

# Akka Streams

Daniel Hinojosa

# Conventions in the slides

The following typographical conventions are used in this material:

## *Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

## **Constant width**

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

**Constant width bold** Shows commands or other text that should be typed literally by the user.

## *Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.

# Shell Conventions

All shells (bash, zsh, Windows Shell) are represented as %

```
% calendar
```

All SBT shells are represented as >

```
> compile
```

All Scala REPL and SBT Consoles are represented as **scala>**

```
scala>
```

# About Akka Streams

## Akka Streams

- Backed by Actors
- Stream processors much like their competitors RxJava, Project Reactor
- Asynchronous Process of Streams can be difficult

# Understanding Java Futures

## Future Defined

Future definition - `Future` represents the lifecycle of a task and provides methods to test whether the task has completed or has been canceled. `Future` can only move forwards and once complete it stays in that state forever.

— Java Concurrency in Practice, Brian Goetz

## Thread Pools

Before setting up a future, a thread pool is required to perform an asynchronous computation. Each pool will return an `ExecutorService`.

There are a few thread pools to choose from:

- `FixedThreadPool`
- `CachedThreadPool`
- `SingleThreadExecutor`
- `ScheduledThreadPool`
- `ForkJoinThreadPool`

## Fixed Thread Pool

- "Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue."
- Keeps threads constant and uses the queue to manage tasks waiting to be run
- If a thread fails, a new one is created in its stead
- If all threads are taken up, it will wait on an unbounded queue for the next available thread

## Cached Thread Pool

- Flexible thread pool implementation that will reuse previously constructed threads if they are available
- If no existing thread is available, a new thread is created and added to the pool
- Threads that have not been used for sixty seconds are terminated and removed from the cache

# Single Thread Executor

- Creates an Executor that uses a single worker thread operating off an unbounded queue
- If a thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.

## Scheduled Thread Pool

- Can run your tasks after a delay or periodically
- This method does not return an `ExecutorService`, but a `ScheduledExecutorService`
- Runs periodically until `cancel()` is called.

## Fork Join Thread Pool

- An `ExecutorService`, that participates in *work-stealing*
- By default when a task creates other tasks (`ForkJoinTasks`) they are placed on the same on queue as the main task.
- *Work-stealing* is when a processor runs out of work, it looks at the queues of other processors and "steals" their work items.
- Not a member of Executors. Created by instantiation
- Brought up since this will be in many cases the "default" thread pool on the JVM

## Basic Future Blocking (JDK 5)

```
ExecutorService fixedThreadPool = Executors.newFixedThreadPool(5);

Callable<Integer> callable = new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        System.out.println("Inside the future: " +
            Thread.currentThread().getName());
        Thread.sleep(5000);
        return 5 + 3;
    }
};

System.out.println("In test:" + Thread.currentThread().getName());
Future<Integer> future = fixedThreadPool.submit(callable);

//This will block
Integer result = future.get(); //block
System.out.println("result = " + result);
```

# Basic Future Asynchronous (JDK 5)

```
ExecutorService cachedThreadPool = Executors.newCachedThreadPool();

Callable<Integer> callable = new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        Thread.sleep(3000);
        return 5 + 3;
    }
};

Future<Integer> future = cachedThreadPool.submit(callable);

//This is proper asynchrony, but rather ugly
while (!future.isDone()) {
    System.out.println("I am doing something else on thread: " +
        Thread.currentThread().getName());
}

Integer result = future.get();
```



Applicable to Java 8

# Java's CompletionStage

## CompletionStage

- A stage of asynchronous computation
- Performs an action or computes a value when the previous stage completes
- An API that cleanly handles asynchronous processing of chained **Future**
- Every stage can be performed as a:
  - **Function**
  - **Consumer**
  - **Runnable**
- Uses an **ExecutorService** to drive each stage
- Is also a basic component of Akka Streams

## CompletableStage initialization with supplyAsync

- **supplyAsync** provides the data contained in the **Supplier**
- Processing takes place on a different **Thread**
- Requires an **ExecutorService** for thread pools

```
ExecutorService executorService = Executors.newCachedThreadPool();

integerFuture1 = CompletableFuture
    .supplyAsync(() -> {
        try {
            System.out.println("intFuture1 is Sleeping in thread: "
                               + Thread.currentThread().getName());
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return 5;
    }, executorService);
```

## Simple Chain using CompletableStage

- Each **CompletableStage** can be interlinked
- This is the simplest flow of data, from source to sink

```
integerFuture1.thenAccept(System.out::println); //5
```

## Functional Map analogy with `thenApply`

- `thenApply` is analogous to a functional `map`
- `map` applies a function to every container providing a copy of the container with the modified contents

```
CompletableFuture<String> future =  
    integerFuture1.thenApply(x -> {  
        System.out.println("In Block:" +  
            Thread.currentThread().getName());  
        return String.valueOf(x + 19);  
    });  
future.thenAccept(s -> {  
    System.out.println("In the accept: " + Thread.currentThread().getName());  
    System.out.println(s);  
});
```

## Inlining `thenApply`

- In the previous example, processing was done separate
- All functions can be inlined to express a chain of processing

```
integerFuture1.thenApply(x -> String.valueOf(x + 19)).thenAccept(System.out::println);
```

## Lab: Running `CompletableFuture` examples

**Step 1:** In the `akka-streams-study` project, under the `src/main/java` folder, open `com.xyzcorp.CompletableFutureTest`.

**Step 2:** Use your IDE's faculties to run the tests individually to see how `CompletableFuture` and `CompletableFutureStage` works

	Running Tests in IntelliJ	Running Tests in Eclipse
Windows/Linux	CTRL + SHIFT + F10 or SHIFT + F10	ALT + SHIFT + X, T
Mac	CTRL + R	CMD + OPTION + X, T



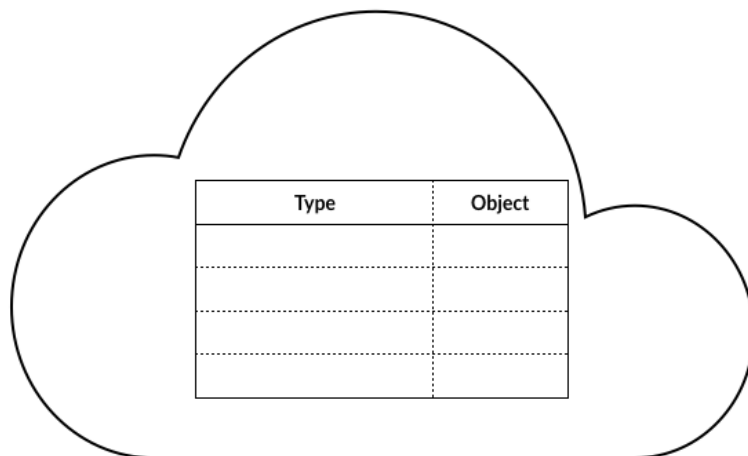
For Eclipse Run the Test in the method header, for IntelliJ, run the Test anywhere within the method



# implicit

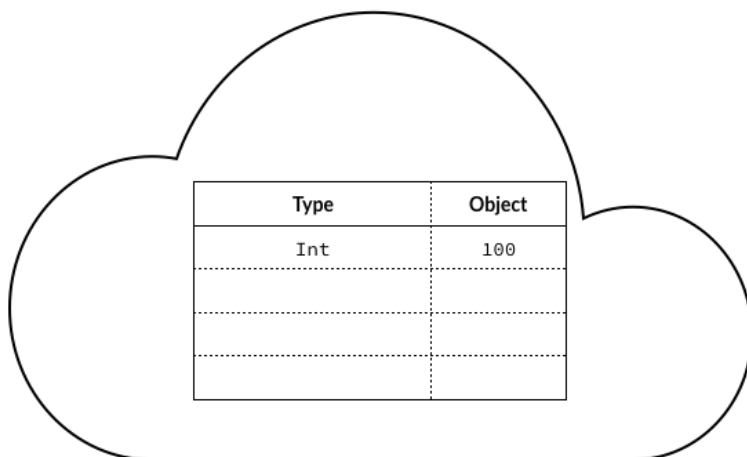
## implicit

- `implicit` is like an invisible `Map[Class[A], A]` where `A` is any object and it is tied into the scope
- Whatever type is required the object that it corresponds to that type will be injected automatically.



## implicit

- `implicit` in this case bounds an `Int` of `100`
- Once established we can call upon the `implicit` binding to a binding parameter group



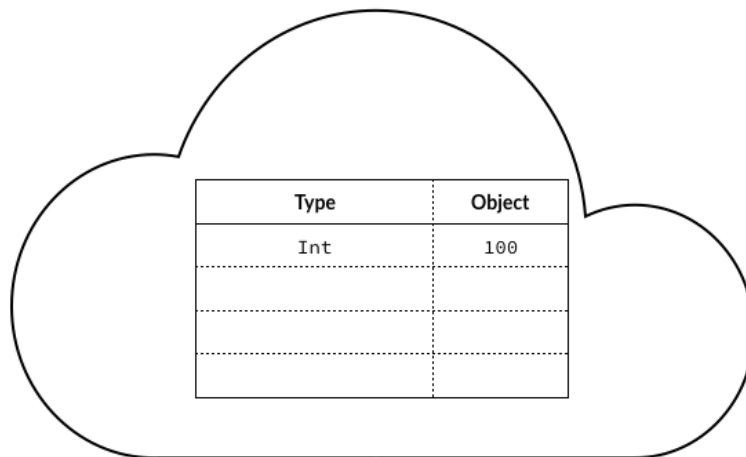
```
implicit val hourlyRate = 100
```

```
def calcPayment(hours: Int)(implicit rate: Int) = hours * rate
```

```
calcPayment(50) should be(5000)
```

# Overriding an **implicit** manually

- You can always override the any **implicit** manually



Type	Object
Int	100

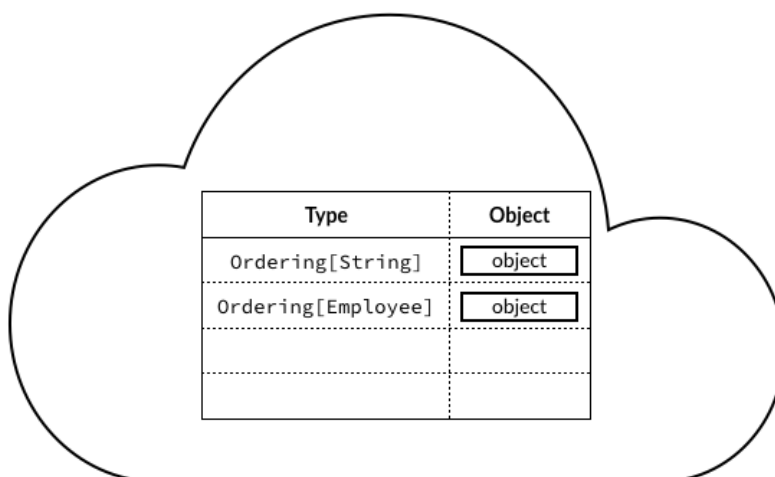
```
implicit val hourlyRate = 100
```

```
def calcPayment(hours: Int)(implicit rate: Int) = hours * rate
```

```
calcPayment(50)(200) should be(10000)
```

## Setting up an **implicit** for **Ordering[T]**

- You can establish an **implicit** for ordering inside of a collection or any construct with **Ordering[T]**
- This is also known as a *Type Class*



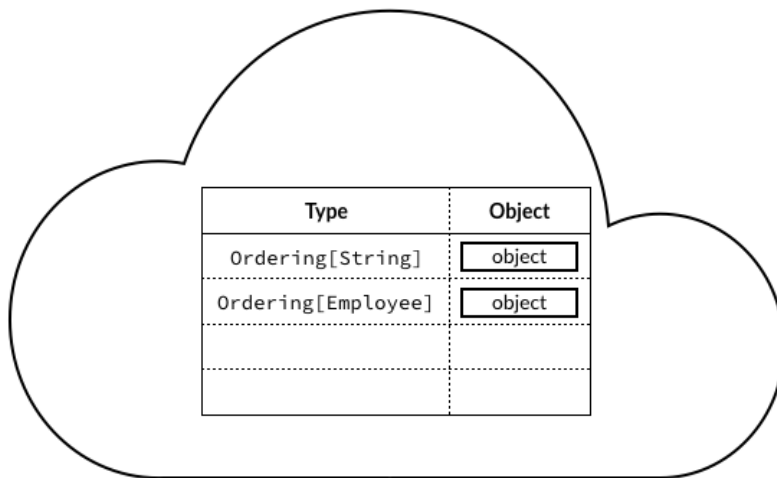
Type	Object
Ordering[String]	<div>object</div>
Ordering[Employee]	<div>object</div>

```
case class Employee(firstName:String, lastName:String)

implicit val employeeOrderingByLastName: Ordering[Employee] =
  new Ordering[Employee] {
    override def compare(x: Employee, y: Employee): Int = {
      x.lastName.compareToIgnoreCase(y.lastName)
    }
  }
}
```

## Applying the `implicit` for `Ordering[T]`

Once this `implicit` is established there is an `implicit` bound on how to order an `Employee`



```
List(new Employee("Eric", "Clapton"),
      new Employee("Jeff", "Beck"),
      new Employee("Ringo", "Starr"),
      new Employee("Paul", "McCartney"),
      new Employee("John", "Lennon"),
      new Employee("George", "Harrison")).sorted
```

Which yields the result:

```
List(new Employee("Jeff", "Beck"),
      new Employee("Eric", "Clapton"),
      new Employee("George", "Harrison"),
      new Employee("John", "Lennon"),
      new Employee("Paul", "McCartney"),
      new Employee("Ringo", "Starr"))
```



In order to avoid any conflicts, it is up to you, the programmer, to decide in what scope `implicit` are applied

# Why did `implicit Ordering[Employee]` work?

- Here is the Scala API signature for `sorted`
- Notice the `implicit` definition in the method signature
- It requires that an implicit bound be available in order to process

```
def sorted[B >: A](implicit ord: math.Ordering\[B\]): List[A]
```

Sorts this sequence according to an Ordering.

The sort is stable. That is, elements that are equal (as determined by `lt`) appear in the same order in the sorted sequence as in the original.

`ord`            the ordering to be used to compare elements.

`returns`      a sequence consisting of the elements of this sequence sorted according to the ordering `ord`.

*Definition Classes*    [SeqLike](#)

*See also*                [scala.math.Ordering](#)

# Establishing an Execution Context

## ExecutionContext

- A `scala.util.concurrent.ExecutionContext`
  - Is a wrapper around and `Executor` that is used to `implicitly` bind a thread pool.
  - Is a `trait`
  - Can be adapted from to wrap a premade `java.util.concurrent.Executor`

## `scala.util.concurrent.ExecutionContext.global`

- `ExecutionContext` backed by a `ForkJoinPool`
- Automatically set to the number of available processors
- Can potentially add more if any of the threads are blocked
- This can be overridden by applying any of the following VM attributes:
  - `scala.concurrent.context.minThreads`
  - `scala.concurrent.context.numThreads`
  - `scala.concurrent.context.maxThreads`

## Retreiving the `ExecutionContext global`

- We can establish an `ExecutionContext` with an `implicit`
- This will make available the `ExecutionContext` whenever required

```
implicit val ec = ExecutionContext.global
```

- This can also be done as an `import`
- It also establishes an `ExecutionContext` implicitly

```
import ExecutionContext.Implicits.global
```

## Converting a Java `Executor` to an `ExecutionContext`

- If there a specialized thread pool that you would like use as an `ExecutionContext`, use `fromExecutor`

```
val executor = ExecutionContext.fromExecutor(Executors.newCachedThreadPool())
```

# Scala Futures

## Semantic Differences with Scala Future

- A Scala **Future** is either *completed* or *not completed*
- When a **Future** is completed with a *value* it is *successfully completed*
- When a **Future** throws a `java.util.Throwable` then it has *failed*

## Creating a Future

- `scala.concurrent.Future[T]` is created with an `apply` in its `object`
- Requires an `ExecutionContext` to be available either explicitly or implicitly
  - `import scala.concurrent.ExecutionContext.Implicits.global`
  - ``implicit val executionContext = scala.concurrent.ExecutionContext.global`
  - ``implicit val executionContext = scala.concurrent.ExecutionContext.fromExecutor(...)`

## Using `foreach` to receive Future results

- To receive the contents of a **Future** you can use `foreach`

```
val future:Future[Int] = ...  
future.foreach(x => println(x))
```

## Using `onComplete` to receive Future results

- `onComplete`
  - Will accept a `Function` that is given a `Try` input
  - The signature is the following:

```
onComplete[U](f: (Try[T]) => U)(implicit executor: ExecutionContext): Unit
```

## Try

- `Try[T]` is an `abstract class` that has two `sealed` children
  - `Success[T]` - Represents a success, contains the answer of type `T`
  - `Failure[T]` - Represents a failure, contains the `Throwable` that caused the failure



`sealed` means an abstraction like `abstract` and `trait` where all the children are declared and no one else can subclass the `sealed` abstraction

## Lab: Creating a Future

**Step 1:** In the akka-streams-study, and in the `src/test/scala` folder, and in the package `com.xyzcorp.futures` open the `FuturesSpec`

**Step 2:** Run the first test in your IDE or on in SBT using

```
> testOnly com.xyzcorp.futures.FuturesSpec -- -z "foreach"
```

**Step 3:** Refactor the `ExecutionContext` so that you do not require to inject it into the `Future` declaration or the `foreach`

## Lab: Processing a Future with onComplete

**Step 1:** Continuing inside `com.xyzcorp.futures.FuturesSpec`, we will have an instructor led exercise and type inside the "A basic future. Processing it using `onComplete`" test

**Step 2:** We will create a future similar to the following:

```
val eventualString: Future[String] = Future {  
  val num = scala.util.Random.nextInt(2)  
  if (num == 1) "Awesome!"  
  else throw new RuntimeException(s"Invalid number $num")  
}
```

**Step 3:** Then we will get the response of the `Future` by using `onComplete`

# Akka

## About Akka

- Set of libraries used to create concurrent, fault-tolerant and scalable applications.
- It contains many API packages:
- Actors, Logging, Futures, STM, Dispatchers, Finite State Machines, and more...
- We are going to only focus on some of the core items.
- Akka's managing processes can run on
  - in the Same VM
  - in a Remote VM
- Akka's Actor's are a replacement Scala's Actors that came in earlier versions

## Starting with Actors

- Based on the Actor Model from Erlang.
- Encapsulates State and Behavior
- Concurrent processors that exchange messages.
- Each message is immutable (cannot be changed, this is required!)
- Each message should not be a closure
- Breath of fresh air if you have suffered concurrency

## Akka Rules

- The messages to and from actors cannot be
  - Mutable
  - Closure

## Immutable Scala Classes

```
case class Person(firstName:string, lastName:string)
```

or



```
class Person(val firstName:String, val lastName:String) {
  override def toString() = {...}
  override def equals(x:AnyRef):Boolean = {...}
  override def hashCode:Int = {...}
}
```

## Recognizing Closures

```
var x = 3
val y = (z:Int) => x + z
def foo(w: Int => Int) = w(5)
println(foo(y)); //8
```

## Location Transparency

- Vertical and Horizontal Growth
- Horizontal Growth driven by Remoting systems
- Vertical Growth driven by routers
- All configuration based

## More about `actorOf()`

- Creates an actor onto a `ActorSystem`
- Creates an actor given the a set of properties to describe the Actor
- Creates an Actor with an identifiable name, if provided
- Returns an `ActorRef`
- If `actorOf` is called *inside* of another actor, that new actor becomes a child of that actor

## `actorSelection()`

- Was `ActorFor` which is now deprecated
- Actor references can be looked up using `actorSystem.actorSelection(...)` method.
- `actorSystem.actorSelection` returns a `ActorSelection` object that abstracts over local or remote reference.
- `ActorSelection` can be used as long as the actor is alive.
- Only ever looks up an existing actor, i.e. does not create one.
- The actor must exist or you will receive an `EmptyLocalActorRef`
- For Remote Actor References, a search by path on the remote system will occur.

# About ActorRefs

- Any subtype of ActorRef
- ActorRef is the only way to interact with an Actor
- Intent is to send messages to Actor that it represents, proxy.
- Each actor has reference to `self()` refers to it's own reference.
- Each actor also has reference to the `sender()`, the actor that sent the message.
- Any reference can be sent to another actor so that actor can send messages to it.

## Lab: Reviewing Akka-Actors

**Step 1:** In the akka-streams-study, in the `src/test/scala` folder and in the `com.xyzcorp.akka` package open the `SimpleActorSpec`

**Step 2:** Run the tests individually in your IDE, below is a table of keymaps to run the test

	Running Tests in IntelliJ	Running Tests in Eclipse
Windows/Linux	CTRL + SHIFT + F10 or SHIFT + F10	ALT + SHIFT + X, T
Mac	CTRL + R	CMD + OPTION + X, T

**Step 3:** Run the tests inside of SBT shell using the command

```
> testOnly com.xyzcorp.akka.SimpleActorTest -- -z "receive our message"
```



"receive our message" is a substring of the test name. That way you can run individual tests

# Reactive Streams

## reactivestreams.org

- <http://reactivestreams.org>
- Standard for asynchronous stream processing with non-blocking back pressure
- Goals is to govern the exchange of stream data across an asynchronous boundary without buffering needless arbitrary data
- Find a minimal set of interfaces, methods and protocols that will describe the necessary operations and entities to achieve the goal
- Not necessary for synchronous process only asynchronous processing

## Reactive Streams API

- API has been standardized as of April 30, 2015
- 1.0.0 version of Reactive Streams API
- Contains:
  - [Java API](#)
  - [Specification](#)
  - [TCK \(Technology Compatibility Kit\)](#)
  - [Implementation Examples](#)

## What Asynchronous Streams?

- Increased need for transferring large quantities of data across asynchronous boundaries
  - CPU Boundaries
  - Networking Systems

## Backpressure

- The producer may not always be the same speed as the consumer
- The consumer will often require that it hold up the source processing
- This primarily occurs when the consumer is slower than the producer
- If no backpressure is defined then data processing will grind to a halt

## Understanding the Reactive Streams API

- A collection of four main **interface**

- **Subscriber**
- **Publisher**
- **Subscription**
- **Processor**

## Publisher

- Simply publishes the data
- Provider of a potentially unbounded number of sequenced elements

```
public interface Publisher<T> {  
    public void subscribe(Subscriber<? super T> s);  
}
```

- Publishing them according to the demand received from its **Subscriber(s)**

## Subscriber

- Will **receive** call to **onSubscribe(Subscription)** after a **Subscriber** is given to the **Publisher**
- Also contains calls sent from the source when:
  - A new element is sent **onNext**
  - When an error occurs **onError**
  - When the stream is possibly done **onComplete**

```
public interface Subscriber<T> {  
    public void onSubscribe(Subscription s);  
    public void onNext(T t);  
    public void onError(Throwable t);  
    public void onComplete();  
}
```

## Subscription

- The **Subscription** represents the **Subscription** itself sent to the **Subscriber**
- Through the **Subscription** a **Subscriber**:
  - Can request more information from its **Source** using **request**
  - Can cancel the flow by calling **cancel**

```
public interface Subscription {  
    public void request(long n);  
    public void cancel();  
}
```

## Processor

- A Combination of both a **Subscriber** and **Publisher**
- Meant to be snapped in to a processing flow between the **Source** and **Sink** and transform the data
- It is the "go-between" linking components in a stream

```
public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {  
}
```

# Akka Streams

## Akka Streams Overview

- Builds on the idea of flows and flow graphs that define how a stream is processed
- Streams are built with reusable pieces either:
  - Prepackaged components
  - Custom made composites

## Relationship between Akka Streams and the Reactive Streams API

- The scope of Reactive Streams is defining a mechanism to move data across an asynchronous boundary
- Akka Streams Implementation Uses Reactive Streams Internally
- Akka Streams API is meant not to expose the Reactive Streams API

## Akka Streams River Analogy



**Source** – One Output, Emits Data Elements

**Flow** – One Input and One Output, Connects up and down streams transforming data.

**Sink** – One Output, requesting and accepting data elements possibly slowing down the upstream producer of elements

## Akka Streams Pipe Analogy



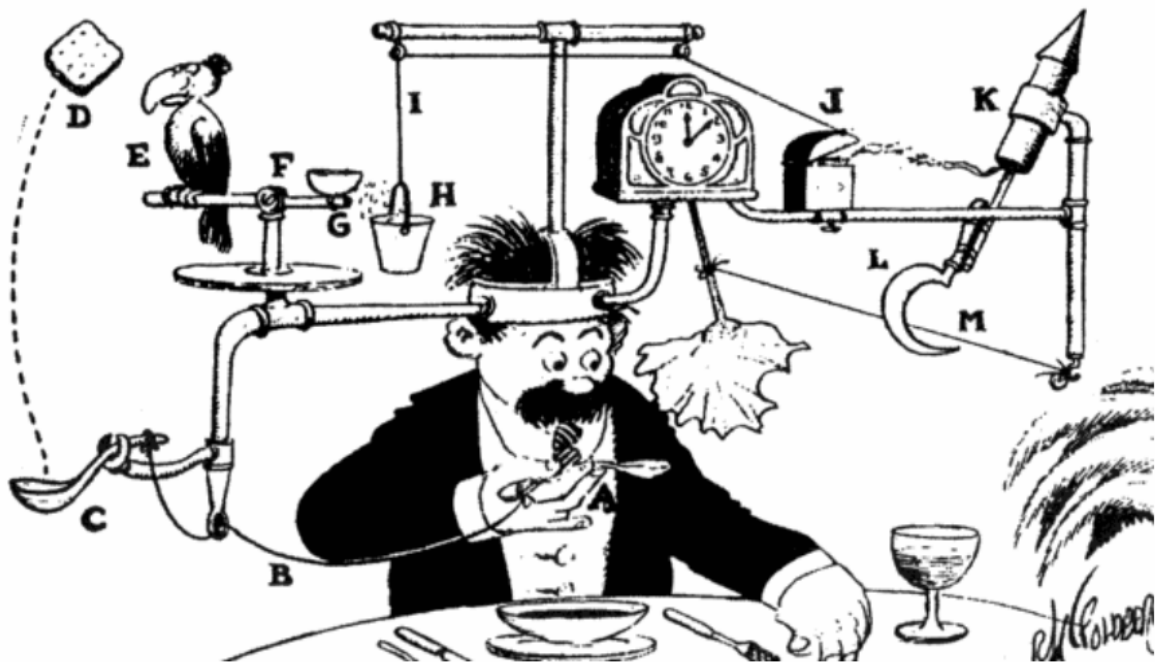
## Akka Streams Lightning Analogy





## **Akka Streams Rube Goldberg Analogy**



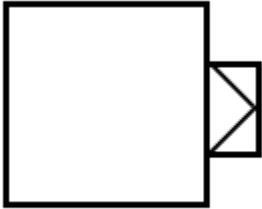


**Think Rube Goldberg machines!**

# Source

## Defining Source

- A Source that accepts a single item
- In the following example the first element `[Int]` is the type of element that this source emits



- The second one is some auxiliary value that certain components may add.
- When no auxiliary value is produced `akka.NotUsed` is used in its place.
- The following is an example of a Source that emits one element

```
val single: Source[Int, NotUsed] = Source.single(3)
```

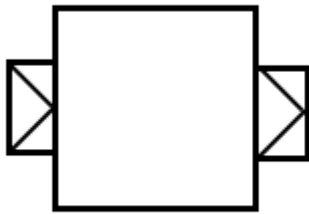
## Other Source

- With each update of Akka Streams another manifestation of Source is developed
- You can create many of these Source
  - `cycle`
  - `fromFuture`
  - `fromPublisher` (from the Reactive Streams Library)
  - `lazily`
  - `more...`

# Flow

## Defining Flow

- **Flow** are the interlinking pieces that are the intermediate computations
- A processing stage which has exactly one input and output
- Connects its upstream and downstream components by transforming the data elements flowing through it
- Are immutable, thread-safe, and freely shareable



## Why Flow?

- A **Source** can be manipulated by anyone of the methods, for example it contains **map**, but it is still a **Source**

```
val result:Source[Int, NotUsed] = Source(1 to 100).map(x => x + 10)
```

- A **Flow** gives us the opportunity to create a separate component that represent the action like **map**

```
val mapIntFlow:Flow[Int, NotUsed] = Flow.apply[Int]() .map(x => x + 10)
```

- The above can be refactored simply to:

```
val mapIntFlow = Flow[Int].map(_ + 10)
```

## Connecting the Flow

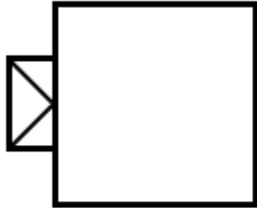
- Once a **Flow** is created, it can then be connected to a **Source** with **via**

```
val mapIntFlow = Flow[Int].map(_ + 10)
val newSource = Source(1 to 10).via(mapIntFlow)
```

# Sink

## Defining Sink

- **Sink** is the where all data is finally accumulated
- A processing stage with exactly one input
- Requesting and accepting data elements possibly slowing down the upstream producer of elements



## Why Sink?

- A **Sink** gives us the opportunity to create a separate component that represents the final stage for stream processing

```
val foldSink = Sink.fold[Int, Int](0)(_ + _)
val printlnSink = Sink.foreach[Int](println)
```

- The sink can then be applied to as a final stage using **to**
- This will create what is called a **RunnableGraph**, which can now be **run**

```
val mapIntFlow: Flow[Int, Int, NotUsed] = Flow[Int].map(x => 10 + x)
val printlnSink = Sink.foreach[Int](println)
val graph: RunnableGraph[NotUsed] =
  Source(1 to 10).via(mapIntFlow).to(printlnSink)
```

# RunnableGraph

## Defining RunnableGraph

- Once **Source**, all **Flow** and **Sink** are connected it is a **RunnableGraph**
- Defined as a Flow that has both ends “attached” to a **Source** and **Sink** respectively
- Is ready to be **run()**

```
val graph: RunnableGraph[NotUsed] = Source(1 to 10).via(mapIntFlow).to(printlnSink)
```

## Running the RunnableGraph

- Once contained as a **RunnableGraph**, you need the following to run
  - An **ActorSystem**
  - An **ExecutionContext**
  - A **Materializer**
- A **Materializer** is an engine, backed by **Actor**, that will process the stream

## A RunnableGraph by example

**Step 1:** Establish all the required elements, **actorSystem**, **materializer**, and **executionContext**

```
implicit val system: ActorSystem = ActorSystem("MyActorSystem")
implicit val materializer: ActorMaterializer = ActorMaterializer()
implicit val executionContext: ExecutionContextExecutor = system.dispatcher
```



**system.dispatcher** merely obtains the thread pool from the **ActorSystem**

**Step 2:** Create a **RunnableGraph**, and run it

```
val mapIntFlow: Flow[Int, Int, NotUsed] = Flow[Int].map(x => 10 + x)
val printlnSink = Sink.foreach[Int](println)
val graph: RunnableGraph[NotUsed] = Source(1 to 10).via(mapIntFlow).to(printlnSink)
graph.run()
```

# Lab: Creating a `RunnableGraph`

**Step 1:** In the akka-streams-study project, and in the `src/test/java` folder and in the `com.xyzcorp.akka.streams` package, and in the `SimpleStreamSpec.scala`, locate the "A stream can be created by independent components"

**Step 2:** Create the example that we have been discussing, inside of the "A stream can be created by independent components" test. Here it is again, but with an explicit `materializer`.

```
val mapIntFlow: Flow[Int, Int, NotUsed] = Flow[Int].map(x => 10 + x)
val printlnSink = Sink.foreach[Int](println)
val graph: RunnableGraph[NotUsed] = Source(1 to 10).via(mapIntFlow).to(printlnSink)
graph.run()(materializer)
Thread.sleep(1000)
```

**Step 3:** Run the `RunnableGraph` created

**Step 4:** Refactor as much cruft as you can away to make it as inline as possible

# Composite Stages

## Composite Source

- A **Source** can be composited with any of its methods, and still be a **Source**

```
val composite:Source[Int, NotUsed] = Source(1 to 10).map(x => x + 10)
```

- A **Source** can be composited also with **via**, which applies a **Flow**, and still be a **Source**

```
val mapIntFlow: Flow[Int, Int, NotUsed] = Flow[Int].map(x => 10 + x)
val composite:Source[Int, NotUsed] = Source(1 to 10)
    .via(mapIntFlow)
```

## Composite Sink

- A **Sink** can be composited with any of its methods, and still be a **Sink**

```
val composite:Source[Int, NotUsed] = Flow[Int].map(x => x + 10).to(String.foreach(println))
```

- A **Sink** can be composited also with **to**, which applies a **Flow**, and can still be a **Sink**

```
val mapIntFlow:Flow[Int, Int, NotUsed] = Flow[Int].map(x => 10 + x)
val compositeSink:Sink[Int, NotUsed] = mapIntFlow.to(Sink.foreach(println))
```

## viaMat and toMat

### Recalling the auxiliary value

- In streams there is an auxiliary value per stream
- This can be used to:
  - to manipulate the stream
  - to provide added information
- It is your choice which you want to use by either pull:
  - The left with `Keep.left`
  - The right with `Keep.right`

### Which one? viaMat or toMat?

- As a mental note:
- `via` is used for `Flow`
- `to` is used for `Sink`
- Therefore:
- `viaMat`, select element as a `Flow`
- `toMat`, select element as a `Sink`

### Lab: toMat

**Step 1:** In the akka-streams-study project, and in the `src/test/scala` folder and in the `com.xyzcorp.akka.streams` package, and in the `SimpleStreamSpec.scala`, locate the "Perform a test with an async boundary which will run on a separate actor and dispatcher" test

**Step 2:** Run the method, in SBT using the following which will only run the single test

```
> testOnly com.xyzcorp.akka.streams.SimpleStreamSpec -- -z "async boundary"
```

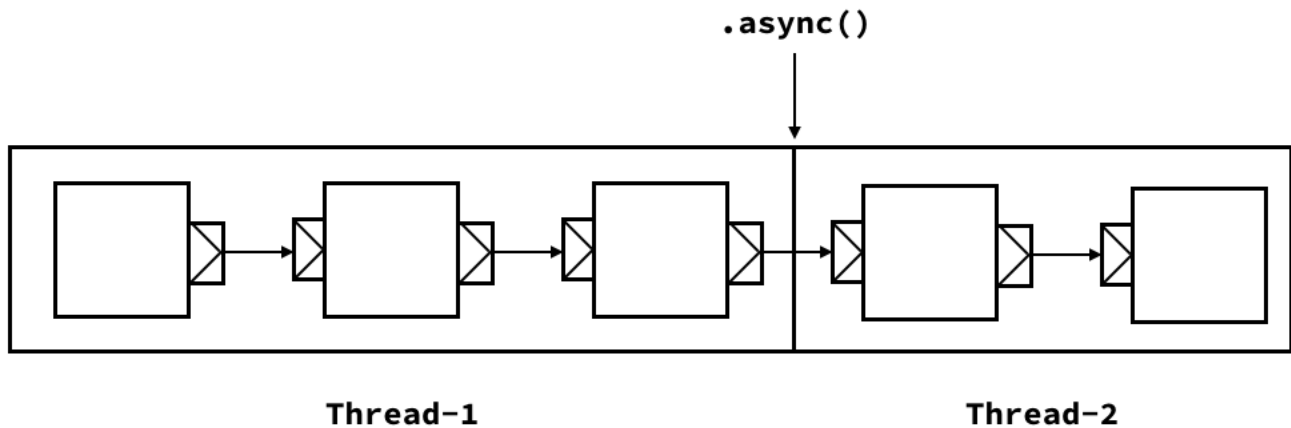
**Step 3:** Disassemble and discover types to understand how all of it fits together. Introduce explicit types to see how it all fits together and try different combinations



# async

## Using async

- **async** is a "combinator" that places a thread before the next component in the Stream
- Included in the boundary is a process that runs within the different **Thread**



## async by Example

```
import scala.language.postfixOps
Source(1 to 10)
  .map(1 +)
  .map(x => {
    println("Top: " + currentThreadName)
    x
  })
  .async //We are setting an async boundary
  .map(x => {
    println("Bottom " + currentThreadName)
    x
  })
  .filter(x => x % 2 == 0)
  .runForeach(println)
```

## Lab: async

**Step 1:** In the akka-streams-study project, and in the `src/test/scala` folder and in the `com.xyzcorp.akka.streams` package, locate the "Perform a test with an async boundary which will run on a separate actor and dispatcher" test in the `SimpleStreamSpec.scala`

**Step 2:** Run the method, in SBT using the following which will only run the single test

```
> testOnly com.xyzcorp.akka.streams.SimpleStreamSpec -- -z "async boundary"
```

**Step 3:** Disassemble and discover types to understand how all of it fits together. Introduce explicit types to see how it all fits together and try different combinations

# Failure

## Dealing with Failure

- Many times dealing with a failure can be very disruptive
- You can deal with failures as
  - `recover` to emit a final element then complete the stream normally on upstream failure
  - `recoverWithRetries` to create a new upstream and start consuming from that on failure
  - Restarting sections of the stream after a backoff
  - Using a supervision strategy for stages that support it

### recover

- Recover will intercept a call, and stop with one element marking the end
- This will provide a non-disruptive exit from the stream

```
Source(10 to 0 by -1)
  .async
  .map(x => Some(100 / x))
  .recover { case t: Throwable => None }
  .runForeach(println)
```

### recoverWithRetries

- This will provide a non-disruptive exit from the stream
- `recoverWithRetries` will:
  - Intercept a call
  - Retry the specified times
  - Stop with multiple elements marking the end

```
Source(10 to 0 by -1)
  .async
  .map(x => Some(100 / x))
  .recover { case t: Throwable => None }
  .runForeach(println)
```

## Lab: Handling Failure

**Step 1:** In the akka-streams-study project, and in the `src/test/scala` folder and in the `com.xyzcorp.akka.streams` package, locate the "Perform a stream with a recover" and "Perform a

stream with a `recoverWithRetries` test

**Step 2:** Run the methods, in SBT, using the following which will only run the single test

```
> testOnly com.xyzcorp.akka.streams.SimpleStreamSpec -- -z "recover"
```

```
> testOnly com.xyzcorp.akka.streams.SimpleStreamSpec -- -z "recoverWithEntries"
```

**Step 3:** Disassemble and discover types to understand how all of it fits together. Introduce explicit types to see how it all fits together and try different combinations

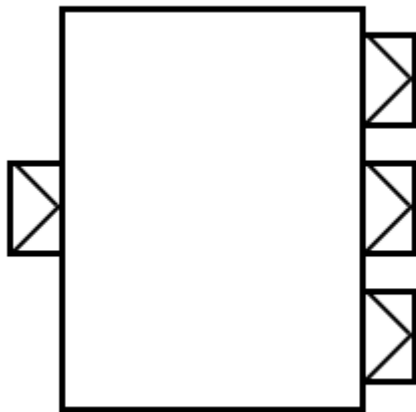
# Graphs

## Designing Graphs

- DSL designed graphs
- Reusable
- Used for complicated fan-in (merge) or fan-out (broadcast scenarios)

## Fan Out

- `Broadcast[T]` - (1 input, N outputs) given an input element emits to each output
- `Balance[T]` - (1 input, N outputs) given an input element emits to one of its output ports
- `UnzipWith[In,A,B,...]` - (1 input, N outputs) takes a function of 1 input that given a value for each input emits N output elements (where  $N \leq 20$ )
- `UnZip[A,B]` - (1 input, 2 outputs) splits a stream of `(A,B)` tuples into two streams, one of type A and one of type B



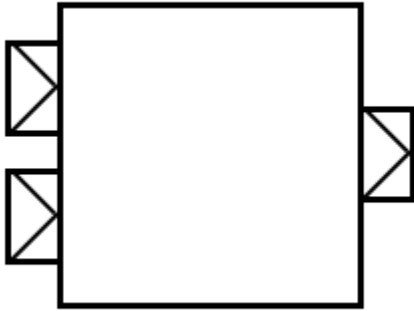
Source: <https://doc.akka.io/docs/akka/2.5.4/scala/stream/stream-graphs.html>

## Fan In

- `Merge[In]` - (N inputs , 1 output) picks randomly from inputs pushing them one by one to its output
- `MergePreferred[In]` - like Merge but if elements are available on preferred port, it picks from it, otherwise randomly from others
- `MergePrioritized[In]` - like Merge but if elements are available on all input ports, it picks from them randomly based on their priority
- `ZipWith[A,B,...,Out]` - (N inputs, 1 output) which takes a function of N inputs that given a value

for each input emits 1 output element

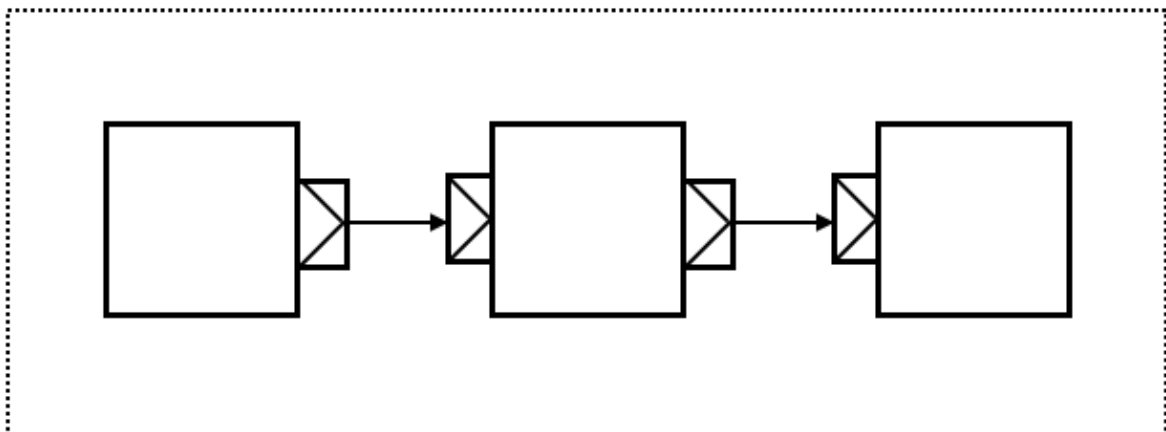
- **Zip[A,B]** - (2 inputs, 1 output) is a ZipWith specialised to zipping input streams of **A** and **B** into a (A,B) tuple stream
- **Concat[A]** - (2 inputs, 1 output) concatenates two streams (first consume one, then the second one)



Source: <https://doc.akka.io/docs/akka/2.5.4/scala/stream/stream-graphs.html>

## Closed Shape

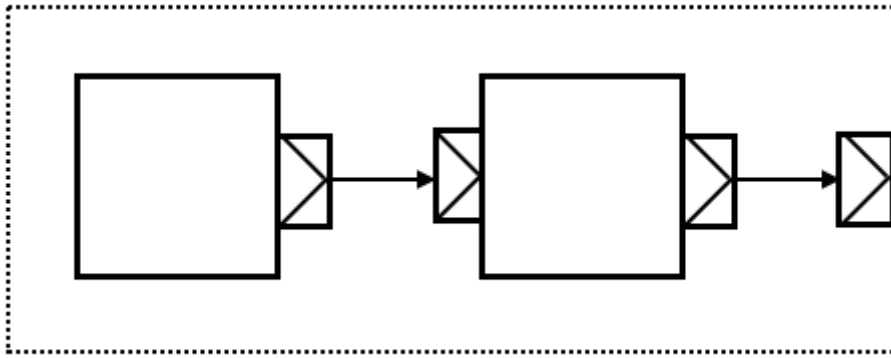
- Graph has all on ports accounted
- Nothing else can be connected
- Potential to be **RunnableGraph**



## Source Shape

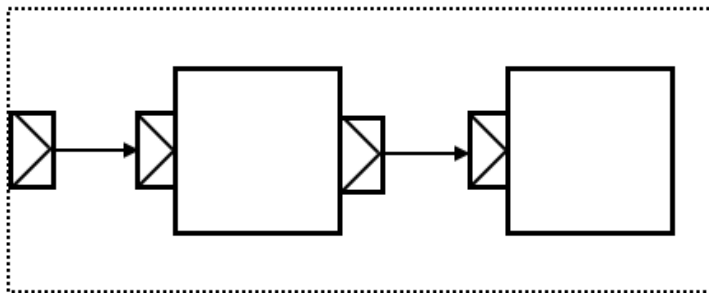
- Contains one open output port

- Requires other components to be closed and runnable



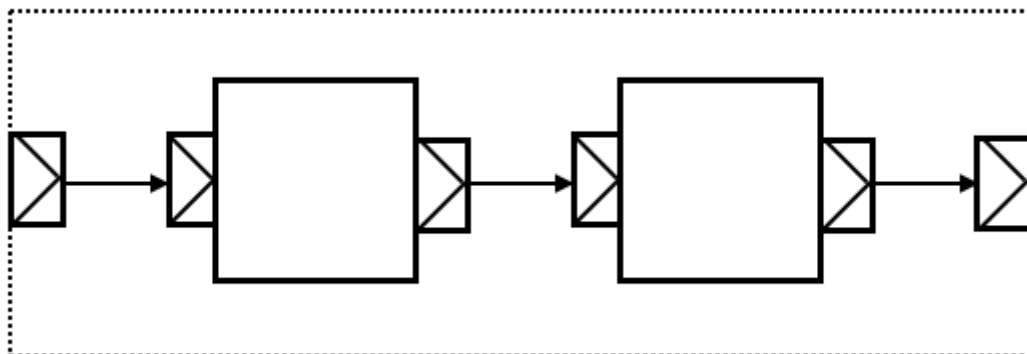
## Sink Shape

- Contains one input shape, outputs are closed
- Requires other components to be closed and runnable



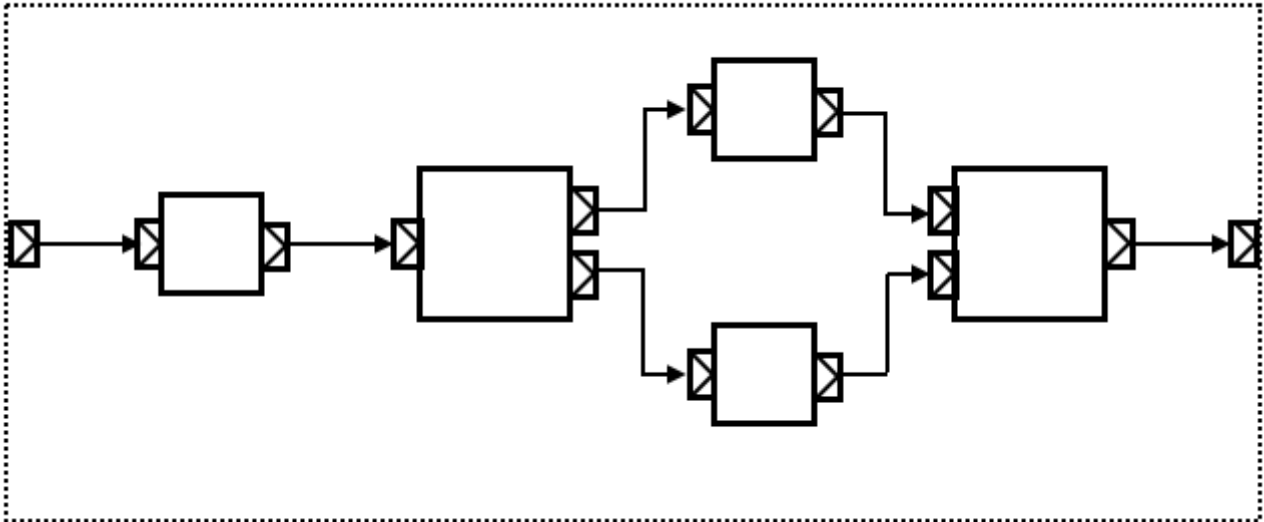
## Flow Shape

- Contains one input, one output
- Composed of other components



# Advanced Shapes

- All shapes can be complicated with more components
- You can determine how advanced or simple the graph can be



## Constructing a Simple Graph

- The following graph has components constructed on the outside
  - `Source`
  - `Sink`
- `RunnableGraph.fromGraph` prepares the graph
- `builder` will allow us to plugin components to be used within the DSL
- `import GraphDSL.Implicits._` will allow DSL connectors like `~>`
- `ClosedShape` indicates the return shape, in this simple instance it is closed

```
val source = Source(1 to 100)
val sink = Sink.foreach[Int](println)

var runnableGraph: RunnableGraph[NotUsed] = RunnableGraph.fromGraph(GraphDSL.create()
{
    implicit builder =>
        import GraphDSL.Implicits._
        source ~> sink
        ClosedShape
})

runnableGraph.run()
```



# Creating a Denser Graph

- In the following Graph:
  - `outputToTwoFiles` is a composed `Graph` of a `Broadcast`
  - `Broadcast[Int](2)` states that there is one input and two output scheme
  - `int2ByteString` and `int2ByteString2` are flows that convert to `ByteString`
  - `ByteString` is wrapper to send `String` over the wire
  - This returns a `SinkShape`

```
val source = Source(1 to 100)

val outputToTwoFiles: Graph[SinkShape[Int], NotUsed] = GraphDSL.create() { implicit b
=>
  import GraphDSL.Implicits._

  val fileSink1 = b.add(FileIO.toPath(Paths.get(s"$userHome/complete1.txt")))
  val fileSink2 = b.add(FileIO.toPath(Paths.get(s"$userHome/complete2.txt")))
  val int2ByteString1, int2ByteString2 = b.add(Flow[Int].map(x => ByteString(x)))
  val splitter = b.add(Broadcast[Int](2))

  splitter ~> int2ByteString1 ~> fileSink1.in
  splitter ~> int2ByteString2 ~> fileSink2.in

  SinkShape.apply(splitter.in)
}

source.runWith(outputToTwoFiles)
```

## Lab: Running the graphs

**Step 1:** In the akka-streams-study project, and in the `src/test/scala` folder and in the `com.xyzcorp.akka.streams` package, locate the "Perform a test with an async boundary which will run on a separate actor and dispatcher" test in `GraphStreamSpec.scala`

**Step 2:** Run each method, in SBT, using the following which will only run the single test. Each test is labeled with the prefix "Case 1:", "Case 2:", "Case 3:" for ease

```
> testOnly com.xyzcorp.akka.streams.SimpleStreamSpec -- -z "Case 1:"
```

**Step 3:** Disassemble and discover types to understand how all of it fits together. Introduce explicit types to see how it all fits together and try different combinations

# Kill Switches

## Kill Switch

- A **Kill Switch** allows the programmer to:
  - complete the graph via `shutdown()`
  - fail the graph via `abort(Throwable t)`
- A **Kill Switch**
  - Can control the completion of one or multiple streams
  - Comes in two flavors:
    - `UniqueKillSwitch`
    - `SharedKillSwitch`

## UniqueKillSwitch

- Always a result of a materialization using `viaMat`
- Allows to control and force the completion of one materialized **Graph** of **FlowShape** == Lab: `UniqueKillSwitch`

**Step 1:** In the akka-streams-study project, and in the `src/test/scala` folder and in the `com.xyzcorp.akka.streams` package, and in the `KillSwitchSpec` locate "Case 1", "Case 2" tests.

**Step 2:** Run the method, in SBT using the following which will only run the single test

```
> testOnly com.xyzcorp.akka.streams.KillSwitchSpec -- -z "Case 1"
```

```
> testOnly com.xyzcorp.akka.streams.KillSwitchSpec -- -z "Case 2"
```

**Step 3:** Disassemble and discover types to understand how all of it fits together. Introduce explicit types to see how it all fits together and try different combinations

Run and review *Case 1* and *Case 2* in the `KillSwitchSpec.scala` in `src/test/scala`

## SharedKillSwitch

- Needs to be constructed before any materialization
- Controls the completion of an arbitrary number graphs of **FlowShape**
- It can be materialized multiple times via its `flow` method
- All materialized graphs linked to it are controlled by the switch

## Lab: SharedKillSwitch

**Step 1:** In the akka-streams-study project, and in the *src/test/scala* folder and in the *com.xyzcorp.akka.streams* package, and in the *KillSwitchSpec* locate "Case 3", "Case 4" tests.

**Step 2:** Run the method, in SBT using the following which will only run the single test

```
> testOnly com.xyzcorp.akka.streams.KillSwitchSpec -- -z "Case 3"
```

```
> testOnly com.xyzcorp.akka.streams.KillSwitchSpec -- -z "Case 4"
```

**Step 3:** Disassemble and discover types to understand how all of it fits together. Introduce explicit types to see how it all fits together and try different combinations