can be joined after that in a single result (e.g. reduce)
- Example: Map – Reduce big data architecture (Google, Hadoop)



Build powerful concurrent & distributed applications **more easily.**

Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications on the JVM.

Simple Concurrency & Distribution

Asynchronous and Distributed by design. High-level abstractions like Actors, Futures and STM.

Resilient by Design

Write systems that self-heal. Remote and/or local supervisor hierarchies.

High Performance

50 million msg/sec on a single machine. Small memory footprint; ~2.5 million actors per GB of heap.

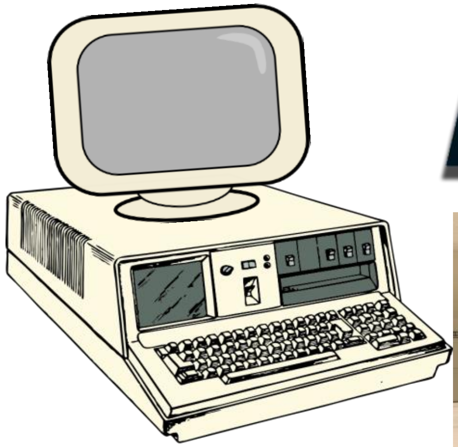
Elastic & Decentralized

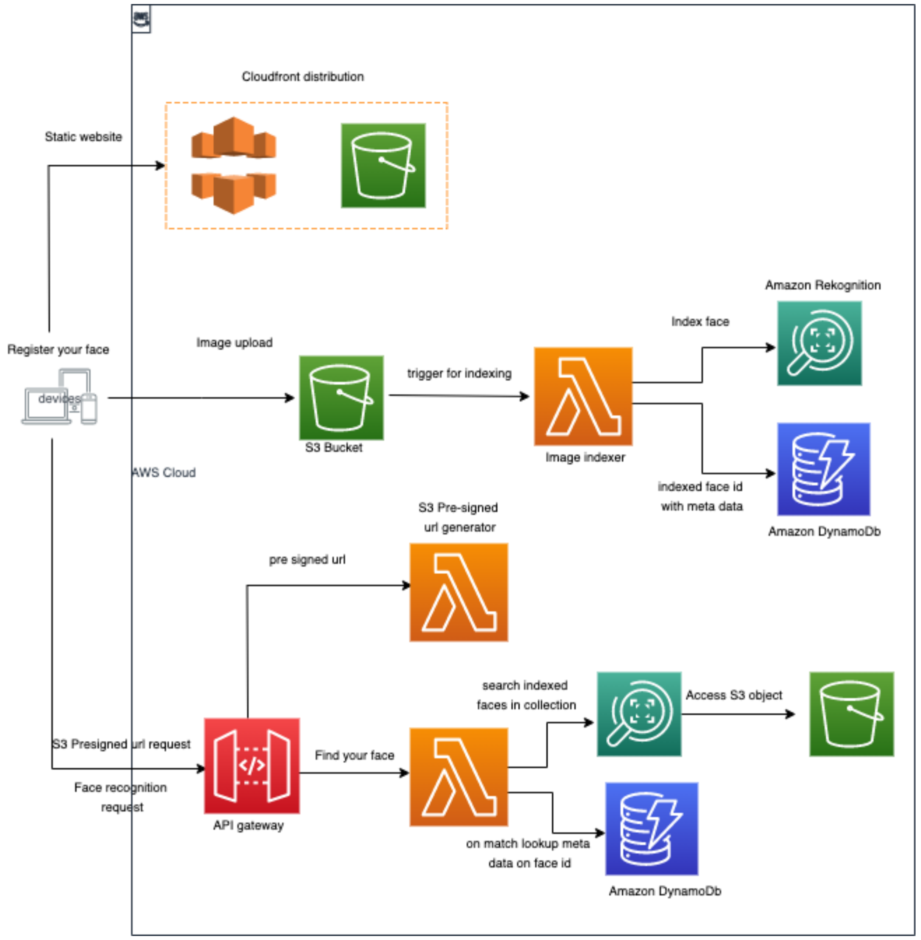
Adaptive load balancing, routing, partitioning and configuration-driven remoting.

Extensible

Use Akka Extensions to adapt Akka to fit your needs.

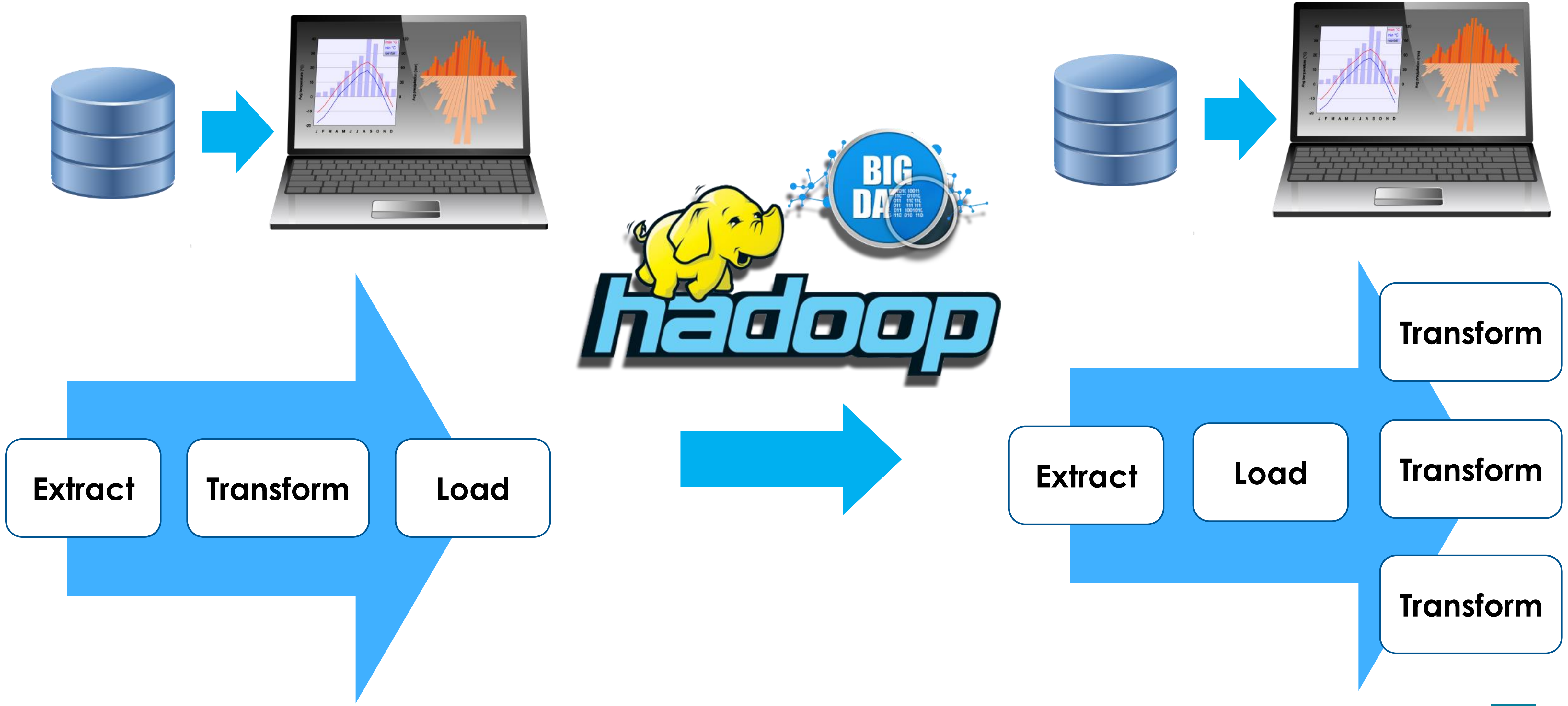
Need for Speed (and Scalability:)



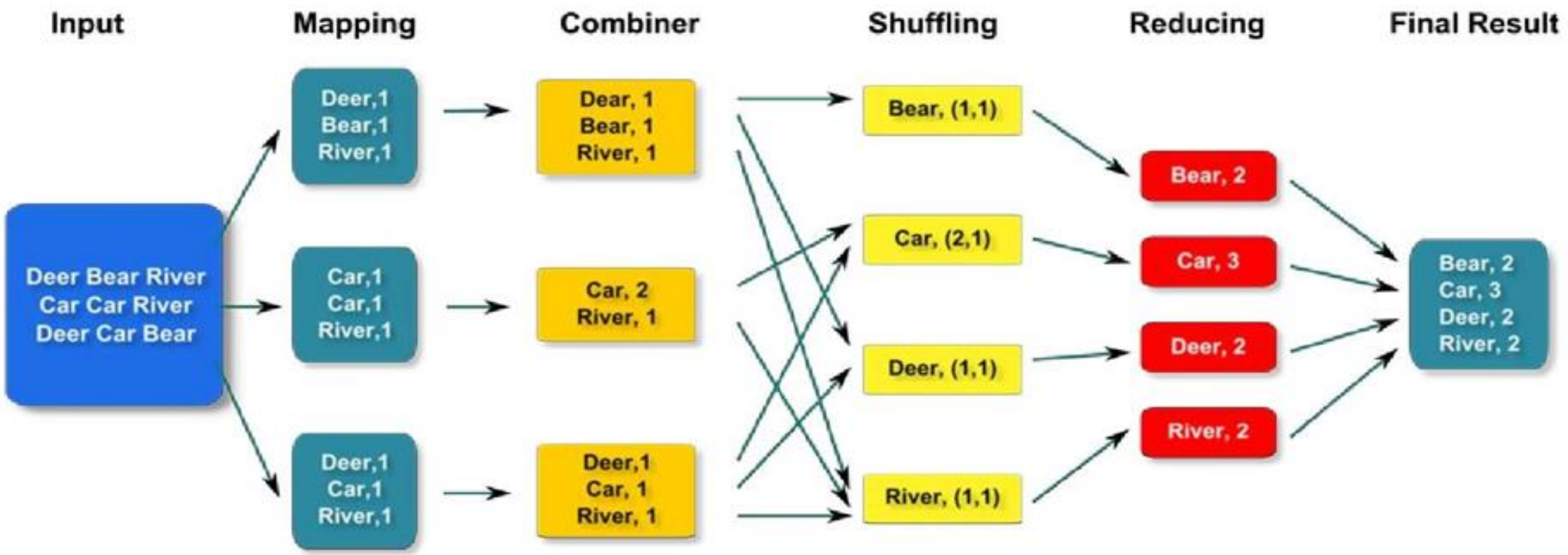


Source: https://aws.amazon.com/blogs/opensource/simplifying-serverless-best-practices-with-aws-lambda-powertools-java/

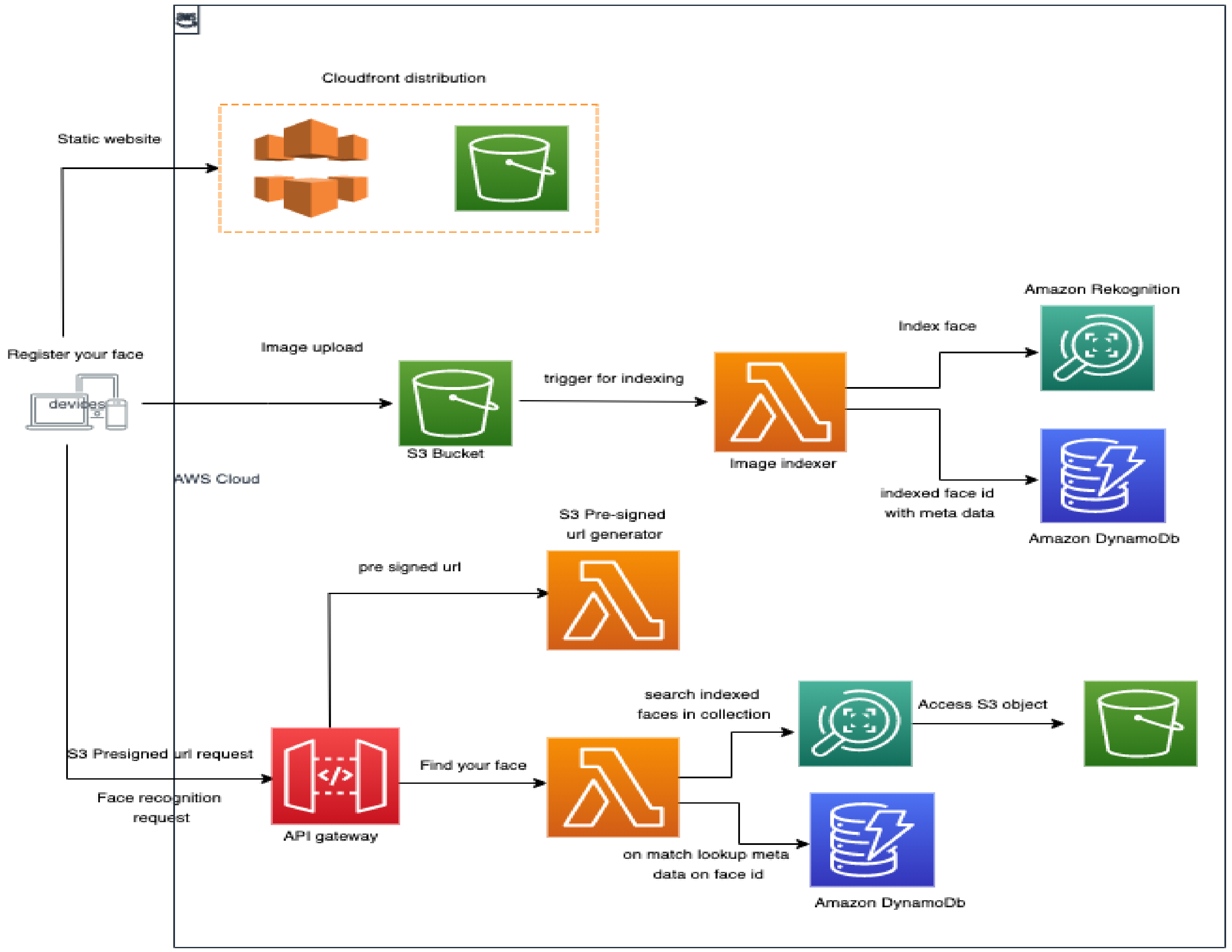
Batch Processing



Map-Reduce Architecture

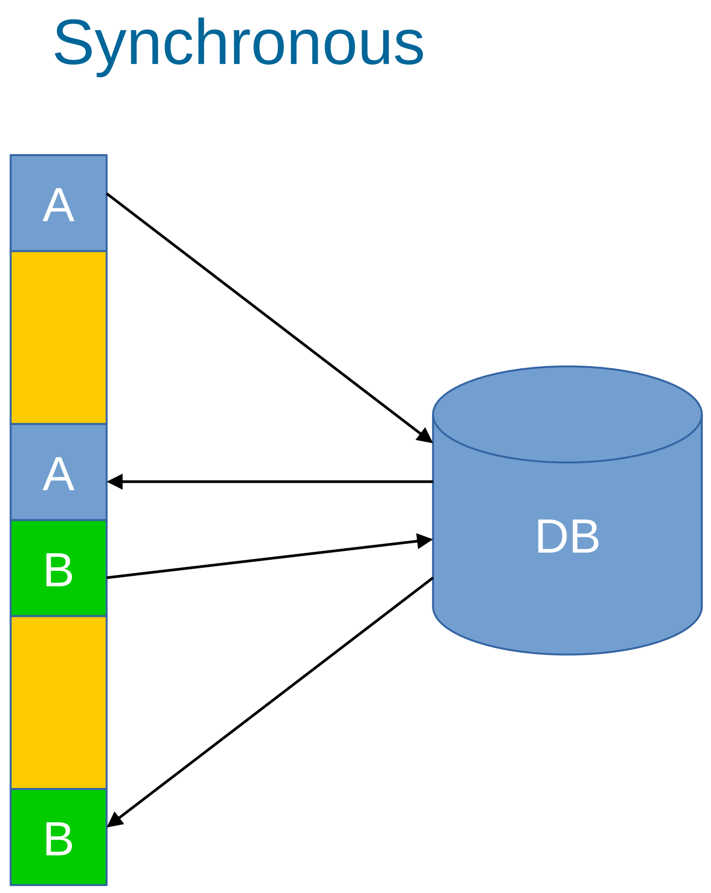


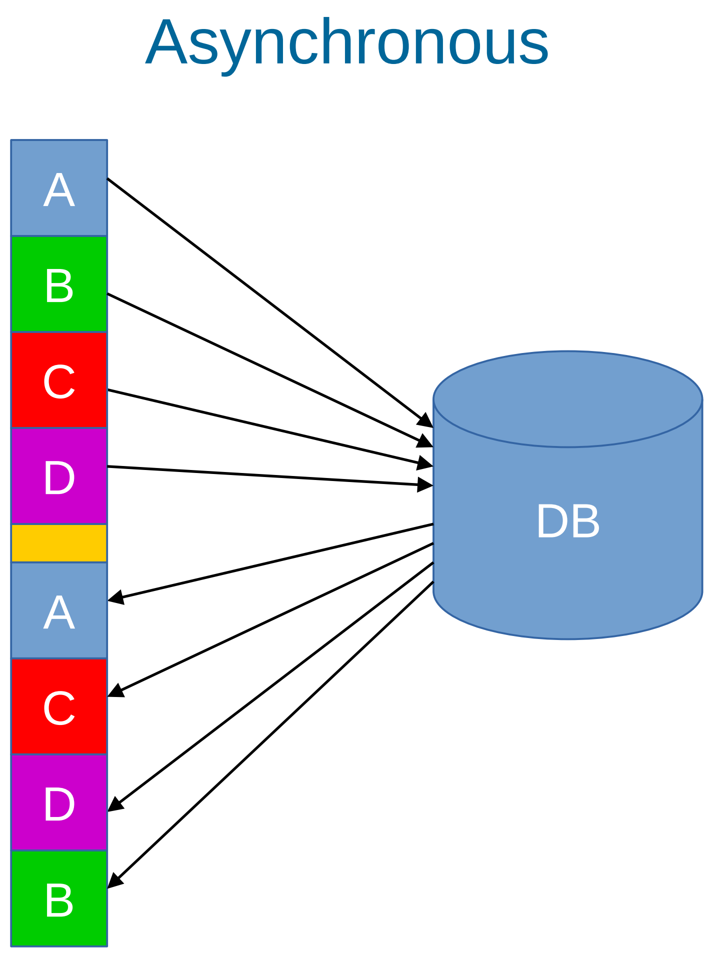
Amazon Lambda



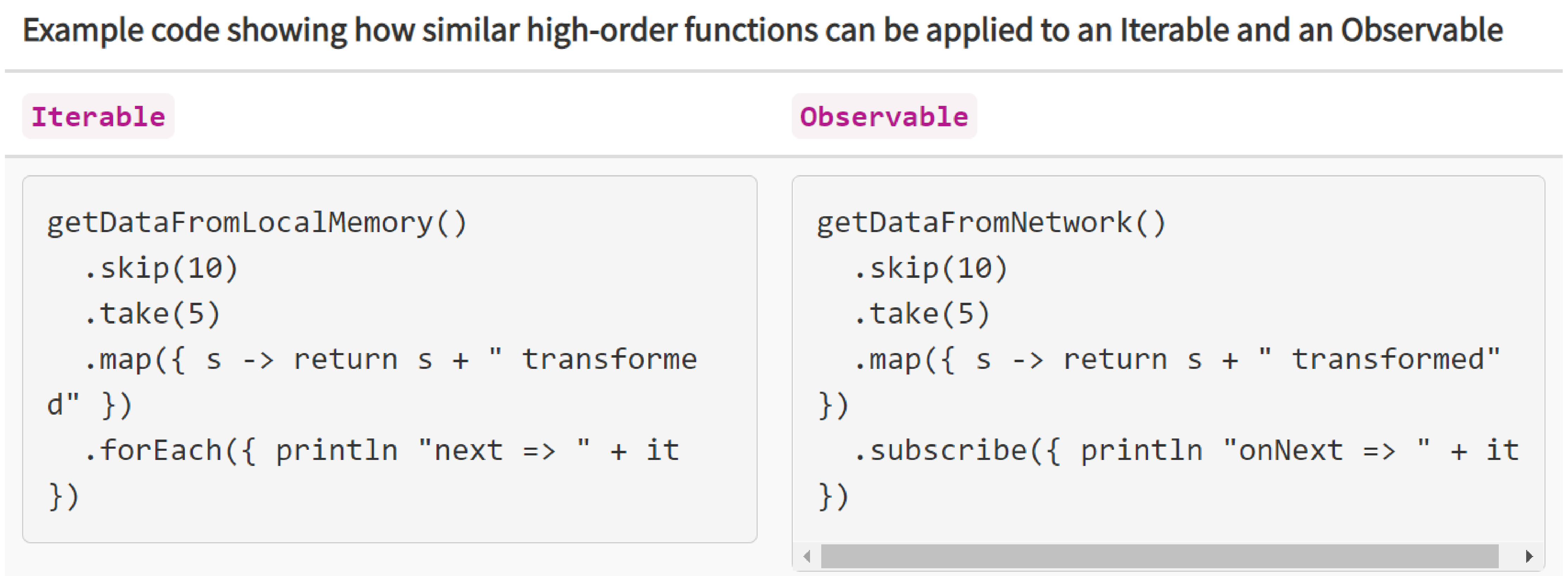
Source: https://aws.amazon.com/blogs/opensource/simplifying-serverless-best-practices-with-aws-lambda-powertools-java/

Synchronous vs. Asynchronous IO





ReactiveX: Observable vs. Iterable



Example code showing how similar high-order functions can be applied to an Iterable and an Observable

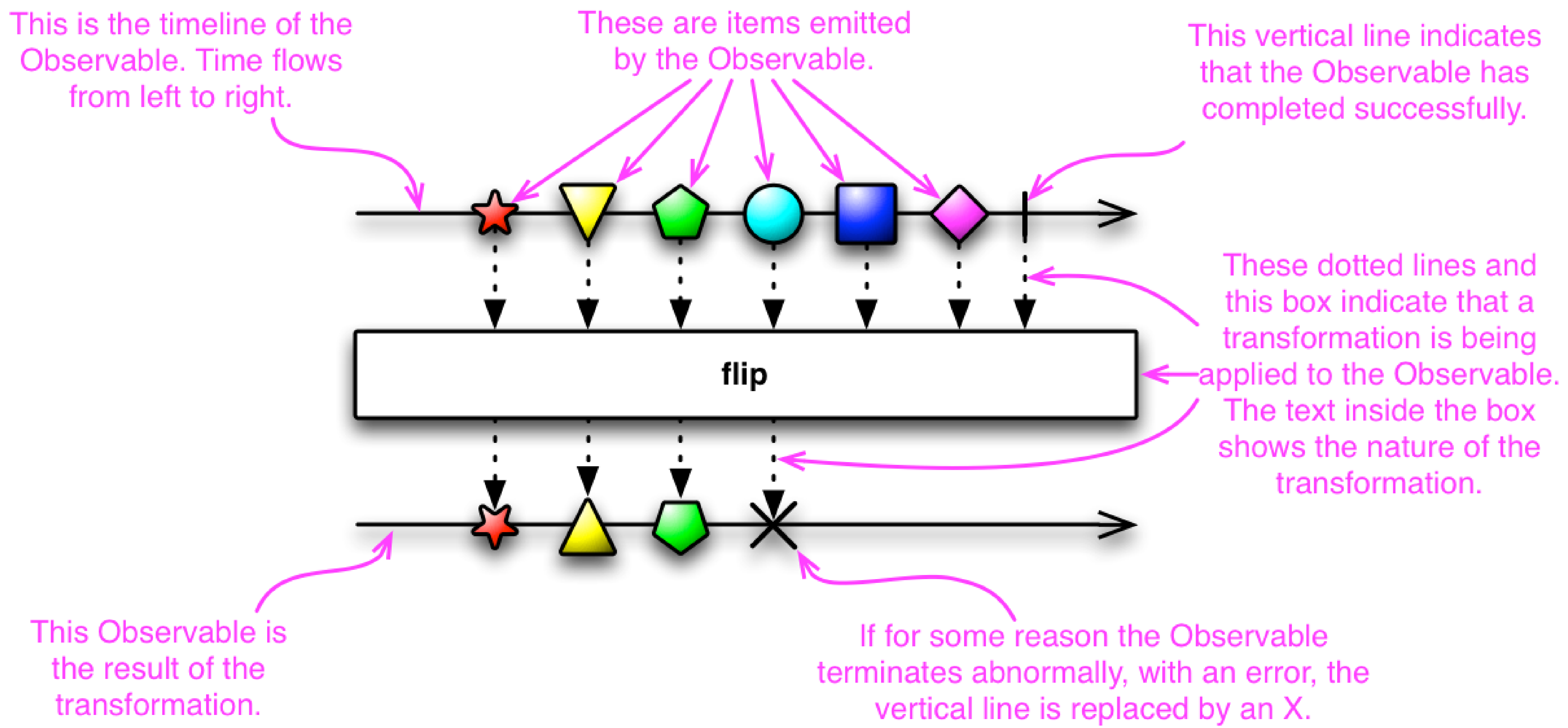
`Iterable`	`Observable`
see below	see below

```
getDataFromLocalMemory()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transforme
d" })
  .forEach({ println "next => " + it
})
```

```
getDataFromNetwork()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transformed"
})
  .subscribe({ println "onNext => " + it
})
```

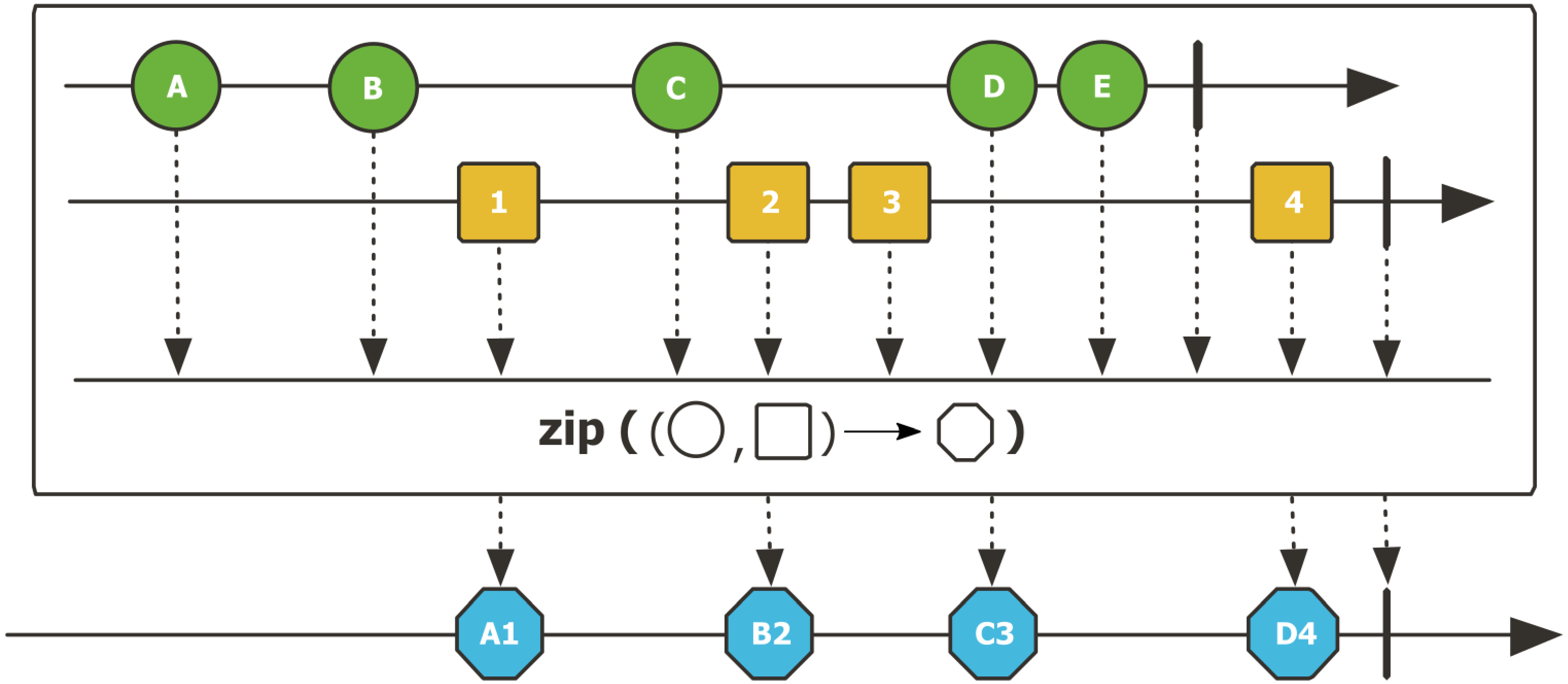
You can think of the Observable class as a **"push"** equivalent to Iterable, which is a **"pull."** With an Iterable, the consumer pulls values from the producer and the thread blocks until those values arrive. By contrast, with an Observable the producer pushes values to the consumer whenever values are available. This approach is more flexible, because values can arrive synchronously or asynchronously.

ReactiveX Observable – Marble Diagrams


This is the timeline of the Observable. Time flows from left to right.
These are items emitted by the Observable.
This vertical line indicates that the Observable has completed successfully.
flip
These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.
This Observable is the result of the transformation.
If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

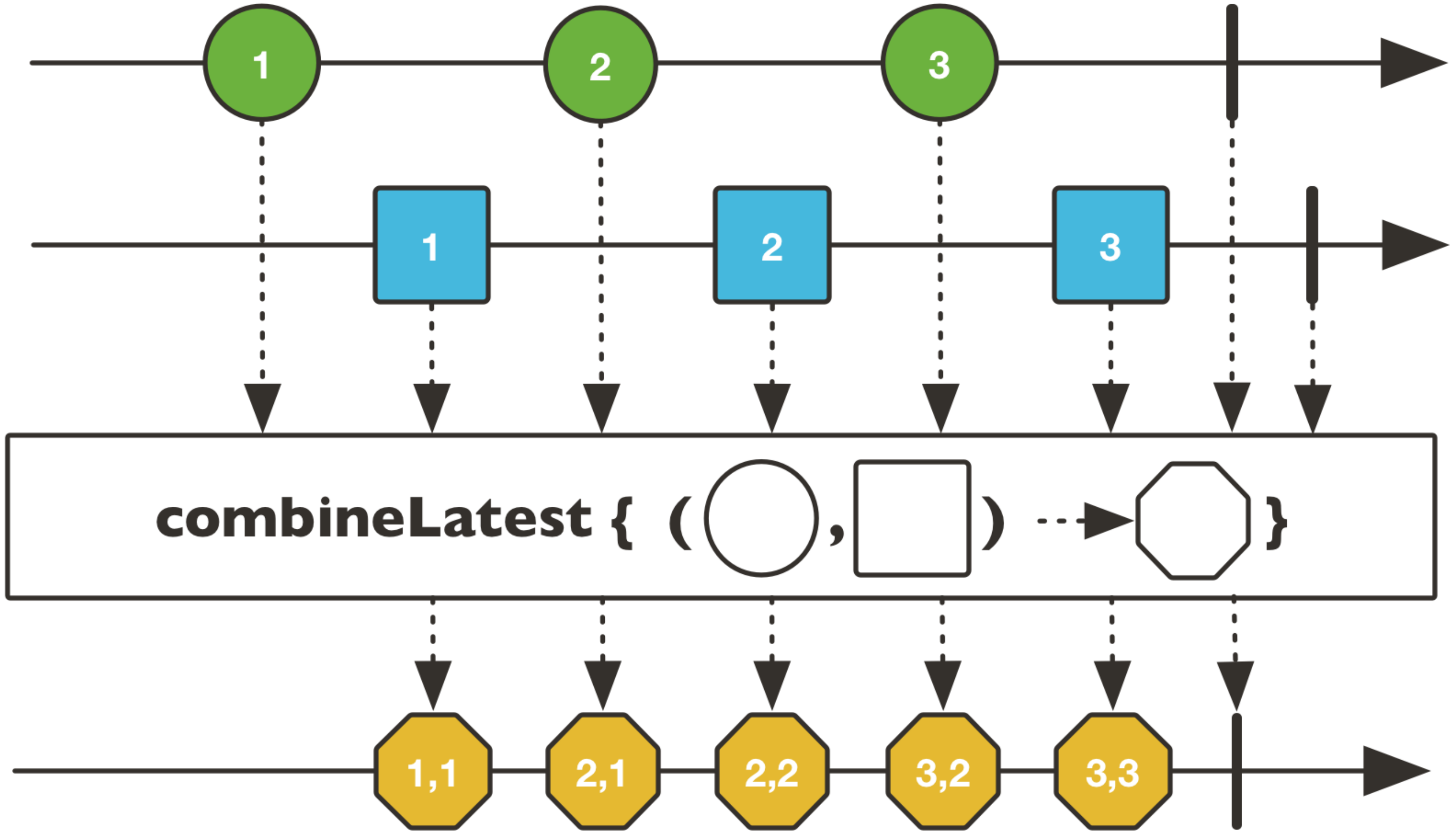
By ReactiveX - Creative Commons Attribution 3.0 License, Source: http://reactivex.io/documentation/observable.html

Example: Zip



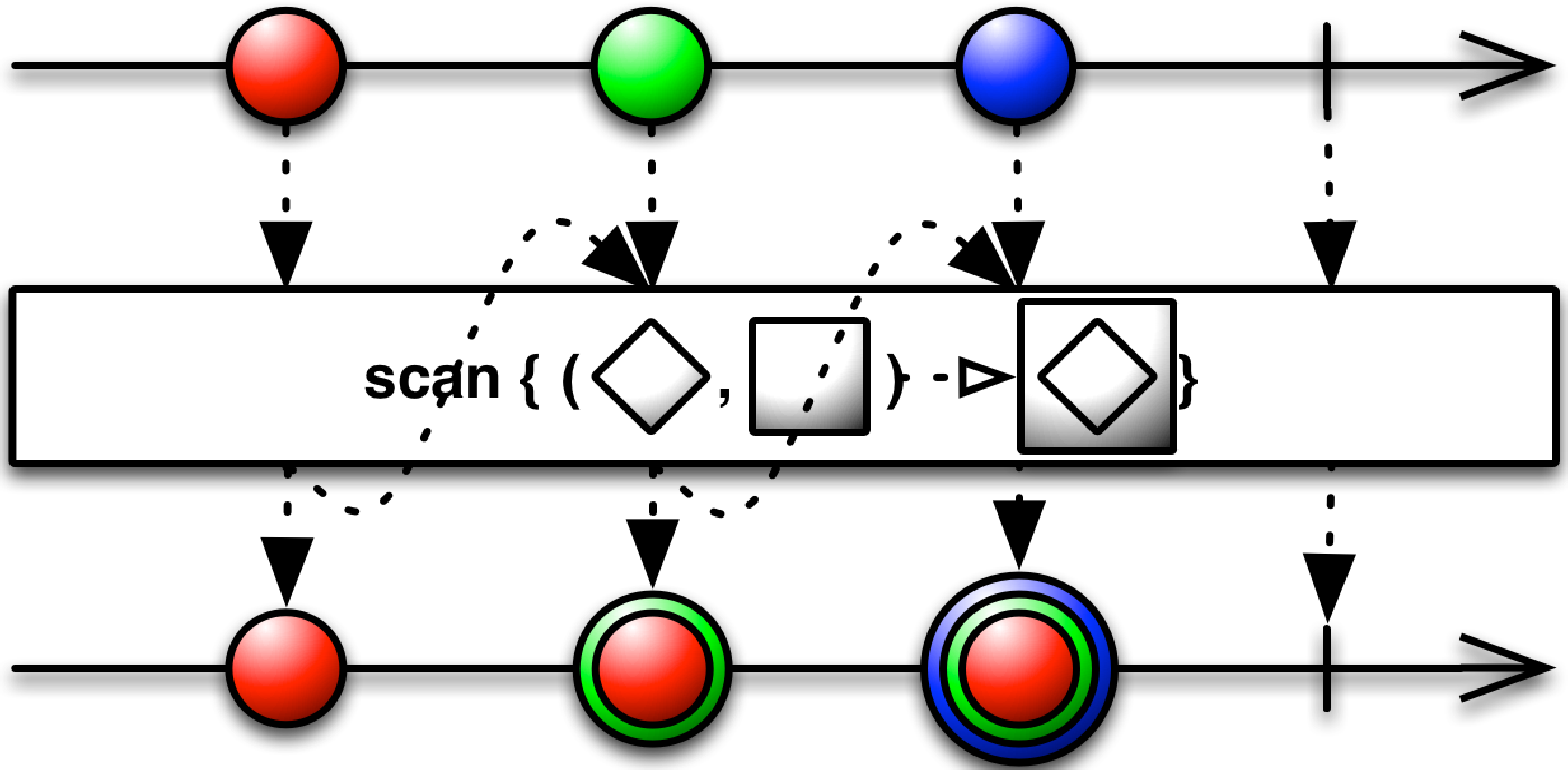
https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html, *Apache Software License 2.0*

Example: CombineLatest



https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html, *Apache Software License 2.0*

Redux == Rx Scan Opearator

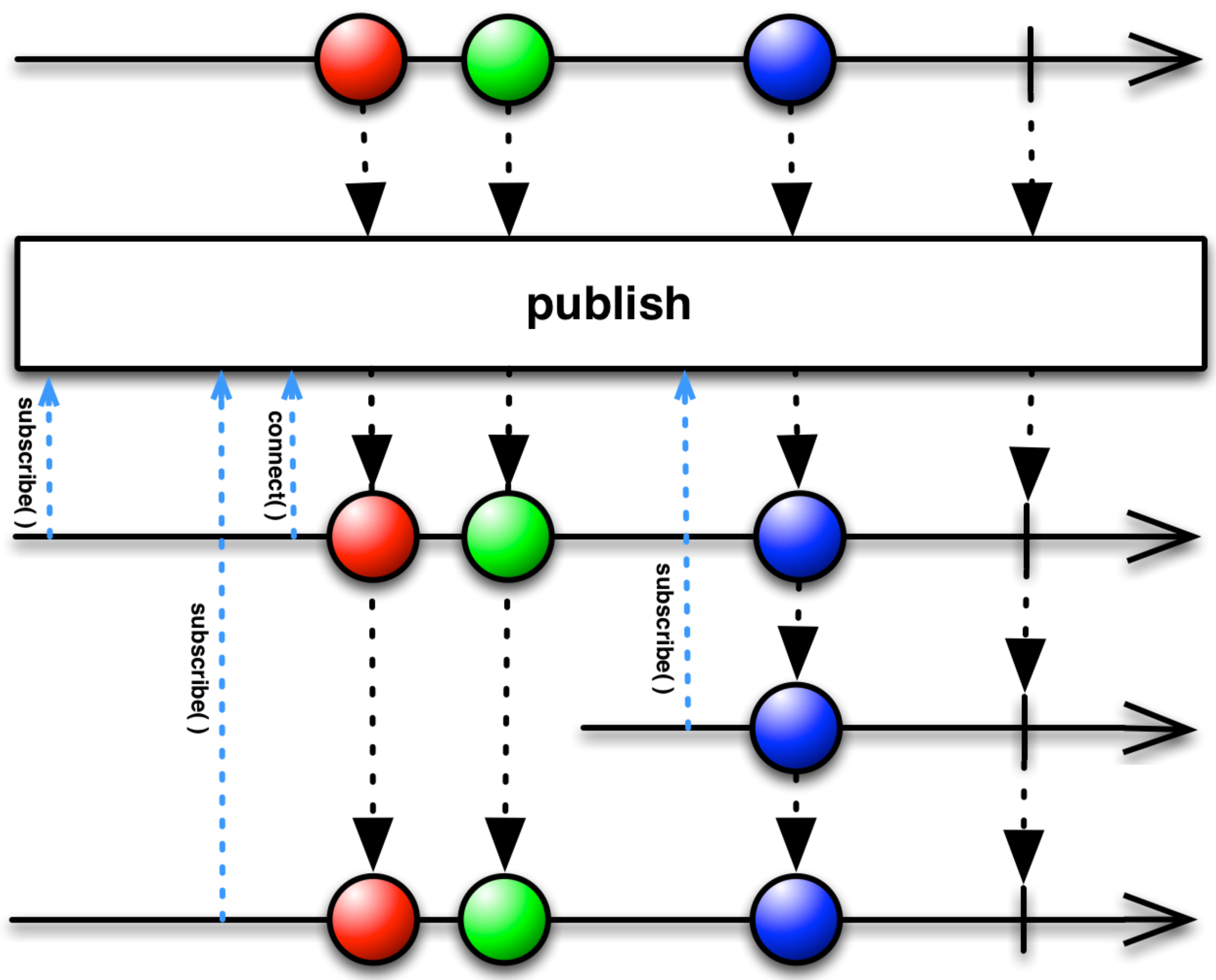


Source: RxJava 2 API documentation, http://reactivex.io/RxJava/2.x/javadoc/

Hot and Cold Event Streams

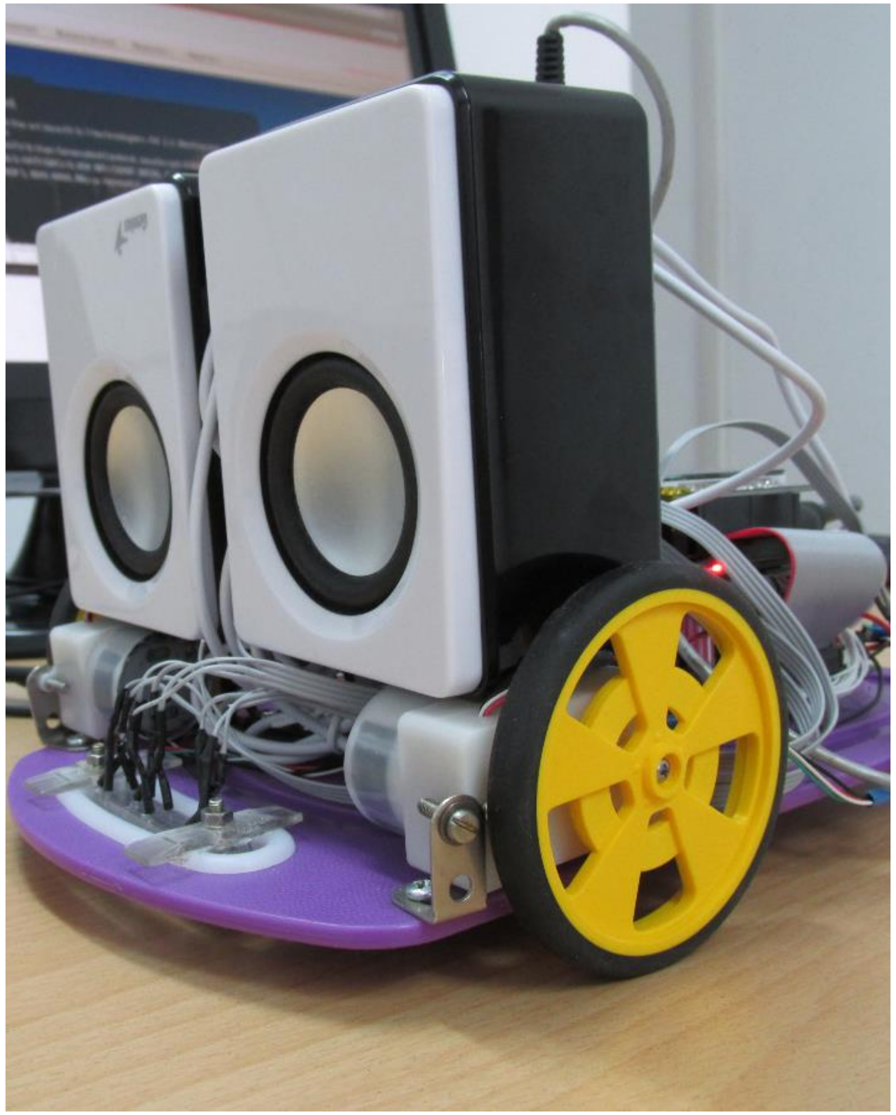
- PULL-based (Cold Event Streams) – Cold streams (e.g. RxJava Observable / Flowable or Reactor Flow / Mono) are streams that run their sequence when and if they are subscribed to. They present the sequence from the start to each subscriber.
- PUSH-based (Hot Event Streams) – emit values independent of individual subscriptions. They have their own timeline and events occur whether someone is listening or not. When subscription is made observer receives current events as they happen.
- *Example:* mouse events

Converting Cold to Hot Stream



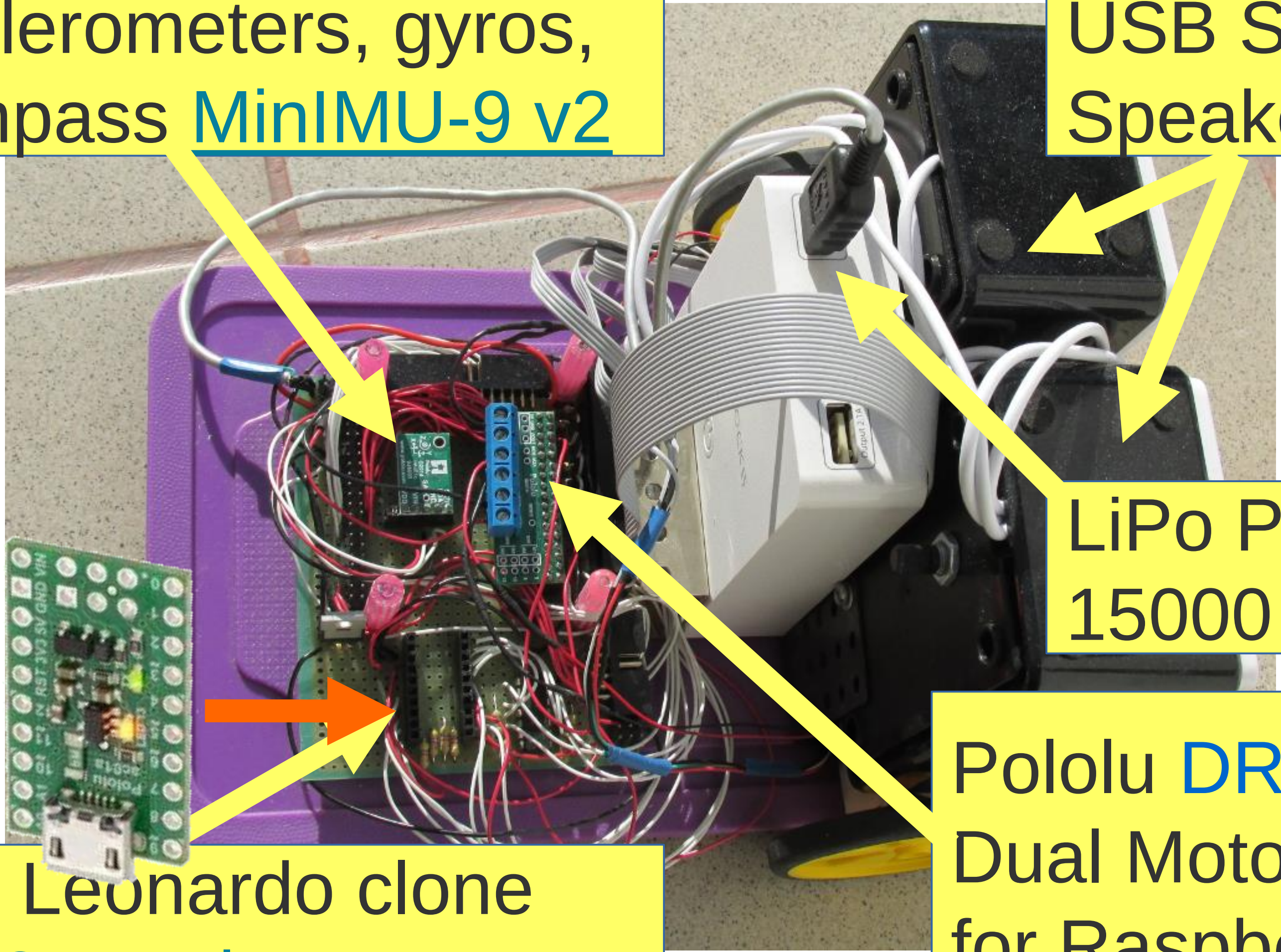
Source: RxJava 2 API documentation, http://reactivex.io/RxJava/2.x/javadoc/

IPTPI: Raspberry Pi + Ardunio Robot



- Raspberry Pi 2 (quad-core ARMv7 @ 900MHz) + Arduino Leonardo clone A-Star 32U4 Micro
- *Optical encoders* (custom), IR optical array, 3D accelerometers, gyros, and compass MinIMU-9 v2
- **IPTPI** is programmed in Python, Java and Go using: Wiring Pi, Numpy, Pandas, Scikit-learn, Pi4J, Reactor, RxJava, GPIO(Go)

IPTPI: Raspberry Pi + Ardunio Robot



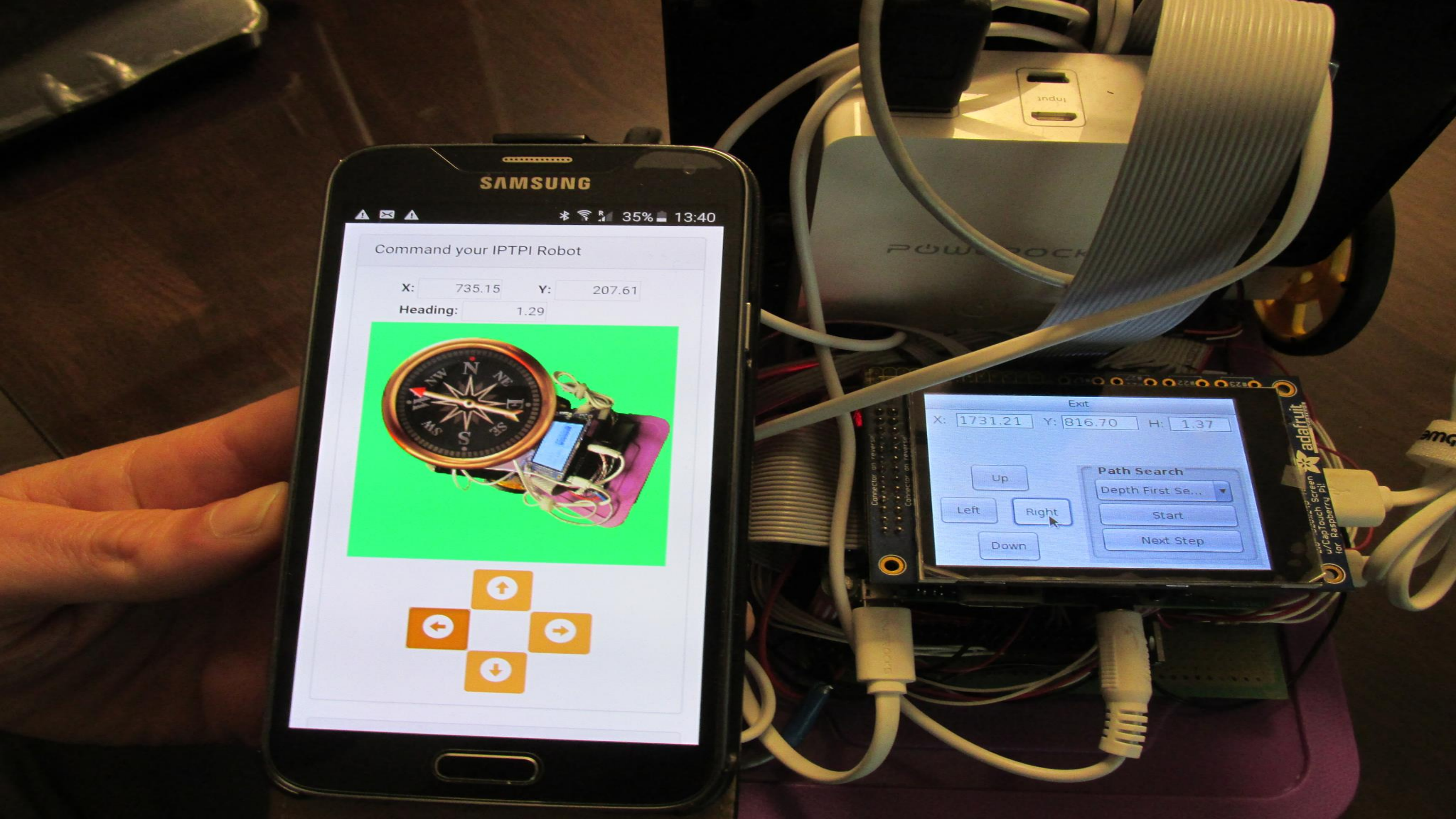
3D accelerometers, gyros, and compass [MinIMU-9 v2]

USB Stereo Speakers - 5V

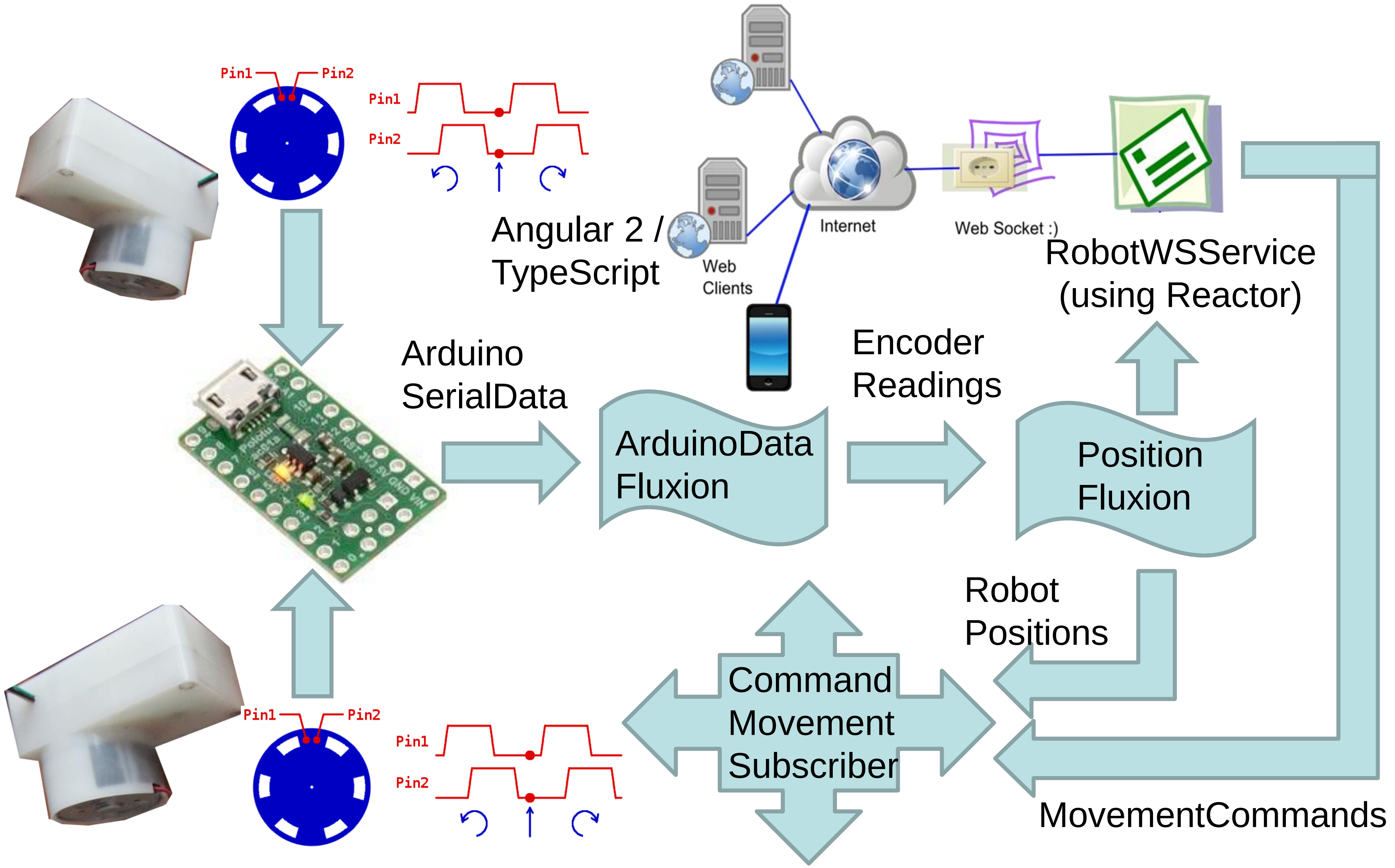
LiPo Powebank 15000 mAh

Pololu DRV8835 Dual Motor Driver for Raspberry Pi

Arduino Leonardo clone [A-Star 32U4 Micro]

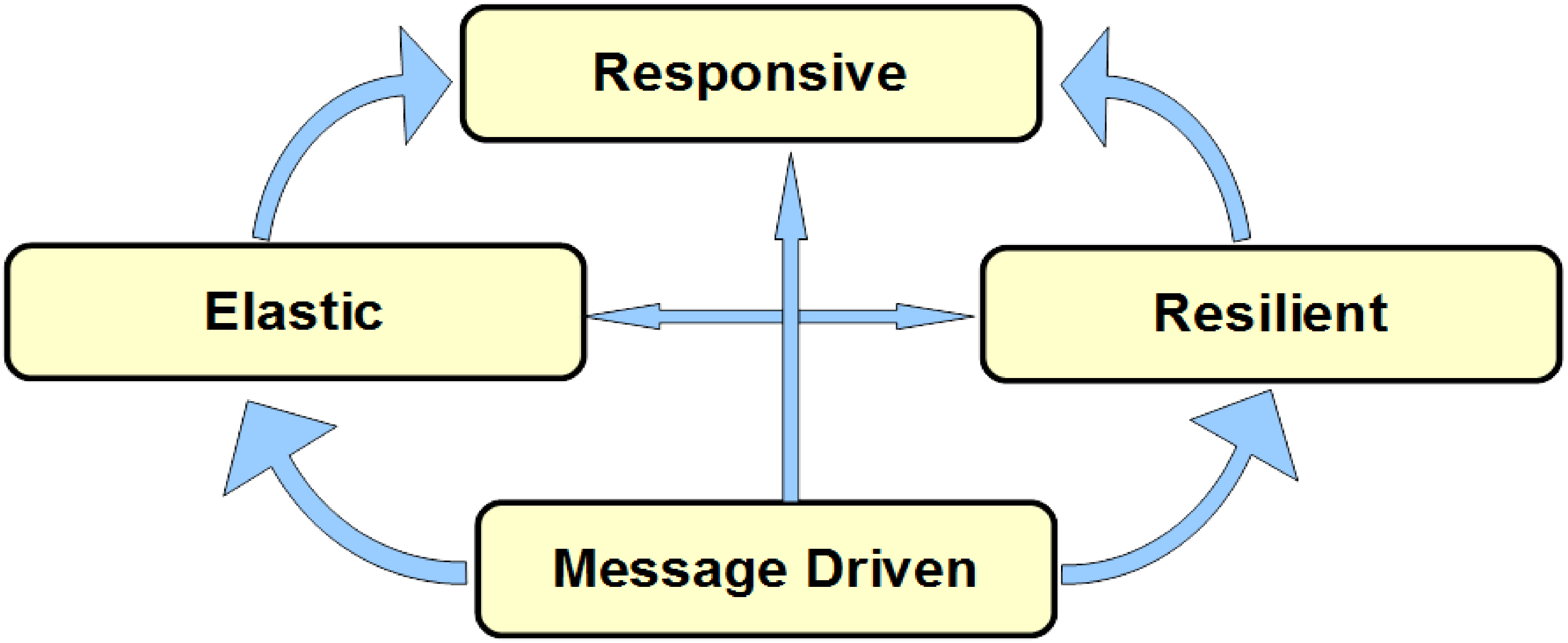

SAMSUNG
35% 13:40
Command your IPTPI Robot
X: 735.15
Y: 207.61
Heading: 1.29
Exit
X: 1731.21
Y: 816.70
H: 1.37
Up
Left
Right
Down
Path Search
Depth First Se...
Start
Next Step
adafruit

IPTPI Reactive Streams



Reactive Manifesto

http://www.reactivemanifesto.org



Scalable, Massively Concurrent

- **Message Driven** – asynchronous message-passing allows to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages [Reactive Manifesto].
- The main idea is to separate concurrent producer and consumer workers by using **message queues.**
- **Message queues** can be **unbounded** or **bounded** (limited max number of messages)
- **Unbounded** message queues can present memory allocation problem in case the producers outrun the consumers for a long period → **OutOfMemoryError**

Data / Event / Message Streams

“Conceptually, a stream is a (potentially never-ending) flow of data records, and a transformation is an operation that takes one or more streams as input, and produces one or more output streams as a result.”

Apache Flink: Dataflow Programming Model

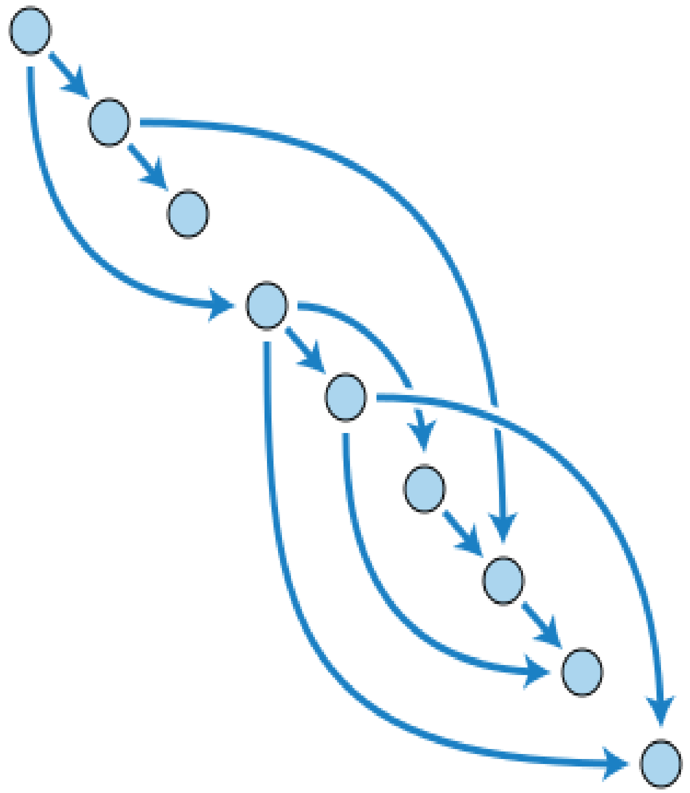
Data Stream Programming

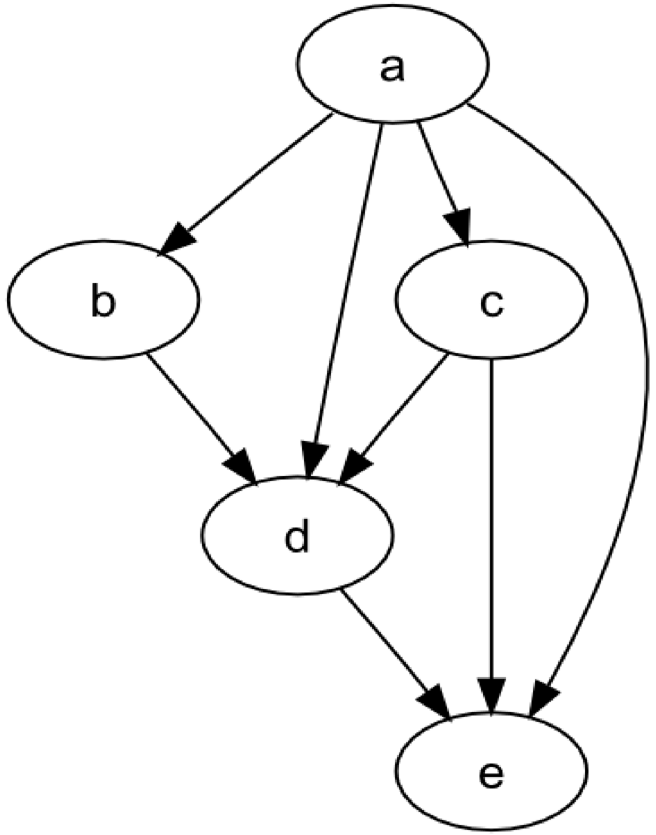
The idea of abstracting logic from execution is hardly new -- it was the dream of SOA. And the recent emergence of microservices and containers shows that the dream still lives on.

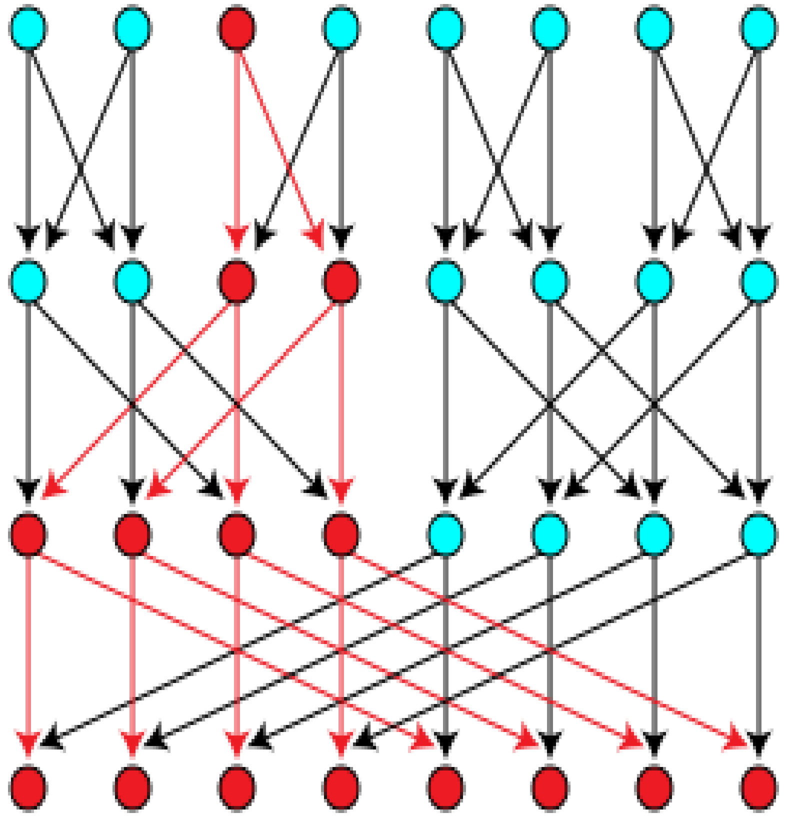
For developers, the question is whether they want to learn yet one more layer of abstraction to their coding. On one hand, there's the elusive promise of a common API to streaming engines that in theory should let you mix and match, or swap in and swap out.

Tony Baer (Ovum) @ ZDNet - Apache Beam and Spark: New coopetition for squashing the Lambda Architecture?

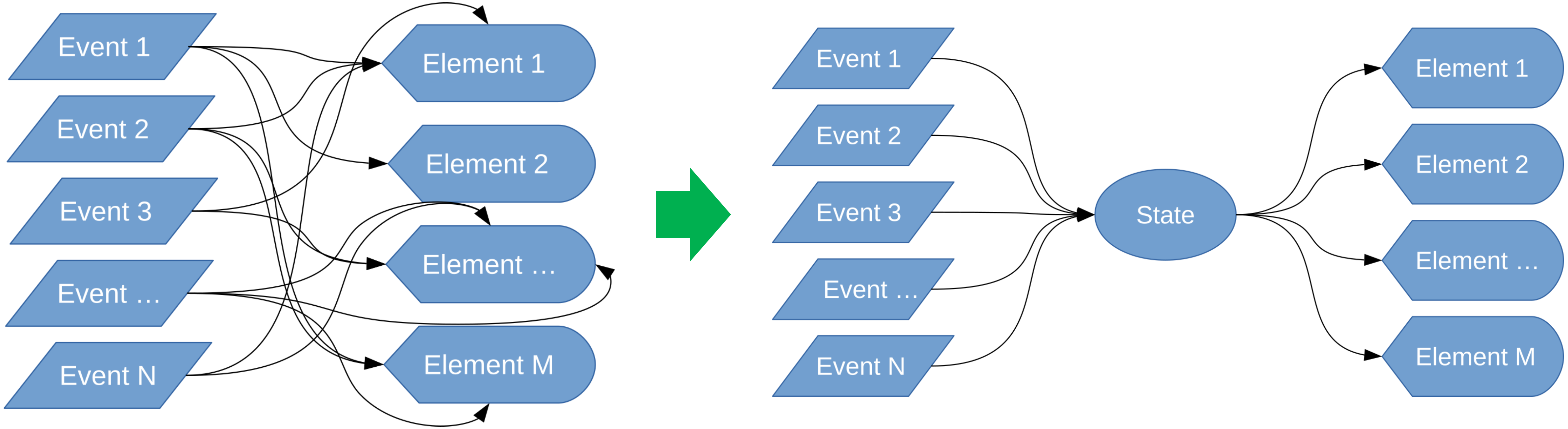
Direct Acyclic Graphs - DAG





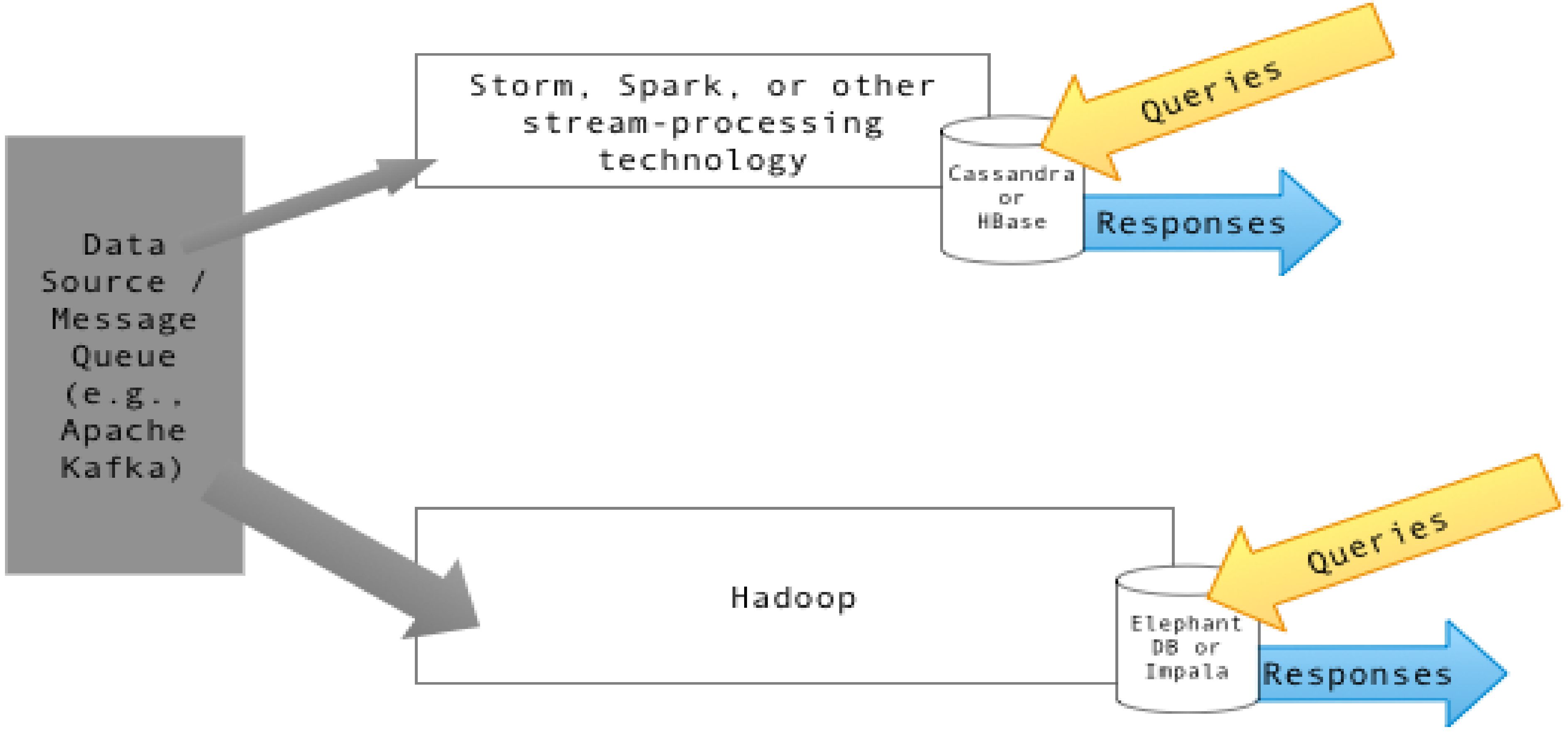


Event Sourcing – Events vs. Sate (Snapshots)



Lambda Architecture - I

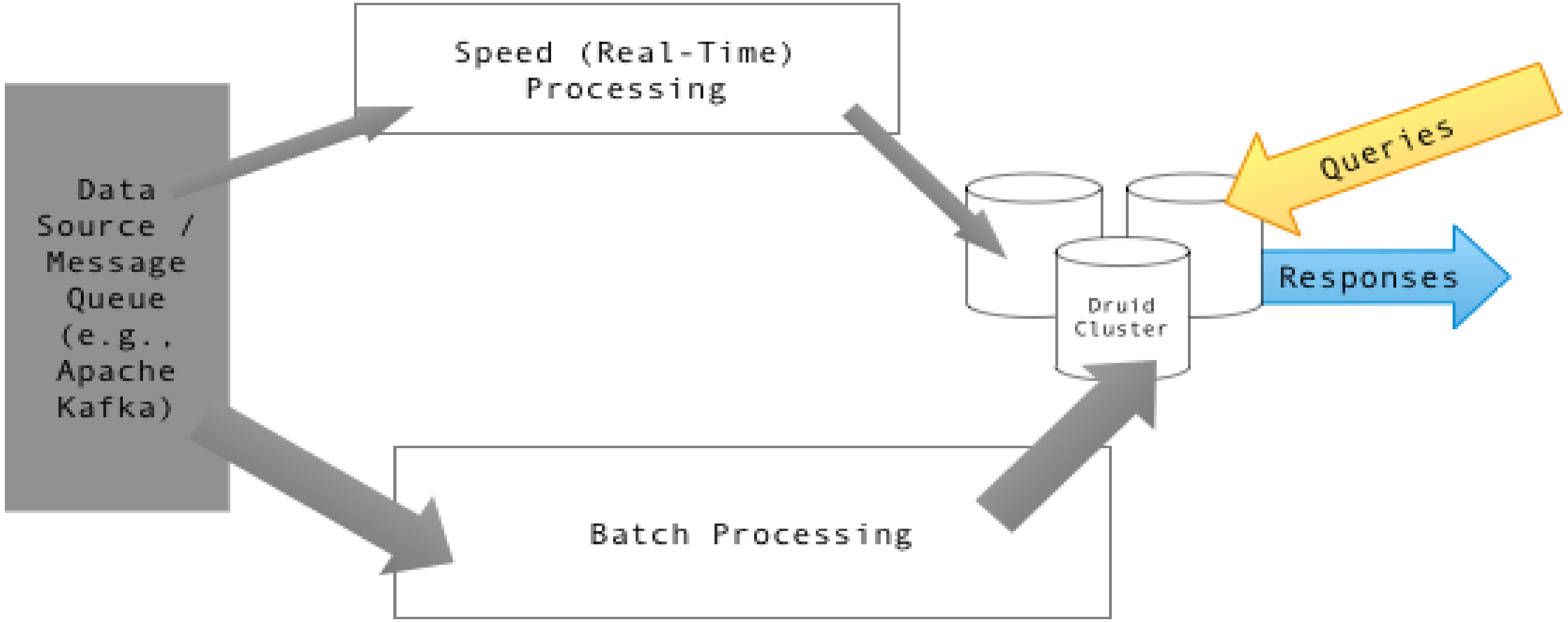
Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)


Storm, Spark, or other stream-processing technology
Cassandra or HBase
Queries
Responses
Data Source / Message Queue (e.g., Apache Kafka)
Hadoop
Elephant DB or Impala
Queries
Responses

https://commons.wikimedia.org/w/index.php?curid=34963987, By Textractor - Own work, CC BY-SA 4

Lambda Architecture - II

Query = λ (Complete data) = λ (live streaming data) * λ (Stored data)


Speed (Real-Time) Processing
Data Source / Message Queue (e.g., Apache Kafka)
Queries
Responses
Druid Cluster
Batch Processing

Kappa Architecture

Query = K (New Data) = K (Live streaming data)

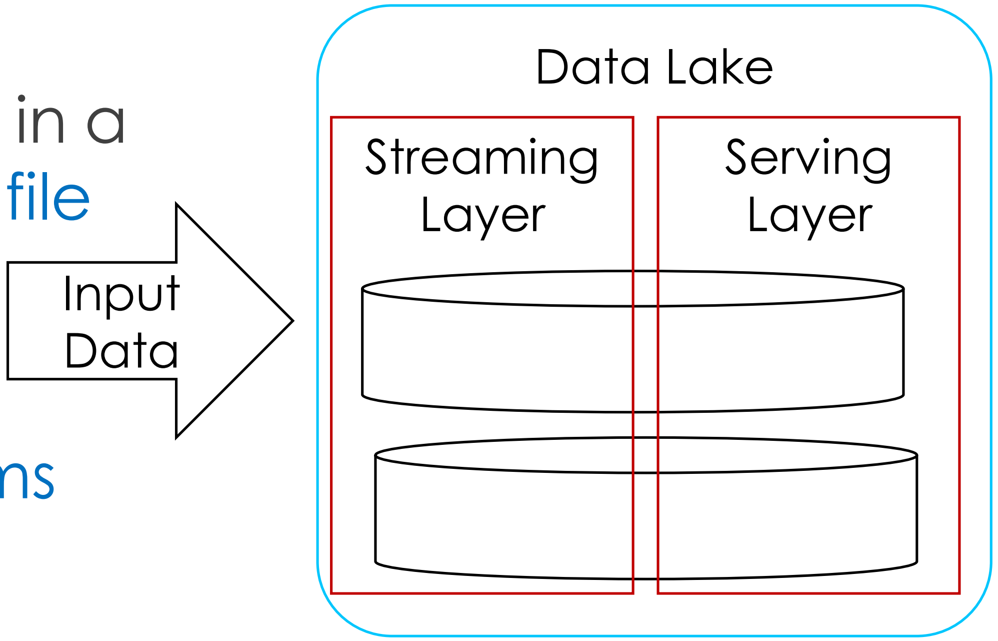
λ vs ϰ

- Proposed by Jay Kreps in 2014
- Real-time processing of distinct events
- Drawbacks of Lambda architecture:
  - It can result in coding overhead due to comprehensive processing
  - Re-processes every batch cycle which may not be always beneficial
  - Lambda architecture modeled data can be difficult to migrate
- Canonical data store in a Kappa Architecture system is an append-only immutable log (like Kafka, Pulsar)

Kappa Architecture II

Query = K (New Data) = K (Live streaming data)

- Multiple data events or queries are logged in a queue to be catered against a distributed file system storage or history.
- The order of the events and queries is not predetermined. Stream processing platforms can interact with database at any time.
- It is resilient and highly available as handling terabytes of storage is required for each node of the system to support replication.
- Machine learning is done on the real time basis



Zeta Architecture

- Main characteristics of Zeta architecture:
  - file system (HDFS, S3, GoogleFS),
  - realtime data storage (HBase, Spanner, BigTable),
  - modular processing model and platform (MapReduce, Spark, Drill, BigQuery),
  - containerization and deployment (cgroups, Docker, Kubernetes),
  - Software solution architecture (serverless computing – e.g. Amazon Lambda)
- Recommender systems and machine learning
- Business applications and dynamic global resource management (Mesos + Myriad, YARN, Diego, Borg).

Distributed Stream Processing – Apache Projects:

- Apache Spark is an open-source cluster-computing framework. Spark Streaming, Spark Mllib

- Apache Storm is a distributed stream processing – streams DAG

- Apache Samza is a distributed real-time stream processing framework.

Distributed Stream Processing – Apache Projects II

- Apache Flink - open source stream processing framework – Java, Scala
- Apache Kafka - open-source stream processing (Kafka Streams), real-time, low-latency, high-throughput, massively scalable pub/sub
- Apache Beam – unified batch and streaming, portable, extensible

What can FP offer to distributed computing ?

- No side-effects and immutable variables – FP facilitates code distribution over several CPU and eases concurrent programming
- Functions are better building components than objects:
  - Functions can be combined, sent remotely
  - Functions can be applied locally on distributed data sets (e.g. parallel stream, using Fork-Join pool underneath).
- In order to do the splitting of the work between multiple threads (forking) the Java Streams use:

  spliterator = split iterator
- The results can be joined after that in a single result (e.g. reduce)
- Example: Map – Reduce big data architecture (Google, Hadoop)

Iterators and Spliterators Example - Zip

```
public static <A, B, C> Stream<C> zip2(Stream<A> streamA, Stream<B> streamB, BiFunction<A, B, C> zipper) {
    Objects.requireNonNull(zipper);
    Spliterator<? extends A> aSpliterator = Objects.requireNonNull(streamA).spliterator();
    Spliterator<? extends B> bSpliterator = Objects.requireNonNull(streamB).spliterator();

    int characteristics = ((aSpliterator.characteristics() & bSpliterator.characteristics()
            & ~(Spliterator.DISTINCT | Spliterator.SORTED))    // Zipping looses DISTINCT and SORTED characteristics
            | (aSpliterator.characteristics() & SIZED | bSpliterator.characteristics() & SIZED));

    long zipSize = (aSpliterator.getExactSizeIfKnown() >= 0) ?
            ((bSpliterator.getExactSizeIfKnown() >= 0) ?
                    Math.min(aSpliterator.estimateSize(), bSpliterator.estimateSize())
                    : aSpliterator.estimateSize())
            :bSpliterator.getExactSizeIfKnown();

    final Iterator<A> iteratorA = Spliterators.iterator(aSpliterator);
    final Iterator<B> iteratorB = Spliterators.iterator(bSpliterator);
    final Iterator<C> iteratorC = new Iterator<C>() {
        @Override
        public boolean hasNext() {
            return iteratorA.hasNext() && iteratorB.hasNext();
        }

        @Override
        public C next() {
            return zipper.apply(iteratorA.next(), iteratorB.next());
        }
    };
    Spliterator<C> split = zipSize > 0 ? Spliterators.spliterator(iteratorC, zipSize, characteristics):
            Spliterators.spliteratorUnknownSize(iteratorC, characteristics);
    return StreamSupport.stream(split, streamA.isParallel() || streamA.isParallel());
}
```

Stream API

Поточно програмиране (1)

Примери:

```
books.stream().map(book ->
    book.getTitle()).collect(Collectors.toList());
books.stream()
       .filter(w -> w.getDomain() == PROGRAMMING)
       .mapToDouble(w -> w.getPrice()) .sum();
document.getPages().stream()
     .map(doc -> Documents.characterCount(doc))
     .collect(Collectors.toList());
document.getPages().stream()
     .map(p -> pagePrinter.printPage(p))
     .forEach(s -> output.append(s));
```

Източник: AdoptOpenJDK/lambda-tutorial (https://github.com/AdoptOpenJDK/lambda-tutorial) Автор: Graham Allan, Лиценз: GNU GENERAL PUBLIC LICENSE v2

Поточно програмиране (2)

Примери:

```
document.getPages().stream()
        .map(page -> page.getContent())
        .map(content -> translator.translate(content))
        .map(translated -> new Page(translated))
        .collect(Collectors.collectingAndThen(
                Collectors.toList(),
                pages -> new
        Document(translator.translate(document.getTitle()), pages)));
```

Exercise 1: Java 8 Stream API Koans – I part

Available @GitHub: https://github.com/iproduct/course-stream-api-2022/tree/main/lambda-tutorial

1. Read carefully the JavaDoc for the unit tests stating the problem to solve: Exercise_1_Test.java, Exercise_2_Test.java, Exercise_3_Test.java, Fill the code in place of comments like: // [your code here]

2. Run the unit tests to check if your proposed solution is correct. If not return to step 1.

Stream Creation

- Empty Stream - generic

```
Stream<User> streamEmpty = Stream.empty();
public Stream<User> createStream(Collection<User> c) {
   return c == null || c.isEmpty() ? Stream.empty() : c.stream();
}
```

- Stream from a Collection

```
Collection<String> collection = Arrays.asList(“hello”, “java”, “streams”);
Stream<String> stream = collection.stream();
```

- Stream from an Array

```
Stream<String> stream = Stream.of(“hello”, “java”, “streams”);
String[] array = {“hello”, “java”, “streams”};
Stream<String> streamOfArrayFull = Arrays.stream(array);
Stream<String> streamOfArrayPart = Arrays.stream(array, 0, 2);
```

Stream Creation II

- Stream.builder()

```
Stream<String> streamBuilder =
    Stream.<String>builder().add(“hello”).add(“java”).add(“streams”).build();
```

- Stream.generate() – using Supplier<T>

```
Stream<String> stream =Stream.generate(() -> “Java Stream API”).limit(5);
```

- Stream.iterate()

```
Stream<Integer> stream = Stream.iterate(1, n -> n + 2).limit(15);
```

Primitive Streams: IntStream, LongStream, DoubleStream

- range(), rangeClosed()

```
IntStream intStream = IntStream.range(1, 15);
LongStream longStream = LongStream.rangeClosed(1, 15);
```

- Random object methods

```
Random rand = new Random();
DoubleStream doubleStream = rand.doubles(15);
```

Streams from String / File

- Stream of String

```
IntStream streamOfChars = “Java Streams".chars();
```

- Stream of File lines

```
Path path = Paths.get("C:/src/main/java/StreamsIntro.java");
Stream<String> linesStream = Files.lines(path);
Stream<String> linesStreamUtf8 = Files.lines(path, Charset.forName(“utf-8"));
```

Stream Variables

```
Stream<String> stream =
 Stream.of(“hello”, “java”, “streams”).filter(element -> element.contains(“a"));
Optional<String> firstElementContainingA = stream.findFirst();
```

```
public static void tryStreamTraversal() {
    Stream<String> bookTitlesStream = bookTitles();
    bookTitlesStream.forEach(System.out::println);
    try {
        bookTitlesStream.forEach(System.out::println);
    } catch(IllegalStateException ex) {
        System.out.println("stream was traversed and closed");
    }
}
```

Stream -> Collection, List, Set, findFirst(), findAny()

```
List<String> allStringsContainingA =
    Stream.of(“hello”, “java”, “streams”).filter(str -> str.contains(“a"))
        .collect(Collectors.toList());

Optional<String> firstStringConatainingA =
    Stream.of(“hello”, “java”, “streams”).filter(str -> str.contains(“a")).findFirst();

Optional<String> anyStringConatainingA =
    Stream.of(“hello”, “java”, “streams”).filter(str -> str.contains(“a")).findAny();
```

Composing Stream Transformations

```
Stream<String> skippingFirstElement =  Stream.of(“hello”, “java”, “streams”)
    .skip(1);

long count = List.of(“hello”, “java”, “string”, “streams”).stream()
    .skip(1)
    .map(str -> str.substring(0, 2))
    .distinct()
    .count();
```

Exercise 2: Text File Keyword Extraction

Using the java.nio.file.Files.lines(path, StandardCharsets.UTF_8) method, implement following functionality:

1. Walk all the lines in the file, split them to words using non-word characters ("\\W+") as separators
2. Filter out the words with length less than 3 and stop words (given as a list of stop words).
3. Count the number of occurrences of each word, and sort words+counts descending on count number.
4. Pritn top 20 most used keyords in the text

Execution Order

```
AtomicInteger counter1 = new AtomicInteger();
var result = list.stream()
        .map(element -> {
            counter1.incrementAndGet();
            System.out.println("Calling map #" + counter1);
            return element.substring(0, 5);
        })
        .skip(2)
        .collect(Collectors.toList());
System.out.println(result);
```

```
Calling map #1

Calling map #2

Calling map #3

[hello]
```

```
AtomicInteger counter2 = new AtomicInteger();
var result2 = list.stream()
        .skip(2)
        .map(element -> {
            counter2.incrementAndGet();
            System.out.println("Calling map #" + counter2);
            return element.substring(0, 5);
        }).collect(Collectors.toList());
System.out.println(result2);
```

```
Calling map #1

[hello]
```

Primitive and Object Stream Reducers

- Primitive stream reducers: min(), max(), sum(), count()
- Object stream reducer (general form):
  - identity – accumulator initial value
  - accumulator – the reducer function - BiFunction:

    (ACC, VAL) -> NEW_ACC
  - combiner – combiner function combines different accumulator values computed in different threads (only used when streams are parallel)

Reducer Demo

```
OptionalInt reduced = IntStream.rangeClosed(1, 20).reduce((a, b) -> a + b);
System.out.println("Reduced: " + reduced);

int reducedWithInitVal = IntStream.rangeClosed(1, 20).reduce(1, (a, b) -> a * b);
System.out.println("Reduced with initial value: " + reducedWithInitVal);

int reducedParallel = IntStream.rangeClosed(1, 100000).boxed().parallel()
        .reduce(0,
                (a, b) -> a + b,
                (a, b) -> {
                    System.out.printf("combiner is called for %s and %s%n", a, b);
                    return a + b;
                });

System.out.println("Reduced with accumulator and combiner: " + reducedParallel);
```

Exercise3: Text File Line Numbering using Reducer

Using the java.nio.file.Files.lines(path, StandardCharsets.UTF_8) method, implement following functionality:

1. Walk all the lines in the file and number them consecutively using only:

reduce(identity, accumulator, combiner) method of Java Stream API

Reducer Demo

```
var path = Paths.get("src/course/stream/demos/StreamApiDemo04.java");
    String fResult = Files.lines(path)
        .reduce(new Tuple2<>("", 1), (acc, line) -> // accumulator
            new Tuple2<>(acc.getV1() + acc.getV2() + ": " + line + "\n", acc.getV2() + 1),
        (acc1, acc2) -> // combimner
            new Tuple2<>(acc1.getV1() + "\n" + acc2.getV2(), 0)).getV1();
    System.out.println(fResult);
```

Collectors API - I

- Collectors are reducers that reduce the stream values to a container or some other type of result, depending on your needs.
- The reduction of a stream is performed by the Collectors.collect() method, accepting an argument of the type Collector
- The Collector provides the actual data structure, and mechanism for the reduction operation implementation.
- The Collectors class provides many ready to use collectors out-of-the-box.

Collectors API - II

- Collector can be specified using following functions:
  - supplier() – creates the accumulating container for collection results
  - accumulator() – adding each value to the accumulating container in a single thread
  - combiner() – combines all containers accu mulated by different threads into single results container
  - finisher() – transforms the result container into other type of result (optional)

Collectors API - III

- Collectors also have a set of characteristics, such as Collector.Characteristics.CONCURRENT, that provide hints that can be used by a reduction implementation to provide better performance.
- A sequential implementation of a reduction using a collector would create a single result container using the supplier function, and invoke the accumulator function once for each input element.
- A parallel implementation would partition the input, create a result container for each partition, accumulate the contents of each partition into a subresult for that partition, and then use the combiner function to merge the subresults into a combined result.

Collectors API Constraints - I

- The first argument passed to the accumulator function, both arguments passed to the combiner function, and the argument passed to the finisher function must be the result of a previous invocation of the result supplier, accumulator, or combiner functions.
- The implementation should not do anything with the result of any of the result supplier, accumulator, or combiner functions other than to pass them again to the accumulator, combiner, or finisher functions, or return them to the caller of the reduction operation.
- If a result is passed to the combiner or finisher function, and the same object is not returned from that function, it is never used again.
- Once a result is passed to the combiner or finisher function, it is never passed to the accumulator function again.

Collectors API Constraints - II

- For non-concurrent collectors, any result returned from the result supplier, accumulator, or combiner functions must be serially thread-confined. This enables collection to occur in parallel without the Collector needing to implement any additional synchronization. The reduction implementation must manage that the input is properly partitioned, that partitions are processed in isolation, and combining happens only after accumulation is complete.

- For concurrent collectors, an implementation is free to (but not required to) implement reduction concurrently. A concurrent reduction is one where the accumulator function is called concurrently from multiple threads, using the same concurrently-modifiable result container, rather than keeping the result isolated during accumulation. A concurrent reduction should only be applied if the collector has the Collector.Characteristics.UNORDERED characteristics or if the originating data is unordered.

Collector Examples

- To Collection/List/Set:

```
List<String> userNames = userList.stream()
      .map(User::getName).collect(Collectors.toList());
```

- min() / max() / sum() / average():

```
double allBooksTotalPrice = booksList.stream()
     .collect(Collectors.summingInt(Book::getPrice));

double allBooksAveragePrice = booksList.stream()
     .collect(Collectors.averagingInt(Book::getPrice));
```

- To String (joining Strings):

```
String userNamesString = userList.stream().map(User::getName)
       .collect(Collectors.joining(", "));
```

Collector Examples

- Statistics about stream values:

```
IntSummaryStatistics booksStatistics = booksList.stream()
        .collect( Collectors.summarizingInt (Book::getPrice) );
```

- Classifying books in two partitions, depending on predicate:

```
Map<Boolean, List<Product>> cheapVsExpensiveBooks = booksList.stream()
  .collect(Collectors.partitioningBy(book -> book.getPrice() >= 50.0));
```

- Grouping using a grouping key extraction function

```
Map<String, List<Book>> booksGroupedByPublisher = booksList.stream()
       .collect(Collectors.groupingBy(Book::getPublisher));
```

Custom Collector

```
var treeSetCollector = Collector.of(
        TreeSet<Double>::new,  // supplier
        TreeSet<Double>::add, //accumulator
        (left, right) -> { left.addAll(right); return left; }, //combiner
        (TreeSet<Double> tsResult) -> tsResult.stream().map(d -> d.toString()) //finisher
                .collect(Collectors.joining(", ")));

var result = new Random().doubles(10).boxed().parallel()
        .collect(treeSetCollector);
System.out.println(result);
```

Custom Collector II   [Open JDK]

- Given a stream of Order, to accumulate the set of line items for each customer:

```
Map<String, Set<LineItem>> itemsByCustomerName
        = orders.stream().collect(
        groupingBy(Order::getCustomerName,
                flatMapping(order -> order.getLineItems().stream(),
                        toSet())));
```

- Given a stream of Employee, to accumulate the employees in each department that have a salary above a certain threshold

```
Map<Department, Set<Employee>> wellPaidEmployeesByDepartment
        = employees.stream().collect(
        groupingBy(Employee::getDepartment,
                filtering(e -> e.getSalary() > 2000,
                        toSet())));
```

Custom Collector III [Open JDK]

- Given a stream of Person, to calculate tallest person in each city:

```
Comparator<Person> byHeight = Comparator.comparing(Person::getHeight);
Map<City, Optional<Person>> tallestByCity
        = people.stream().collect(
        groupingBy(Person::getCity,
                reducing(BinaryOperator.maxBy(byHeight))));
```

- Ggiven stream of Person, calculate the longest last name of residents in each city:

```
Comparator<String> byLength = Comparator.comparing(String::length);
Map<City, String> longestLastNameByCity
        = people.stream().collect(
        groupingBy(Person::getCity,
                reducing("",
                        Person::getLastName,
                        BinaryOperator.maxBy(byLength))));
```

Collector Examples [Open JDK]

```
var results = Files.lines(path, StandardCharsets.UTF_8)
        .flatMap(line -> Arrays.stream(line.split("\\W+")))
        .map(String::toLowerCase)
        .filter(word -> word.length() > 2)
        .filter(not(STOP_WORDS::contains))
        .collect(Collectors.collectingAndThen(
                Collectors.groupingBy(Function.identity(), Collectors.counting()),
                (Map<String, Long> wordCounts) -> wordCounts.entrySet().stream()
                        .sorted(Map.Entry.<String, Long>comparingByValue().reversed())
                        .limit(20).collect(Collectors.toList())
        ));
```

Collector Examples – Line Numbers (Stateful Aggregation)

```
Files.lines(path)
        .collect(HashMap<Integer, String>::new, (map, line) -> map.put(map.size(), line),
Map::putAll) // Create a map of the index to the object
        .forEach((i, o) -> { // Now we can use a BiConsumer forEach!
            System.out.println(String.format("%d: %s", i+1, o));
        });
```

Using peek() for Side Effects

```
Stream.of("hello", "functional", "java", "streams", "api")
        .filter(s -> s.length() > 4)
        .peek(s -> System.out.println("After filter: " + s))
        .map(String::toUpperCase)
        .peek(s -> System.out.println("After map: " + s))
        .collect(Collectors.toList());


Stream<User> userStream =
                    Stream.of(new User("George"), new User("Hristo"), new User("Vesko"));

userStream.peek(u -> u.setName(u.getName().toUpperCase()))
        .forEach(System.out::println);
```

Parallel Streams and Side Effects [Open JDK]

- Laziness - intermediate operations are lazy (evaluated only when it is required) - do not start processing the contents of the stream until the terminal operation commences.
- Interference - lambda expressions in stream operations should not interfere. Interference occurs when the source of a stream is modified while a pipeline processes the stream – e.g. attempt to add a string to the source list of the stream throws a ConcurrentModificationException.
- Stateful lambda expressions - avoid using them as parameters in stream operations. A stateful lambda expression is one whose result depends on any state that might change during the execution of a pipeline. When a stream is executed in parallel, the map operation processes elements of the stream specified by the Java runtime and compiler, and can vary every time the code is run. For deterministic and predictable results, ensure that lambda expression parameters in stream operations are not statefull.

Exercise4: Walking + Filtering File Tree Using Streams

Using the java.nio.file.Files.walk(Path start, FileVisitOption... options) method, implement following functionality:

1. Walk the ./src directory of the project and print names of all files with .java extension recursively in all subdirectories.
2. Calculate the total number and size statistics (min / max / average size of java files, and the sum of sizes) of all java files in the project.

Exercise 5: Functional JavaDoc Processing

Using the java.nio.file.Files.walk(Path start, FileVisitOption... options) method, implement following functionality:

1. Walk the ./src directory of the project and process all files with .java extension recursively in all subdirectories.

2. For each file print filename and all comments starting with // (till the end of line)

3. *For each java file extract as text all JavaDoc comments (can span on multiple lines, and the comments are syntactically correct, starting with /** and ending with */)

4. *Print the filenames and extracted JavaDoc comments to the console

Method References, Default and Static Method in Interfaces

Method References

- Static method in a class – Class::staticMethod
- Methods of concrete object instances – object::instanceMethod
- Instance methods referred using the class – Class::instanceMethod
- Object constructors from given class – Class::new

```
Comparator<Person> namecomp = Comparator.comparing(Person::getName);

Arrays.stream(pageNumbers).map(doc::getPageContent).forEach(Printers::print);

pages.stream().map(Page::getContent).forEach(Printers::print);
```

Static and Default Methods in Interfaces

- Methods with default implementation in interfaces are also referred as virtual extension methods or defender methods, because they provide possibility to extend interfaces without breaking their existing clients (implementations).

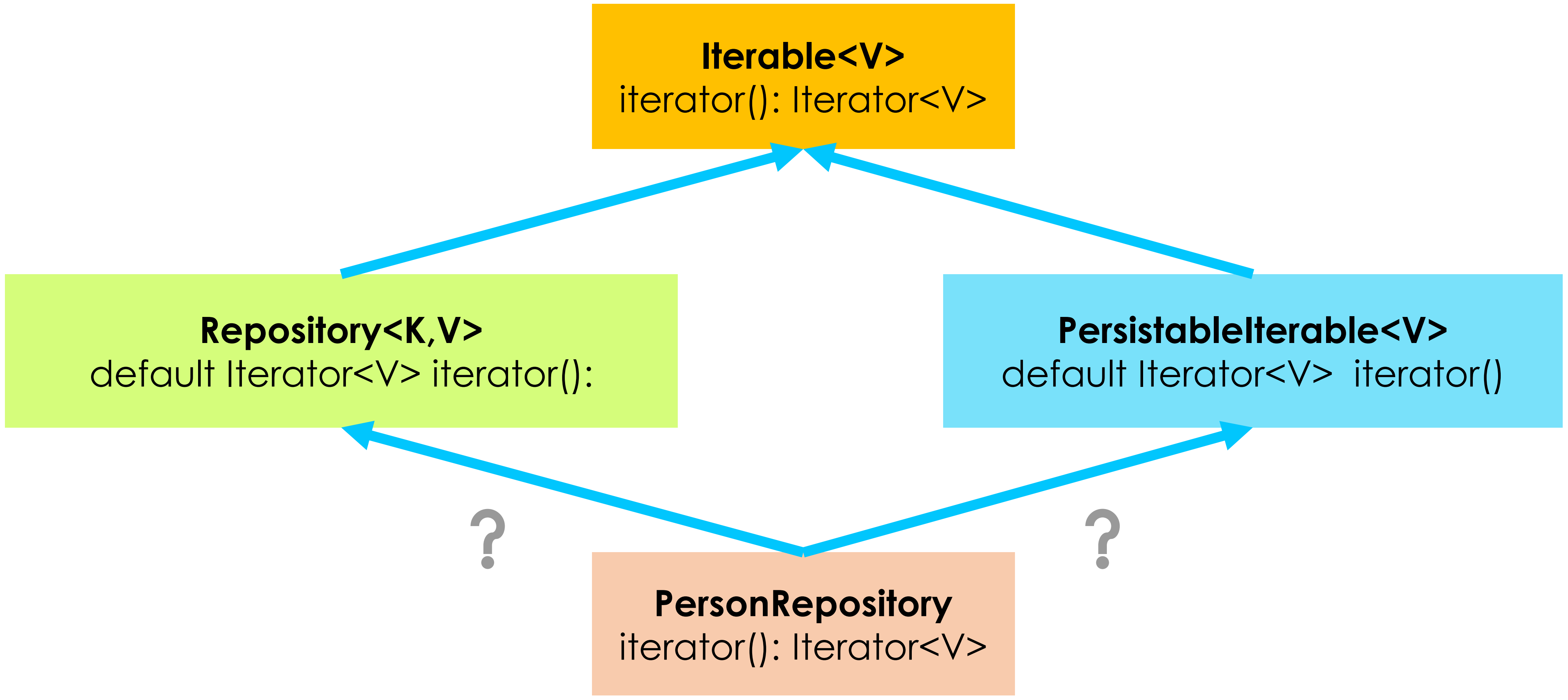
- Static methods provide ability to add helper (utility) methods – for example factory methods, directly in the corresponding interfaces, for which they produce objects from that interface type (e. g. Comparator interface).

Default and Static Interface Methods Example

- @FunctionalInterface

```
interface Event {
    Date getDate();
    default String getDateFormatted() {
            return String.format("%1$td.%1$tm.%1$tY", getDate());
    }
    public static <T, U extends Comparable<? super U>>
    Comparator<T> comparing(Function<T, U> getKey) {
        return (c1, c2) -> getKey.apply(c1).compareTo(getKey.apply(c2));
    }
}
Event current = () -> new Date();
System.out.println(current.getDateFormatted());
```

The Diamomd Problem



Rules when inheriting default implmentations

Following are the rules to follow when a class inherits a method with the same signature from multiple places (another class or interface):

- Classes always win. A method declaration in the class or a superclass takes priority over any default method declaration.
- Otherwise, sub-interfaces win: the method with the same signature in the most specific defaultproviding interface is selected. (for example in your case method from Second interface should run as Second extends First).
- Finally, if the choice is still ambiguous, the class inheriting from multiple interfaces has to explicitly select which default method implementation to use by overriding it and calling the desired method explicitly.

Exercise 6: Java 8 Stream API Koans – II part

Available @GitHub: https://github.com/iproduct/course-stream-api-2022/tree/main/lambda-tutorial

1. Read carefully the JavaDoc for the unit tests stating the problem to solve: Exercise_4_Test.java and Exercise_5_Test.java
2. Fill the code in place of comments like: // TODO [your code here]
3. Run the unit tests to check if your proposed solution is correct. If not return to step 1.

Functional Programming and Monads

- Concept of **monad** in functional programming (Categories theory)
- The **monad** formally is a set of three elements:

– Parameterized Type **M<T>**

– “**unit**” function: **T -> M<T>**

– “**bind**” operation: **bind(M<T>, f:T -> M<U>) -> M<U>**

- Example in Java 8 – the type: **java.util.Optional<T>**

Parameterized type: **Optional<T>**

– “**unit**” functions: **Optional<T> of(T value) , Optional<T> ofNullable(T value)**

– “**bind**” operation: **Optional<U> flatMap(Function<? super T,Optional<U>> mapper)**

Recommendations

Parallel Streams Usage Recommendations

- In certain use cases, when we process big amount of data and/or apply computationally intensive processing operations, the parallel streams can bring performance benefits. But they can also slow down the processing in other cases. So we should use sequential streams as a default, only converting them to parallel, if actual performance requirements arise for that.

- Given the use case requirements a sequential stream can be converted to a parallel after careful performance profiling, and identifying the need for stream parallelization.

- Parallel streams incur multiple overheads such as: multiple threads creation and management, splitting the stream in parts, merging the results, dereferencing cost and lost memory locality when using objects containers.

- Arrays of primitives bring best locality in Java, and they can be split fast into even ranges. Fetching multiple references in parallel can slow down the data processing.

Литература и интернет ресурси

- Oracle tutorial – lambda expressions - http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- Java SE 8: Lambda Quick Start - http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
- OpenJDK Lambda Tutorial - https://github.com/AdoptOpenJDK/lambda-tutorial

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD