Reactive Streams
(Akka Streams)

[http://doc.akka.io/docs/akka/current/scala/stream]

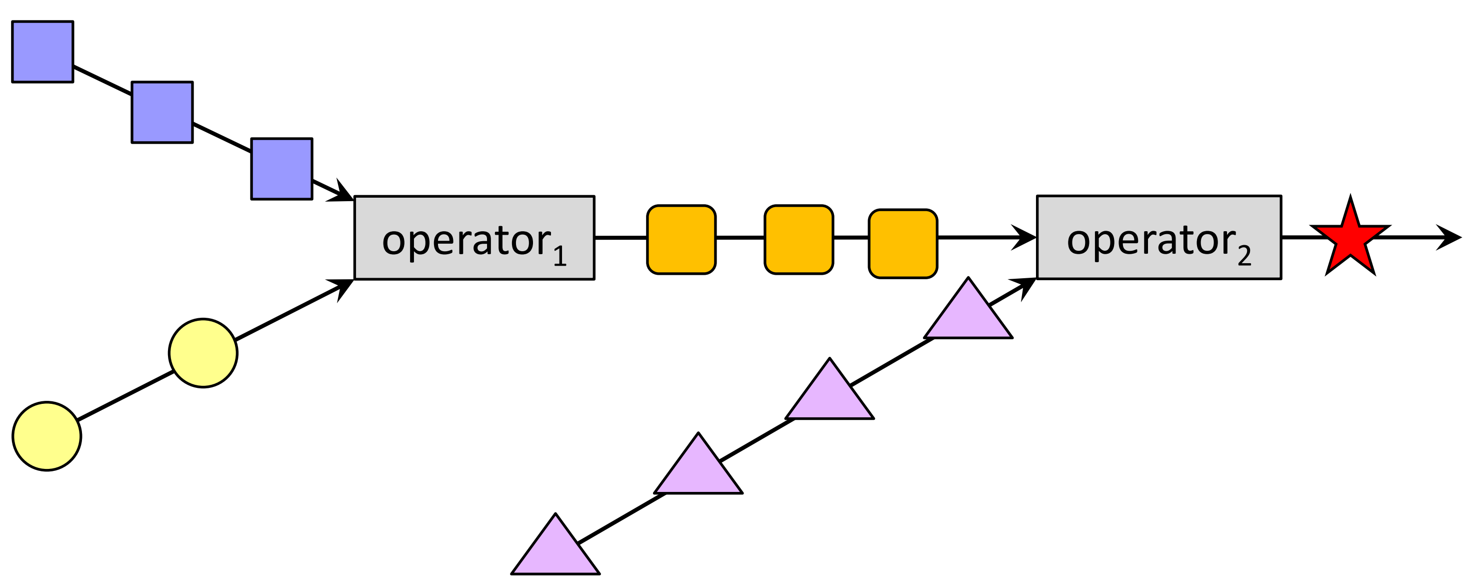
[Roestenburg et al., 2017; p. 171 ff.]

Reactive Streams

- **Streams** are sequences of data, where items can be continuously processed before the sequence ends.
- **Stream processing** is a programming style where the program logic is modeled in terms of composable streams.
- The primary goal of **Reactive Streams** is
  - to manage the flow of stream data **across asynchronous boundaries**
  - ensuring that the receiving side can handle the data without being overwhelmed
    → **backpressure**.
- The **Reactive Streams Specification** aims to standardize stream processing.
- Reactive Streams **implementations**:
  - Akka Streams
  - RxJava (ReactiveX)
  - RxJS (ReactiveX)
  - Project Reactor (Spring)
  - MongoDB, Reactive Rabbit, …

Basic Principle

- The program is driven by different event streams: user input, IoT device stream, stream of stock quotes, etc.
- Event streams can be combined using different *operators* (high-order functions).
- Events can be *processed synchronously* and *asynchronously*.

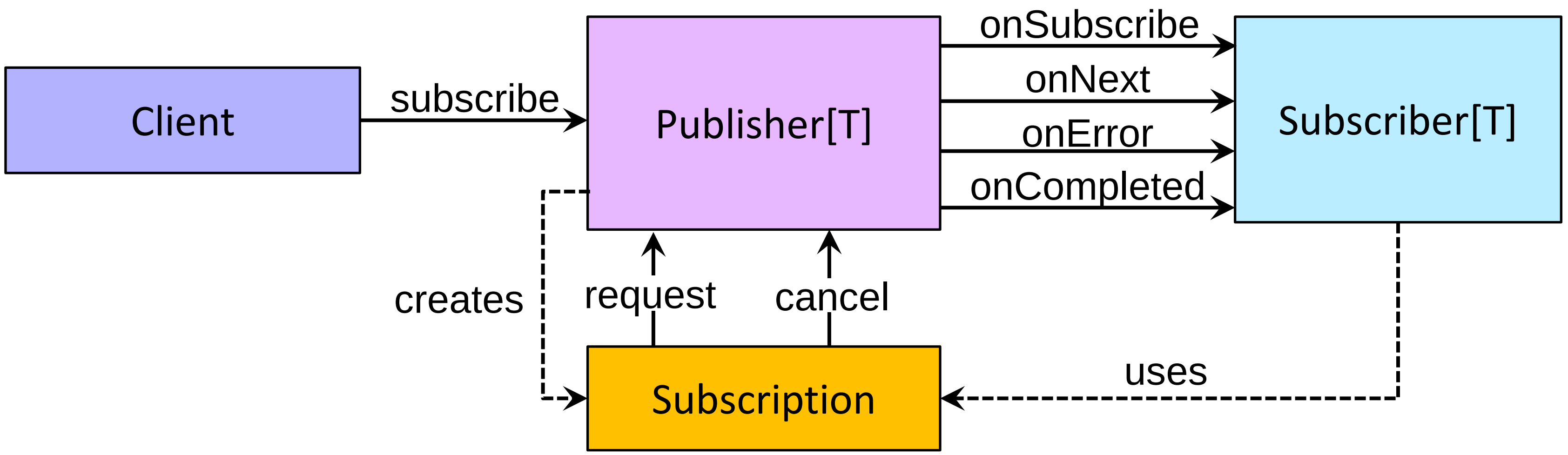


Reactive Streams Specification

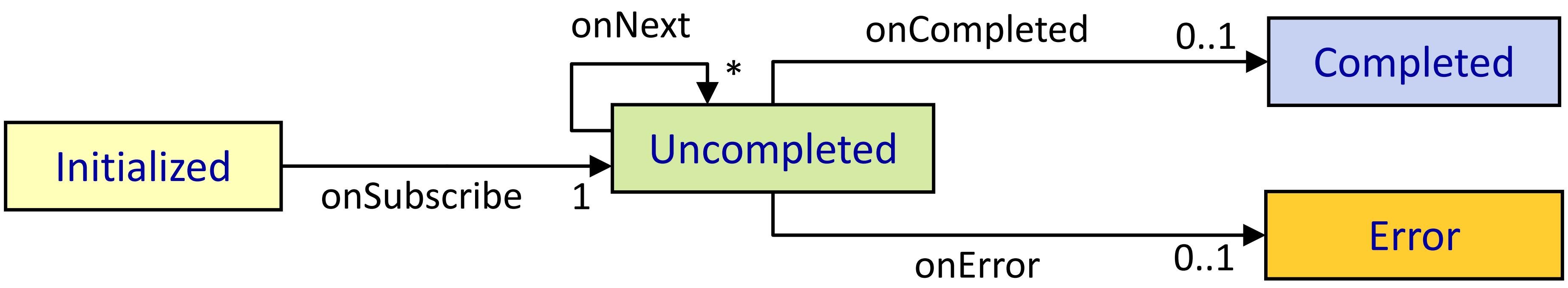
- The Reactive Streams specification provides
  - a set of **JVM interfaces (API)** that describe the necessary operations
  - the **basic semantics** of the operations (transmission of elements, handling of backpressure, etc.)
  - a **TCK** (Technology Compatibility Kit): test suite that checks if an implementation conforms to the specification.
- Design principles:
  - **Asynchronous operations**: Ensuring data streams can be processed asynchronously, allowing for efficient use of resources across different threads.
  - **Handling of backpressure**: Fast producers should not lead to inefficient resource consumption.
- Scope:
  - Streams API defines how data is passed between nodes internally.
  - Implementers can provide a totally different end-user API.

Basic Concept of Stream Processing

- **Publishers** provide a potentially unbounded number of elements. The elements are published according to the demand of a subscriber.
- Every time the publisher produces an *event*, its **Subscriber** gets notified.
  - `onSubscribe`: First event. Provides **Subscription** object.
  - `onNext`: Regular event of type T. Called for each event in the stream.
  - `onError`: Observable throws an exception. Called once. No more events are produced after an error occurred.
  - `onCompleted`: Fired after the last (regular) event.



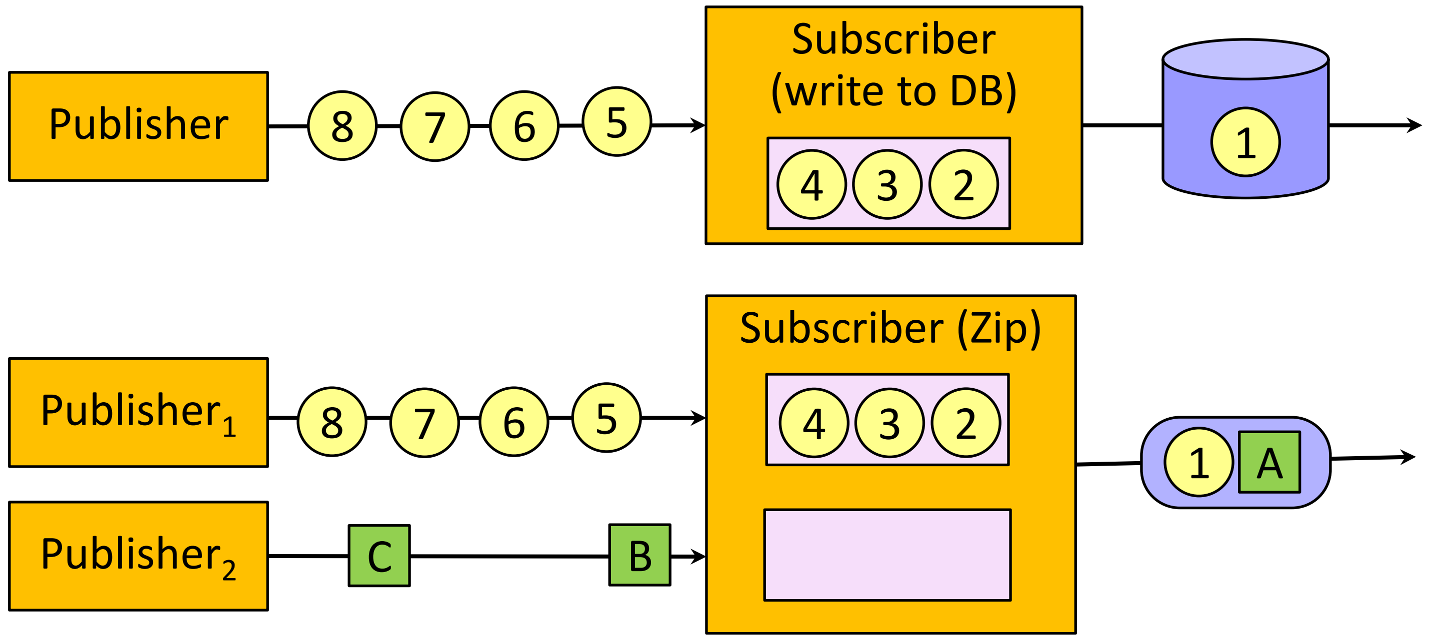
Semantics of Reactive Streams



- The total number of *onNext* notifications must be less than or equal to the total number of requested elements.
- Once a terminal state is signaled (*onComplete, onError*) it is required that no further signals occur.
- All event methods must be signaled in a thread-safe manner.
- A subscriber must signal demand via *request* to receive *onNext* signals.
- A subscriber must be prepared to receive *onComplete/onError* events without a preceding *request* call.
- And many other rules.

Backpressure: Problem

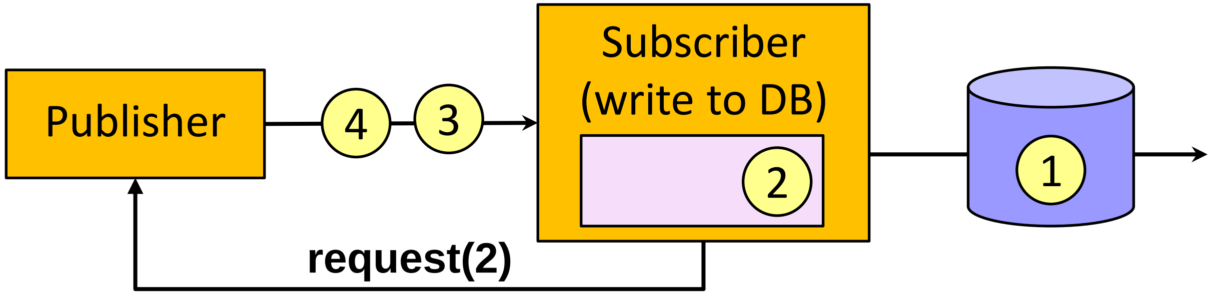
- Often publishers produce items faster than subscribers can consume.



- Subscribers can buffer items to some extent.
  - Results in high memory usage.
  - My even lead to memory exhaustions → program crashes.

Backpressure: Solution

- The subscriber repeatedly requests the number of items it may process (or buffer).
  - Requesting items is also an asynchronous operation.
- The publisher must not send more items then requested in total.



- The publisher can handle backpressure in one of the following ways:
  - Forward backpressure to publisher it is subscribed to.
  - Not generate elements, if possible.
  - Buffer elements.
  - Drop elements.
  - Tear down the stream if no other strategy can be applied.

Reactive Streams API

```
trait Publisher[+T]:
  def subscribe(subscriber: Subscriber[T]): Unit
```

```
trait Subscriber[T]:
  def onNext(value: T): Unit
  def onError(error: Throwable): Unit
  def onCompleted(): Unit
  def onSubscribe(subscription: Subscription): Unit
```

```
trait Subscription:
  def request(n: Long): Unit
  def cancel(): Unit
```

```
abstract class Processor[T,R] extends Subscriber[T] with Publisher
```

Duality of Collections, Futures, and Publishers

- Publishers allow to process sequences of data, like collections.
- Publishers allow to process data asynchronously, like futures.
  But futures are restricted to a single item.

	single item	multiple items
synchronous	`getData: T`	`getData: Iterable[T]`
asynchronous	`getData: Future[T]`	`getData: Publisher[T]`

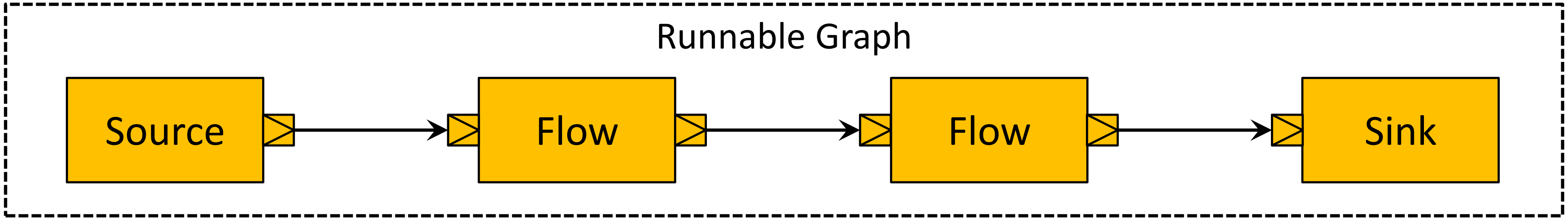
- Collections (iterables) and publishers share the same mechanisms to iterate through sequences of data.
- Clients *pull* data out of *iterables*, *publishers push* data to their clients.

Action/Event	Iterable	Publisher
get item	`T next()`	`onNext(T)`
Exception	`throws Exception`	`onError(Exception)`
complete	`! hasNext()`	`onCompleted()`

Akka Streams: Basic Concepts

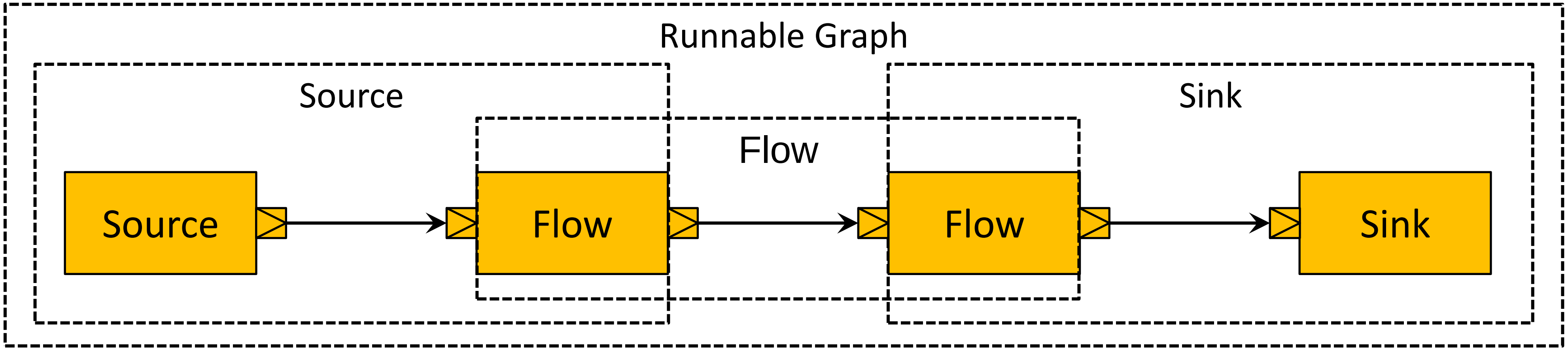
- Akka Streams implement the *Reactive Streams specification*.
- The Akka Streams API is *decoupled* from the Reactive Streams API.
  - Akka Streams are end-user-oriented. Focused on stream transformations.
  - Reactive Streams define how data can be moved across asynchronous boundaries.
- Akka Streams are implemented using Akka Actors.
- Differences between Akka Streams and Actors:
  - Streams handle backpressure automatically.
  - Streams cannot be distributed.

Terminology (1)



- **Processing Stage**: Building block in the graph (operations, …)
  - **Source**: Processing stage with *no input* and *one output.* Generates elements.
  - **Flow**: Processing stage with *one input* and *one output.* Transforms elements.
  - **Sink**: Processing stage with *one input* and *no output.* Processes elements (with side-effects).
- **Graph**: Description of the topology of the stream.
  - **Runnable Graph**: Graph with no input and output ports.
- **Element**: Processing unit of streams.

Terminology (2)



- Procession Stages can be *composed* to new processing stages.
  - Source/Flow → Source, Flow/Flow → Flow, Flow/Sink → Sink.
  - Enables structuring of programs.
- **Materialization** is the process of allocating all resources to be able to run the computation described by the graph.
  - Graphs are a description of the processing pipeline (immutable, thread-safe).
  - Materialization enables the actual processing of elements.
  - In Akka Streams processing stages are mapped to actors.

Sources

- Create a source that emits one element:

```
val s1: Source[String, NotUsed] = Source.single("one")          // "one"
```

- Convert a `collection` to a source:

```
val s2: Source[Int, NotUsed] = Source(List(1, 2, 3))            // 1, 2, 3
```

- Convert an iterator to a source:

```
val s3: Source[Int, NotUsed] = Source.fromIterator(
  () => Iterator.from(1))                                       // 1, 2, 3, …
```

- Emit elements periodically:

```
val s4: Source[String, Cancellable] =
  Source.tick(0.second, 500.millis, "x")                        // "x", "x", …
```

- Convert a `Future` object to a source:

```
val s5: Source[Int, NotUsed] =
  Source.fromFuture(Future.successful("future"))                // "future"
```

Sinks

- Execute a given function for each received element:

```
val s1: Sink[Int, Future[Done]] = Sink.foreach[Int](println(_))
```

- Return the first received element as a future:

```
val s2: Sink[Int, Future[Int]] = Sink.head[Int]            // 1,2,3 -> 1
```

- Fold the received elements using a given function:

```
val s3: Sink[Int, Future[Int]] = Sink.fold(0)(_ + _)       // 1,2,3 -> 6
```

- Insert all received elements into a collection:

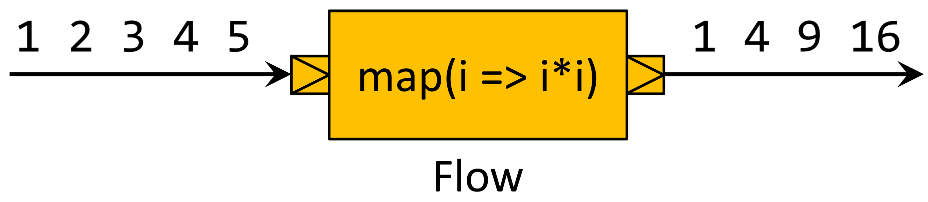
```
val s4: Sink[Int, Future[immutable.Seq[Int]]] = Sink.seq[Int]
                                                           // 1,2,3 -> Seq(1,2,3)
```

- Discard all received elements:

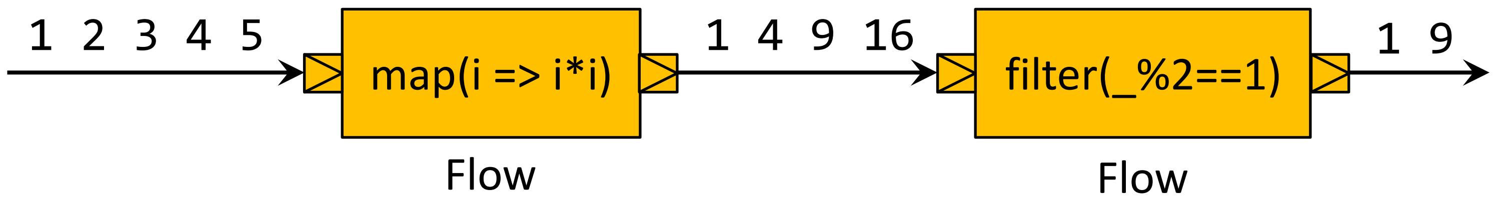
```
val s3 = Sink.ignore
```

Flows

- Reactive streams unfolds its full power with *operators* that allow to transform, filter, or combine processing stages.
- In this way the relationship between processing stages is *described declaratively*.
- In Akka Streams these operators are called flows.
- Flows can (but need not) be executed *asynchronously*.



- Flows can be composed, resulting in new flows.
  - This is the basic structuring element in stream-based programs.



Creating Flows

- Apply a function to all received elements and emit the function value:

```
val flow1: Flow[Int, Double, NotUsed] = Flow[Int].map(_*2.0)
                                                         // 1,2,3 -> 2.0, 4.0, 6.0
```

- Only pass on those elements that satisfy a given predicate:

```
val flow2: Flow[Int, Int, NotUsed] = Flow[Int].filter(_%2==0)
                                                         // 1,2,3,4 -> 2,4
```

- Map input elements to a collection and flatten result:

```
val flow3 = Flow[Int].mapConcat(i => 10*i to 10*i+1)
                                                         // 1,2,3 -> 10,11,20,21,30,31
```

- Map each input element to a source and pass on elements emitted by sources:

```
val flow4 = Flow[Int].flatMapConcat(i => Source(10*i to 10*i+1))
                                                         // 1,2,3 -> 10,11,20,21,30,31
```

- Emit input elements with speed limited to elements per time unit. Put elements into a bounded buffer. When buffer is full, signal backpressure (`Shaping`) or fail (`Enforce`).

```
val flow5 = Flow[Int].throttle(2, 1.second, 5, ThrottleMode.Shaping)
```

Combing Processing Stages

- Sources, flows and sinks can be composed using the methods `via` and `to`.
  - source **via** flow → source
  - flow **via** flow → flow
  - flow **to** sink → sink
  - source **to** sink → runnable graph

```
val source: Source[Double, NotUsed] = source.via(Flow[Int].map(_*2.0))
```

```
val flow1: Flow[Int, Int, NotUsed] = …
val flow2: Flow[Int, Int, NotUsed] = flow1.via(Flow[Int].mapConcat(i => 10*i to 10*i+1))
```

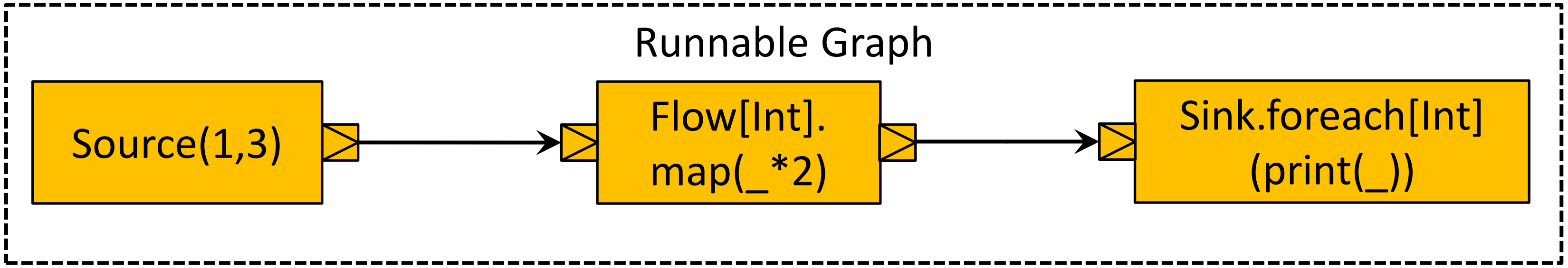
- Sources and flows offer *convenience methods* for this purpose:

```
val source: Source[Double, NotUsed] = source.map(_*2.0)
```

```
Val flow1: Flow[Int, Int, NotUsed] = …
val flow2: Flow[Int, Int, NotUsed] = flow1.mapConcat(i => 10*i to 10*i+1)
```

RunnableGraph

- A graph with no open input and output ports is called a **runnable graph**.
  - It is a blueprint for a stream, ready to be executed.




```
val source = Source(1 to 10)
val flow   = Flow[Int].map(_ * 2)
val sink   = Sink.foreach[Int](i => print(s"$i "))
val graph: RunnableGraph[NotUsed] = source.via(flow).to(sink)
```

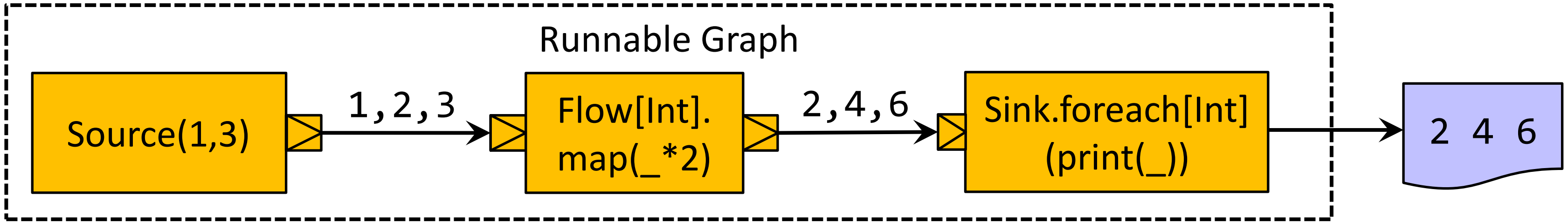
Materialization

- Materialization is the process of *allocating all resources* needed to execute a runnable graph.
- In Akka Streams an actor system executes a materialized stream.
- Materialization and execution is initiated by method `run`:

```
final case class RunnableGraph[+Mat] extends Graph[ClosedShape, Mat]:
  def run()(using materializer: Materializer)
```

- Usage:

```
given ActorSystem = ActorSystem(Root(), "MyActorSystem")

val graph: RunnableGraph[NotUsed] = source.via(flow).to(sink)
graph.run() // ActorSystem is implicitly converted to a Materializer
```



Materialized Values (1)

- Each processing stage can produce a materialized value.
  - The type of the materialized value is the 2nd type parameter of the processing stage.

```
val source: Source[Int, NotUsed]    = Source(1 to 10)
val sink:   Sink[Int, Future[Int]] = Sink.fold[Int, Int](0)(_ + _)
```

- Passing materialized values through streams
  - When combining two stages with `to`/`via`, the materialized value of the left stage is preserved:

```
val graph1: RunnableGraph[NotUsed] = source.to(sink)
```

  - With `toMat`/`viaMat` the value that is passed on can be computed using a function that maps *(leftValue, rightValue)* to *newValue*.

```
val graph2: RunnableGraph[NotUsed]      = source.toMat(sink)(Keep.left)
val graph3: RunnableGraph[Future[Int]] = source.toMat(sink)(Keep.right)
val graph4: RunnableGraph[(NotUsed, Future[Int])] =
                                          source.toMat(sink)(Keep.both)
val graph5: RunnableGraph[Future[Int]] =
              source.toMat(sink)((l, r) => r) // same as Keep.right
```

Materialized Values (2)

- Accessing the materialized value
  - `runnableGraph.run()` returns the materialized value:

```
val graph: RunnableGraph[Future[Int]] = source.toMat(sink)(Keep.right)
val sumFuture: Future[Int] = graph.run()
sumFuture.foreach(sum => println(s"sum=${sum}"))
```

  - `source.runWith(sink)`
    - runs the source with the given sink and
    - returns the materialized right value of the sink

```
val sumFuture: Future[Int] = source.runWith(sink)
sumFuture.foreach(sum => println(s"sum=${sum}"))
```

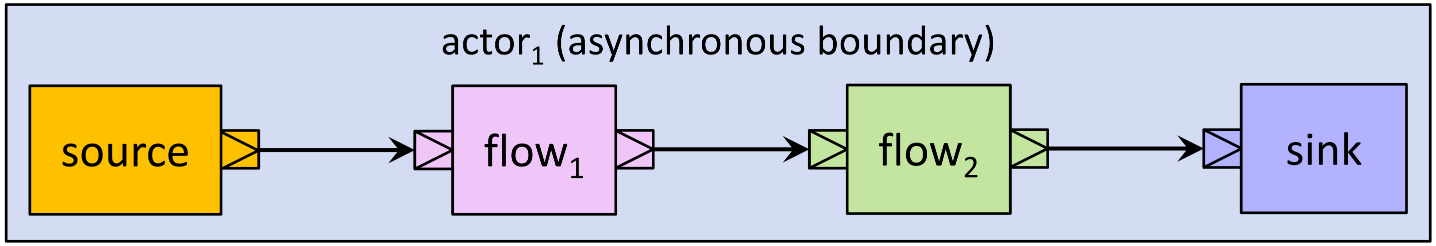
  - There are also convenience methods for creating sinks out of functions and executing the graph: `runForeach`, `runFold`, etc.

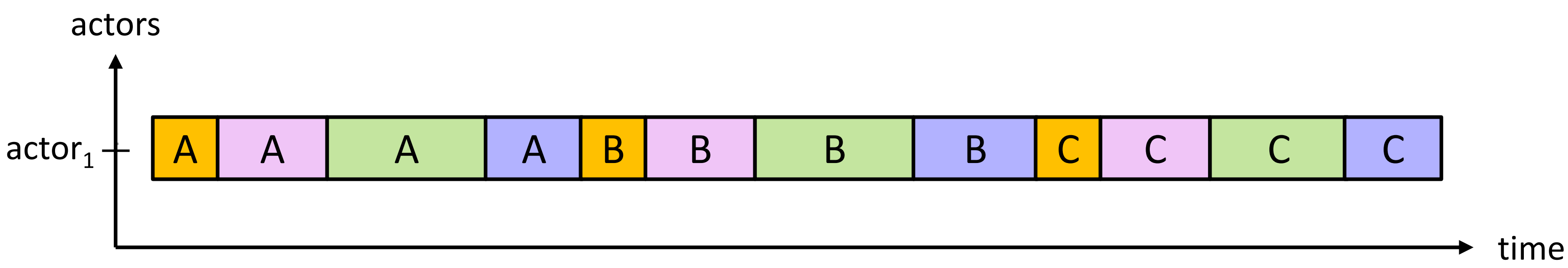
```
val sumFuture: Future[Int] = source.runFold(0)(_ + _)
```

Operator Fusion

- By default, all processing stages are executed on the same actor → **operator fusion**.
  - Passing elements from one stage to the next is faster.
  - Fused stages does not run in parallel.

```
source.via(flow1).via(flow2).runWith(sink)
```

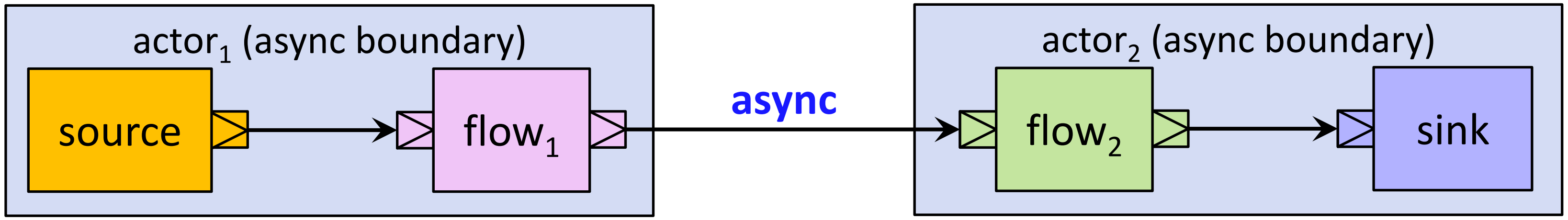


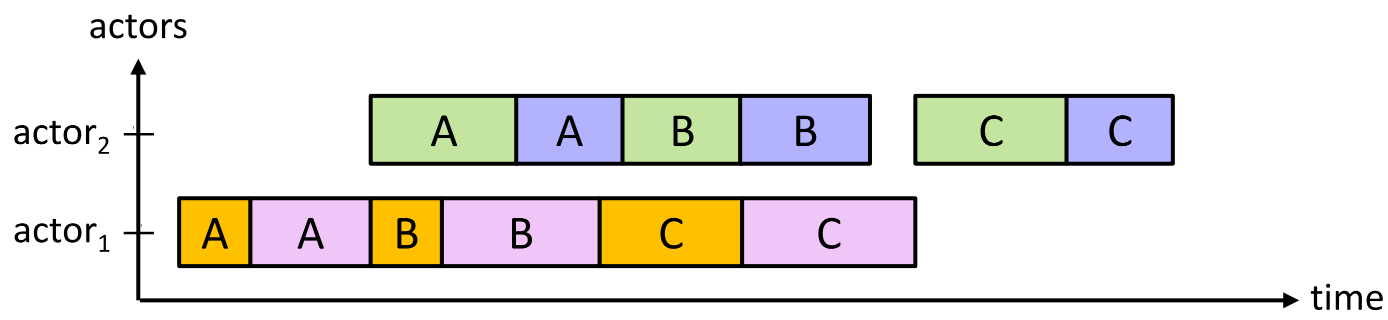


Parallel Execution of Streams

- To allow parallel processing asynchronous boundaries have to be inserted into the stages using the method `async`.
  - Stages marked as asynchronous demarcate execution units.
  - Stages in the same execution unit are fused.
  - Stages in different execution units may be executed asynchronously.

```
source.via(flow1).async.via(flow2).runWith(sink)
```





Parallel Execution with `mapAsync`

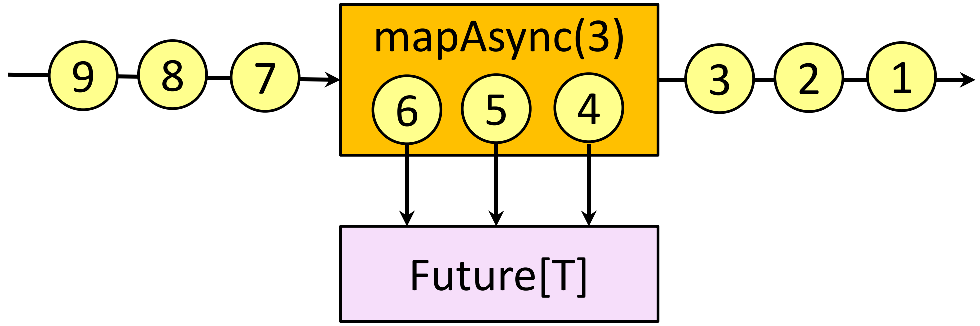
- With `async` groups of processing stages are assigned to different actors.

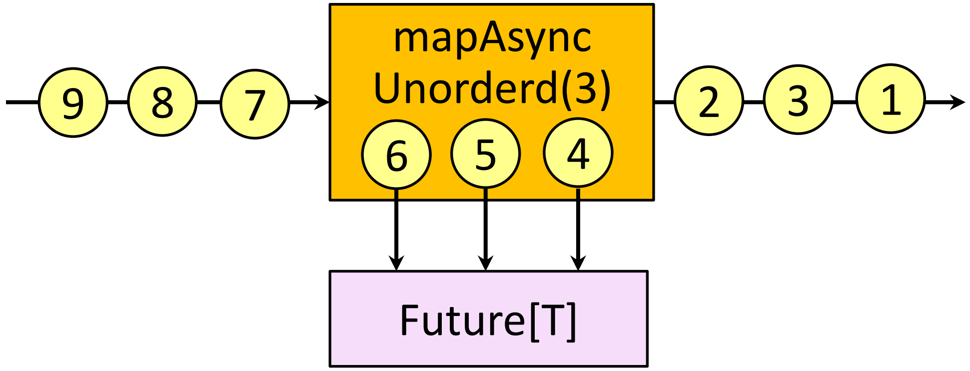
```
source.map(expensiveTask1).async.map(expensiveTask2).withSink(sink)
```

- By using `mapAsync` one has more control over the level of parallelism:

```
source
 .mapAsync(3)(param => Future { expensiveTask1(param) })
 .mapAsync(3)(param => Future { expensiveTask2(param) })
 .runWith(sink)
```

  - Backpressures when the number of futures reaches the configured parallelism (first parameter) or the downstream backpressures.





Error Handling

- When an exception is thrown in a stage the entire stream is shut down.

```
val errorSource = Source(1 to 6).map(n =>
  if (n < 4) n.toString
  else throw new RuntimeException("error in source")
)

errorSource.runForeach(print(s"$i "))                     // 1 2 3
```

- This can be avoided by implementing one of the following error handling strategies:
  a) Recover: Emit a final element and then complete.
  b) Recover with retries: Replace stream by a new one.
  c) Repeatedly restart stream.
  d) Use actor supervision.

Error Handling: Recover

- Recover
  - Exceptions are transformed into a final element using a partial function.
  - Stream is completed afterwards.

```
var recoverSource = errorSource.recover {
  case e: RuntimeException => "stream interrupted"
}
errorSource.runForeach(i => print(i => s"$i "))   // 1 2 3 stream interrupted
```

- Recover with retries
  - Put a new stream in place of the failed one.
  - Maximum number of retries (`attempts`) can be specified.

```
val alternativeSource = Source(10 to 50 by 10).map(_.toString)
val retrySource = errorSource.recoverWithRetries(attempts = 1, {
  case _: RuntimeException => alternativeSource
})
retrySource.runForeach(i => print(s"$i "))       // 1 2 3 10 20 30 40 50
```

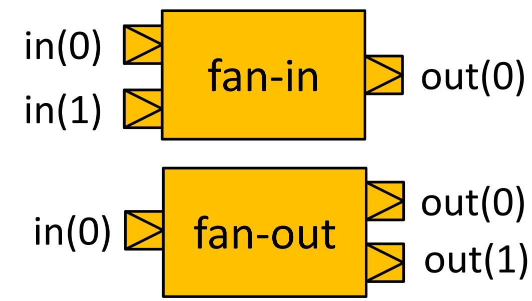
Error Handling: Delayed Restarts

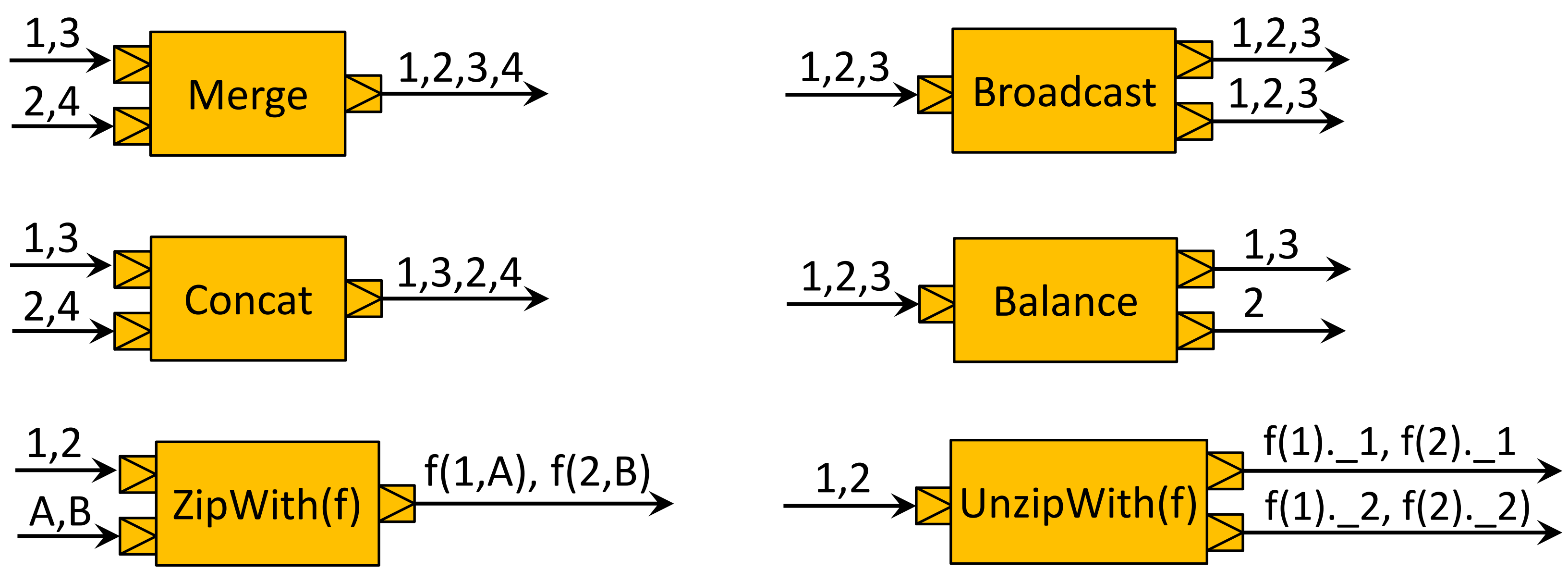
- `RestartSource/RestartFlow/RestartSink` implement the *exponential backoff supervision strategy*.
  - The stage is restarted when the stream *fails,* or it is *completed successfully*.
  - Useful when some external resource is not available to give it time to start-up again.
  - Restarting is done in exponentially increasing intervals.

```
val temperatureSource = Source.fromIterator(() => new Iterator[Double] {
  override def hasNext: Boolean = true
  override def next():  Double  = temperature() // throws Exception
})
val restartSource = RestartSource.withBackoff
  (minBackoff = 1.second,            // restart intervals:
   maxBackoff = 10.seconds,          // 1, 2, 4, 8, 10, 10, … seconds
   randomFactor = 0.2) // add 20% noise
  (() => temperatureSource)
restartSource.runForeach(i => print(s"$i "))
// 10.7 15.3 17.5 sensor_error 20.1 18.9 sensor_error 15.8 14.4 …
```

Graphs

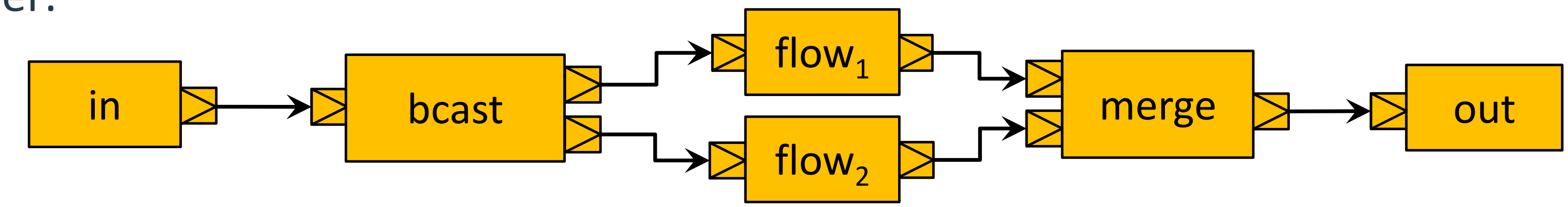
- General graphs are needed when you what to perform
  - *fan-in* operations (multiple inputs) or
  - *fan-out* operations (multiple outputs).
- Akka Streams provides predefined fan-in and fan-out stages:





Graph DSL

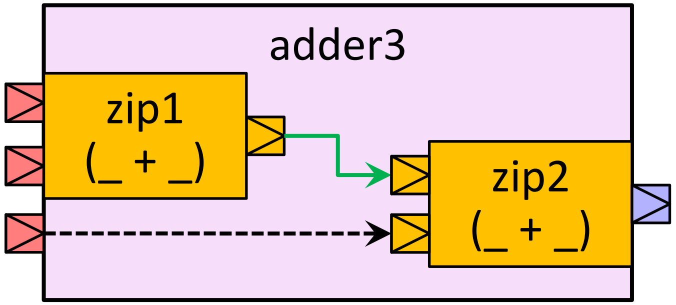
One of the goals of the Graph DSL is to look similar to how one would sketch a graph on a sheet of paper.




```
val graph = GraphDSL.create() { implicit builder: GraphDSL.Builder[NotUsed] =>
  import GraphDSL.Implicits._
  val in = Source(1 to 8)
  val out = Sink.foreach[Int](i => print(s"$i "))
  val bcast = builder.add(Broadcast[Int](2))
  val merge = builder.add(Merge[Int](2))
  val flow1 = Flow[Int].filter(_ % 2 == 1).map(_*10)
  val flow2 = Flow[Int].filter(_ % 2 == 0).map(_*100)
  in ~> bcast ~> flow1 ~> merge ~> out
        bcast ~> flow2 ~> merge
  ClosedShape
}
RunnableGraph.fromGraph(graph).run()         // 10 200 30 400 50 600 70 800
```

Partial Graphs (1)

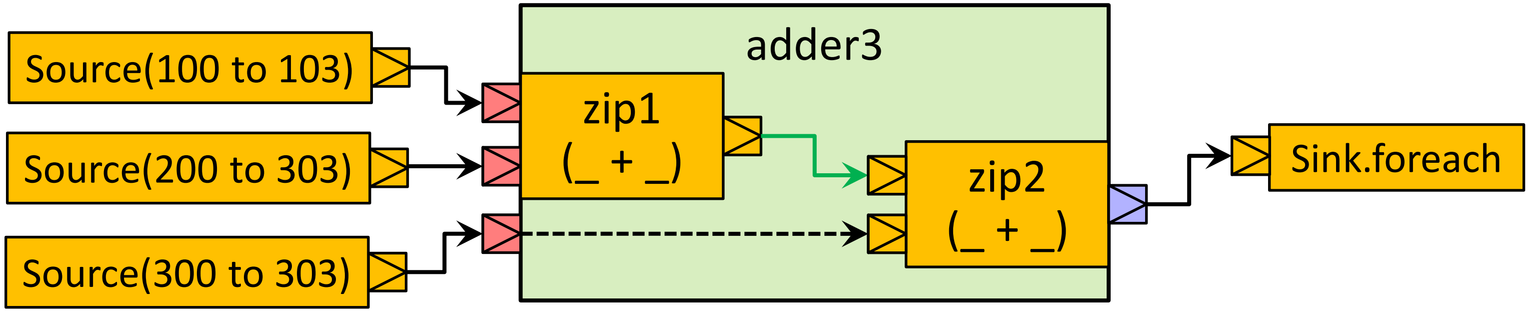
- Graphs with open input and/or output ports are called *partial graphs*.
- Partial graphs are *reusable components*.
- These building blocks can be assembled to *hierarchies of graphs*.
- This way we can build *modular* stream-based programs.




```
val adder3Graph: Graph[UniformFanInShape[Int, Int], NotUsed] =
  GraphDSL.create() { implicit b =>
    val zip1 = b.add(ZipWith[Int, Int, Int](_ + _))
    val zip2 = b.add(ZipWith[Int, Int, Int](_ + _))
    zip1.out ~> zip2.in0
    UniformFanInShape(zip2.out, zip1.in0, zip1.in1, zip2.in1)
  }
```

Partial Graphs (2)

- Partial graphs can be embedded into runnable graphs by wiring-up all open input and output ports.




```
RunnableGraph.fromGraph(GraphDSL.create() { implicit b =>
  val adder = b.add(adder3Graph)
  val sink  = Sink.foreach[Int](i => print(s"$i "))
  Source(100 to 103) ~> adder
  Source(200 to 203) ~> adder
  Source(300 to 303) ~> adder
  adder              ~> sink
  ClosedShape
}).run()                                                     // 600 603 606 609
```

Simplified API for Combining Sources and Sinks

- There is a simplified API that can be used to combine multiple sources and sinks.
- `Source.combine` creates *fan-in shaped graphs* using junctions like `Merge` or `Concat`.

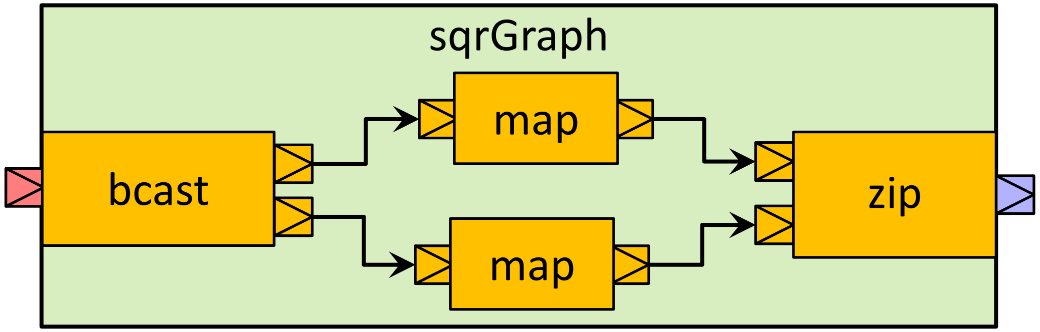
```
val source1 = Source(1 to 50)
val source2 = Source(10 to 50 by 10)
val merged = Source.combine(source1, source2)(Merge(_))
merged.runWith(Sink.foreach(i => print(s"$i ")))    // 1 10 2 20 3 30 4 40 5 50
```

- `Sink.combine` creates *fan-out shaped graphs* using junctions like `Broadcast` or `Balance`.

```
val consoleSink = Sink.foreach[Int](println)
val file = Paths.get("out.txt")
val intToByteString = Flow[Int].map(i => ByteString(s"$i\n"))
val fileSink = intToByteString.to(FileIO.toPath(file))
val sink = Sink.combine(consoleSink, fileSink)(Broadcast[Int](_))
source1.runWith(sink)
```

Building Sources, Sinks, and Flows from Partial Graphs

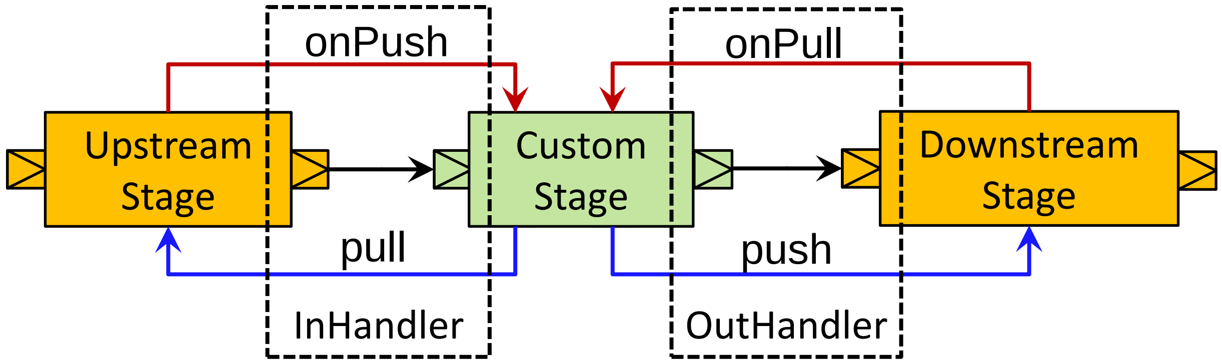
- Partial graphs with no input/one input, one input/one output, and one input/no output can be exposed as sources, flows, or sinks.




```scala
val sqrGraph: Graph[FlowShape[Int, (Int, Int)], NotUsed] =
  GraphDSL.create() {  implicit b =>
    val bcast = b.add(Broadcast[Int](2))
    val zip   = b.add(Zip[Int, Int]())

    bcast ~> Flow[Int].map(identity)   ~> zip.in0
    bcast ~> Flow[Int].map(i => i * i) ~> zip.in1

    FlowShape(bcast.in, zip.out)
  }

val source = Source(1 to 3)
val sink = Sink.foreach[(Int, Int)](pair => print(s"$pair "))
val done = source.via(Flow.fromGraph(sqrGraph)).runWith(sink) // (1,1) (2,4) (3,9)
```

Custom Stages

- A *graph stage* represents an elementary building block of a stream which is defined by
  - its *shape* and
  - the *graph stage logic*.
- The shape defines the number and type of input and output ports.
  - `SourceShape/FlowShape/SinkShape`: no/one input/output port.
  - `UniformFan[In|Out]Shape`: n inputs, one output; one input, n outputs.
- The *Graph Stage API* defines the methods
  - `def shape: S <: Shape`
  - `def createLogic(…): GraphStageLogic`: Implements the stage's logic by providing an input and an output handler.



Custom Stages: Example (1)

```
class MyFilter[T](predicate: T => Boolean) extends GraphStage[FlowShape[T, T]]:
  val input  = Inlet[T] ("MyFilter.in")
  val output = Outlet[T]("MyFilter.out")
  override def shape: FlowShape[T, T] = FlowShape(input, output)
  override def createLogic(inheritedAttributes: Attributes): GraphStageLogic =
    new GraphStageLogic(shape) {
      setHandler(input, new InHandler {
        override def onPush(): Unit = {
          val element = grab(input)
          if(predicate(element))
            push(output, element)
          else
            pull(input)
        }
      })
      setHandler(output, new OutHandler {
        override def onPull() = pull(input)
      })
    }
```

Custom Stages: Example (2)

- A graph with shape source/flow/sink can be converted into a shape source/flow/sink:

```
object MyFilter:
  def apply[T](predicate: T => Boolean): Flow[T, T, NotUsed] =
        Flow.fromGraph(new MyFilter(predicate))
```

- Usage of custom stage:

```
val source = Source(1 to 10)
source.via(MyFilter(_ % 2 == 0)).runForeach(println)
```