Akka

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 Scala shells are represented as `scala>`

```
scala> 4 * 10
```

Introduction to Akka

About Akka

- Set of libraries used to create concurrent, fault-tolerant and scalable applications.
- It contains many API packages:
  - Actors
  - Logging
  - Futures
  - Finite State Machines
  - Persistence
  - Clustering
- 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
- Most components are open source, with some proprietary components

History of Actors

- Developed by Carl Hewitt, Peter Bishop, and Richard Steiger in 1973 paper: *"Universal Modular Actor Formalism for Artificial Intelligence."*
- Based on the Actor Model from Erlang.

Actor Model Tenets

- Encapsulates State and Behavior
- Internally we focus as if ***everything is single threaded***
- Communication between other `Actor` is done with message passing
- Is the center architecture of Akka and it's libraries

To implement the Actor Model

- All computation is performed within an actor
- Actors can communicate only through messages
- In response to a message, an actor can:
  - Change its state or behavior
  - Send messages to other actors
  - Create a finite number of child actors

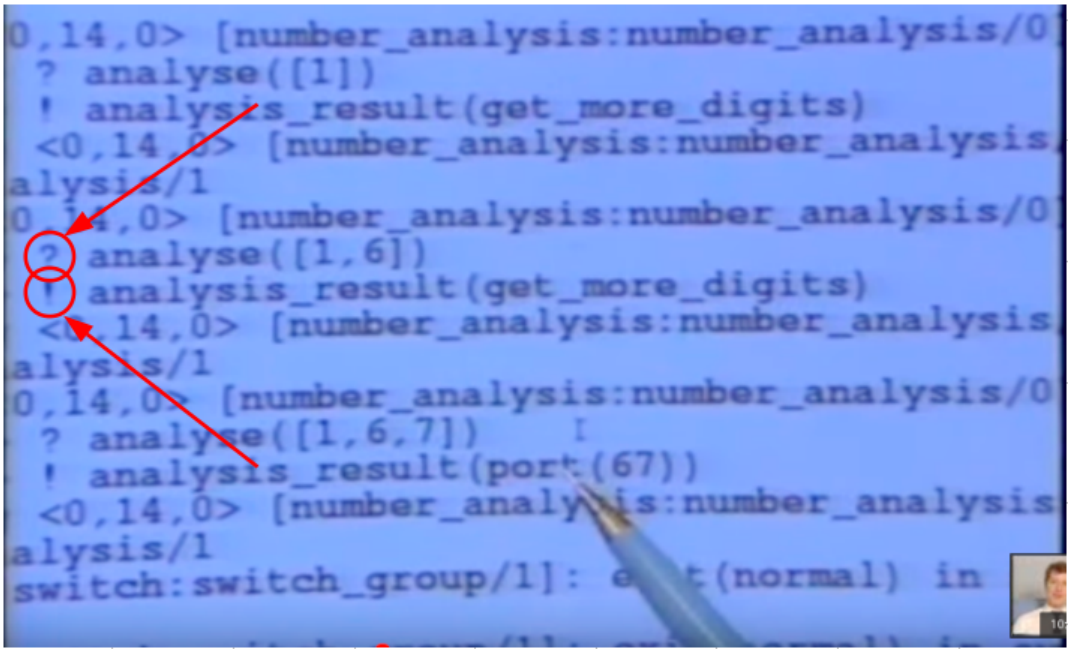
Actor Model in Action

▶ ./images/akka_basic_video.mp4 *(video)*

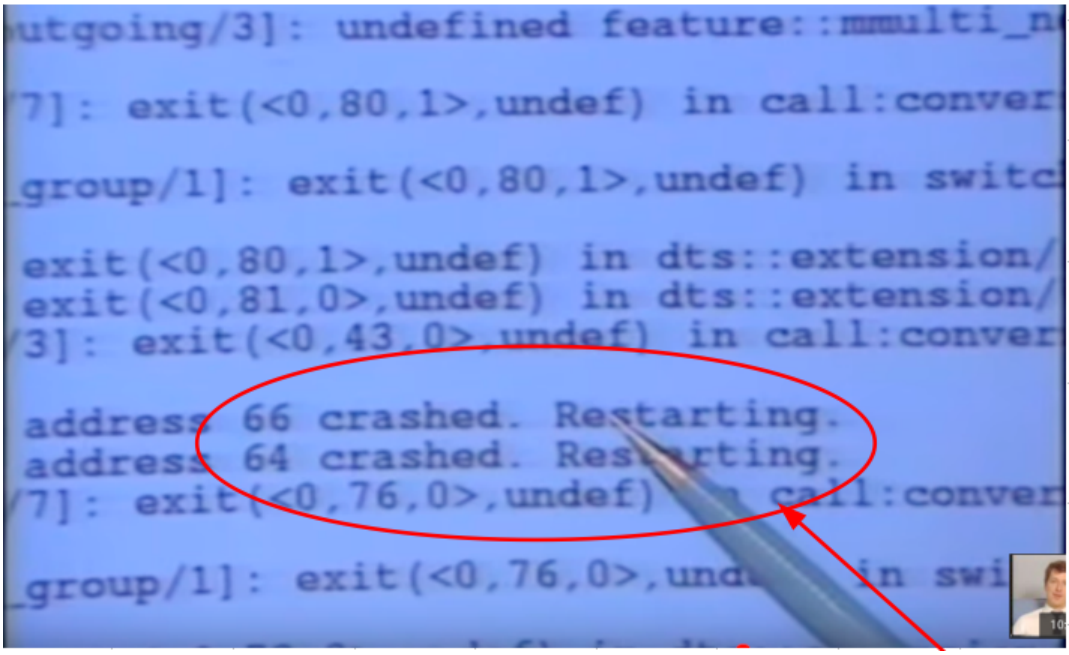
Actors in Erlang

Erlang the Movie

Actors in Erlang Sending Messages



Actors in Erlang Fault Tolerance



Actor Strategies

- An Actor can then respond to the message by:
  - Processing the message
  - Passing it on
  - Act as a proxy before sending to another

All these are transparent to the sender

Difference between Actors and other Messaging Queues

- Transactional Persistent Queues may or may not be implemented in standard MQ*
- If you going to code in JVM, Akka is a great choice
- If you going to require polyglot integration than an MQ
- Akka will scale better than an MQ, since MQ will require more fixed threads*
- Message Queues have better guarantees

This is ***not*** a sales pitch, and it is tough to compare every MQ with Akka

Many large companies use both MQ and Akka

Akka vs. Message Queues

Here is a table comparing the two different models.

Akka	Generic MQ
Multiple Consumers	Polyglot Solutions
Low Latency	Better Message Reliability (Higher latency)
Automatic Concurrency Management	Some may offer different takes on architecture (Kafka)
Better Interop with AkkaHTTP and Play	Better Integration with a particular framework

Setup

Typical Setup Instructions

Before we begin it is assumed that all of you have the following tools installed:

- JDK 1.8 (latest java is 1.8.0_162)
- Scala (latest scala is 2.12.6)
- SBT (latest sbt is 1.1.6)

To verify that all your tools work as expected

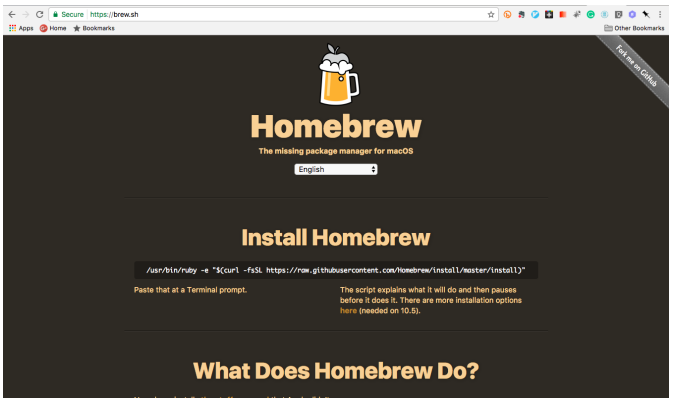
```
% javac -version
javac 1.8.0_162

% scala -version
Scala code runner version 2.12.6 -- Copyright 2002-2018, LAMP/EPFL

% java -version
java version "1.8.0_162"
Java(TM) SE Runtime Environment (build 1.8.0_1.8.0_162-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

% sbt sbtVersion
[info] Set current project to scala (in build file:/<folder_location>)
[info] 1.1.6
```

Installing Java, Scala, SBT on a Mac Automatically with Brew



If you have a mac and brew installed, you can run the following ***and be done!***:

```
% brew update
% brew cask install java
% brew install scala
% brew install sbt
```

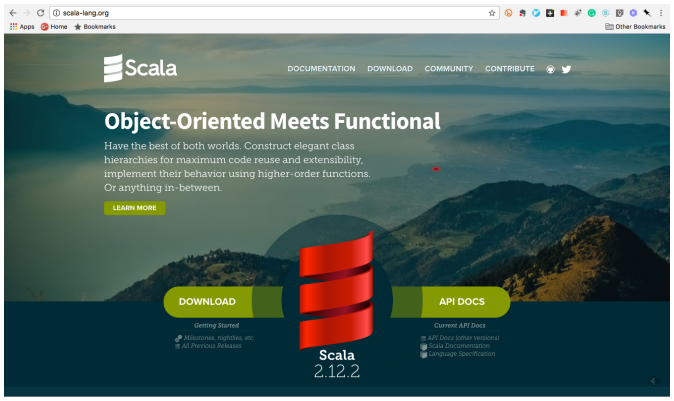
This will require an install of Homebrew. Visit https://brew.sh/ for details of installation if you want to use brew.

Depending on your company's software and security constraints, you may not be able to use brew

If you don't have Java 8 installed

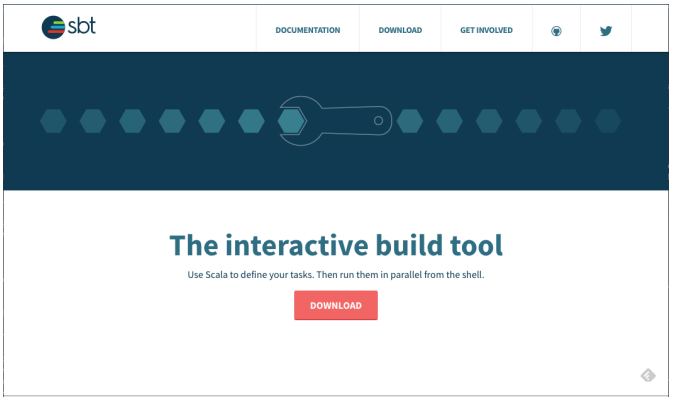
- Visit: http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html
- Select: *Accept License Agreement*
- Download the appropriate Java version based on your architecture.

If you do not have Scala installed



- Visit http://scala-lang.org
- Click the *Download* Button
- Download the appropriate binary for your system:
  - Mac and Linux will load a *.tgz* file
  - Windows will download an *.msi* executable
- For Mac and Linux you can expand with `tar -xvfz scala-2.12.6.tgz`

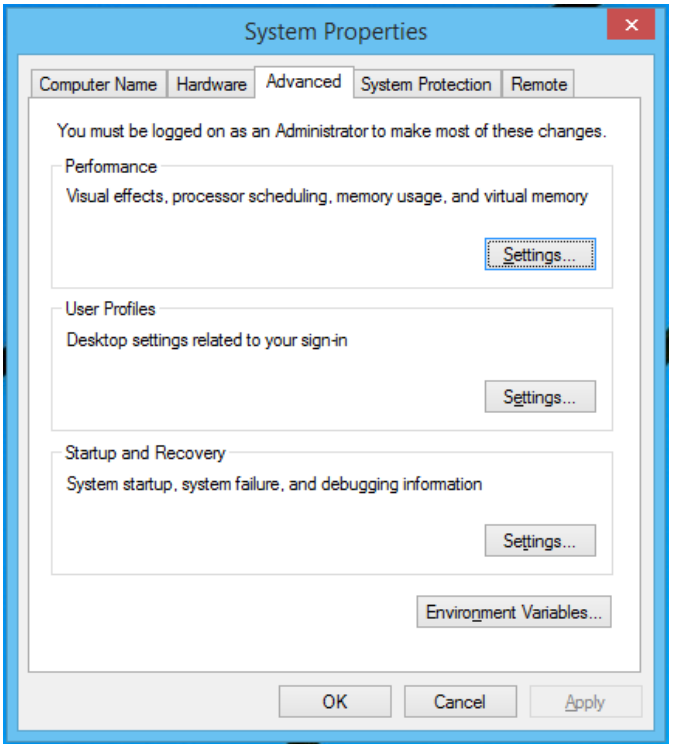
If you do not have SBT installed

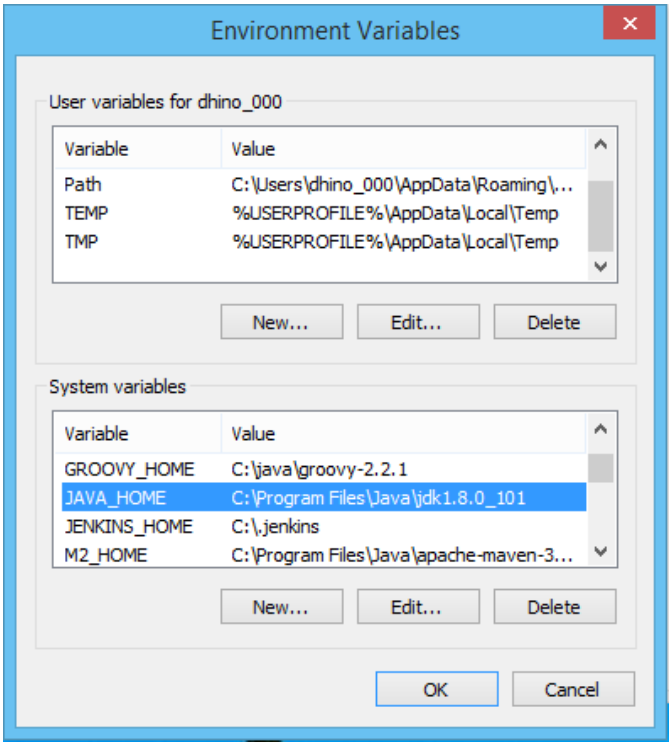


- Visit http://scala-sbt.org
- Click the *Download* Button
- Download the appropriate binary for your system:
  - Mac and Linux will load a *.tgz*, or a *.zip* file
  - Windows will download an *.msi* executable
- For Mac and Linux you can expand with `tar -xvfz scala-2.12.6.tgz`

Windows Users Only: Setting up the Windows Environment Variables for Java

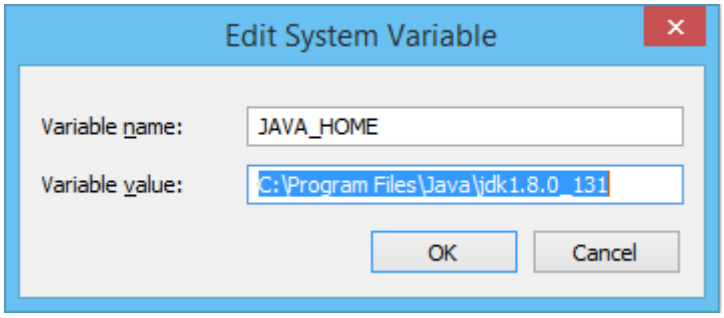
- Go to your *Environment Variables*, typically done by typing the Windows key(⊞) and type `env`





Windows Users Only: Setting up `JAVA_HOME`

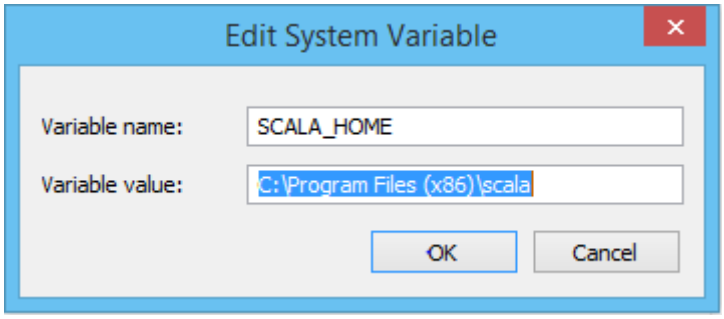
- Edit `JAVA_HOME` in the System Environment Variable window with the location of your JDK



Using `jdk1.8.0_131` in the image. Your version may vary.

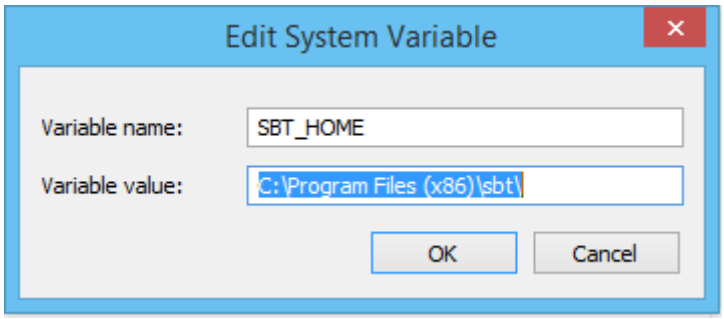
Windows Users Only (Optional): Setting up `SCALA_HOME`

- This setting is not necessary with Scala on Windows since the .msi file installs everything required
- If you do have problems where a tool is unable to locate Scala, set up an environment variable `SCALA_HOME`



Windows Users Only: Setting up `SBT_HOME`

- This setting is not necessary since SBT on Windows since the .msi file installs everything required
- If you do have problems where a tool is unable to locate SBT, set up an environment variable `SBT_HOME`



Windows Users Only: Setting up `PATH`

- Once you establish `JAVA_HOME`, possibly `SCALA_HOME`, ***append*** to the `PATH` setting the following:

```
;%JAVA_HOME%\bin;%SCALA_HOME%\bin
```

Windows Users Only: Restart All Command Prompts And Try Again

```
C:\> javac -version
javac 1.8.0_162

C:\> scala -version
Scala code runner version 2.12.6 -- Copyright 2002-2018, LAMP/EPFL

C:\> java -version
java version "1.8.0_162"
Java(TM) SE Runtime Environment (build 1.8.0_1.8.0_162-b17)
Java HotSpot(TM) 64-Bit Server VM (build 25.65-b01, mixed mode)

C:\> sbt sbtVersion
[info] Set current project to scala (in build file:/<folder_location>)
[info] 1.1.6
```

Changes won't take effect until you open a new command prompt!

Mac Users Only: Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using `nano`

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

- Make sure the following contents are in your *.bash_profile*
- If you already have a `PATH`, append the new values to the end.

```
export SCALA_HOME= <location_of_scala>
export SBT_HOME= <location_of_sbt>
export JAVA_HOME=$(/usr/libexec/java_home)
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$SBT_HOME/bin
```

If you used `brew`, many of these application will not require their `PATH` setup.

You can locate where `scala` is by either doing

```
% which scala
% whereis scala
```

When done open a new terminal or if already on an open terminal type:

- For bash: **`source .bash_profile`**
- For zsh: **`source .zshrc`**

Linux Users Only: Editing your *.bash_profile* or *.zshrc*

- If you are using the Bash shell, edit the your *.bash_profile* in your home directory using your favorite editor
- If you are using the Zsh shell, edit the your *.zshrc* in your home directory using your favorite editor

For example, if using `nano`

```
% nano ~/.bash_profile
```

Replace *nano* with your favorite editor *vim*, *emacs*, *atom*, etc.

- Make sure the following contents are in your *.bash_profile*
- If you already have a `PATH`, append the new values to the end.

```
export SCALA_HOME= <location_of_scala>
export SBT_HOME= <location_of_sbt>
export JAVA_HOME= <location_of_jdk>
export PATH=$PATH:$JAVA_HOME/bin:$SCALA_HOME/bin:$SBT_HOME/bin
```

When done open a new terminal or if already on an open terminal type:

- For bash: **`source .bash_profile`**
- For zsh: **`source .zshrc`**

Basic API

Akka Messages

Akka Messages

- Everything in an actor model is message based
- Messages are sent asynchronously to other actors

Akka Message Rules

- To maintain thread encapsulation, messages to and from actors ***cannot*** be:
  - Mutable
  - Closure

What is a closure in Java?

```
import java.util.function.Function;

public class Closures {

  public static Integer foo(Function<Integer, Integer> w) {
     return w.apply(5);
  }

  public static void main(String[] args) {
      Integer x = 3;
      Function<Integer, Integer> y = (Integer z) ->  x + z;
      System.out.println(foo(y)); //8
  }
}
```

What is a closure in Scala?

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

What is a immutability in Java?

```
public class Person {
    private final String firstName;
    private final String lastName;
    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public String getLastName() {
        return lastName;
    }
    @Override
    public String toString() {...}

    @Override
    public boolean equals(Object o) {...}

    @Override
    public int hashCode() {...}
}
```

What is immutability in Scala?

```
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 = {...}
}
```

The Mailbox

- Every actor contains a mailbox
- The mailbox exists with the actor
- Once in the mailbox it is outside the control of the sender
- The Actor Model provides an *At Most Once Guarantee*
- There are other Akka tools that provide *At Least Once Delivery*
- **IMPORTANT** No actor can receive more than one message at a time!
- This is called the *single threaded illusion*

The `ActorSystem`

- An actor system is a hierarchical group of actors which share common configuration
  - Dispatchers
  - Deployments
  - Remote capabilities
  - Addresses.
- It is also the entry point for ***creating or looking up actors***
- Analogous to a server

`ActorSystem` examples

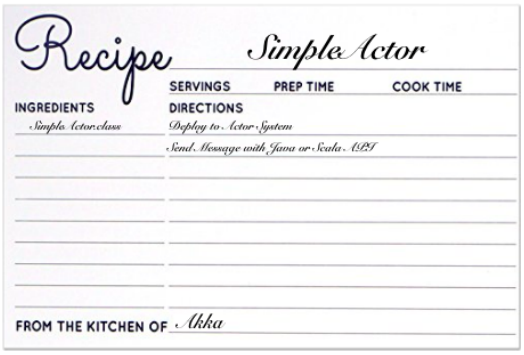
Scala:

```
val system = ActorSystem("MySystem")
```

Java:

```
ActorSystem system = ActorSystem.create("MySystem");
```

A `Prop`



- Configuration class to specify how to create recipe
- Analogous to a recipe
- Immutable class

`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`, `ActorFor` 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 `ActorRef`

- 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.

Other Types of References

- Depending how you configure Akka, you might have a different kind of `ActorRef` implementations
  - Pure Local Actor References
  - Local Actor References
  - Local Actor References with Routing
  - Remote Actor References
  - Promise Actor References
  - Dead Letter Actor References
  - Cluster Actor References

Untyped Actors

Setting up your first actor in Java

```
public class SimpleActorJava extends AbstractActor {
    private LoggingAdapter log =
       Logging.getLogger(getContext().system(), this);

    @Override
    public Receive createReceive() {
        return receiveBuilder().match(String.class, message -> {
            System.out.println(Thread.currentThread().getName());
            log.info("Received String message in Simple Actor Java {}",
                    message);
        }).build();
    }
}
```

- Untyped Actors in Java extend from `AbstractActor`
- Actors are infused now with Java 8 Lambdas
- This is a newer API than from the past

Setting up your first actor in Java Continued

```
public class SimpleActorJava extends AbstractActor {
    private LoggingAdapter log =
            Logging.getLogger(getContext().system(),
                    this);

    @Override
    public Receive createReceive() {
        return receiveBuilder().match(String.class, message -> {
            System.out.println(Thread.currentThread().getName());
            log.info("Received String message in Simple Actor Java {}",
                    message);
        }).build();
    }
}
```

- Logging is provided by `akka.event.LoggingAdapter`
- Pattern matching in Java is provided now by `receiveBuilder()`
- A `match` will offer many different variants
- `getContext` is a reference call to get the `ActorSystem` in which the actor is housed

Creating an `ActorSystem` with an `Actor`

```
ActorSystem actorSystem = ActorSystem.create("My-Actor-System");
Props props = Props.create(SimpleActorJava.class);
ActorRef actorRef = actorSystem.actorOf(props);
actorRef.tell("Hello", ActorRef.noSender());
```

- An `ActorSystem` is a container for multiple `Actor`
- A `Props`
  - A recipe as to how to create an `Actor`
  - Is a message and can be used to send to other actors
- `actorSystem.actorOf` creates the actor
- `actorRef.tell` sends the message to an actor in a fire and forget fashion
- This will run locally on a single JVM

Lab: Setting up a project

Step 1: Ensure that Java, Scala, and SBT is installed

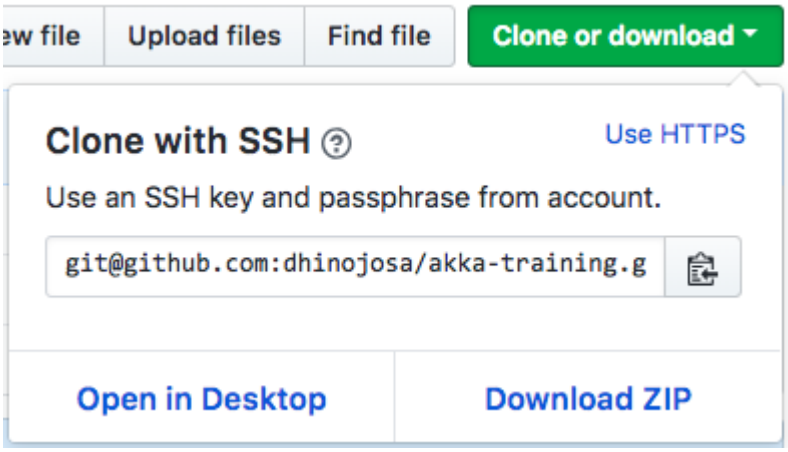
Step 2: Clone akka-training, from the github repository: https://github.com/dhinojosa/akka-training

- Select the "Clone or Download" Button
- Select SSH or HTTPS
- run `git clone <url>`

or

Step 2: Download the project from the github repository: https://github.com/dhinojosa/akka-training by clicking on the "Clone or Download" button

- Select the "Clone or Download" Button
- Click Download
- Unzip the project



The pdf, and zip of the slides for this course are also available in the repo.

Lab: Setting up a project continued

Step 3: In the akka-training folder, run the following command to download dependencies

```
akka-training % sbt reload update
```

This will take a while, and you can ignore initial warnings, particularly if running Java 9 or higher

Bringing your project into your IDE

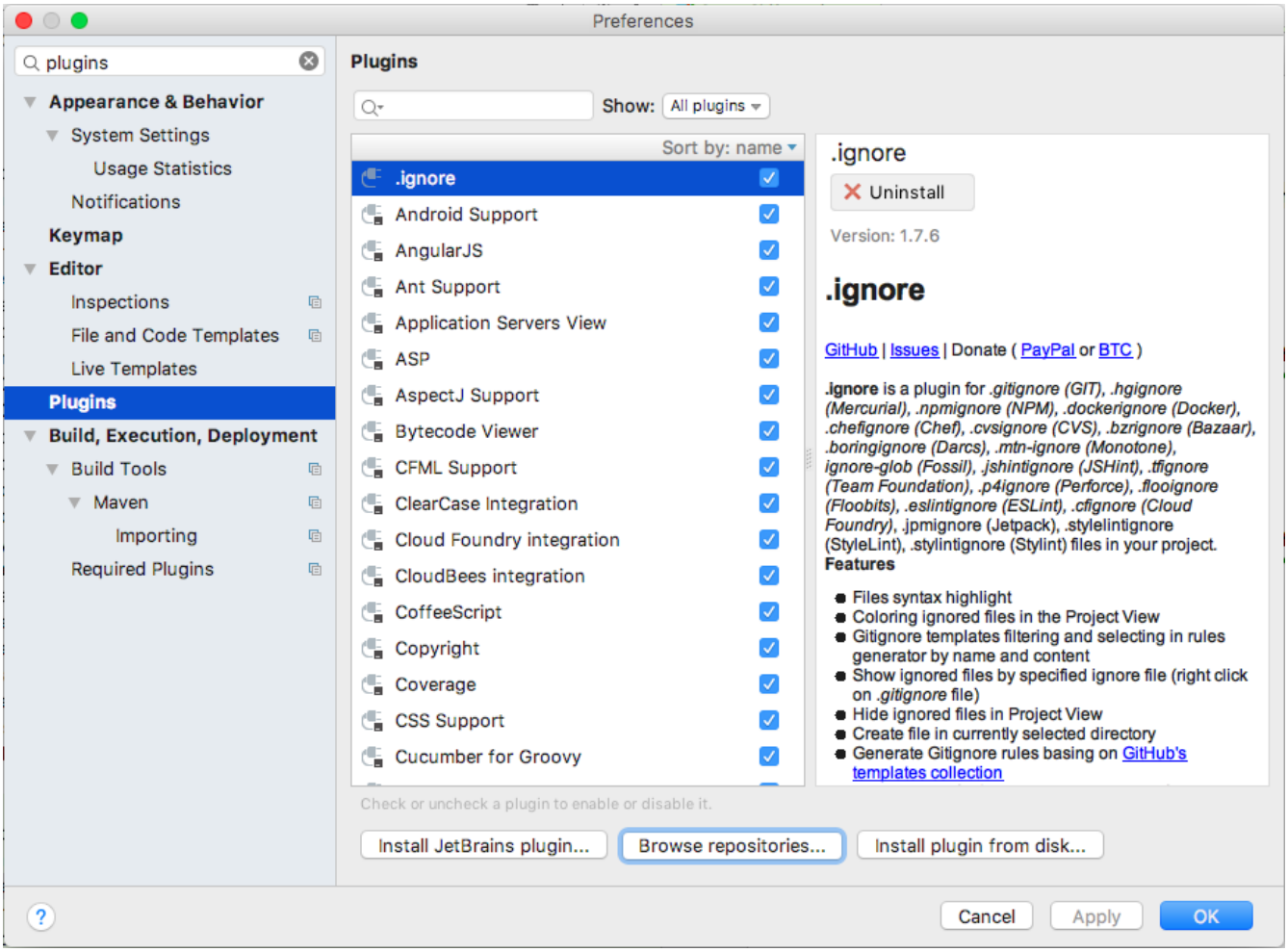
- **IMPORTANT Be sure to download and configure the latest version of your IDE!**
- Eclipse - Be sure to have Neon or Oxygen (Oxygen 3 Preferred)
- IntelliJ IDEA (Professional or Community): 2018-1
- **IMPORTANT Be sure to backup any IDE settings that you believe are critical**

Lab (For IntelliJ Users): Install the IntelliJ Scala Plugin

IntelliJ from JetBrains offers support for Scala through one of its plugins

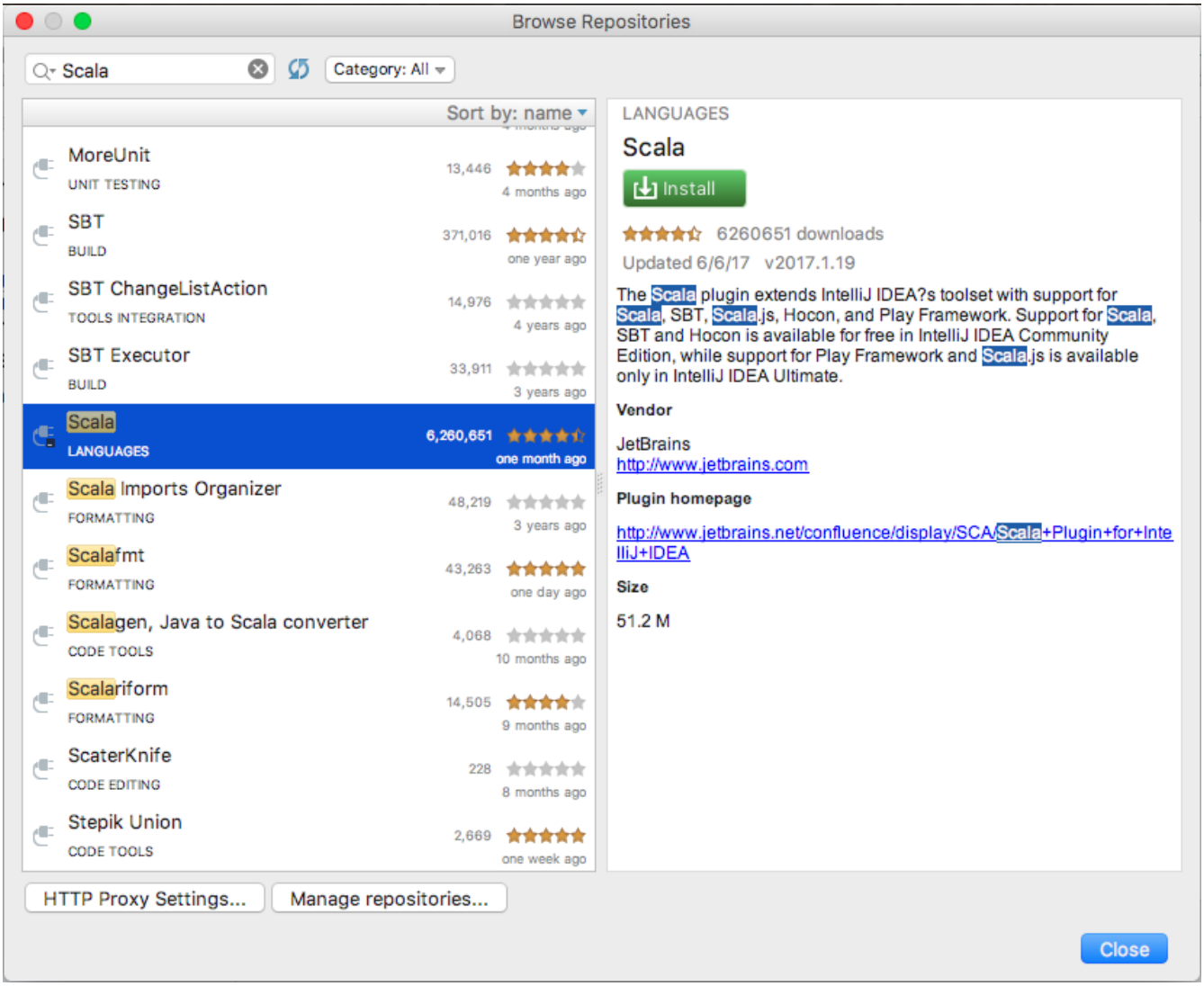
Step 1: Go to:

- For Windows and Linux: File | Settings | Plugins
- For MacOSX: IntelliJ IDEA | Preferences | Plugins
- Click on Browse Repositories



Lab (For IntelliJ Users): Install the IntelliJ Scala Plugin Continued

Step 2: Search for Scala in the filter on the upper-left hand corner



Step 3: Click *Install* on the right pane window

Step 4: Restart is necessary

Do not install the SBT plugin, the IntelliJ Plugin already has SBT support

Lab (For IntelliJ Users): Open your project in IntelliJ Continued

Step 1: Go to:

- File | Open
- Locate your project, akka-training
- Click the Open button

Eclipse

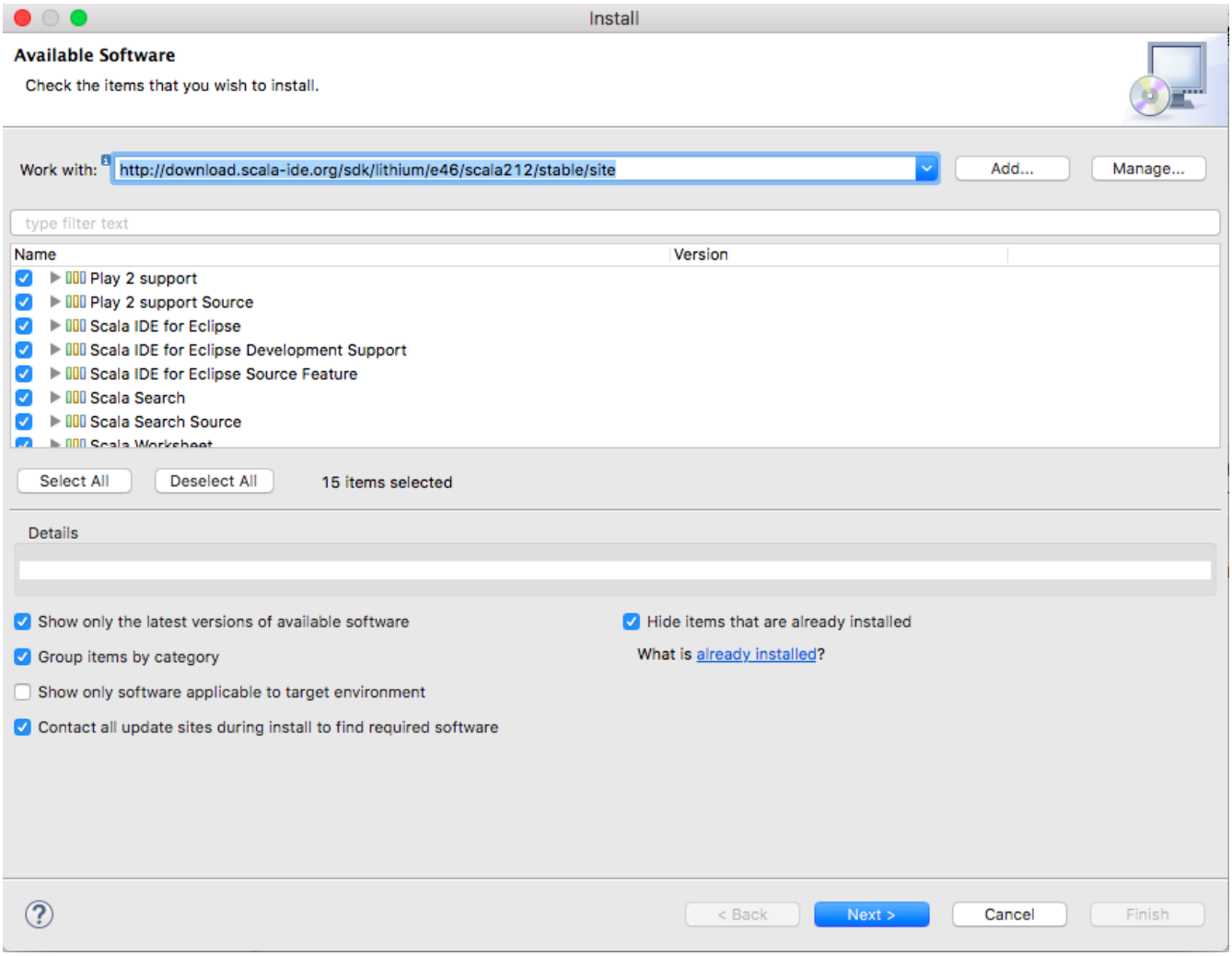
- Be sure to have one of the latest Eclipse installation as plugins tend to not work with older

plugins

- If you do not have any Eclipse you can just go straight to http://scala-ide.org and download a complete bundle and skip the Eclipse setup and jump to ***Installing the SBT Eclipse Plugin***

Optional Lab (Eclipse Users Only): Installing the Scala-IDE Plugin

If you wish to install the Scala-IDE plugin manually instead of downloading the complete Scala-IDE package…



Step 1: Go to Help | Install New Software

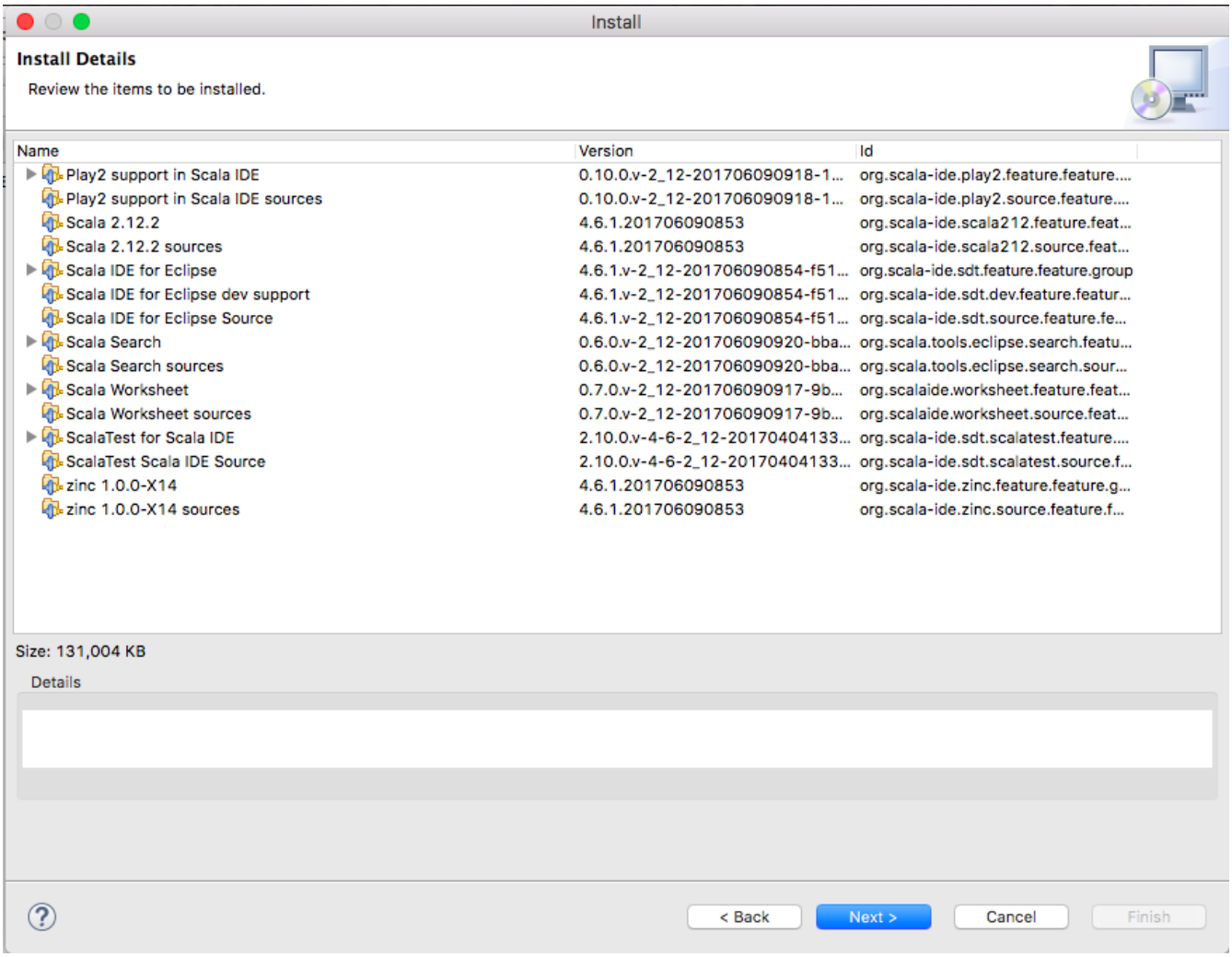
Step 2: In the *Work With:* field enter the URL of the Scala IDE Plugin found in http://scala-ide.org/download/current.html

Step 3: Hit Enter, and select all the Scala modules that appear

Step 4: Hit Next

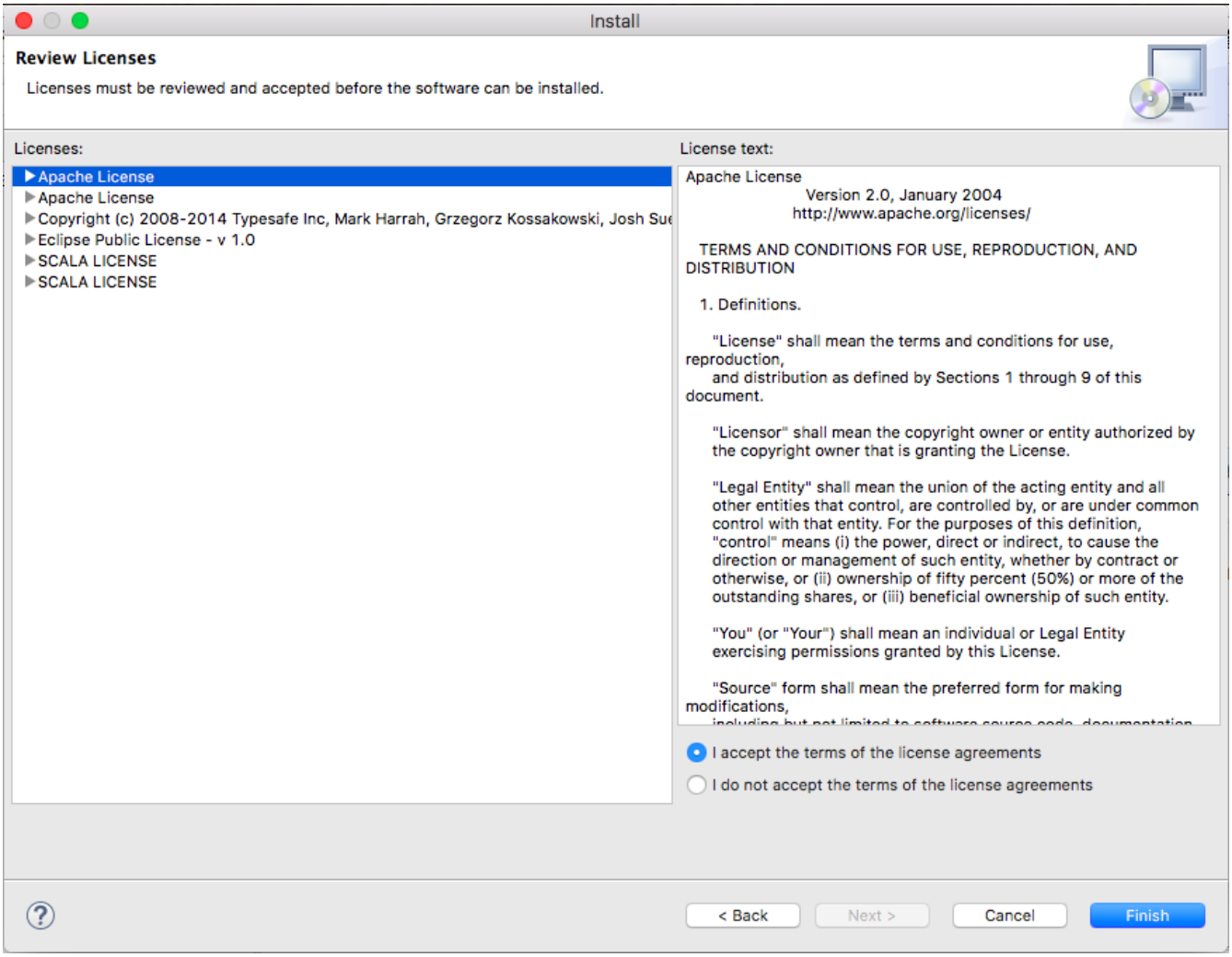
Optional Lab (Eclipse Users Only): Install Details

If you wish to install the Scala-IDE plugin manually instead of downloading the complete Scala-IDE package…



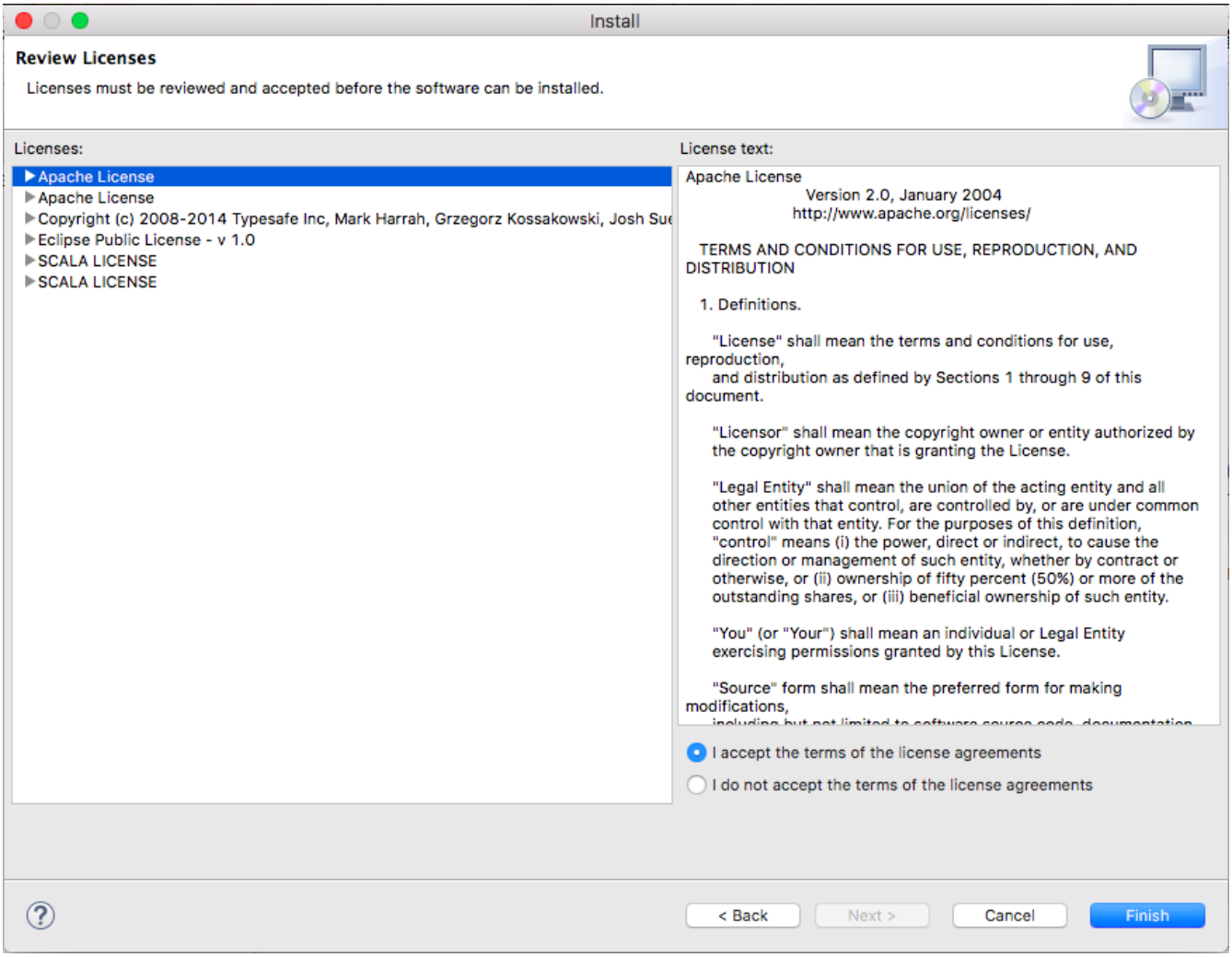
Step 1: Ensure the details for the plugin and hit *Next*

Optional Lab (Eclipse Users Only): License Details



Step 1: Ensure the details for the licenses used and hit *Next*

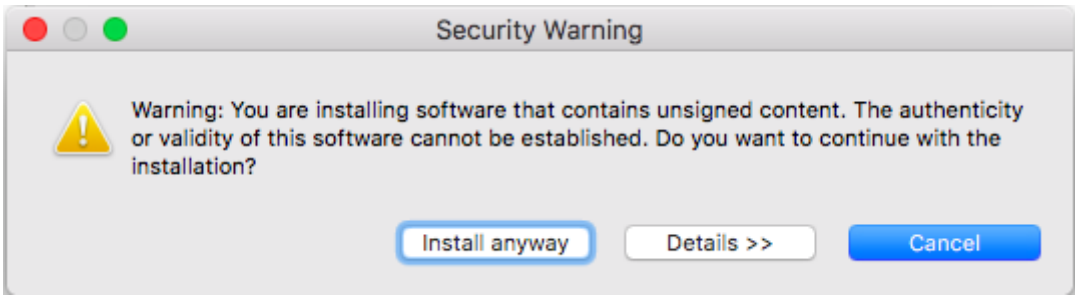
Optional Lab (Eclipse Users Only): Unsigned Content



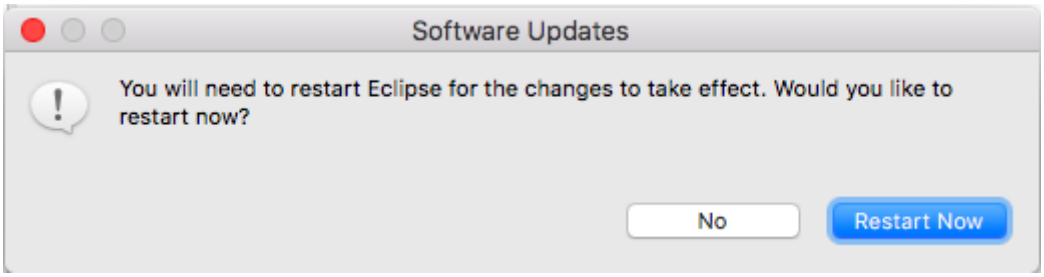
Step 1: Ensure the details for the licenses used and hit *Next*

Optional Lab (Eclipse Users Only): Last Details

Step 1: An image may appear warning about unsigned content, choose *Install Anyway*



Step 2: When prompted to Restart Machine, select *Restart Now*.



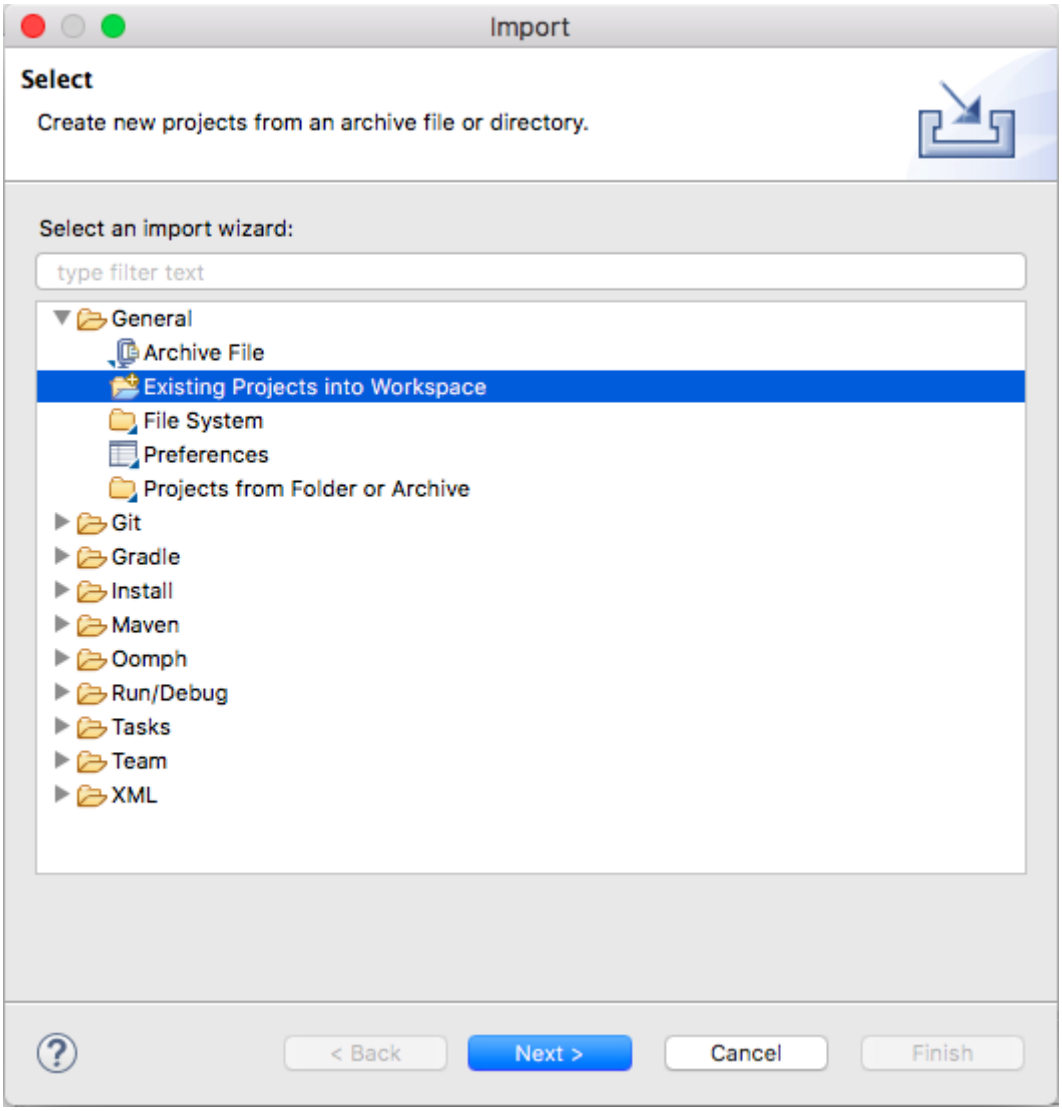
Lab (Eclipse Users Only): Generating Eclipse Files

Step 1: Enter the `sbt` shell and enter Eclipse.

```
> eclipse
```

After you invoke this command in SBT, this will create all the eclipse files needed.

Lab (Eclipse Users Only): Importing into Eclipse



In Eclipse, in the menus, go to:

Step 1: File | Import

Step 2: Select: General | Existing Project in Workspace

Step 3: Click Next

Lab (Eclipse Users Only): Selecting the Project

Step 1: In *Select Root Directory* locate your project

Step 2: Select *Finish*

Lab: Creating an Actor in Java

Step 1: In *src/main/java*, create a package called `com.akkatraining.java`

Step 2: Create `SimpleActorJava` in `com.akkatraining.java` with the following:

```
package com.akkatraining.java

public class SimpleActorJava extends AbstractActor {
    private LoggingAdapter log =
            Logging.getLogger(getContext().system(),
                    this);

    @Override
    public Receive createReceive() {
        return receiveBuilder().match(String.class, message -> {
            System.out.println(Thread.currentThread().getName());
            log.info("Received String message in Simple Actor Java {}",
                    message);
        }).build();
    }
}
```

Lab: Creating an `ActorSystem` in a main method:

Step 1: In the `SimpleActorJava` that has just been created, create a `public static void main` method to create an `ActorSystem`

```
package com.akkatraining.java

public class SimpleActorJava extends AbstractActor {

    //...Previous Content Elided

    public static void main(String[] args) extends Exception {
        ActorSystem actorSystem =
           ActorSystem.create("My-Actor-System");
        Props props = Props.create(SimpleActorJava.class);
        ActorRef actorRef = actorSystem.actorOf(props);
        actorRef.tell("Hello", ActorRef.noSender());
    }
}
```

Lab: Running through SBT

Step 1: Run the main method either in your IDE or in SBT by running `run` in SBT, or in the command line, and selecting the class that you want to run:

```
% sbt run
```

or

```
% sbt
> run
```

Step 3: Manually stop the server by stopping the VM in your IDE , is you used your IDE, or Control+D in SBT

Step 4: Discuss the results

Step 5: `actorOf` also takes a name as a parameter, add one, discuss the results

Shutting down gracefully

- Last lab we shut down harshly
- We can programatically shut down gracefully if we call `terminate`
- Remember any non-daemonic thread in the JVM will prevent the JVM from shutting down
- Typically since this is usually when an application is terminating as will we will block and wait for the termination to end
- **IMPORTANT** Blocking is frowned upon in the reactive architectures except in rare cases like this, shutting down
- `actorSystem.terminate` returns a `Future` that can be used to determine when shutting down.

```
Await.result(actorSystem.terminate(),
   Duration.create(1, TimeUnit .SECONDS));
```

Lab: Terminate an `ActorSystem`

Step 1: In `SimpleActorJava`, create a terminate the `ActorSystem` and await the shutdown

Step 2: Handle any `Exception` that might be thrown

Step 3: Verify by running the example in either your IDE or in the command line

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 with 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 `canceled()` 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

Lab: Review how a `Future` behaves

Step 1: In your favorite IDE, and in the *src/main/java* directory and in the `com.akkatraining.java.future` package review `BasicFuturesTest`

Step 2: Now, review `AsychronousFuturesTest`, and note the difference with `BasicFuturesTest`

Step 3: Run both in SBT (or your IDE). As a review, run either:

```
% sbt run
```

or enter the `sbt` shell and run from there:

```
% sbt
> run
```

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 and Akka Streams API

`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);
```

`thenApplyAsync`

- `thenApplyAsync` will run the next stage of the future onto another thread given an `Executor`
- This provides asynchronous boundaries to process information

```
CompletableFuture<String> thenApplyAsync =
        integerFuture1.thenApplyAsync(x -> {
            System.out.println("In Block:" +
                    Thread.currentThread().getName());
            return "" + (x + 19);
        });

thenApplyAsync.thenAcceptAsync((x) -> {
    System.out.println("Accepting in:" + Thread.currentThread().
getName());
    System.out.println("x = " + x);
}, executorService);
```

Inlining `thenApplyAsync`

```
integerFuture1.thenApplyAsync(x1 -> {
    System.out.println("In Block:" +
            Thread.currentThread().getName());
    return "" + (x1 + 19);
}).thenAcceptAsync((x) -> {
    System.out.println("Accepting in:" + Thread.currentThread().
getName());
    System.out.println("x = " + x);
}, executorService);
```

Running a process after the Future Completion

- `thenRun`
  - Will ignore the value of the `CompletableFuture`
  - Proceed with a `Runnable` to finalize any processes

```
integerFuture1.thenRun(new Runnable() {
    @Override
    public void run() {
        String successMessage =
                "I am doing something else once" +
                        " that future has been triggered!";
        System.out.println
                (successMessage);
        System.out.println("Run inside of " +
                Thread.currentThread().getName());
    }
});
```

Inlining the above:

```
integerFuture1.thenRun(() -> {
    String successMessage =
            "I am doing something else once" +
                    " that future has been triggered!";
    System.out.println
            (successMessage);
    System.out.println("Run inside of " +
            Thread.currentThread().getName());
});
```

Handling Exceptions in `CompletableFuture`

- There are two methods that can be used with a `CompletableFuture` to handle exceptions
  - `exceptionally`
  - `handle`

`exceptionally`

- In the example below assume that `stringFuture` will return a `String`
- `exceptionally` will trap any exception and proceed down the process
- This will not interrupt the process and provide an alternative

```
stringFuture1
   .thenApply(Integer::parseInt)
   .exceptionally(t -> return -1})
   .thenAccept(System.out::println);
```

handle

- In the example below assume that `stringFuture` will return a `String`
- `handle` will trap any and any exception and proceed down the process
- This will not interrupt the process and provide an alternative in case of `Exception`

```
stringFuture1
    .thenApply(Integer::parseInt)
    .exceptionally(t -> return -1})
    .thenAccept(System.out::println);
```

compose

- `compose` is analogous to `flatMap` with `CompletableFuture`
- `flatMap`
  - `map` has the signature `coll.map(x → y)`
  - `flatMap` has the signature `coll.map(x → coll(y))`
  - Where, `coll` is any collection or container like `CompletableFuture`

Given:

```
public CompletableFuture<Integer> getTemperatureInFahrenheit(final
String cityState) {
    return CompletableFuture.supplyAsync(() -> {
        //Contact Web Service and parse for temperature
    });
}
```

Then:

```
stringFuture1.thenCompose(
    cityState -> getTemperatureInFahrenheit(cityState));
```

Lab: Running `CompletableFuture` example

Step 1: In the akka-training project, and in the *src/main/java* and in the `com.akkatraining.java.futures` package create a class called `CompletableFutureJava` with the following content in the class declaration

```java
package com.akkatraining.java;

public class CompletableFutureJava {
  static void sleepFor(long millis) {
      try {
          Thread.sleep(millis);
      } catch (InterruptedException ignored) {
      }
  }

  private static class WebService {
      CompletableFuture<String> getHighestRankedStock
          (LocalDate localDate) {
          sleepFor(1000);
          return CompletableFuture.supplyAsync(() -> "MSFT");
      }

      CompletableFuture<Float> getStockPrice(String symbol) {
          sleepFor(2000);
          Random random = new Random();
          //Mimicking going to a webservice
          return CompletableFuture.supplyAsync(random::nextFloat);
      }
  }
}
```

Step 2: Create a `main` method in `CompletableFutureJava` that will do the following

- Instantiate `WebService`
- Call the webService `getHighestRankedStock` and given the stock, use `CompletableFuture` to process the price
- Make all necessary imports

Step 3: Run the example, discuss other options

Reintroduction to Scala

`val`

- `val` is called a *value*
- It is immutable, therefore unchangeable
- No reassignment
- Since Scala is functional, it is *preferred*

```
val a = 10
```

Reassignment of `val`

A reassignment is out of the question with a `val`

```
val a = 10
a = 19 //error: reassignment to val
```

`var`

- Called a *variable*
- It is mutable, therefore changeable
- Reassignable
- Used sparingly
- Usually kept from being changed from the outside

```
var a = 10
a = 19
println(19) //That's how you println
```

Scala `class`

- Classes are the templates or blueprints of a construct that encapsulates state and manages behavior.
- Classes have been around for a long time and in every object oriented language.
- Since Scala is a half object oriented language, half functional language, there are naturally classes.

A Scala `class`

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

- A `class` is public by default, so no need for a `public` modifier
- The `(firstName:String, lastName:String)` is the primary constructor!
- In Scala the primary constructor is "top-heavy" with a constructor that contains all the information
- Other constructors are smaller constructors that feed the top constructor
- The reason for the top heavy constructor is immutability

Can't access or modify class?

- As it stands in our `class`, we can neither access or modify our `class`
- Most of the time we don't want to modify our `class` for immutability purposes
- To be able to access the members, we will predicate each of the values with `val`
- To be able to mutate the member variables, we predicate each of the values with `var` (Not recommended)

```
package com.xyzcorp

class Employee(val firstName:String, var lastName:String)
```

```
val emp = new Employee("Dennis", "Ritchie")
emp.firstName should be ("Dennis") //Works because of val
emp.lastName should be ("Ritchie") //Works because of var
emp.lastName = "Hopper"            //Works because of var
emp.lastName should be ("Hopper")
```

Scala Methods

- There is a differentiation between *methods* and *functions*
- Methods in Scala belong to context like a `class`, `object`, `trait`, a script, or another method.
- In Scala, a method starts with `def`
- Parameters are in reverse of what is expected in Java, value or variable before the type, e.g `age:Int`

A Basic Non-Concise Method

- The `:Int` is the return type
- If you expect something in return you need, an equal sign (`=`)
- If you do not then leave it out

```
def add(x: Int, y: Int):Int = {
      return x + y
}
```

Cleaning up our Method

- A method can make use of *type inference*
- The braces are optional
- `return` is optional, in fact, it is rarely used

Therefore...

```
def add(x: Int, y: Int) = x + y
```

Discuss: Why there no longer is `:Int` at the end by referencing the API at http://www.scala-lang.org/api/current/

When type inference is not good enough

- There are times when you need the type:
  - To make things clear
  - Because the type inferencer could be wrong
  - Your method might be overloaded and it would be needed disambiguate from other methods
  - You're doing recursion
  - You're doing method overloading

Figuring out the type

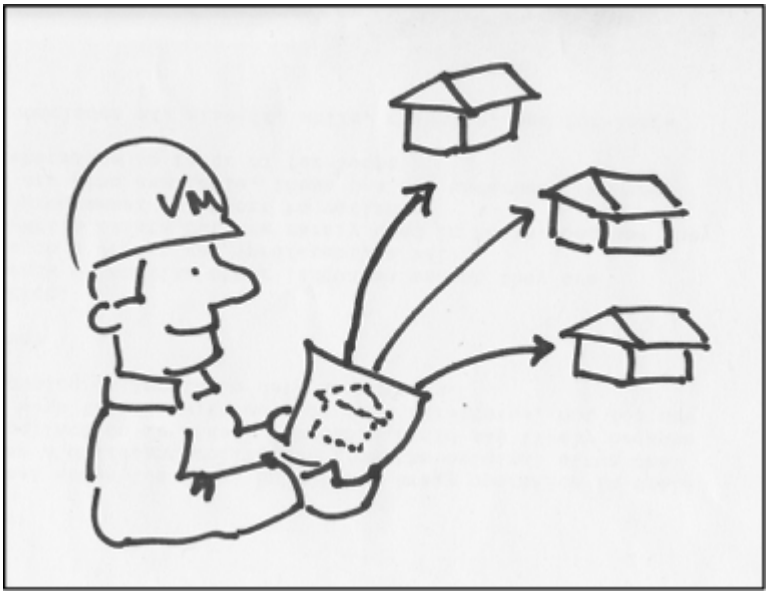
Given the following what is the type?

```
def numberStatus(a:Int) =
   if (a < 10) "Less than 10"
   else if (a > 10) "Greater than 10"
   else "It is 10!"
```

```
val result = numberStatus(3)
result should be ("Greater than 10")
```

Scala Type Hierarchy

- A type is a `class`, `trait`, a **primitive**, or an `object` in Scala.
- A `class` for those who don't know is a code template, or blueprint, that describes what the objects created from the blueprint will look like.



- `Int` is a type
- `String` is a type
- If we created a `Car` class, `Car` would be a type
- If we created `InvoiceJSONSerializer`, `InvoiceJSONSerializer` would be a type

Matching Types

Given these two methods:

```
def add(x:Int,y:Int):Int = x + y
def subtract(x:Int, y:Int):Int = x - y
```

To use them together, you would just need to match the types:

```
add(subtract(10, 3), subtract(100, 22))
```

If we changed `subtract` to use `Double` instead:

```
def add(x:Int,y:Int):Int = x + y
def subtract(x:Double, y:Double):Double = x - y
```

We would just need to make sure that that type matches:

```
add(subtract(10.0, 3.0).round.toInt, subtract(100.0, 22.0).round.toInt)
```

Primitives Are Objects

- In Scala we treat everything like an object
- Therefore, you can treat all numbers, boolean, and characters as types
- Every primitive is a member of `AnyVal`

`java.lang.Object` has been demoted

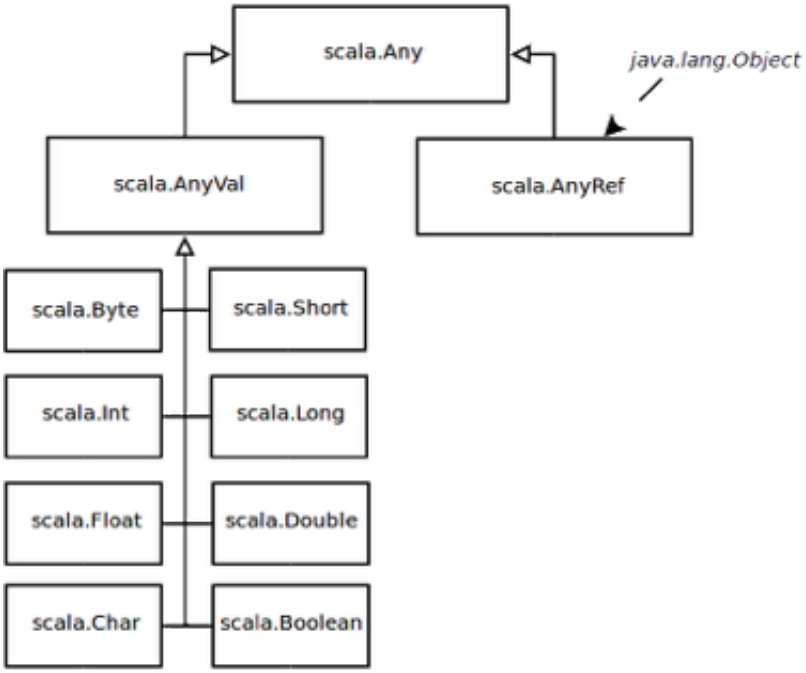
- `java.lang.Object` is called `AnyRef` in Scala
- `AnyRef` will be the super type of all object references
- This includes:
  - Scala API Classes
  - Java API Classes
  - Your custom classes

`Any` is the new leader

- `Any`
  - Super type of:
    - `AnyVal` (Primitive Wrappers)
    - `AnyRef` (Object References)

Scala Family Tree

Scala's Family Tree:



Lab: Polymorphism

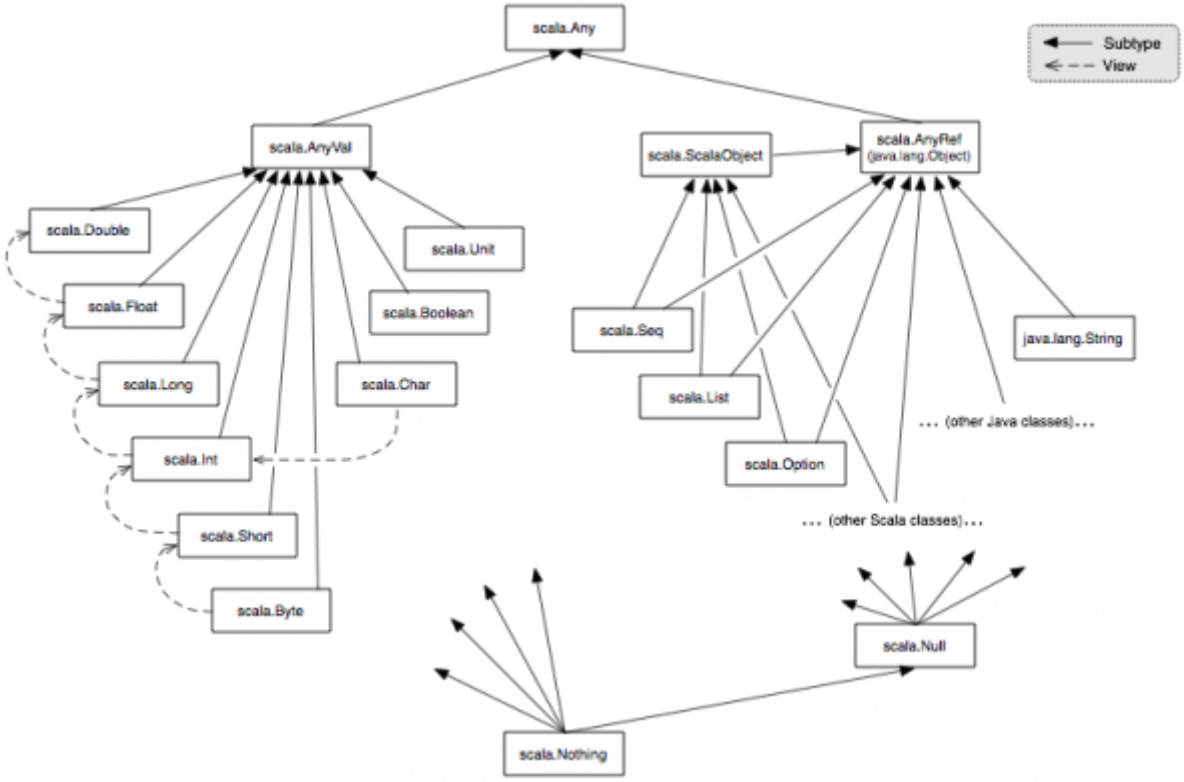
Step 1: Open a REPL either by typing `scala` or by running SBT's `console`.

Step 2: Try various combinations of references and see how everything is match together. Try some of these combinations that may or may not work to be sure:

```
scala> val a:AnyVal = 40
scala> val b:Any = a
scala> val c:AnyRef = "A String"
scala> val d:Any = c
scala> val e:AnyRef = 40
scala> val f:AnyVal = "A String"
```

Step 3: Discuss and ask any questions.

The Complete Hierarchy



Nothing

- `Nothing` is the sub type of everything
- It is used for collections and other containers with no defined parameterized type
- Used to represent a "bottom", a type that covers among other things, methods, that only throw `Throwable`

Lab: Where is `Nothing`?

Step 1: In a REPL try the following, create an empty `List` with no type:

```
> val list = List()
```

Step 2: Create a `List` with a type, and note the difference

```
> val list2 = List[Int]()
```

Step 3: Create a method called `ohoh` that returns only an `Exception`. Throwing an `Exception` in particular is always of type `Nothing` since nothing really ever gets returned.

```
> def ohoh(i:Int):Nothing = { throw new Exception() }
```

It shouldn't matter if it is an `Exception` or a `RuntimeException`

Link: http://www.scala-lang.org/api/current/scala/Nothing.html

`Null`

- `Null` is a type in Scala that represents a `null`
- `Null` is the sub type of all object references
- `null` is not used in pure Scala applications, but only for those that interop with Java

Lab: `Null` type

Step 1: In a REPL, type the following, and noticed the type:

```
val f = null
```

Step 2: In the REPL, next try the following, and analyze the type:

```
val x:String = null
```

Step 3: In the REPL, next try the following, and analyze the results, can you guess why there is a strange message?

```
val x:Int= null
```

Conclusion

- Types are templates that make up an object.
- Primitives are also types, `Int`, `Short`, `Byte`, `Char`, `Boolean`, etc.
- Types need to matched up like a puzzle. If it isn't the type system at compile time will tell you there is a type mismatch
- Every type is in a relationship.
- `Any` is the parent for all types
- `AnyVal` is the parent for all primitives
- `AnyRef` is the parent for all Scala, Java, and custom classes that you create
- `Nothing` is the subtype for everything
- `Null` is the subtype of all references

Scala `Unit`

- There is a `Unit` and it can be invoked using `()`
- A `Unit` is analogous to `void` in Java, C, C++
- Used as an interpretation of not returning a something
- Another way to interpret `Unit` is means there is nothing to give

```
val g = ()
```

Where have we used `Unit`?

- There is a popular form of `Unit`
- It is called `println`,
- `println` doesn't return anything, therefore it is return `Unit`

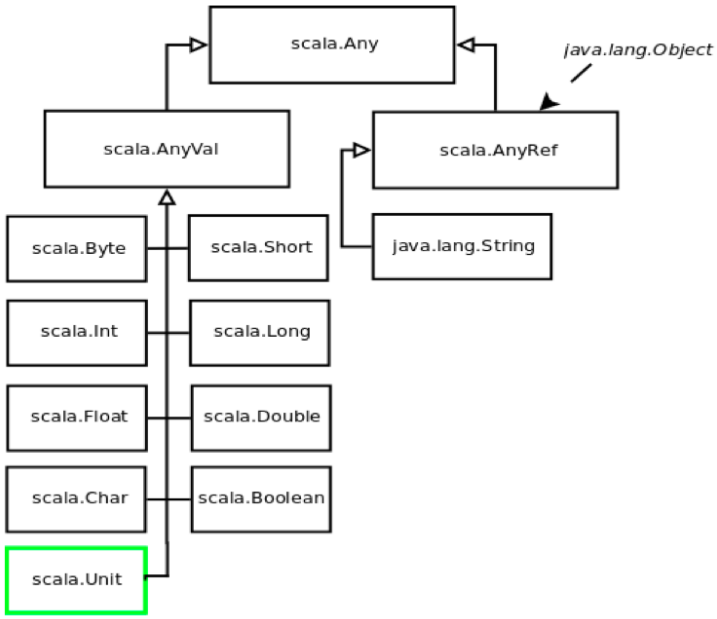
https://www.scala-lang.org/api/current/scala/Unit.html

https://www.scala-lang.org/api/current/scala/Predef$.html

Determining the type of `println`

```
> val h = println("Hello, Scala")
```

Where is `Unit` in the Scala hierarchy?



What is the return type of this?

Without running, what do you think the return type is for the following?

```
def add(x: Int, y: Int) = {
  if (x > 10) println(x)
  else x + y
}
```

`Unit` can be used anywhere

- Unit is just an object that represents a void
- The following is a purely nonsensical method to prove that it can be treated like an object

```
def nonsense(g:Unit):Int = 50
```

Unplanned `Unit`

- In methods, if you do not use the `=` that will implicitly mean that you are returning `Unit`

Given the following which is correct, this will add `x` and `y`:

```
def add(x:Int, y:Int) = {
   x + y
}
```

If we didn't use `=` this will implicitly return `Unit`

```
def badAdd(x: Int, y: Int) {
   x + y
}

println(badAdd(4, 5)) //Returns a ()!
```

Explicitly putting `Unit`

If we can to write out the `Unit` explicitly we can do so, by stipulating `Unit` as the return type

```
def addUnit(x: Int, y: Int):Unit = {
   x + y
}
println(addUnit(4,5)) //Returns a ()!
```

It's called `addUnit` because we didn't create it on purpose.

Side Effects and `Unit`

- When seeing `Unit` in functional programming when being returned from a method, it means it likely is creating a side effect
- A side effect is when some is changed to the outside world either
  - Printing to a screen
  - Printing to a printer
  - Saving to Storage
  - Changing State

The following is changing state. Notice that this is returning `Unit`!
Also, notice that there is `var`!
A `Unit` return will typically mean that there is a side-effect

```
var a = 0
def sideEffect() {
   a = a + 1
}
```

You don't have to be too stringent in avoiding state, although some FP purist will disagree

Scala Tuples

- Tuples are dumb containers
- Perfect for grouping items
- Perfect for return two items or three or four particularly if they are different types
- Perfect for Akka messages

Creating a `Tuple`

```
val ta = (1, "cool", 402.00)
val tb:(Int, String, Double) = (1, "cool", 402.00)
val tc:Tuple3[Int, String, Double] = (1, "cool", 402.00)
```

There is a `Tuple1`, `Tuple2`, `Tuple3`,...`Tuple22`

Getting the values from a `Tuple`

```
val ta = (1, "cool", 402.00)
println(ta._1)
println(ta._2)
println(ta._3)
```

The `_1`, `_2` comes from Haskell's `first` and `second` functions

Swapping `Tuple`

- Only a `Tuple2` can be swapped

```
val t2 = ("Foo", 40.00)
println(t2.swap) //(40.00, "Foo")
```

Syntactic Sugar for Tuples

- `Tuple2` can also be created with `->`
- The `->` is a method that is on every object that accepts another object

```
val t2 = "Foo" -> 40.00 // (40.00, "Foo")
```

You can use the `-` and `>` or you can use the unicode rightwards arrow: `→`

The Magical `apply` method

Considering the following code once again:

```
class Foo(x:Int) {
   def bar(y:Int) = x + y
}
```

If we run it, it would like this and return 50

```
val foo = new Foo(40)
foo.bar(10)
```

Replacing `bar` with `apply`

Let's take the previous code and use `apply` instead of `bar`:

```
class Foo(x:Int) {
   def apply(y:Int) = x + y
}
```

If we run it, it would like this and return 50

```
val foo = new Foo(40)
foo.apply(10)
```

Where thing are different, is that **apply is not required method call and you can leave the word out!**

Therefore...since we used apply it looks like this:

```
val foo = new Foo(40)
foo(10)
```

Scala Functions

- Think of functions as something that take input or multiple input and returns ***only one*** output
- Functions are based on traits, `Function1`, `Function2`, ..., `Function22`

Functions the hard way

Given: An anonymous instantiation

```
val f1:Function1[Int, Int] = new Function1[Int, Int] {
   def apply(x:Int) = x + 1
}

println(f1.apply(4)) //5
```

But since, the name of the method is `apply`....

```
println(f1(4)) //5
```

Note: In 2.12, it may not be like this, since they will integrate with Java 8's backed `java.util.function`

Different Signatures of Function (The long way)

- Here are some functions declared the long way
- We explicitly declare the types on the left hand side `Function1`, `Function0`
- We are instantiating the `traits`

```
val f1:Function1[Int, Int] = new Function1[Int, Int] {
     def apply(x:Int) = x + 1
}

val f0:Function0[Int] = new Function0[Int] {
     def apply() = 1
}

val f2:Function2[Int, String, String] = new Function2[Int, String,
String] {
     def apply(x:Int, y:String) = y + x
}
```

Signatures of Function (The medium way)

- Here are some functions declared the medium way
- We explicitly declare the types on the left hand side as:
  - `Int ⇒ Int` to represent `Function1[Int, Int]`
  - `() ⇒ Int` to represent `Function0[Int]`
  - `(Int, String) ⇒ String` to represent `Function2[Int, String, String]`
- Of course these functions are still anonymous instantiating of `traits`

```
val f1:(Int => Int) = new Function1[Int, Int] {
     def apply(x:Int) = x + 1
}

val f0:() => Int = new Function0[Int] {
     def apply() = 1
}

val f2:(Int, String) => String = new Function2[Int, String, String] {
     def apply(x:Int, y:String) = y + x
}
```

Trimming down the functions

- Give the functions from the previous page, we can clean up the functions
- The left hand side uses a short hand notation to declare the types as before
- The right hand side uses a short hand notation to create the function!

```
val f1:Int => Int = (x:Int) => x + 1

val f0:() => Int = () => 1

val f2:(Int, String) => String = (x:Int, y:String) => y + x
```

Making it concise: Using type inference with functions

- Since there is heavy type inference in Scala, we can trim everything down
- The left hand side can be left with little or no type annotation since the right hand side is declaring most of the information

```
val f1 = (x:Int) => x + 1

val f0 = () => 1

val f2 = (x:Int, y:String) => y + x
```

Choosing the type inference, left hand side or right hand side

Using right hand side verbosity, and left hand side type inference

```
val f1 = (x:Int) => x + 1

val f0 = () => 1

val f2 = (x:Int, y:String) => y + x
```

Using right hand side inferred, and left hand side verbosely declared

```
val f1:Int => Int = x => x + 1

val f0:() => Int = () => 1

val f2:(Int, String) => String = (x,y) => y + x
```

Returning more than one result

- Every function has one return value so how do we return multiple items?
- Use a `Tuple`!

```
val f3 = (x:String) => (x, x.size)  // Right hand side declaration
println(f3("Laser"))                // ("Laser", 5)
```

Trimming the result with _

- Given the function where the left hand side is declared with the type
- We can provide a shortcut using the _ that represents one of the arguments
- Seasoned Scala developers will recognize this shortcut
- Be aware that this may be confusing for novices

Given:

```
val f5: Int => Int = x => x + 1
```

Can be converted to:

```
val f5: Int => Int = _ + 1
```

Works the same when invoking the function

```
f5(40)  //41
```

Advanced Feature: Postfix Operators

- If the argument on the right most position is a `_`, it can be omitted
- This may require that an `import` feature may need to be turned on

Given:

```
val f5: Int => Int = x => x + 1
```

Can be converted to:

```
val f5: Int => Int = _ + 1
```

Since addition is commutative, it can be rearranged jj

```
val f5: Int => Int = 1 + _
```

Since, the `_` is at the right most position it can be dropped

```
val f5: Int => Int = 1+
```

Works the same when invoking the function

```
f5(40)  //41
```

Advanced Feature: Postfix Operators, Clearing Warnings

- If you attempted to clear out the most `_` on the right hand side you may have received this warning:

```
warning: postfix operator + should be enabled
by making the implicit value scala.language.postfixOps visible.
This can be achieved by adding the import clause 'import
scala.language.postfixOps'
or by setting the compiler option -language:postfixOps.
See the Scaladoc for value scala.language.postfixOps for a discussion
why the feature should be explicitly enabled.
```

- This means that to avoid the warning you can turn that feature on by an `import` statement

```
import scala.language.postfixOps
val f5: Int => Int = 1+
```

Works the same when invoking the function, and you get no warnings

```
f5(40)  //41
```

Trimming the result with multiple _

- If a function has two arguments, that too can use the _ shortcut
- The only thing is that _ can only be applied to **one** argument

Here is the before:

```
val sum: (Int, Int) => Int = (x, y) => x + y
```

Here is the after:

```
val sum: (Int, Int) => Int = _ + _
```

Note in the above:

- The first _ represents the first argument, or `x` in the above example
- The second _ represents the second argument, or `y` in the above example

Scala Partial Functions

- A Scala partial function is must ask permission first.
- A Partial Function uses `isDefinedAt` to determine if applicable
- If applicable it will run using `apply`

```
val doubleEvens = new PartialFunction[Int, Int] {
  override def isDefinedAt(x:Int) = x % 2 == 0
  override def apply(x:Int) = x * 2
}

val tripleOdds = new PartialFunction[Int, Int] {
  override def isDefinedAt(x:Int) = x % 2 != 0
  override def apply(x:Int) = x * 3
}
```

Combining Partial Functions

```
val combinedPartialFunction = doubleEvens.orElse(tripleOdds)
```

Applying the above yields

```
combinedPartialFunction.apply(10) //20
combinedPartialFunction.apply(15) //45
```

Making Partial Functions Concise

```
val doubleEvens = {case x:Int if x % 2 == 0 => x * 2}


new PartialFunction[Int, Int] {
  override def isDefinedAt(x:Int) = x % 2 == 0
  override def apply(x:Int) = x * 2
}

val tripleOdds = new PartialFunction[Int, Int] {
  override def isDefinedAt(x:Int) = x % 2 != 0
  override def apply(x:Int) = x * 3
}
```

Combining Partial Functions

```
doubleEven.orElse(tripleOdds)
```

Scala `object`

- We know that objects are instantiated from classes
- We can also create an object, a single object, without a class
- This is called an `object`
- **IMPORTANT:** An `object` is a **singleton**. There is only one
- It is how we avoid `static` methods

Lab: Trying out the `object`

Step 1: Go to a Scala REPL or sbt console and type the following

```
scala> object MyObject
```

Step 2: Verify that it is truly a singleton

```
scala> val a = MyObject
scala> val b = MyObject
scala> a == b
scala> a eq b
```

How we avoid `static`

- We can add values, or variables, and methods to an `object`
- When we invoke those values, variable, and methods it looks and feels like invoking something `static`

Adding methods to `MyObject`

```
object MyObject {
   def foo(x:Int, y:Int) = x + y
}
```

To invoke this `object` we call:

```
MyObject.foo(4, 2) should be (6)
```

When do you need objects? When do you need singletons?

- For Classes
  - Need to define a template to create multiple instances
  - Every instance has a state
- For Objects
  - You need a singleton
  - You need a factory pattern, which defined as: Creating families of related or dependent objects without specifying or hiding their concrete classes.
  - You need to implement pattern matching logic
  - You need create a utility that doesn't require and instance or a state.
  - You have some default values or constants

The `main` method

- The `main` method in Java requires all of this so that we can execute it as an application

```
public static void main(String[] args) {
   System.out.println("Hello, Scala")
}
```

- The same elements apply in Scala although the components are different
  - `object` instead of `static`
  - `Unit` instead of `void`
  - Everything is `public` by default
  - `Array[String]` instead of `String[]`

Therefore the `main` method in Scala looks like:

```
object Runner { //Call it whatever you want
  def main(args:Array[String]):Unit = println("Hello, Scala")
}
```

Alternative `main` method

- You can also create a `main` method doing the following:

```
object Runner extends App {
   println("Hello, Scala")
}
```

Untyped Actors in Scala

Setting up your first actor in Scala

- Actors in Scala extend from `Actor`
- By extending `Actor` you must override `receive`
- `receive` must return a `Receive` object
- The `Receive` object is a `PartialFunction[Any, Unit]`

```
class SimpleActorScala extends Actor{
    override def receive: Receive = new Receive {
        override def isDefinedAt(x: Any): Boolean = ???

        override def apply(v1: Any): Unit = ???
    }
}
```

Using a refined a `receive` with actual `PartialFunction`

```
class SimpleActorScala extends Actor{
   val log = Logging(context.system, this)

   def receive:Receive = {
     case x:String =>
       log.info("Received message in Simple Actor Scala {}", x)
   }
}
```

Using a refined a `receive` with actual `PartialFunction`

- We can use the `trait`, `ActorLogging` to mix in logging

```
class SimpleActorScala extends Actor with ActorLogging {
   def receive:Receive = {
     case x:String =>
       log.info("Received message in Simple Actor Scala {}", x)
   }
}
```

Creating an `ActorSystem` with an `Actor` in Scala

```
package com.akkatraining.scala

object SimpleActorScalaRunner {
   def main(args:Array[String]):Unit = {
      val actorSystem = ActorSystem("My-Actor-System")
      val props = Props[SimpleActorScala]
      val actorRef = actorSystem.actorOf(props, "simpleActorScala")
      actorRef ! "Hello"

      println("Press Enter to Terminate System")
      scala.io.StdIn.readLine()

      Await.result(actorSystem.terminate(),
         Duration(10, TimeUnit.SECONDS))
   }
}
```

- A `Props`
  - A recipe as to how to create an `Actor`
  - Is a message and can be used to send to other actors
- `actorSystem.actorOf` creates the actor
- `actorRef !` sends the message to an actor in a fire and forget fashion
- This will run locally on a single JVM

Lab: Creating an Actor in Scala

Step 1: In *src/main/scala*, create a package called `com.akkatraining.scala`

Step 2: Create `SimpleActorScala` in `com.akkatraining.scala` with the following:

```
package com.akkatraining.scala

class SimpleActorScala extends Actor {
    val log = Logging(context.system, this)

    override def receive:Receive = {
        case message:String =>
            log.info("""Received String message
                        in Simple Actor Scala {}""", message)
    }
}
```

Step 3: Ensure that all imports are done (not shown)

Lab: Creating an `ActorSystem` in a main method:

Step 1: In the folder *src/main/scala* create an `object` called `SimpleActorScalaRunner`

```
package com.akkatraining.scala

object SimpleActorScalaRunner {
    def main(args:Array[String]):Unit = {
        val actorSystem = ActorSystem("My-Actor-System")
        val props = Props[SimpleActorScala]
        val actorRef = actorSystem.actorOf(props)
        actorRef ! "Hello"

        println("Press Enter to Terminate System")
        scala.io.StdIn.readLine()

        Await.result(actorSystem.terminate(),
           Duration(10, TimeUnit.SECONDS))
    }
}
```

Step 2: Repair any imports

Step 3: Run the main method either in your IDE or in SBT by running `run` in SBT, or in the command line, and selecting the class that you want to run:

```
% sbt run
```

or

```
% sbt
> run
```

Lab: Creating an `ActorSystem` in a main method (Continued):

Step 4: Discuss the results

Step 5: Give `actorOf` a name of for your actor, run again, discuss the results

Step 6: Discuss where `apply` was used

Step 7: Discuss termination

Actor Lifecycle Methods

preStart

- Lifecycle hook that is notified before an `Actor` is started

```
def preStart(): Unit = ()
```

postStop

- Lifecycle hook that is notified after an `Actor` is stopped

```
def postStop(): Unit = ()
```

preRestart

- Lifecycle hook that is notified before a restart
- `reason` is a `Throwable` containing the reason for the `restart`
- `message` will contain any message from the `restart`
- Used to clean up any resources or "unwatch" children

```
def preRestart(reason: Throwable, message: Option[Any]): Unit = ()
```

postRestart

- Lifecycle hook that is notified after restart
- Typically calling `preStart` since the actor has since restarted

```
def postRestart(reason: Throwable): Unit = {
  preStart()
}
```

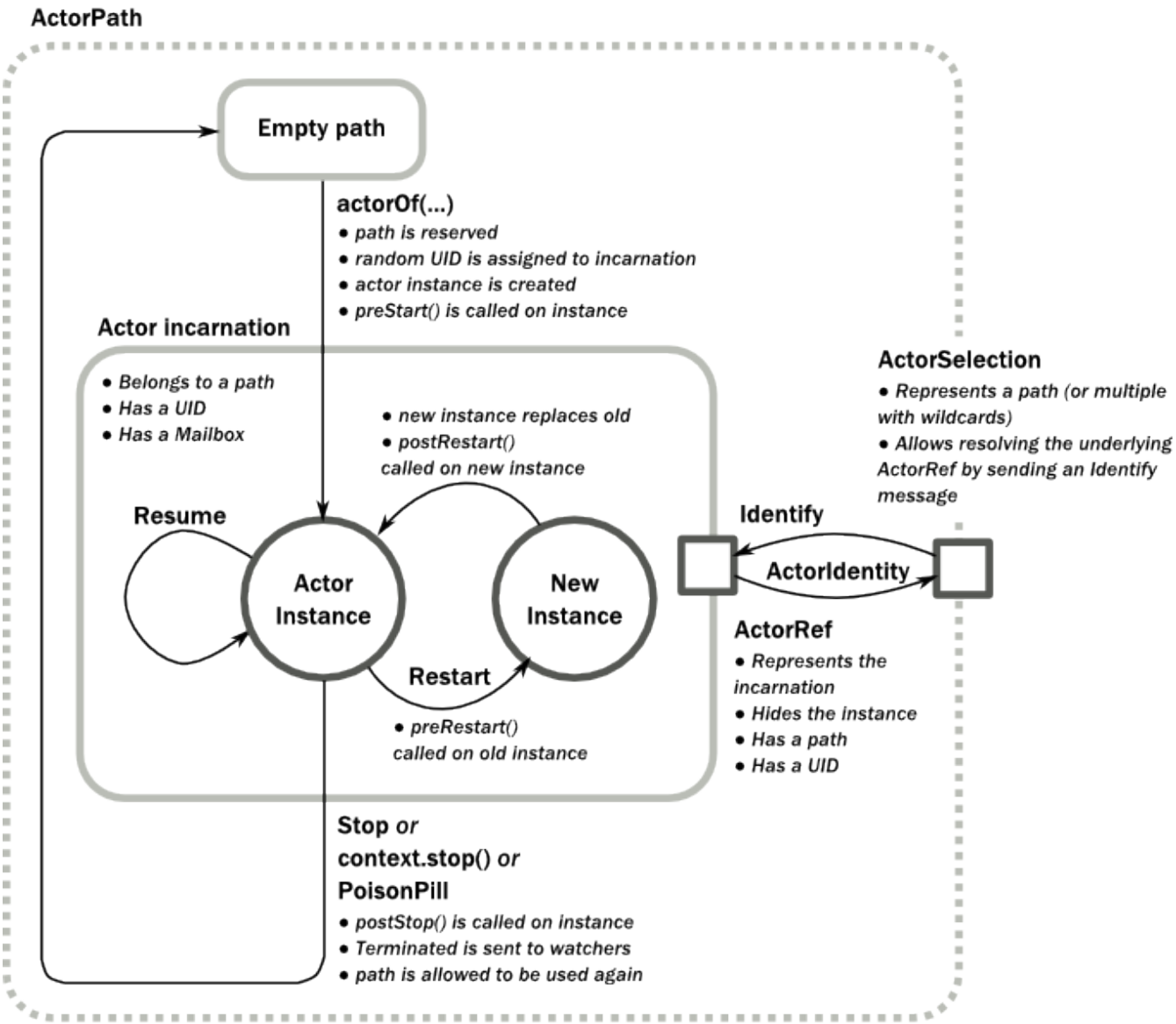
Newer lifecyle methods

- There are some newer lifecycle methods that are wrappers around events
- These have not been documented on the Akka website
- They are:
  - `aroundReceive`
  - `aroundPreStart`

- aroundPreRestart
- aroundPostRestart

```
override def aroundPostStop(): Unit = {
  log.debug("The actor is about to stop")
  super.aroundPostStop()
  log.debug("The actor has stopped")
}
```

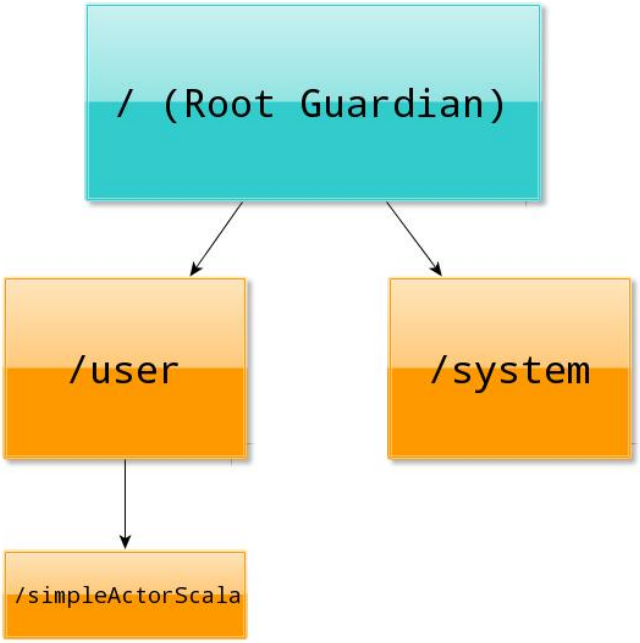
Actor lifecycle map



References

How are actors referenced from an `ActorSystem`?

- Given an actor deployed by the name of `simpleActorScala`



How are actors mapped as a URI?

Given an `ActorSystem` called `actorSystem`...

A local reference URL

```
akka://my-sys/user/service-a/worker1
```

A remote reference URI

```
akka.tcp://my-sys@host.example.com:5678/user/service-b
```

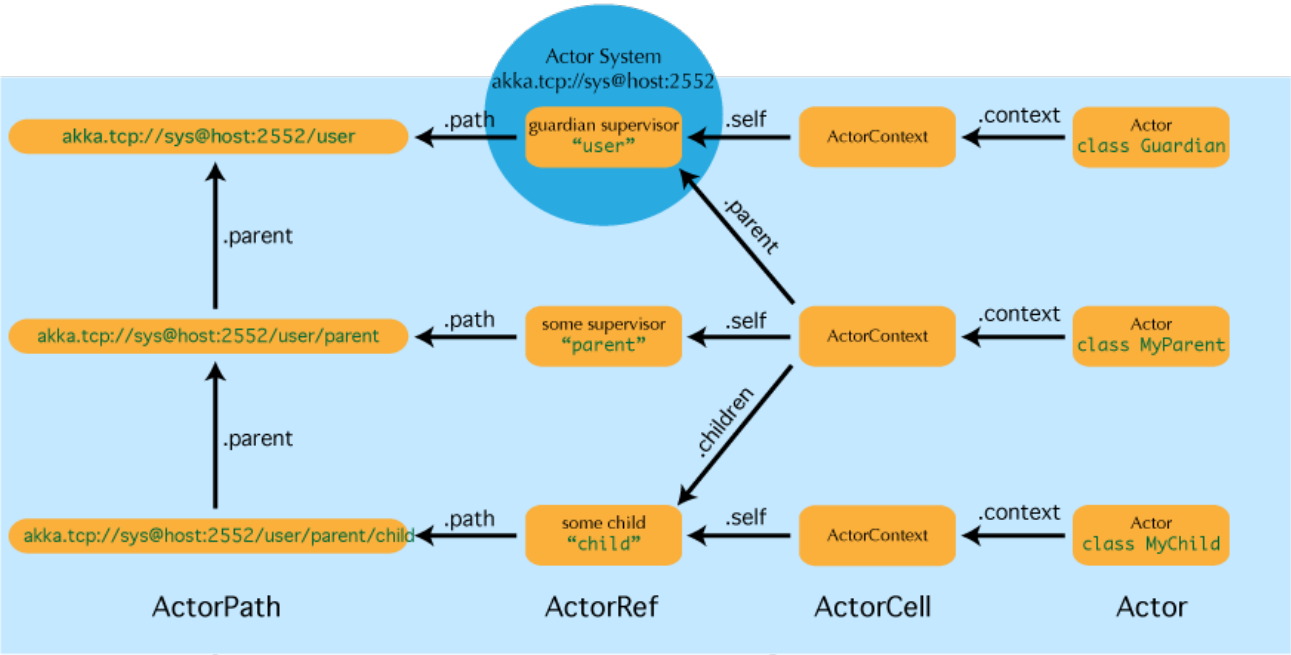
References within an Actor

Within an actor, we can use the following references:

`sender`	`ActorRef` that sent the message
`self`	`ActorRef` of the actor that it is running on

`context`	The environment that the actor is in, `context.system` will return the `ActorSystem`

How the references are resolved



Location Transparency

- There is nearly no API for the remoting layer of Akka
- It is purely driven by configuration.
- Just write your application according to the principles
- Then specify remote deployment of actor sub-trees in the configuration file.
- This way, your application can be scaled out without having to touch the code

Peer to Peer Philosophy

1. Communication between involved systems is symmetric: *if a system A can connect to a system B then system B must also be able to connect to system A independently*
2. The role of the communicating systems are symmetric in regards to connection patterns: *there is no system that only accepts connections, and there is no system that only initiates connections.*

Lab: Conversing multiple actors

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.conversation` called `ActorA`

Step 2: Inside of `ActorA`, override the lifecycle methods `preStart`, and `aroundPostStop` and `log.info` when these events take place.

Step 3: Inside of `ActorA` respond to these messages

- When the `String` `"Hello"` is received tell the `sender`, `"Hello to You"`
- When the `String` `"How are you doing?"` tell the `sender`, "Fine"
- When anything else is sent, `log.info` with `log.info("Got something else {}", msg)`

Lab: Conversing multiple actors (continued)

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.conversation` called `ActorB`

Step 2: Inside of `ActorB`, override the lifecycle methods `preStart`, and `aroundPostStop` and `log.info` when these events take place.

Step 3: Inside of `ActorB` respond to these messages

- When any `ActorRef` is received tell that `ActorRef`, `"Hello"`
- When the `String` `"Hello to you"` tell the `sender`, "How are you doing?"
- When anything else is sent, `log.info` with `log.info("Got something else {}", msg)`

Lab: Conversing multiple actors (continued)

Step 1: In the project, akka-training, Create a runnable `object` inside of *src/main/scala* and in the package `com.akkatraining.scala.conversation` called `ActorConversationRunner`

Step 2: In the `ActorConversationRunner`:

- Create an `ActorSystem`
- Create `ActorA` in the `actorSystem` and obtain its `ActorRef`
- Create `ActorB` in the `actorSystem` and obtain its `ActorRef`
- Send `!` the reference of `actorA` to `actorB`

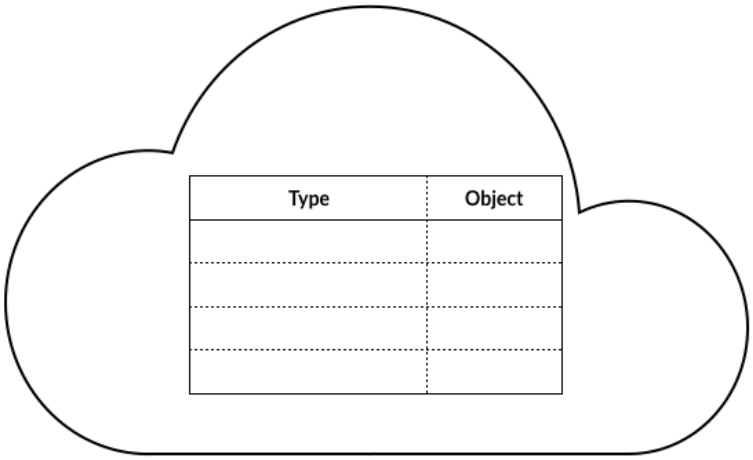
If there are issued it may be due to spelling, messages are case sensitive.

Step 3: Add a key listener to shut down the `actorSystem` when the ***ENTER*** key is pressed

implicit

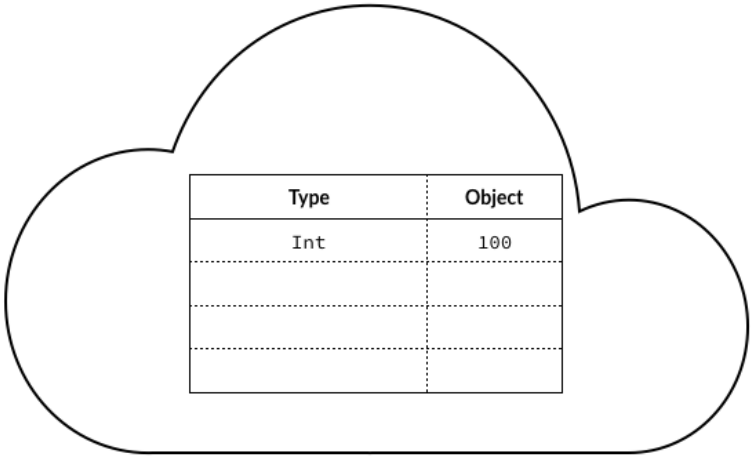
implicit

- `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



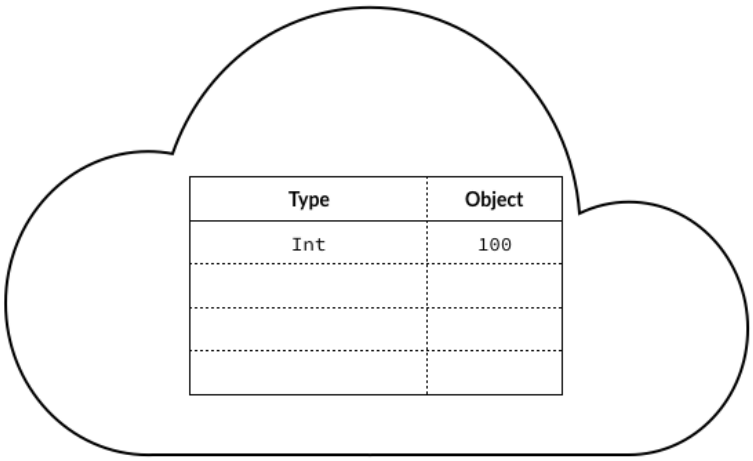

```
implicit val hourlyRate = 100

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



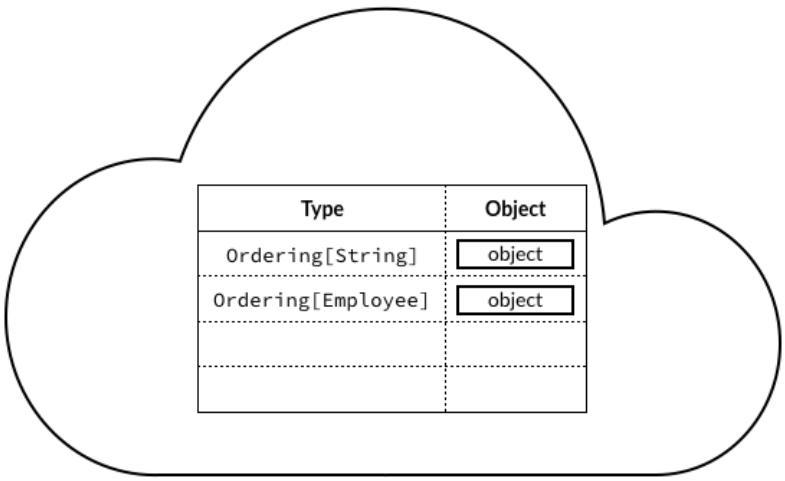

```
implicit val hourlyRate = 100

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*

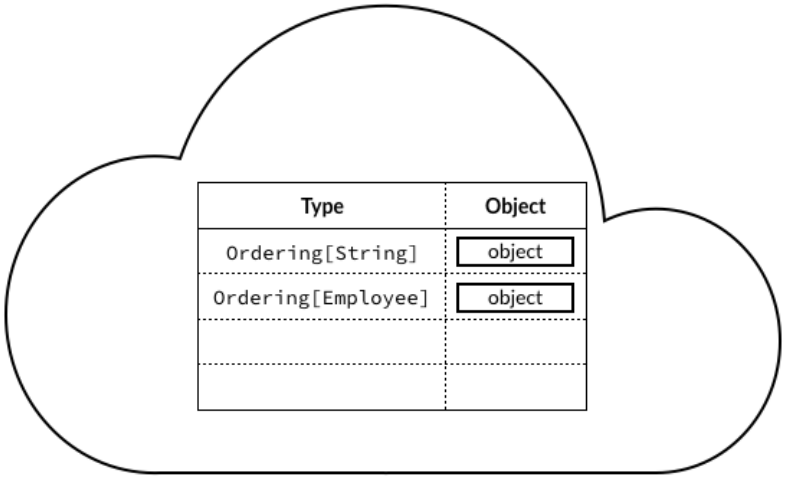


```
case class Employee(firstName:String, lastName:String)

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
```

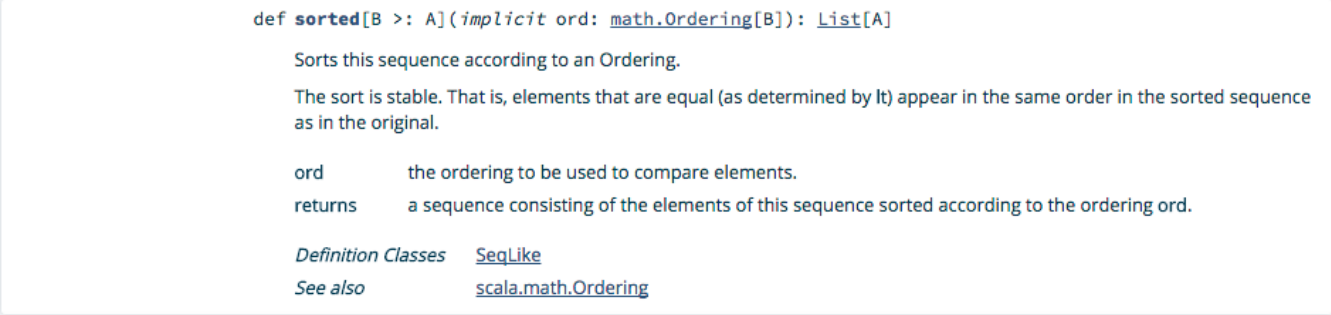
Which yields the result:

```
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

Scala Futures

Semantic Differences with Java `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.lang.Throwable` then it has *failed*

Creating a `Future`

- `scala.concurrent.Future[T]` is created with an `apply` in its `object`
- Optionally, you can call `Future.successful` or `Future.failed` as a quick alternative
- 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 a the contents of a `Future` you can use `foreach`

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

Lab: Creating a `Future` in Scala

Step 1: In the *src/test/scala* directory and in the `com.akkatraining.scala.futures` package, open *FuturesSpec.scala*

Step 2: Go to the specification "Case 1: Create a basic Future in Scala"

Step 3: Follow along with instructor in creating a future

Step 4: Run the test in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.futures.FuturesSpec -- -z "Case 1:"
```

It would be easier the first time to run `test:compile` first and use tab completion for the above line

Using `Await.ready`

- `scala.concurrent.Await.ready` will block and await the answer of the `Future`
- When the future is resolved either successfully or in failure it will return the same `Future`
- Think of it as a "waiting gate"
- You can then use any `Future` methods to perform any operation needed.
- Uses a `scala.concurrent.duration.Duration` as a timeout

```
import scala.concurrent.duration._

Await.ready(future, Duration.apply(2, TimeUnit.SECONDS))
      .map(x => x + 10)
      .foreach(println)
```

Using `Await.result`

- `scala.concurrent.Await.result` will block and await the answer of the `Future`
- When the future is resolved successfully it will return the result from the `Future`
- Also, think of it as a "waiting gate", but this time it just returns what is "inside" of the `Future`
- Uses a `scala.concurrent.duration.Duration` as a timeout

```
import scala.concurrent.duration._

val result: Int = Await.result(future, Duration.apply(2, TimeUnit
.SECONDS))
```

The above future is a `Future[Int]`

Lab: Seeing `ready` and `result` in action

Step 1: In the *src/test/scala* directory and in the `com.akkatraining.scala.futures` package, open *FuturesSpec.scala*

Step 2: Go to the specifications "Case 3:" and "Case 4:" and review.

Step 3: Run "Case 3:" and "Case 4:" in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.futures.FutureSpec -- -z "Case 3:"
```

It would be easier the first time to run `test:compile` first and use tab completion for the above line

Step 4: Change code and ask questions

Future.successful

- You can just get straight to the point with a `Future` and call `successful` to return a value.
- This is used if the Future being created has no potential of failure, like performing I/O.

```
Future.successful("Hello")
```

Future.failed

- You can just get straight to the point with a `Future` and call `failed` to return a failure.
- This is used if the Future being created has no potential of failure, like performing I/O.

```
Future.failed[Int](new Throwable("Unable to complete"))
```

recover

- Creates a new future that will handle any matching `Throwable` that this future might contain.
- If there is no match, or if this future contains a valid result then the new future will contain the same.

The following will return `10`

```
Future.successful(10).recover { case e: Throwable => -1 }
```

The following will return `-1`

```
Future.failed(new Throwable("Unable to complete"))
      .recover { case e: Throwable => -1 }
```

recoverWith

- Creates a new `Future` that will handle any matching `Throwable` that this future might contain.

- Assigns it a value of another future.
- If there is no match, or if this future contains a valid result then the new future will contain the same result.

The following will return `10`

```
Future.successful(10)
      .recoverWith { case e: Throwable => Future.successful(-1) }
```

The following will return `-1`

```
Future.failed(new Throwable("Unable to complete"))
      .recoverWith { case e: Throwable => Future.successful(-1) }
```

Lab: Seeing `successful`, `failed`, `recover` and `recoverWith` in Action

Step 1: In the *src/test/scala* directory and in the `com.akkatraining.scala.futures` package, open *FuturesSpec.scala*

Step 2: Go to the specifications "Case 5:" through "Case 8:" and review.

Step 3: Run "Case 5:" through "Case 8:" one at a time in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.futures.FutureSpec -- -z "Case 5:"
```

Step 4: Change code and ask questions

`Try`

- `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
- Used to model an attempt of an action.

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

`Try` by Example

- Any that has the potential of an exception can be trapped within a `Try`

```
def parseNumber(s: String): Try[Int] = Try {
  Integer.parseInt(s)
}
```

- We can then do typical functional programming

```
parseNumber("x").map(x => x + 30)
parseNumber("50").map(x => x + 30)
```

- We can do typical methods found in an `Option`

```
triedInt1.getOrElse(-1) should be(-1)
triedInt2.getOrElse(-1) should be(80)
```

Lab: Seeing `Try` up close

Step 1: In the *src/test/scala* directory and in the `com.akkatraining.scala.futures` package, open *FuturesSpec.scala*

Step 2: Go to the specifications "Case 9:" and review.

Step 3: Run "Case 9:" in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.futures.FutureSpec -- -z "Case 9:"
```

Step 4: Change code and ask questions

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
```

Lab: Processing a `Future` with `onComplete`

Step 1: Continuing inside `com.xyzcorp.futures.FuturesSpec`, we will have an

Step 2: Go to the specification "Case 10:" and review.

Step 3: Given the following `Future` called `eventualString` handle the response using `onComplete` given the signature you see in the previous page. Use `Await.ready` or `Await.result`, however you see fit.

```
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 4: Be sure to take out the `pending` at the beginning of the test

Step 5: We will then run the test in your IDE or on in SBT using

```
> testOnly com.xyzcorp.futures.FuturesSpec -- -z "Case 10:"
```

Using an `ActorSystem` dispatcher for an `implicit` thread pool

- You can use an `ActorSystem` dispatcher as a thread pool for any `Future`

```
val actorSystem = ActorSystem("TestSystem")
import actorSystem.dispatcher
```

Lab: Importing an `ActorSystem.dispatcher` up close

Step 1: In the *src/test/scala* directory and in the `com.akkatraining.scala.futures` package, open *FuturesSpec.scala*

Step 2: Go to the specification "Case 11:" and review.

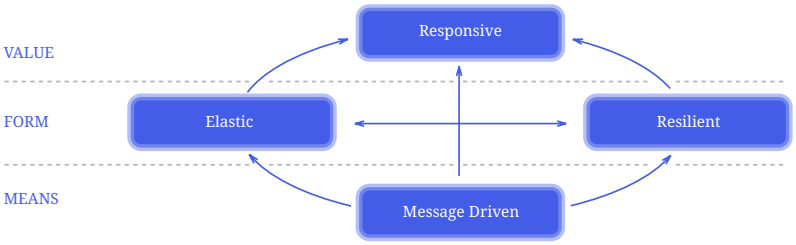
Step 3: Run "Case 11:" in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.futures.FutureSpec -- -z "Case 11:"
```

Step 4: Change code and ask questions

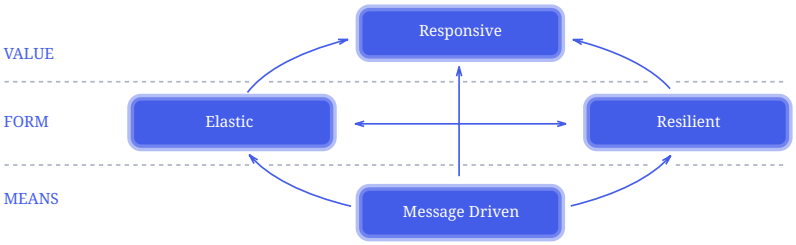
Reactive

Reactive Architecture



Reactive Manifesto (http://reactive-manifesto.org)

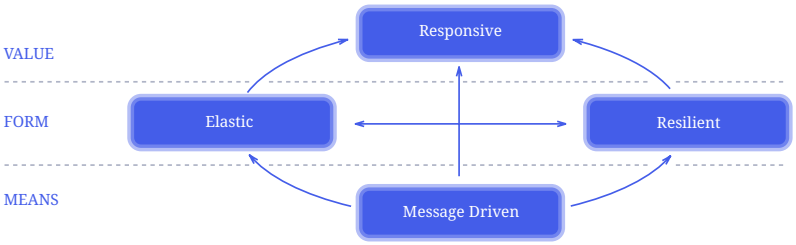
Responsive



- System responds in a timely manner if at all possible.
- Problems may be detected quickly and dealt with effectively.
- Responsive systems focus on providing rapid and consistent response times,
- Establishing reliable upper bounds so they deliver a consistent quality of service.
- Consistent behavior simplifies
  - Error handling
  - Builds end user confidence
  - Encourages further interaction.

Reactive Manifesto (http://reactive-manifesto.org)

Resilient

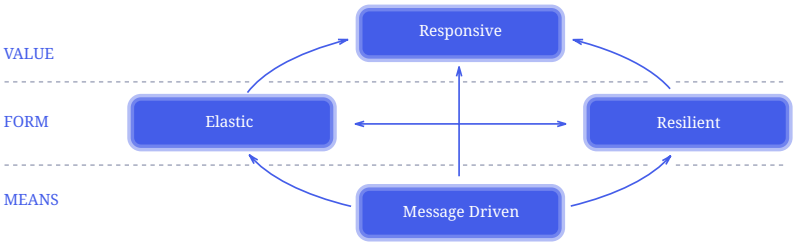


- The system stays responsive in the face of failure.

- Any system that is not resilient will be unresponsive after a failure.
- Achieved by replication, containment, isolation and delegation.
- Failures are contained within each component
- Isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole.
- Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary.
- The client of a component is not burdened with handling its failures.

Reactive Manifesto (http://reactive-manifesto.org)

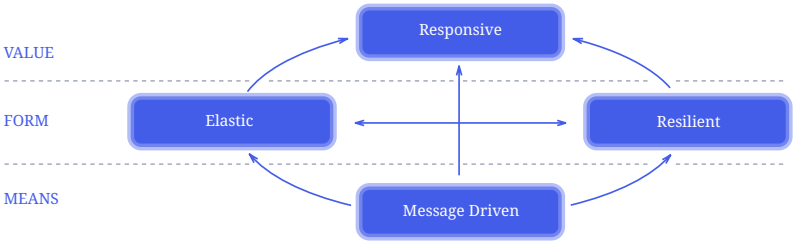
Elastic



- The system stays responsive under varying workload.
- React to changes in the input rate by increasing or decreasing the resources allocated to service these inputs.
- Implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them.
- Reactive Systems support predictive, as well as reactive, scaling algorithms by providing relevant live performance measures.
- They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

Reactive Manifesto (http://reactive-manifesto.org)

Message-Driven



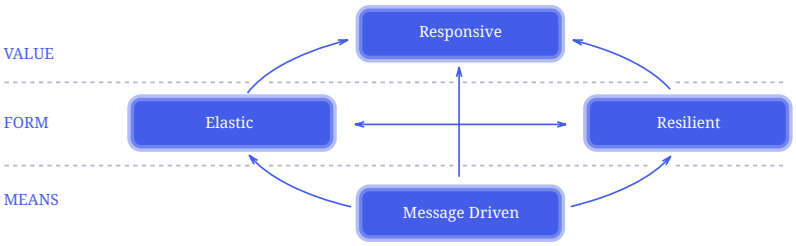
- Rely on asynchronous message-passing to establish a boundary between components that ensures:
  - Loose coupling

- Isolation
- Location Transparency.

- This boundary also provides the means to delegate failures as messages.

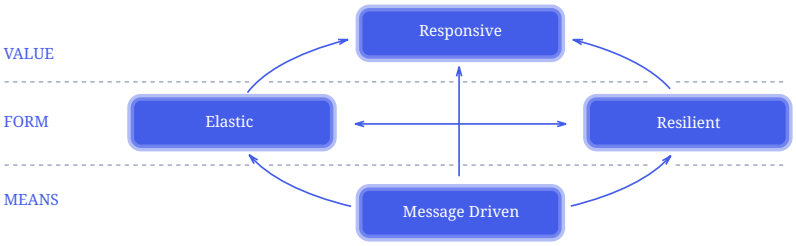
Reactive Manifesto (http://reactive-manifesto.org)

Message-Driven Continued



- Employing explicit message-passing enables:
  - Load management
  - Elasticity
  - Flow control

Message-Driven Flow Control



- Flow control can be used to:
  - Shape and monitor message queues
  - Apply back-pressure when necessary.
- Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host.
- Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

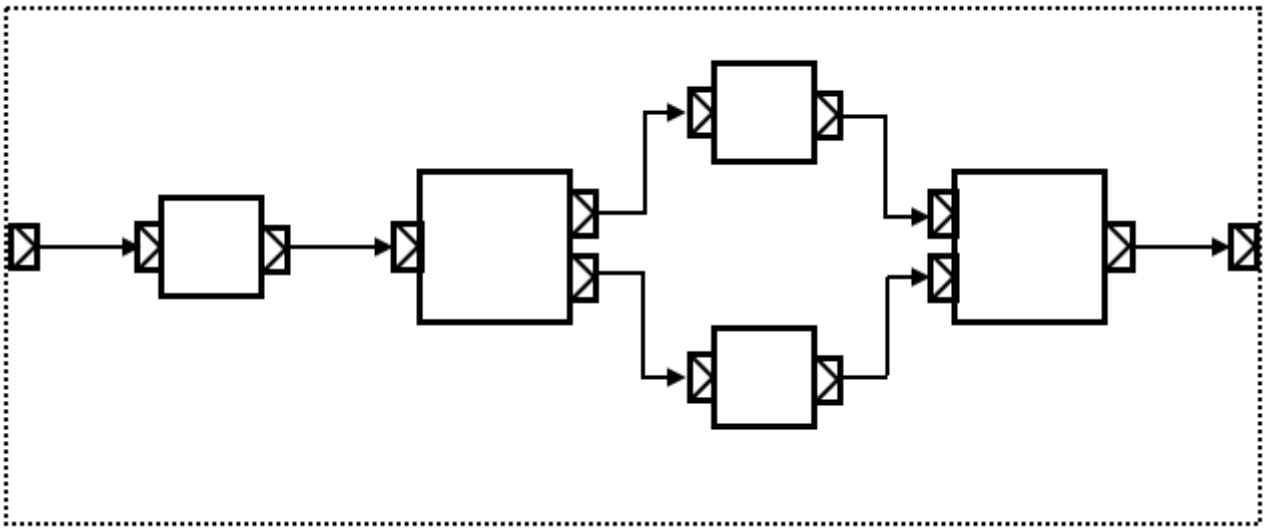
Reactive Manifesto (http://reactive-manifesto.org)

Non Blocking Communication

- Reactive is also about:

- Non-blocking processes (`Future`) that do work
- Knowing where to perform blocking and where not to
- Prefer non-blocking in most cases
- Tell the computer what to do, leave instructions when the future return, and leave

Reactive Programming



- Not be confused with Reactive Architecture
- Be able to process information as forks
- Each fork is able may or may not converge
- Use of functional programming to perform task(s)
- Ability to add asynchronous boundaries
- We will see more with Akka Streams

Configuration

Lightbend Config

- https://github.com/lightbend/config
- Implemented in plain Java with no dependencies
- Supports files in three formats: Java properties, JSON, HOCON
- Merges multiple files across all formats
- Can load from files, URLs, or classpath
- Support for "nesting" (treat any subtree of the config the same as the whole config)
- Override the config with Java system properties, `java -Dmyapp.foo.bar=10`
- Configuring an app, with its framework and libraries, all from a single file such as *application.conf*
- Parses duration and size settings, `512k` or `10 seconds`
- Converts types
  - If you ask for a `boolean` and the value is the string "yes"
  - If you ask for a `float` and the value is an `int`, it will figure it out.

A quick intro to HOCON

- "Human-Optimized Config Object Notation"
- JSON-Like Features
- Typical configuration file: *application.conf*

Sample HOCON Configuration

```
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:mem:play"
db.default.user=sa
db.default.password=""
```

Sample HOCON Configuration (Alternate 1)

```
db {
  default.driver=org.h2.Driver
  default.url="jdbc:h2:mem:play"
  default.user=sa
  default.password=""
}
```

Sample HOCON Configuration (Alternate 2)

```
db {
  default {
    driver=org.h2.Driver
    url="jdbc:h2:mem:play"
    user=sa
    password=""
  }
}
```

HOCON features:

- Comments (Using `#`)
- Includes
- Substitutions (`"foo" : ${bar}`, `"foo" : Hello ${who}`)
- Properties-like notation (`a.b=c`)
- Less noisy, more lenient syntax
- Substitute environment variables (`logdir=${HOME}/logs`)
- API based on immutable `Config` instances, for thread safety and easy reasoning about config transformations

application.conf

- Default configuration file name
- All HOCON
- One ***application.conf*** per application
- Typically stored ***src/main/resources***
- But you can also use:
  - *application.json*
  - *application.properties*

reference.conf

- Default HOCON settings
- Overridable by *application.conf*
- Typically packages by library with default settings
  - e.g. Akka contains *reference.conf*
  - Akka's `reference.conf`

All `akka` information is stored naturally within `akka` tag

Here is a sample configuation of a sample *application.conf* with `akka` configuration:

```
akka {
  loggers = ["akka.event.slf4j.Slf4jLogger"]

  loglevel = "DEBUG"

  stdout-loglevel = "DEBUG"

  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"

  actor {
    provider = "cluster"

    default-dispatcher {
      throughput = 10
    }
  }

  remote {
    netty.tcp.port = 4711
  }
}
```

Reading a configuration from a `String`

- Though not quite as useful, we can put our configuration into a `String` and parse it

```
val str = """a{b = 4}"""
val config = ConfigFactory.parseString(str)
config.getInt("a.b")
```

```
val str = """a.b = 50"""
val config = ConfigFactory.parseString(str)
config.getInt("a.b")
```

Lab: Viewing the parsing of a `String`

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.config` located the `ConfigSpec`.

Step 2: Go to the specifications "Case 1:" through "Case 2:" and review.

Step 3: Run "Case 1:" through "Case 2:" one at a time in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.config.ConfigSpec -- -z "Case 1:"
```

Step 4: Change code and ask questions

Reading a configuration from an file

- We can read from the actor when ever we need using our configuration by calling

```
val config = ConfigFactory.load()
```

- We can also read from a specific file by using the file name and leaving out the extension

```
val config = ConfigFactory.load("myconfig")
```

Reading from *application.conf*

- You can read from *application.conf* which is the configuration standard using the *resources prefix* in this case that is: `application`

```
val config = ConfigFactory.load("application")
val loggingFilter = config.getString("akka.logging-filter")
```

- You can also read from the *application.conf* by calling just `load()` since reading from the *application.conf* is the default

```
val config = ConfigFactory.load()
val loggingFilter = config.getString("akka.logging-filter")
```

Lab: Parsing of the *application.conf*

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.config` located the `ConfigSpec`.

Step 2: Go to the specifications "Case 3:" through "Case 4:" and review.

Step 3: Run "Case 3:" through "Case 4:" one at a time in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.config.ConfigSpec -- -z "Case 3:"
```

Step 4: Change code and ask questions

Overriding properties using system properties

- You can override properties from either the command line or via `System.setProperty` and that setting will take hold.
- This is a great way to change settings, like servers for deployment
- To set a system property from the command, use the form:

```
-Dfoo.bar.baz="true"
```

- To do so programmatically, use `System.setProperty(…)`
- To clear all settings programatically before `ConfigFactory.load` use `invalidateCaches`

Lab: Overriding properties using system properties

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.config` located the `ConfigSpec`.

Step 2: Go to the specifications "Case 5:" and review.

Step 3: Run "Case 5:" in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.config.ConfigSpec -- -z "Case 5:"
```

Step 4: Change code and ask questions

Reading a different configuration within a configuration file

- A configuration file can be subdivided into logical sections that can inherit one another
- These are called a `config`
- In the configuration, `context-a` can be read as a configuration with it's own `a.b.c` value
- You can also use the root as a fallback in case a value is unavailable in a config, like `a.b.c.d`

```
a {
   b {
      c = 19
      d = true
   }
}

context-a {
   a {
      b {
         c = 30
      }
   }
}
```

Programmatically reading in configurations

- Given the previous configuration, assuming the file prefix is called *sample3* and reading in `a.b.c` will render `30` (See previous slide)

```
val config = ConfigFactory
               .load("sample3")
               .getConfig("context-a")
config.getDouble("a.b.c")
```

- Since `a.b.c.d` is not available in `context-a`, we can still read `d` if the fallback is the root
- This will require that you assign the original configuration to a value in order to do so

```
val originalConfig = ConfigFactory.load("sample3")
val config = originalConfig
               .getConfig("context-a")
               .withFallback(originalConfig)
originalConfig.getBoolean("a.b.d")
```

Lab: Reading a different configuration within a configuration file

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.config` located the `ConfigSpec`.

Step 2: Go to the specifications "Case 6:" through "Case 7:" and review.

Step 3: Run "Case 6:" through "Case 7:" one at a time in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.config.ConfigSpec -- -z "Case 6:"
```

Step 4: Change code and ask questions

Reading in JSON

- A configuration file can also be read as JSON.
- JSON is more verbose that HOCON, so favor HOCON whenever possible

Sample JSON File

```
{
  "d": {
    "d1": "Hello",
    "d2": "World"
  },
  "e": {
    "e1": "Bon Jour",
    "e2": "Monde"
  }
}
```

- Reading from code doesn't look any different

Reading from Scala file

```
val originalConfig = ConfigFactory.load("sample5")
originalConfig.getString("d.d2")
```

Lab: Reading in JSON

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.config` located the `ConfigSpec`.

Step 2: Go to the specifications "Case 8:" and review.

Step 3: Run "Case 8:" in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.config.ConfigSpec -- -z "Case 8:"
```

Step 4: Change code and ask questions

Including configuration files

- You can include from other configuration files using the `include` directive.
- It will essentially graft the entries from one file into the exact location where `include` is located.
- Can be used with both HOCON, and JSON

JSON file

```
{
  "d": {
    "d1": "Hello",
    "d2": "World"
  },
  "e": {
    "e1": "Bon Jour",
    "e2": "Monde"
  }
}
```

HOCON File with an include

```
a {
   b {
      c = 19
      d = true
      include "/sample5.json"
   }
}
```

- Therefore there will be `a.b.c.d.d1` coordinate in the file

Lab: Including configuration files

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.config` located the `ConfigSpec`.

Step 2: Go to the specification "Case 9:" review.

Step 3: Run "Case 9:" in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.config.ConfigSpec -- -z "Case 9:"
```

Step 4: Change code and ask questions

Substitutions with HOCON

- Substitutions can be made from previously set values within the configuration file
- Substitutions are made using `${…}` and using the coordinates within

```
a {
   b {
      c = 19
      d = true
      e = ${a.b.d}
   }
}
```

Lab: Substitutions with HOCON

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.config` located the `ConfigSpec`.

Step 2: Go to the specification "Case 10:" review.

Step 3: Run "Case 10:" in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.config.ConfigSpec -- -z "Case 10:"
```

Step 4: Change code and ask questions

Reading in Lists

- Lists can also be read in from the Configuration API.
- Unfortunately, determining the type is up to you, and will return an `AnyRef` in Scala, or `Object` in Java
- To get the underlying list, you will need to called `unwrapped`

The HOCON with a list at `a.b.d`

```
a {
   b {
      c = 30
      d = ["Red", "Orange", "Green", "Blue"]
   }
}
```

```
val originalConfig = ConfigFactory.load("sample1")
val configList: ConfigList = originalConfig.getList("a.b.d")
val list:List[AnyRef] = configList.unwrapped
```

Lab: Reading in Lists

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.config` located the `ConfigSpec`.

Step 2: Go to the specification "Case 11:" review.

Step 3: Run "Case 11:" in either your IDE, or within the SBT shell by running:

```
> testOnly com.akkatraining.scala.config.ConfigSpec -- -z "Case 11:"
```

Step 4: Change code and ask questions

Lab: Reading an entry from *application.conf*

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.config` located the `ConfigSpec`.

Step 2: Locate the test "Lab 1: Using the Config API find the netty hostname in the remote-akka configuration"

Step 3: Using the Config API locate the Netty hostname in the `remote-akka` configuration

Step 4: Run the test to ensure success

Ask Actors in Scala

Using the `!` or `?`

- `!` means “fire-and-forget”
  - e.g. send a message asynchronously and return immediately
  - Also known as "tell"
- `?` sends a message asynchronously and returns a `Future`
  - Represents a possible reply.
  - Also known as `ask`

This should look very familiar in the movie we saw, Erlang the Movie

Using the `sender()` reference

- A `sender` reference
  - Can reply to an `Actor` or `Future` that made a request
  - Is of type `ActorRef`
  - Can be reused and sent to other actors as a message

Sending a response back in Scala

```
sender() ! replyMsg
```

Using the `Ask` pattern

- To bring in functionality to use a `?`, then import `akka.pattern._`.
- This will implicitly provide a `?` method to an `ActorRef`

Importing `ask`

```
import akka.pattern._
```

- To use, make the call with `?` this returns a `Future[Any]`
- To convert it to something meaningful you can use `mapTo[…]`

Using `mapTo`

```
val future = actorRef ? Address("123 Main St", "San Diego", "CA")
future.mapTo[String]
```

Using `pipe` and `ask` combined with `Future` manipulation

```
import akka.pattern.{ ask, pipe }
import system.dispatcher // The ExecutionContext that will be used

final case class Result(x: Int, s: String, d: Double)
case object Request

implicit val timeout = Timeout(5 seconds) // needed for <code>?</code>
below

val f: Future[Result] =
  for {
    x ← ask(actorA, Request).mapTo[Int] // call pattern directly
    s ← (actorB ask Request).mapTo[String] // call by implicit
conversion
    d ← (actorC ? Request).mapTo[Double] // call by symbolic name
  } yield Result(x, s, d)

f pipeTo actorD // .. or ..
pipe(f) to actorD
```

Source: https://doc.akka.io/docs/akka/2.5/actors.html

Lab: `ask` Actor as a calculation

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.ask` locate the `AskActorSpec`.

Step 2: Go to the specification "Lab 1:", here you will create an `ActorSystem` called `actorSystem`

Step 3: In the test, add the following as a way to manually shut down the server. This assumes you called your `ActorSystem`, as `actorSystem` (See Step 2)

```
println("Press Enter to terminate")
scala.io.StdIn.readLine()
Await.ready(actorSystem.terminate(), 10 seconds)
```

Step 4: Make any necessary `import`s

Step 5: In *src/main/scala* create an actor in the `com.akka.training.scala.ask` package called `AskActor` that will take in a message that are `case class` called `Add` and `Subtract` that take two `Int` parameters. Use the ask pattern and `sender()` to return the result of the computation.

Step 6: In the `AskActorSpec` in *src/main/scala* and in the `com.akka.training.scala.ask`

package, create an instance of `AskActor` and ask a `Add` or `Subtract` instance to obtain the future reference using `?` and process the result by making an ScalaTest assertion.

Step 7: Run your lab using the following at the command line, or an IDE. Be aware that your IDE may or may not be aware of the "Press Enter to continue"

```
> testOnly com.akkatraining.scala.ask.AskActorSpec -- -z "Lab 1:"
```

Lab: Using two `ask` Actors and composing the result

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.ask` locate the `AskActorSpec`.

Step 2: In the `AskActorSpec`, and in "Lab 2:" create an instance of `AskActor` and ask a `Add` or `Subtract` actor to obtain the future reference using `?` and set it to a `val` with a name of your choosing

Step 3: Ask with another message to `Add` or `Subtract` and obtain that reference using `?` and set that to `val` with a name of your choosing

Step 4: Compose the two `Future` by multiplying both results using some of the `Future` techniques we discussed.

Step 5: Place code that shuts down the `ActorSystem` with `Await.ready` or `AwaitResult`

Step 6: Run your lab using the following at the command line, or an IDE.

```
> testOnly com.akkatraining.scala.ask.AskActorSpec -- -z "Lab 2:"
```

Lab: Using two `ask` Actors and composing the result with a for comprehesion

Step 1: In the project, akka-training, locate *src/test/scala* and in the package `com.akkatraining.scala.ask` locate the `AskActorSpec`.

Step 2: In the `AskActorSpec` in Lab 3, do the same thing as "Lab 2" and use for comprehensions to process both `Future` and multiply both results

Step 3: Run your lab using the following at the command line, or an IDE.

```
> testOnly com.akkatraining.scala.ask.AskActorSpec -- -z "Lab 3:"
```

Ask Actors in Java

Using Ask Actors with Java

- For the `ask` pattern in Java, use `PatternCS`
- `pipe` can be used to send messages over to an actor

```
import static akka.pattern.PatternsCS.ask;
import static akka.pattern.PatternsCS.pipe;

import java.util.concurrent.CompletableFuture;
Timeout t = Timeout.create(Duration.ofSeconds(5));

// using 1000ms timeout
CompletableFuture<Object> future1 =
  ask(actorA, "request", 1000).toCompletableFuture();

// using timeout from above
CompletableFuture<Object> future2 =
  ask(actorB, "another request", t).toCompletableFuture();

CompletableFuture<Result> transformed =
  CompletableFuture.allOf(future1, future2)
  .thenApply(v -> {
    String x = (String) future1.join();
    String s = (String) future2.join();
    return new Result(x, s);
  });

pipe(transformed, system.dispatcher()).to(actorC);
```

Stopping Actors

How to stop an actor?

- An actor can be stopped any of these ways:
  - Manual Stop
  - `PoisonPill`
  - Killing the `Actor`
  - `GracefulStop`

Actor `stop`

- This is a manual `stop` of the `Actor`
- To perform a `stop` use the `ActorSystem` and `stop`
- `stop` is an asynchronous computation
- Does not immediately terminate the `Actor` and will continue processing the message

Actor `stop` in Scala

```
val actorSystem = ...
actorSystem.stop(myActorRef)
```

Actor `stop` in Java

```
ActorSystem actorSystem = ...
actorSystem.stop(myActorRef)
```

Actor `PoisonPill`

- A `PoisonPill` poisons the queue with a message
- Will be handled after messages that were already queued in the mailbox.
- Once the `Actor` consumes the `PoisonPill` it is killed off

```
myActor ! PoisonPill.getInstance
```

Actor `Kill`

- `akka.actor.Kill` will kill the actor
- Killing will cause `ActorKilledException` to be thrown from the `Actor`
- The `ActorKilledException`
  - Will trigger supervision if programmed
  - Causes parent how to handle the exception (or it's parent)

```
myActor ! Kill
```

DeathWatch

- `DeathWatch`
- An `Actor` can receive a message when ever another actor is `terminated`
- Call `context` and watch to watch when an actor is `terminated`
- Given that we have a parent actor, we can `watch` like the following:
- If the parent in turn does not handle the `Terminated` message it will itself fail with an `DeathPactException`

```
simpleActor = context.actorOf(Props[SimpleActor], "SimpleActor")
context.watch(simpleActor)
```

We can then setup message listeners when needed:

```
override def receive: PartialFunction[Any, Unit] = {
    ...
    case Terminated(actorRef) =>
       log.info("""Simple Actor has been terminated,
                   we know this because this actor
                   is a death watcher""");
}
```

GracefulStop

- Useful if you need to wait for termination or compose ordered termination of one or several actors
- When returned successfully, the actor's `postStop()` hook will have been executed
- There exists a happens-before edge between the end of `postStop()` and the return of `gracefulStop`()
- This would be a perfect time to use `Future.sequence` or `Future.traverse` to gracefully stop a series of `Actors`

```
import akka.pattern.gracefulStop
import scala.concurrent.Await

try {
  val stopped: Future[Boolean] =
      gracefulStop(actorRef, 5 seconds,
                   PoisonPill.instance)
  Await.result(stopped, 6 seconds)
  // the actor has been stopped
} catch {
  // the actor wasn't stopped within 5 seconds
  case e: akka.pattern.AskTimeoutException ⇒ ...
}
```

Stopping Actor Differences

`stop`	• Method can be applied to the `self` reference from inside an Actor then that Actor • Will not terminate immediately • Requires a `context`
`Kill`	• Throw an `ActorKilledException` • Will trigger supervision. • Similar to `PoisonPill`, processed as message
`PoisonPill`	• Will terminate the `Actor` permanently. • Added to mailbox. • Allows messages to finish processing
`GracefulStop`	• Uses terminal operation `Kill`, or `Poison Pill, or custom message • Returns a Future given a message when done • Use custom message type to interact with other actors before termination

Lab: Stopping an Actor using `DeathWatch` and `stop`

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.stopping` called `DeathListenerActor`

Step 2: Inside of `DeathListenerActor`

- Listen to a message of `ActorRef`, and `watch` it
- Listen to a message of `Terminated` and `log.info` which `Actor` terminated

Step 3: Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.stopping` called `VictimActor`, and just make this actor accept any `String` and print it to `log.info`

Lab: Stopping an Actor using `DeathWatch` and `stop` (Continued)

Step 4: In *src/test/scala* and in the package `com.akkatraining.scala.stopping` locate the `StoppingSpec` ang go to the specification in "Case 1:"

Step 5: Inside of "Case 1":

- Create an `ActorSystem` called `actorSystem`
- Create `DeathListenerActor` in `actorSystem`
- Create `VictimActor` in `actorSystem`
- Register `VictimActor` with `DeathListenerActor`
- Send five `String` messages to `VictimActor` and then `stop` the `Victim Actor`
- Place code that shuts down the `ActorSystem` with `Await.ready` or `AwaitResult`

Step 6: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.stopping.StoppingSpec -- -z "Case 1:"
```

Step 7: Notice the behavior

Lab: Poison Pill an Actor using `DeathWatch`

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.stopping` locate the `StoppingSpec` and go to the specification in "Case 2:"

Step 2: Inside of "Case 2":

- Create an `ActorSystem` called `actorSystem`
- Create `DeathListenerActor` in `actorSystem`
- Create `VictimActor` in `actorSystem`
- Register `VictimActor` with `DeathListenerActor`
- Send five `String` messages to `VictimActor` and then send a `PoisonPill` instance
- Place code that shuts down the `ActorSystem` with `Await.ready` or `AwaitResult`

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.stopping.StoppingSpec -- -z "Case 2:"
```

Step 4: Notice the behavior

Lab: Poison Pill an Actor with a `GracefulStop` using `DeathWatch`

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.stopping` locate the `StoppingSpec` and go to the specification in "Case 3:"

Step 2: Inside of "Case 3":

- Create an `ActorSystem` called `actorSystem`
- Create `DeathListenerActor` in `actorSystem`
- Create `VictimActor` in `actorSystem`
- Register `VictimActor` with `DeathListenerActor`
- Send five `String` messages to `VictimActor`
- Perform a `GracefulStop` with `PoisonPill` with a timeout of `3 seconds`
- Capture the `Future` of a graceful stop and either print the value of the `Future` or make an assertion on its value
- Place code that shuts down the `ActorSystem` with `Await.ready` or `AwaitResult`

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.stopping.StoppingSpec -- -z "Case 3:"
```

Step 4: Notice the behavior

Fault Tolerance

Child `Actor`

- The ability for an `Actor` to create a child gives it *fault tolerance*.
- Errors have a path on which to propagate
- Akka passes any `Exception` or `Error` to the parent, or *supervisor*
- Any child `Actor` that is killed, will throw an `ActorKilledException`
- The Supervisor `Actor` then dictates what to do next

Types of Strategies

- **One-for-one strategy** - Each child is treated separately
- **All-for-one strategy** - Any decision is applied to all children of the supervisor, not only the failing one

Options with problematic children

- Some options on how to handle a child `Actor` that caused an exception
  - Stop the `Actor`
  - Ignore the Failure
  - Continue as if nothing happened
  - Restart the `Actor`

This is where Akka's popular term "Let it crash" comes from

Example of a Supervisor `Actor`

```
import akka.actor.Actor

class Supervisor extends Actor {
  import akka.actor.OneForOneStrategy
  import akka.actor.SupervisorStrategy._
  import scala.concurrent.duration._

  override val supervisorStrategy =
    OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1 minute) {
      case _: ArithmeticException      ⇒ Resume
      case _: NullPointerException     ⇒ Restart
      case _: IllegalArgumentException ⇒ Stop
      case _: Exception                ⇒ Escalate
    }

  def receive = {
    case p: Props ⇒ sender() ! context.actorOf(p)
  }
}
```

- `maxNrOfRetries` the number of times a child actor is allowed to be restarted, negative value means no limit, if the limit is exceeded the child actor is stopped
- `withinTimeRange` duration of the time window for `maxNrOfRetries`, `Duration.Inf` means no window

Directives

- The match statement which forms the bulk of the body is of type `Decider` which is a `PartialFunction[Throwable, Directive]`
- The `Directives` that can be used are the following:
  - `Resume` - Resumes message processing for the failed Actor
  - `Restart` - Discards the old Actor instance and replaces it with a new then resumes message processing.
  - `Stop` - Stops the Actor
  - `Escalate` - Escalates the failure to the supervisor of the supervisor, by rethrowing the cause of the failure, i.e. the supervisor fails with the same exception as the child.

Example of a Child `Actor`

```
import akka.actor.Actor

class Child extends Actor {
  var state = 0
  def receive = {
    case ex: Exception ⇒ throw ex
    case x: Int        ⇒ state = x
    case "get"         ⇒ sender() ! state
  }
}
```

Default Supervisor Strategy

- `Escalate` is used if the defined strategy doesn't cover the exception that was thrown.
- When the supervisor strategy is not defined for an actor the following exceptions are handled by default:
  - `ActorInitializationException` will stop the failing child actor
  - `ActorKilledException` will stop the failing child actor
  - `DeathPactException` will stop the failing child actor
  - `Exception` will restart the failing child actor

If the exception escalate all the way up to the root guardian it will handle it in the same way as the default strategy defined above.

Lab: Creating Fault Tolerance in Akka

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.faulttolerance` called `AllForOneParentActor`

Step 2: Inside of `AllForOneParentActor`

- Listen to a message of `Props`, using the given `Props` create 10 children within the parent
- Send randomly one of children back to the `sender` using `scala.util.Random.nextInt`

Step 3: Define a `supervisorStrategy`,

- Make it a `AllForOneStrategy` with 3 retries, and a `Duration` of `1 second`.
- Program it so that an `IllegalArgumentException` is thrown we send it the `Restart` directive
- Program it so that anything else is given the `Escalate` directive

Lab: Creating Fault Tolerance in Akka (Continued)

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.faulttolerance` called `ChildActor`

Step 2: Inside of `ChildActor`

- Listen to a message of type `String` that says "IllegalArgumentException"
- Then throw an `IllegalArgumentException` with the message "Oh No" and the `self.path.name`

Step 3: Put in place the following lifecycle methods and `log.info` the events as they happen and call the `super` methods when done.

- `preStart`
- `postStop`
- `preRestart`
- `postRestart`

Lab: Creating Fault Tolerance in Akka (Continued)

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.faulttolerance` locate the `FaultToleranceSpec` and go to the specification in "Lab 1:..."

Step 2: Inside of "Lab 1:..."

- Create an `ActorSystem` called `actorSystem`
- Create an `AllForOneParentActor` in `actorSystem`
- Create `ChildActor` in `actorSystem`
- Create an `implicit` timeout using the following form:

```
implicit val timeout: Timeout = Timeout(10L, TimeUnit.SECONDS)
```

- ***Ask*** the all-for-one parent `ActorRef` with a `Prop` of `ChildActor`
- Retain the `Future` from the ask, and once returned send the child the message `"IllegalArgumentException"`
- Place code that shuts down the `ActorSystem` with `Await.ready` or `Await.result`

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.faulttolerance.FaultToleranceSpec --
-z "Lab 1:"
```

Step 4: Notice the behavior

Lab: Creating Fault Tolerance in Akka (Continued)

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.faulttolerance` called `OneForOneParentActor`

Step 2: Inside of `OneForOneParentActor`

- Listen to a message of `Props`, using the given `Props` create 10 children within the parent
- Send randomly one of children back to the `sender` using `scala.util.Random.nextInt`

Step 3: Define a `supervisorStrategy`

- Make it a `OneForOneStrategy` with 3 retries, and a `Duration` of `1 second`.
- Program it so that an `IllegalArgumentException` is thrown we send it the `Restart` directive
- Program it so that anything else is given the `Escalate` directive
- You can pretty much copy and paste from `AllForOneParentActor` and exchange the `SupervisorStrategy`

Lab: Creating Fault Tolerance in Akka (Continued)

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.faulttolerance` locate the `FaultToleranceSpec` and go to the specification in "Lab 2: ..."

Step 2: Inside of "Lab 2: ..."

- Create an `ActorSystem` called `actorSystem`
- Create a `OneForOneParentActor` in `actorSystem`
- Create `ChildActor` in `actorSystem`
- Create an `implicit` timeout
- ***Ask*** the parent `ActorRef` and give it a `Prop` of `ChildActor`
- Retain the `Future` from the ask, and once returned send the child the message `"IllegalArgumentException"`
- Place code that shuts down the `ActorSystem` with `Await.ready` or `Await.result`

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.faulttolerance.FaultToleranceSpec --
-z "Lab 2:"
```

Step 4: Notice the behavior

Lab: Creating Fault Tolerance in Akka (Continued)

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.faulttolerance` locate the `FaultToleranceSpec` and go to the specification in "Lab 3: ..."

Step 2: Inside of "Lab 3: ..."

- Create an `ActorSystem` called `actorSystem`
- Create a `OneForOneParentActor` in `actorSystem`
- Create `ChildActor` in `actorSystem`
- Create an `implicit` timeout
- ***Ask*** the parent `ActorRef` and give it a `Prop` of `ChildActor`
- Retain the `Future` from the ask, and once returned ***kill*** the `ChildActor`
- Place code that shuts down the `ActorSystem` with `Await.ready` or `AwaitResult`

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.faulttolerance.FaultToleranceSpec --
-z "Lab 3:"
```

Step 4: Notice the behavior

Timers

Making events happen

- You can cause events to happen based on time:
  - Schedule Once
  - Periodic Events
- Events are messages

Use Cases

- Timeout
- Keeping a pulse for an operation
- Circuit Breaker Pattern

For cron operations, use Akka's quartz scheduler

Stopping Events

- You will get a `Cancellable` back that you can call `cancel` to cancel the execution of any scheduled operation

Schedule Once Example

```
import system.dispatcher
import scala.concurrent.duration.Duration._

system.scheduler.scheduleOnce(50 milliseconds, testActor, "foo")
```

Schedule Once Function

```
import system.dispatcher
import scala.concurrent.duration.Duration._

system.scheduler.scheduleOnce(50 milliseconds) {
  testActor ! System.currentTimeMillis
}
```

Perpetual Event

- `schedule` can be given an `initialDelay`, and `interval` to schedule a perpetual event

```
val tickActor = ...
val cancellable =
  system.scheduler.schedule(
    0 milliseconds,
    50 milliseconds,
    tickActor,
    tick)

cancellable.cancel()
```

Lab: Creating a Scheduler

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.schedulers` locate the `SchedulersSpec` and go to the specification in "Lab 1:..."

Step 2: Inside of "Lab 1:..."

- Create an `ActorSystem` called `actorSystem`
- Create a `SimpleActorScala` in `actorSystem`, the same `SimpleActorScala` you created at the beginning of class
- Schedule a message of `"Ping"` every 5 seconds

Step 3: Sleep on the `main` thread for 30 seconds, and then cancel the scheduler

Step 4: Shut down the `actorSystem` appropriately

Step 5: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.schedulers.SchedulersSpec -- -z "Lab
1:"
```

Step 6: Notice the behavior

Dispatchers

About Dispatchers

- An Akka `MessageDispatcher` is what makes Akka Actors “tick”
- It is the engine or thread pool of the `ActorSystem` so to speak
- All `MessageDispatcher` implementations are also an `ExecutionContext`
  - That means that they can be used to execute arbitrary code, for instance `Future`

Default Dispatcher

- Every `ActorSystem` will have a default dispatcher if one is not configured
- The `default-dispatcher` can be configured in the *application.conf*
- If an `ActorSystem` is created with an `ExecutionContext` then that will be the default dispatcher
- If no dispatcher is configured or supplied, then the default is a Fork-Join pool specified in `akka.actor.default-dispatcher.default-executor.fallback`

Fork-Join Dispatcher

- Employing work-stealing:
  - All threads in the pool attempt to find and execute tasks submitted to the pool and/or created by other active tasks
  - Eventually blocking waiting for work if none exist)
  - This enables efficient processing when most tasks spawn other subtasks
- More information about Java’s Fork Join Pool Executor
- `type` descibes that the configuration is a `Dispatcher`
- `executor` is the type of executor

```
my-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "fork-join-executor"
  # Configuration for the fork join pool
  fork-join-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 2
    # Parallelism (threads) ... ceil(available processors * factor)
    parallelism-factor = 2.0
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 10
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

`parallelism-max` and `parallelism-min` are the number of *hot* threads to avoid latency

Thread Pool Executor

- An `ExecutorService` that executes each submitted task using one of possibly several pooled threads
- Urged to use:
  - `Executors.newCachedThreadPool()`
  - `Executors.newFixedThreadPool(int)`
  - `Executors.newSingleThreadExecutor()`

```
blocking-io-dispatcher {
  type = Dispatcher
  executor = "thread-pool-executor"
  thread-pool-executor {
    fixed-pool-size = 32
  }
  throughput = 1
}
```

Creating an Actor with a certain dispatcher

From the `ActorSystem`

```
import akka.actor.Props
val myActor = context.actorOf(Props[MyActor], "myactor")
```

Inside `application.conf`

```
akka.actor.deployment {
   /myactor {
       dispatcher = my-dispatcher
   }
}
```

Alternate to create `Actor` with a `Dispatcher`

Creating an `Actor` *with a certain* `Dispatcher` *from application.conf*

```
import akka.actor.Props
val props = Props[MyActor].withDispatcher("my-dispatcher")
val myActor = context.actorOf(props, "myactor1")
```

That `my-dispatcher` is the path in *application.conf* and you can put full paths(e.g. `"foo.bar.my-dispatcher"`)

Standard Form `Dispatcher`

- This is an event-based dispatcher that binds a set of Actors to a thread pool.
- It is the default dispatcher used if one is not specified.
- Sharability: Unlimited
- Mailboxes: Any, creates one per Actor
- Use cases: Default dispatcher, Bulkheading
- Driven by: `java.util.concurrent.ExecutorService`.
- For `executor`
  - `fork-join-executor`
  - `thread-pool-executor`
  - FQCN of an `akka.dispatcher.ExecutorServiceConfigurator`

Source: https://doc.akka.io/docs/akka/current/dispatchers.html

`PinnedDispatcher`

This dispatcher dedicates a unique thread for each actor using it i.e. each actor will have its own

thread pool with only one thread in the pool.

- Sharability: None
- Mailboxes: Any, creates one per Actor
- Use cases: Bulkheading
- Driven by: Any `akka.dispatch.ThreadPoolExecutorConfigurator`. By default a `thread-pool-executor`

Source: https://doc.akka.io/docs/akka/current/dispatchers.html

`CallingThreadDispatcher`

This dispatcher runs invocations on the current thread only. This dispatcher does not create any new threads, but it can be used from different threads concurrently for the same actor.

- Sharability: Unlimited
- Mailboxes: Any, creates one per Actor per Thread (on demand)
- Use cases: Testing
- Driven by: The calling thread

Source: https://doc.akka.io/docs/akka/current/dispatchers.html

`Affinity Pool Dispatcher`

- Increase throughput in cases where there is relatively small number of actors that maintain some internal state.
- Tries its best to ensure that an actor is always scheduled to run on the same thread.
- Aims to decrease CPU cache misses which can result in significant throughput improvement.

```
affinity-pool-dispatcher {
  # Dispatcher is the name of the event-based dispatcher
  type = Dispatcher
  # What kind of ExecutionService to use
  executor = "affinity-pool-executor"
  # Configuration for the thread pool
  affinity-pool-executor {
    # Min number of threads to cap factor-based parallelism number to
    parallelism-min = 8
    # Parallelism (threads) ... ceil(available processors * factor)
    parallelism-factor = 1
    # Max number of threads to cap factor-based parallelism number to
    parallelism-max = 16
  }
  # Throughput defines the maximum number of messages to be
  # processed per actor before the thread jumps to the next actor.
  # Set to 1 for as fair as possible.
  throughput = 100
}
```

Source: https://doc.akka.io/docs/akka/current/dispatchers.html

CPU Cache Miss is a failed attempt to read or write a piece of data in the cache, which results in a main memory access with much longer latency. There are three kinds of cache misses: instruction read miss, data read miss, and data write miss.

Lab: Running with a Default Dispatcher

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.dispatchers` locate the `DispatcherSpec` and go to the specification in "Case 1:..."

Step 2: This test runs with a `BlockingActor` and a `PrintActor`. Review the code in *src/main/scala*

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.faulttolerance.FaultToleranceSpec --
-z "Case 1:"
```

Step 4: Notice the behavior

Lab: Running with a Default Dispatcher and a Future

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.dispatchers` locate the `DispatcherSpec` and go to the specification in "Case 2:..."

Step 2: This test runs with a `BlockingFutureActor` and a `PrintActor`. Review the code in *src/main/scala*. Notice the `executionContext` in `BlockingFutureActor`

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.faulttolerance.FaultToleranceSpec --
-z "Case 2:"
```

Step 4: Notice the behavior

Lab: Running with different dispatchers

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.dispatchers` locate the `DispatcherSpec` and go to the specification in "Case 3:..."

Step 2: This test runs with a `SeparateDispatcherFutureActor` and a `PrintActor`. Review the code in *src/main/scala*. Notice the `executionContext` in `SeparateDispatcherFutureActor`

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.faulttolerance.FaultToleranceSpec --
-z "Case 3:"
```

Step 4: Notice the behavior

Mailboxes

Mailboxes

- An Akka Mailbox holds the messages that are destined for an `Actor`.
- Normally each Actor has its own mailbox
- Can be customized using for example a `BalancingPool` where all routees will share a single mailbox instance.

Built In Mailbox Types

Akka comes shipped with a number of mailbox implementations:

- `UnboundedMailbox` – The default mailbox
  - Backed by a java.util.concurrent.ConcurrentLinkedQueue
  - Blocking: No
  - Bounded: No
  - Configuration name: `unbounded` or `akka.dispatch.UnboundedMailbox`
- `SingleConsumerOnlyUnboundedMailbox`
  - Backed by a very efficient Multiple Producer Single Consumer queue, cannot be used with `BalancingDispatcher`
  - Blocking: No
  - Bounded: No
  - Configuration name: `akka.dispatch.SingleConsumerOnlyUnboundedMailbox`

Built In Mailbox Types (Continued)

- `BoundedMailbox`
  - Backed by a `java.util.concurrent.LinkedBlockingQueue`
  - Blocking: Yes
  - Bounded: Yes
  - Configuration name: “bounded” or “akka.dispatch.BoundedMailbox”
- `UnboundedPriorityMailbox`
  - Backed by a `java.util.concurrent.PriorityBlockingQueue`
  - Blocking: Yes
  - Bounded: No
  - Configuration name: `akka.dispatch.UnboundedPriorityMailbox`

Built In Mailbox Types (Continued)

- `BoundedPriorityMailbox`
  - Backed by a `java.util.PriorityBlockingQueue` wrapped in an `akka.util.BoundedBlockingQueue`
  - Blocking: Yes
  - Bounded: Yes
  - Configuration name: `akka.dispatch.BoundedPriorityMailbox`

Requiring a mailbox per dispatcher

- For whatever dispatcher is configured a mailbox can be set
- This requires a `mailbox-requirement` per dispatcher

```
my-dispatcher {
  mailbox-requirement = org.example.MyInterface
}
```

Specifying a default mailbox type

- To select a default mailbox, you can configure that at `akka.actor.default-mailbox`

```
akka.actor.default-mailbox {
  mailbox-type = "akka.dispatch.SingleConsumerOnlyUnboundedMailbox"
}
```

Specifying a mailbox per actor

Inside the application.conf

```
non-blocking-bounded {
   mailbox-type = "akka.dispatch.NonBlockingBoundedMailbox"
   mailbox-capacity = 10
}

akka.actor.deployment {
   /myActor {
      mailbox = non-blocking-bounded
   }
}
```

When creating the actor

```
import akka.actor.Props
val myActor = actorSystem.actorOf(Props[MyActor], "myActor")
```

Specifying a mailbox per actor, alternative

Inside the application.conf

```
non-blocking-bounded {
   mailbox-type = "akka.dispatch.NonBlockingBoundedMailbox"
   mailbox-capacity = 10
}
```

- In the following we call `withMailbox` from the `Props` to specify mailbox

When creating the actor

```
import akka.actor.Props
val props = Props[MyActor].withMailbox("non-blocking-bounded")
val myActor = actorSystem.actorOf(props, "myActor")
```

Specifying `mailbox` semantics per actor

- First specify what kind of Message Queue an `Actor` will require
- Basic configurations are located in `akka.dispatch` and `Mailbox`
- These are just marker `trait` that certain `Mailboxes` will specify
- This is a way to demand that an `Actor` always has a certain mailbox

`MyBoundedActor` *demands* `BoundedMessageQueueSemantics`

```
import akka.dispatch.RequiresMessageQueue
import akka.dispatch.BoundedMessageQueueSemantics

class MyBoundedActor extends MyActor
  with RequiresMessageQueue[BoundedMessageQueueSemantics]
```

Seeking the right implementation

Some implementations, consulting the Akka API:

- `BoundedControlAwareMailbox.MessageQueue`
- `BoundedDequeBasedMailbox.MessageQueue`
- `BoundedMailbox.MessageQueue`

- `BoundedNodeMessageQueue`
- `BoundedPriorityMailbox.MessageQueue`
- `BoundedStablePriorityMailbox.MessageQueue`

https://doc.akka.io/japi/akka/current/akka/dispatch/BoundedMessageQueueSemantics.html

Mapping the mailbox semantics in *application.conf*

```
bounded-mailbox {
  mailbox-type = "akka.dispatch.NonBlockingBoundedMailbox"
  mailbox-capacity = 1000
}

akka.actor.mailbox.requirements {
  "akka.dispatch.BoundedMessageQueueSemantics" = bounded-mailbox
}
```

How Mailboxes are resolved

Prioritization how mailboxes are resolved:

1. If the actor's deployment configuration section contains a `mailbox` key then that names a `configuration` section describing the mailbox type to be used.
2. If the actor's `Props` contains a mailbox selection—i.e. `withMailbox` was called on it — then that names a configuration section describing the mailbox type to be used (note that this needs to be an absolute `config` path, for example `myapp.special-mailbox`, and is not nested inside the `akka` namespace).
3. If the dispatcher's configuration section contains a `mailbox-type` key the same section will be used to configure the mailbox type.
4. If the actor requires a `mailbox` type as described using semantic mapping for that requirement will be used to determine the mailbox type to be used; if that fails then the dispatcher's requirement—if any—will be tried instead.
5. If the dispatcher requires a mailbox type as described then the mapping for that requirement will be used to determine the mailbox type to be used.
6. The default mailbox `akka.actor.default-mailbox` will be used.

Source: https://doc.akka.io/docs/akka/2.5/mailboxes.html

Lab: View the default behavior of a mailbox

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.mailboxes` locate the `MailboxSpec` and go to the specification in "Case 1:..."

Step 2: "Case 1:" Runs an `Actor` using the default mailbox

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.mailboxes.MailboxesSpec -- -z "Case
1:"
```

Step 4: Notice the behavior

Lab: Run a custom mailbox using *application.conf*

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.mailboxes` locate the `MailboxSpec` and go to the specification in "Case 2:..."

Step 2: "Case 2:" Runs an `Actor` using a mapping in the *application.conf*. Look for the configuration `custom-mailbox-akka` and see how it is mapped to `mailboxActor`

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.mailboxes.MailboxesSpec -- -z "Case
2:"
```

Step 4: Notice the behavior

Lab: Run a custom mailbox using directly using *application.conf*

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.mailboxes` locate the `MailboxSpec` and go to the specification in "Case 3:..."

Step 2: "Case 3:" Runs an `Actor` using a mapping in the *application.conf*. Look for the configuration `custom-mailbox-akka`. Only this time notice how we are not mapping directly to the actor name in *application.conf* but when we create the `Actor`.

Step 3: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.mailboxes.MailboxesSpec -- -z "Case
3:"
```

Step 4: Notice the behavior

Lab: Run an `Actor` with a semantic constraint

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.mailboxes` locate the `MailboxSpec` and go to the specification in "Case 4:..."

Step 2: "Case 4:" Runs an `Actor` that has a semantic constraint. It uses a mapping in the *application.conf*. Look for the configuration `custom-priority-mailbox-akka` in the *application.conf*. Noticed that we are defining an implementation for the semantic requirement: `akka.dispatch.BoundedMessageQueueSemantics` which is tied to the declaration for `non-blocking bounded`

Step 3: In *src/main/scala* and in the package `com.akkatraining.scala.mailboxes`, open `MailboxWithRequirementActor` and noticed the requirements `trait` with RequiresMessageQueue[BoundedMessageQueueSemantics] and match it with the mapping in the *application.conf* in Step 2.

Step 4: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.mailboxes.MailboxesSpec -- -z "Case
4:"
```

Step 5: Notice the behavior

Lab: Run a custom `Mailbox`

Step 1: In *src/test/scala* and in the package `com.akkatraining.scala.mailboxes` locate the `MailboxSpec` and go to the specification in "Case 5:..."

Step 2: "Case 5:" Uses a custom mailbox, where it is configured in configuration `"custom-priority-mailbox-akka"`. In this case the mailbox used is `"custom-priority-mailbox"` where it will give messages with the `Symbol 'highpriority` first.

Step 3: In *src/main/scala* and in the package `com.akkatraining.scala.mailboxes`, locate `MyPriorityMailbox` and see how it is implemented

Step 4: Run the test in either your IDE or in SBT using the following

```
> testOnly com.akkatraining.scala.mailboxes.MailboxesSpec -- -z "Case
5:"
```

Step 4: Notice the behavior

Remoting

About Remoting

- When sending messages to actors you do not need to know whether it is on the local machine or remote
- This is referred to as *location transparency*
- Remoting will:
  - Serialize and Deserialize messages
  - Send over the network
- Akka will take care of the mechanics for us
- Discovery is only available to clustering

Serialization

- Serialization is pluggable
- Java Serialization is often times slow
- May need to swapped out for better options
  - Protocol Buffers
  - Kryo
  - etc.

Remoting Setup

- Remoting Setup is done by
  - Configuration
  - Programmatic
- We require the dependency in SBT (or your preferrable build tool)

```
libraryDependencies += "com.typesafe.akka" % "akka-remote_2.12" %
"2.5.13"
```

Location Transparency

- Vertical and Horizontal Growth
- Horizontal Growth driven by Remoting systems
- Vertical Growth driven by routers

- All configuration based
- Be able to perform calculations without being concerned about the location

Establishing Remoting

```
akka {
  actor {
    provider = remote
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = 2552
    }
 }
}
```

- Change `provider` from `local` to `remote`
- Add a `hostname`, it must be identifiable on the network
- Add a port number

Types of remoting interaction

- **Lookup** - Lookup actors remotely with interaction `actorSelection(path)`
- **Creation** - Create actors remotely with interaction `actorOf(Props(…))`

Looking up Remote Actors

```
val selection =
  context.actorSelection
("akka.tcp://actorSystemName@10.0.0.1:2552/user/actorName")
```

Once obtained, you can send information to that `Actor`

```
selection ! "Pretty awesome feature"
```

Create an actor remotely

- An `Actor` can also be created on another machine as well
- Amend your *application.conf* and add under `akka.actor.deployment` what seems to be a local

address but mapped to a remote location

```
akka {
  actor {
    deployment {
      /sampleActor {
        remote = "akka.tcp://sampleActorSystem@127.0.0.1:2553"
      }
    }
  }
}
```

```
val actor = system.actorOf(Props[SampleActor], "sampleActor")
actor ! "Pretty slick"
```

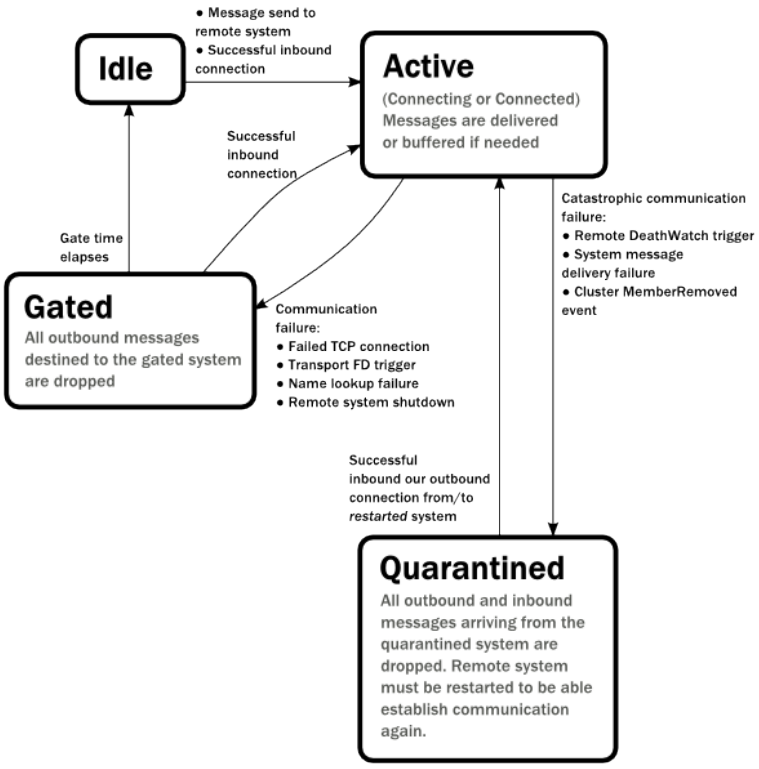
Remote Deployment Whitelist

- You can specify which actors can deployed on an akka system by specifying a white
- *Remote Deployment is not remote class loading*
- Classes would need to exist on the Akka instance
- To enable, set `akka.remote.deployment.enable-whitelist` to `on`

```
akka.remote.deployment {
  enable-whitelist = on

  whitelist = [
    "com.xyzcorp.MyActor"
  ]
}
```

Failure Detection Model



Description of the Detection Model

- `Idle` - Initial State
- `Active`
  - The first time a message is attempted to be sent to the remote system
  - Or an inbound connection is accepted
  - No failures
- `Gated`
  - Communication failure happens and the connection is lost between the two systems
  - The amount of retries before switching back to `Idle` is done `akka.remote.retry-gate-closed-for`
  - If successful, then it goes back to `Active`
- `Quarantined`
  - Used for unrecoverable failures
  - Permanent until system is back

Lab: Creating Remoting in Akka

Step 1: In the project, akka-training, Create two actors inside of *src/main/scala* and in the package `com.akkatraining.scala.remoting` called `ActorA` and `ActorB`

Step 2: Inside of `ActorA` and `ActorB`, print `log.info` everything

Step 3: Inside of *application.conf*, create a configuration called `remote-a` that connects to port `2551`

Step 4: Inside of *application.conf*, create a configuration called `remote-b` that connects to port `2552`

Lab: Creating Remoting in Akka (Continued)

Step 1: In the project, akka-training, Create a Runnable `object` of *src/main/scala* and in the package `com.akkatraining.scala.remoting` called `RemoteARunner`

Step 2: In `RemoteARunner`, load the `remote-a` configuration with a fallback to `ConfigFactory.load()`

Step 3: Create an `actorSystem` with the custom configuration

Step 4: Place code that shuts down the `ActorSystem` with `Press Enter to terminate` behavior

Step 5: Run the server in either your IDE or using SBT

```
> run
```

Do not create `ActorA` onto this system

Lab: Creating Remoting in Akka (Continued)

Step 1: In the project, akka-training, Create a Runnable `object` of *src/main/scala* and in the package `com.akkatraining.scala.remoting` called `RemoteBRunner`

Step 2: In `RemoteBRunner`, load the `remote-b` configuration with a fallback to `ConfigFactory.load()`

Step 3: Create an `actorSystem` with the custom configuration

Step 4: Place code that shuts down the `ActorSystem` with `Press Enter to terminate` behavior

Step 5: Run the server in either your IDE or using SBT

```
> run
```

Do not create `ActorB` onto this system

Lab: Creating Remoting in Akka (Continued)

Step 1: In the project, akka-training, go to *application.conf* create a configuration called `remote-client` that uses remoting and connects to port `2553`

Step 2: In *application.conf*, and in the `remote-client` configuration:

- map `akka.actor.deployment` for `/actorA` to `remote-a` ip-address and port associated
- map `akka.actor.deployment` for `/actorB` to `remote-b` ip-address and port associated

Step 3: Create a Runnable `object` in *src/main/scala* and in the package `com.akkatraining.scala.remoting` called `RemoteClientRunner`

Step 4: In `RemoteClientRunner`, load the `remote-client` configuration with a fallback to `ConfigFactory.load()`

Step 5: Create an `actorSystem` with the custom configuration

Step 6: Send messages to both `actorA` and `actorB` and notice the message on each of the terminals

Step 7: Place code that shuts down the `ActorSystem` with `Press Enter to terminate` behavior

Step 8: Run the server in either your IDE or using SBT

```
> run
```

Routers

About Routers

- Actor that reroutes message to other actors
- Different Strategies available
- Create your own programmatically
- A Router to the sender is just an `ActorRef`
- A Router will just distribute to routees

Out of the Box Routers

- `RoundRobinRoutingLogic`
  - Round Robin Distribution
  - Chooses either number of Routees **or** a list of Routees **not both**.
- `RandomRoutingLogic`
  - Chooses a routee randomly
- `SmallestMailboxRouter`
  - Chooses non-suspended routee with the least messages in its mailbox
- `BroadcastRouter`
  - Sends messages to all the routees!

Out of the Box Routers (Continued)

- `ScatterGatherFirstCompletedRouter`
  - Sends messages to all routees, gets a `Future`,
  - Whichever routee responds first, that reponse will be sent to the `sender()`
- `ConsistentHashingRouter`
  - Consistent Hashing
  - Special type of map where remapping of keys is limited

Programmatic Router

```
class Master extends Actor {
  var router = {
    val routees = Vector.fill(5) {
      val r = context.actorOf(Props[Worker])
      context watch r
      ActorRefRoutee(r)
    }
    Router(RoundRobinRoutingLogic(), routees)
  }

  def receive = {
    case w: Work ⇒
      router.route(w, sender())
    case Terminated(a) ⇒
      router = router.removeRoutee(a)
      val r = context.actorOf(Props[Worker])
      context watch r
      router = router.addRoutee(r)
  }
}
```

Source: https://doc.akka.io/docs/akka/current/routing.html#a-simple-router

Different Router Strategies

- Pool
  - The router creates `routees` as child actors
  - Removes them from the router if they terminate.
- Group
  - The routee actors are created externally to the router
  - Router sends messages to the specified path using actor selection
  - Does not watch for termination.

Using external configuration from *application.conf*

- The settings for a router actor can be defined in configuration or programmatically.
- To make use of an externally configurable router the `FromConfig` props wrapper must be used
- If the props of an actor is ***not*** wrapped in `FromConfig` it will ignore the router section of the deployment configuration.

Pool Configuration

- The following creates a round-robin router
- Sent to 5 workers

Setting up the router

```
akka.actor.deployment {
  /parent/router1 {
    router = round-robin-pool
    nr-of-instances = 5
  }
}
```

Using the router

```
val router1: ActorRef =
  context.actorOf(FromConfig.props(Props[Worker]), "router1")
```

Using a router programmatically

```
val router2: ActorRef =
  context.actorOf(RoundRobinPool(5).props(Props[Worker]), "router2")
```

The problem with `sender`

- When a routee sends a message, it will implicitly *set itself* as the `sender`

```
sender() ! x
```

- Since the `parent` is the parent of all routees, send the `parent`

```
sender().tell("reply", context.parent)
sender().!("reply")(context.parent)
```

Supervision of a router

- Routees that are created by a *pool router* will be created as the router's children
- The router is therefore also the children's supervisor
- The supervision strategy of the router actor can be configured with the `supervisorStrategy` property of the Pool
- If no configuration is provided, routers default to a strategy of ***always escalate***

- This means that errors are passed up to the router's supervisor for handling
- The router's supervisor will decide what to do about any errors
- This means that without a supervisor strategy, both router and routee have the same fate

Setting a Supervisor Strategy for a routee

```
val escalator = OneForOneStrategy(...) {
  case e:IllegalArgumentException
            ⇒ SupervisorStrategy.Escalate
}
val router = system.actorOf(
      RoundRobinPool(1, supervisorStrategy = escalator)
        .props(routeeProps = Props[MyActor]))
```

Group Routees Via Configuration

- You create the routees and you feed them to the `Router`
- Messages will be sent to `actorSelection`
- For example, the following routees have been pre-created

Establishing the Routees

```
akka.actor.deployment {
  /parent/router3 {
    router = round-robin-group
    routees.paths = ["/user/workers/w1",
                     "/user/workers/w2",
                     "/user/workers/w3"]
  }
}
```

Calling the routees

```
val router3: ActorRef =
  context.actorOf(FromConfig.props(), "router3")
```

Group Routees Via Programmatic Configuration

```
context.actorOf(
    RoundRobinGroup(List("/user/routee1",
                         " user/routee2")).props(),
                    "router4")
```

Lab: Creating a router in Akka

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.router` called `WorkerActor`

Step 2: In `WorkerActor`

- If the message received is the `String "terminate"`, stop the actor
- If the message received is any other `String "terminate"`, stop the actor
- If the message is not a `String`, send it to method `unhandled`

Lab: Creating a router in Akka (Continued)

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.router` called `BossActor`

Step 2: In `BossActor`

- Create a programmatic `RoundRobinPoolLogic` router
- If the message received is a `String`, then route to any of the programmatic routees
- If the message received is a `Terminated` object, remove the `routee` and add a replacement
- Decide on logging which will help you

Lab: Creating a router in Akka (Continued)

Step 1: In the project, akka-training, Create a runnable `object` inside of *src/main/scala* and in the package `com.akkatraining.scala.router` called `RouterRunner`

Step 2: Create an `actorSystem`

Step 3: Import the `actorSystem` dispatcher

Step 4: Create the `BossActor` inside of `ActorSystem` and obtain an `ActorRef` to it

Step 5: Create a `scheduler` that sends a message every 50 milliseconds. Be sure to obtain the `Cancellable` instance so you can cancel after you hit ***ENTER***. Send the current date and time using `java.time.LocalDateTime.now()`

Step 4: Place code that will listen to the user's `Press Enter to terminate` behavior, this time cancel the `Cancellable` from the `scheduler` and then shut down the `actorSystem`

Step 5: Run the server in either your IDE or using SBT

```
> run
```

Clustering

Automatic Management of Membership

- Relatively new component to Akka
- Akka Clustering adds a means to setup remoting over a number of remote systems
- Offers:
  - Location independence and transparency, even more so!
  - Actors communicate with one another, across the cluster
  - Addressing where actors believe that they are on the same cohesive system

Before Clustering

- Akka Actors needed some remote addressing to send information
- At times not truly as transparent as conceived

Akka Clustering Communication

- To message to a cluster, one node only needs access to *seed nodes*
- Seed nodes are specially designed node that can be used to connect to a cluster by new nodes
- Recommended you have more than one seed node
- As long as one of the seed nodes is available you can join
- No special code is required, configuration informs the location of the seed nodes

New Node Lifecycle

- New node then goes through a series of lifecycle steps before becoming a member
- A node can either
  - Start new actors in its own `ActorSystem`
  - Start new actors in another node in the cluster
- ***The Cluster*** becomes a single virtual actor system.

Gossiping and Convergence

- A *gossip* protocol
  - Is used to send messages to one another
  - Random peers communicate what information they have about cluster membership

- *Convergence*
  - Once the gossiping is done
  - When every node knows the membership structure of the cluster

Vector Clocks

- A data structure and algorithm
- Used to detect causality violations
- Updates using (node, counter) tuples
- Each update has an accompanying update to the vector clock

Clustering Dependency

```
"com.typesafe.akka" %% "akka-cluster" % "2.5.5"
```

Terms

node	A logical member of a cluster. There could be multiple nodes on a physical machine. Defined by a hostname:port:uid tuple.
cluster	A set of nodes joined together through the membership service.
leader	A single node in the cluster that acts as the leader. A leader manages cluster convergence and membership state transitions.

Node identifiers

- Nodes are addressed by `hostname:port:uid` tuple
- Any node doesn't necessary have to house any actors
- Joining a clustered is done by issuing a `Join` command

Rejoining a Cluster

- Once an `ActorSystem` has left a cluster it cannot rejoin without restart
- Restart provides a new UID

Lab: Setting up an Actor to use in a cluster

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.cluster` called `SimpleActor`

Step 2: Inside of `SimpleActor`, simply log the whatever information comes to it

Step 3: Go to the *application.conf* and review the configuration in the `clustering_akka` section. Notice the following:

- The `seed-nodes` established
- The deployment of a router in our cluster
- `akka.actor.provider` is set to `cluster`
- We are using remoting much in the same way we did in the remoting section
- We are also enhancing our clusters with the `MetricsExtension` and the `ClusterClientReceptionist`

Lab: Creating a `ClusterListener`

Step 1: In the project, akka-training, Create an actor inside of *src/main/scala* and in the package `com.akkatraining.scala.cluster` called `SimpleClusterListener` with the following content:

```
class SimpleClusterListener extends Actor with ActorLogging {

    val cluster = Cluster(context.system)

    // subscribe to cluster changes, re-subscribe when restart
    override def preStart(): Unit = {
        cluster.subscribe(self, initialStateMode = InitialStateAsEvents,
            classOf[MemberEvent], classOf[UnreachableMember])
    }
    override def postStop(): Unit = cluster.unsubscribe(self)

    def receive: Receive = {
        case MemberUp(member) ⇒
            log.info("Member is Up: {}", member.address)
        case UnreachableMember(member) ⇒
            log.info("Member detected as unreachable: {}", member)
        case MemberRemoved(member, previousStatus) ⇒
            log.info(
                "Member is Removed: {} after {}",
                member.address, previousStatus)
        case _: MemberEvent ⇒ // ignore
    }
}
```

Step 2: Discuss what this will do...also notice `ActorLogging` as a way to establish logging

Lab: Starting our cluster

Step 1: In the project, akka-training, Create a Runnable object inside of *src/main/scala* and in the

package `com.akkatraining.scala.cluster` called `SimpleClusterRunner` with the following content:

```
object SimpleClusterRunner extends App {

    val rootConfig = ConfigFactory.load()
    val seqPorts = Seq("2551", "2552", 0, 0)

    seqPorts foreach { p =>
        val config = ConfigFactory
            .parseString(s"akka.remote.netty.tcp.port=$p")
            .withFallback(rootConfig....)
        val actorSystem = ActorSystem("My-Cluster", config)
        val actorRef = actorSystem
            .actorOf(Props[SimpleClusterListener], name
="clusterListener")
        val router = actorSystem
            .actorOf(FromConfig.props(Props[SimpleActor]), name
="simpleRouter")
        ClusterClientReceptionist(actorSystem).registerService(router)
    }
}
```

Step 2: Discuss what we are creating with the following points:

- All `ActorSystem` must have the same name
- We are creating a distributed router with `SimpleActor`
- We also are establishing a `ClusterClientReceptionist` and registering the router
- We are actually running 4 total systems with a different port
- The first two are mandatory since they are the seeds

Lab: Creating a client to access from the outside

Step 1: In the project, akka-training, Create a Runnable object inside of *src/main/scala* and in the package `com.akkatraining.scala.cluster` called `ClusterClientRunner` with the following content:

```
object ClusterClientRunner extends App {

    val actorSystem = ActorSystem("Non-Member-Actor-System")
    val initialContacts = Set(
        ActorPath.fromString("akka.tcp://My-
Cluster@127.0.0.1:2552/system/receptionist"),
        ActorPath.fromString("akka.tcp://My-
Cluster@127.0.0.1:2551/system/receptionist"))

    val c: ActorRef = actorSystem.actorOf(
        ClusterClient.props(ClusterClientSettings(actorSystem)
            .withInitialContacts(initialContacts)))
    c ! ClusterClient.Send("/user/simpleRouter", "Hello!", false)

    println("Press ENTER to continue")
    StdIn.readLine()

    Await.ready(actorSystem.terminate(), 10 seconds)
}
```

Step 2: Discuss what we are creating with the following points:

- All `ActorSystem` has a different name
- We are sending to the `Receptionist` that will take the message and find a routee
- `c` is just an `ActorRef`, the fine grained mechanics are hidden

Akka Streams

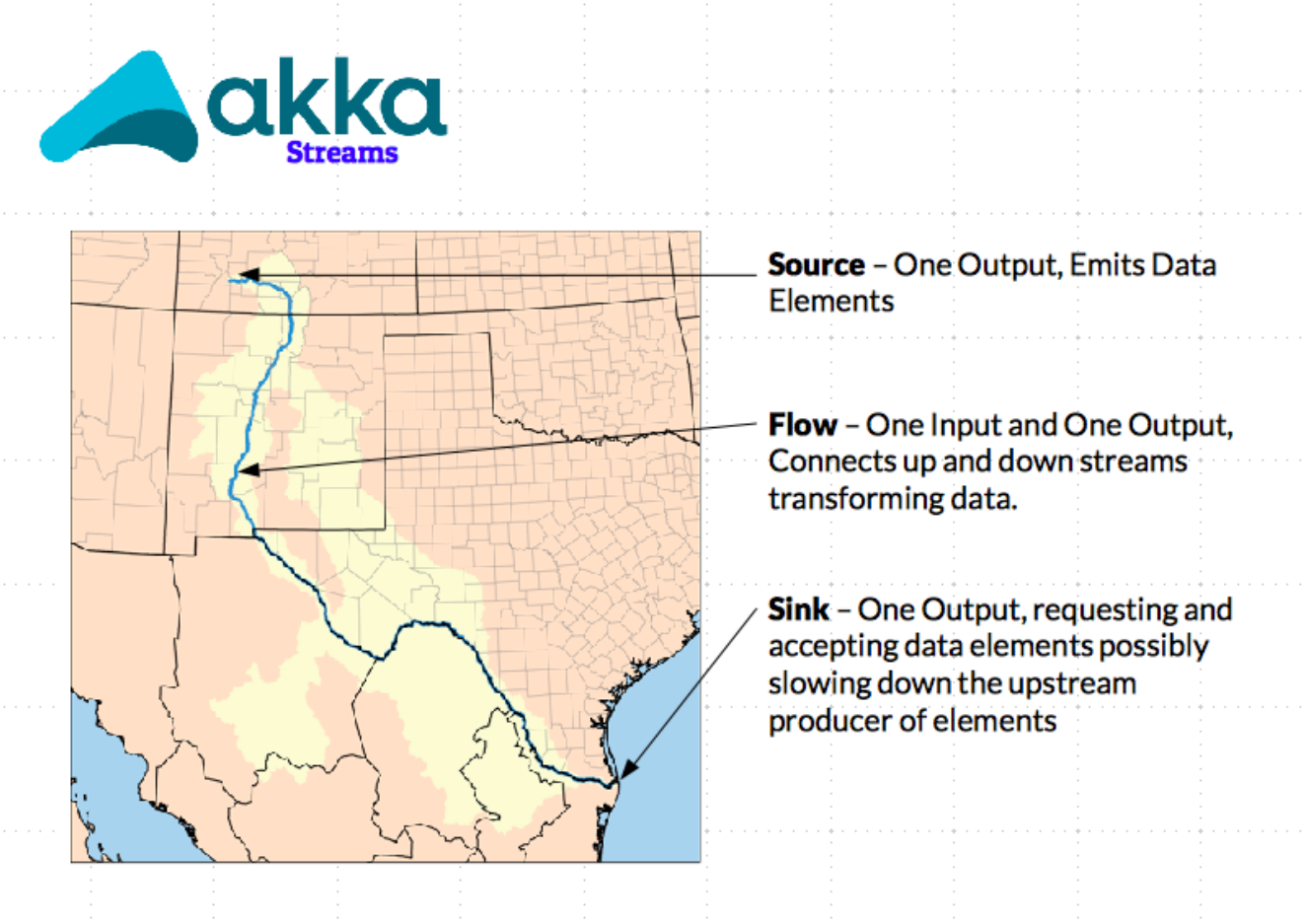
Akka Streams Overview

- 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

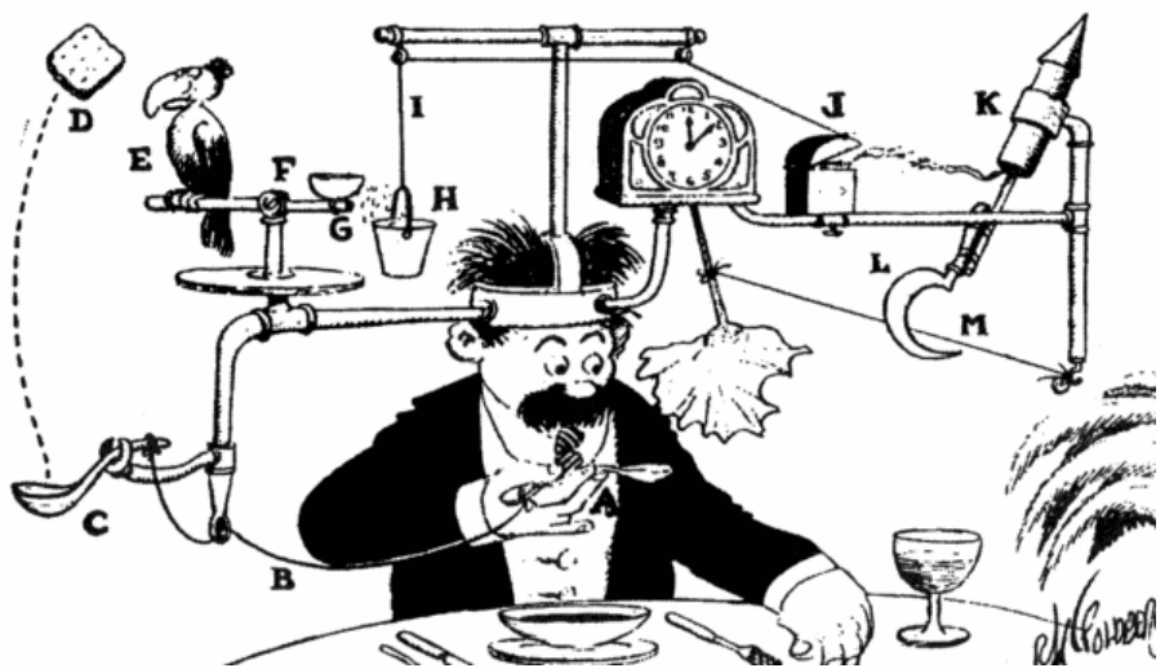
Akka Streams River Analogy



Akka Streams Pipe Analogy

Akka Streams Lightening Analogy

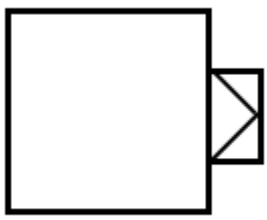
Akka Streams Rube Goldberg Analogy



Think Rube Goldberg machines!

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

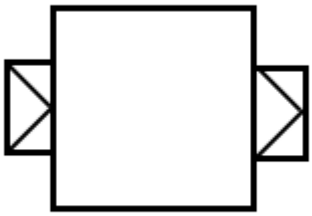
```
val single: Source[Int, NotUsed] = Source.single(3)
```

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...

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)
```

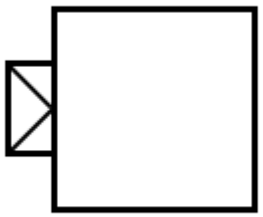
Connecting the `Flow`

- 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)
```

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)
```

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 a an engine, backed by `Actor`, that will process the stream

A `RunnableGraph` by example

```
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`

```
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()
```

Composite `Source`

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

```
val compositeSource:Source[Int, NotUsed] =
   Source(1 to 10)
     .filter(x => x % 2 == 0)
```

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

```
val compositeSource: Source[Int, NotUsed] = {
    val flow = Flow[Int].filter(x => x % 2 == 0)
    Source(1 to 10).via(flow)
}
```

Composite `Sink`

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

```
val compositeSink =
    Flow[Int]
      .map(x => x + 10)
      .toMat(Sink.fold(0)(_ + _))(Keep.right)
```

- 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))
```

Recalling the auxiliary value

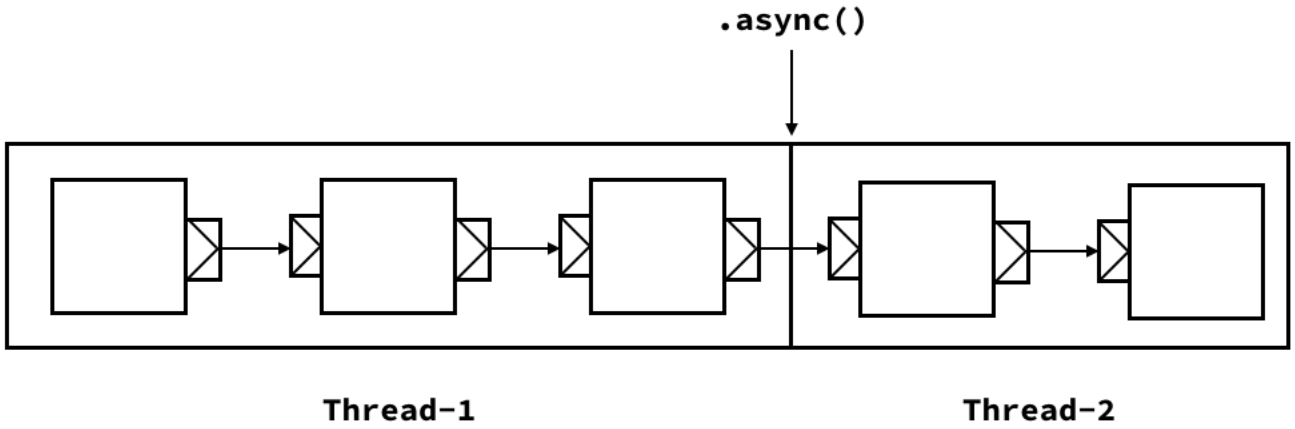
- In streams there is an auxialliary value per stream
- This can be used to:
  - to manipulate the stream
  - to provide added information
- It is your choice which you want to use by either pull:
  - The left with `Keep.left`
  - The right with `Keep.right`

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`

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)
```

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

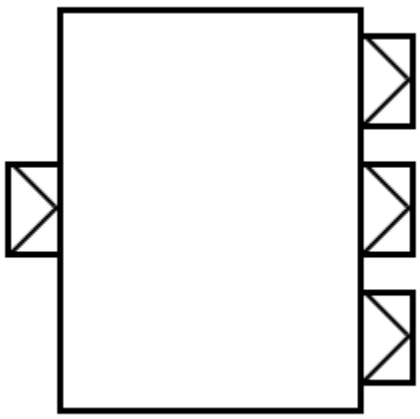
```
Source(10 to 0 by -1)
      .async
      .map(x => Some(100 / x))
      .recover { case t: Throwable => None }
      .runForeach(println)
```

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 ⇐ 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



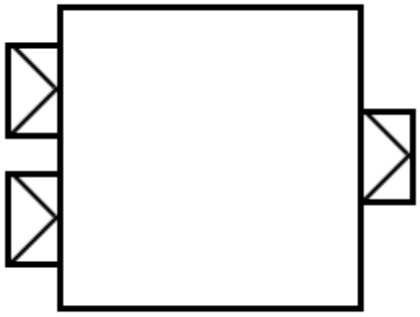
Source: https://doc.akka.io/docs/akka/2.5.4/scala/stream/stream-graphs.html

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

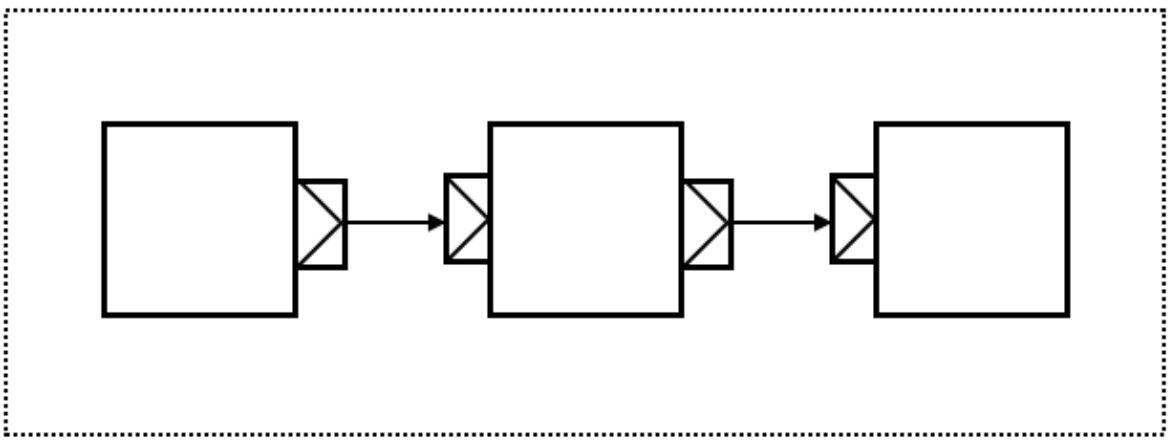
- `MergePrioritized[In]` - like Merge but if elements are available on all input ports, it picks from them randomly based on their priority
- `ZipWith[A,B,…,Out]` - (N inputs, 1 output) which takes a function of `N` inputs that given a value for each input emits 1 output element
- `Zip[A,B]` - (2 inputs, 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

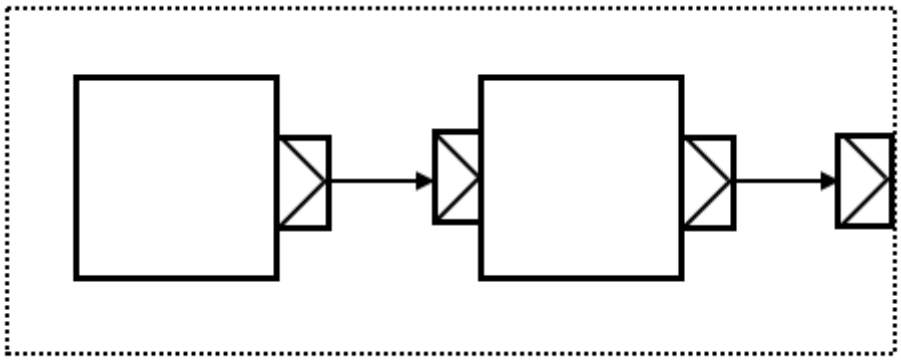
Closed Shape

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



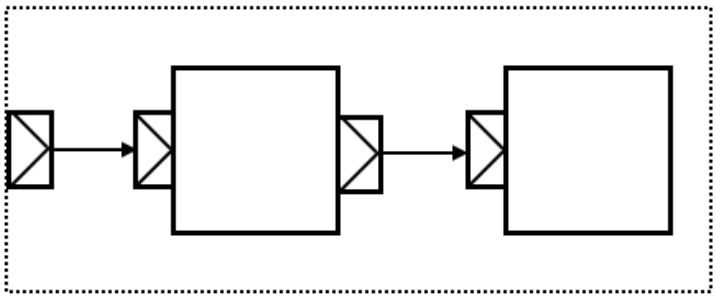
Source Shape

- Contains one open output port
- Requires other components to be closed and runnable



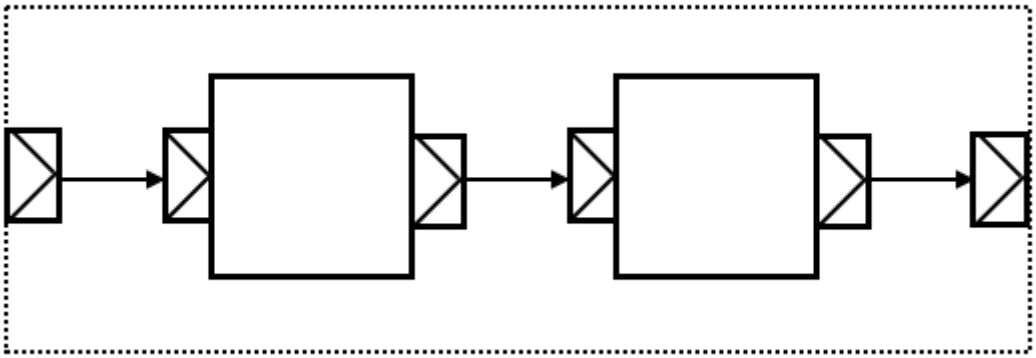
Sink Shape

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



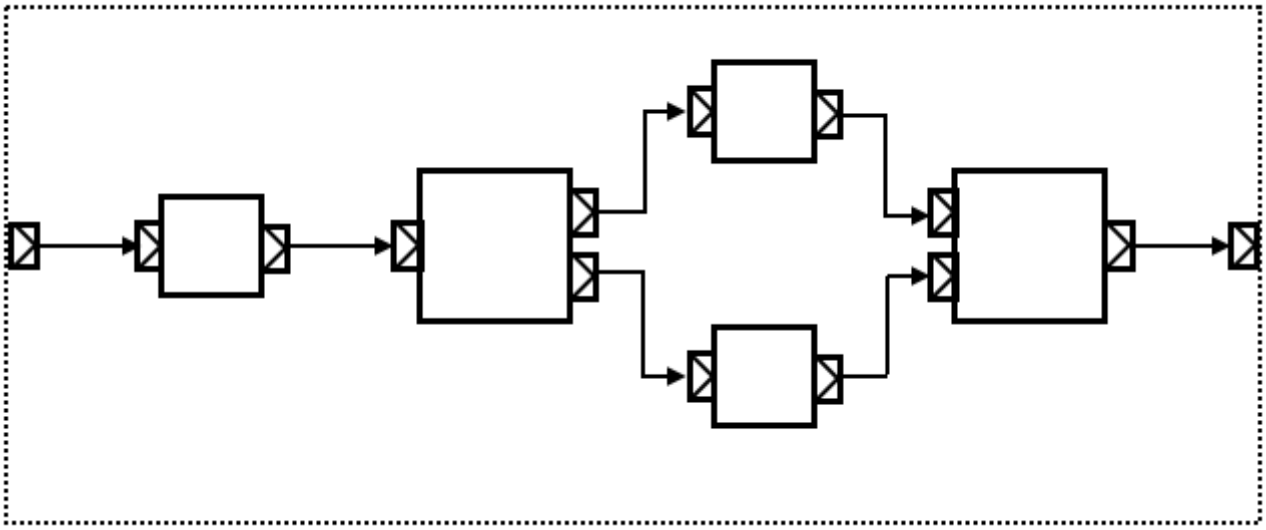
Flow Shape

- 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`
  - `Broadcat[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 userHome: String =
    System.getProperty("user.home")

val source = Source(1 to 100)

val outputToTwoFiles =
  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)
```

Akka Http

Akka Http Basics

- RESTFul content server
- Microservice face to rest of enterprise
- Processes using:
  - Akka Actors
  - Akka Streams
- Used with Play Framework

Akka Http Further Features

- High-level functionality including:
  - (De)Compression
  - (Un)Marshalling

Akka APIs

- Server APIs
- Client APIs

Akka Http Server APIs

- **Akka Http Core**
  - Mostly low-level server and client-side implementation of HTTP
- **Akka Http DSL**
  - DSL (Domain Specific Language) for defining HTTP-based APIs on the server-side

Akka Client API

- Consuming HTTP-based services offered by other endpoints
- Currently provided by the `akka-http-core` module.
- Used to interact in parallel with RestFUL endpoints

Akka Http Mashallers

- Akka-Http-Spray-XML - (De)Marshallers for XML

- Akka-Http-Spray-JSON - (De)Marshallers for JSON

Akka-Http Dependency

```
"com.typesafe.akka" %% "akka-http" % "10.1.3"
```

Required Components for a WebServer

```
object WebServer {
  def main(args: Array[String]) {
    implicit val system = ActorSystem("my-system")
    implicit val materializer = ActorMaterializer()
    implicit val executionContext = system.dispatcher
  }
}
```

- An `ActorSystem` is required since this is an Akka instance
- A `Materializer` is required as an engine to process streaming data
- An `executionContext` is required for `Future` processing and shut down

Akka-Http Routing DSL

```
object WebServer {
  def main(args: Array[String]) {
    //implicits elided

    val route =
      path("hello") {
        get {
          complete(HttpEntity(ContentTypes.<code>text/html(UTF-8)</
code>,
                   "<h1>Say hello to akka-http</h1>"))
        }
    }
  }
}
```

- `path` will map the `/hello` on the web server
- `get` is the HTTP method
- `complete` signifies a complete payload with the content-type and HTML

Start Server and provide a shut down the server

```
object WebServer {
  def main(args: Array[String]) {
    //implicits elided
    //route elided

    val bindingFuture = Http().bindAndHandle(route, "localhost", 8080)

    println(s"Server online at http://localhost:8080/\nPress RETURN to
stop...")
    StdIn.readLine()
    bindingFuture
      .flatMap(<em>.unbind())
      .onComplete(</em> => system.terminate())
  }
}
```

- `bindingFuture` has a `Future` handle on the server given a `route`
- `unbind` will unbind from the port
- When all done we `terminate` our Actor System

Mapping an actual route, a real world example

```
val route: Route =⍰path("order" / IntNumber) { id =>⍰
    get {
       complete {
          "Received GET request for order " + id
       }
    } ~⍰put {
       complete {
          "Received PUT request for order " + id
    }
  }
}
```

Mapping an actual route, a more complex example

```
val route =⊠path("order" / IntNumber) { id =>⊠
   (get | put) { ctx =>
       ctx.complete(
           s"""Received ${ctx.request.method.name}
             request for order $id""")⊠
       }⊠
   }
```

Using an `Extractor` in a DSL

```
val route =⊠path("order" / IntNumber) { id =>
⊠ (get | put) {
      extractMethod { m =>
         complete(s"""Received ${m.name}
                     request for order $id""")
         }
  }
}
```

Composability using an `Extractor` in a DSL

```
val getOrPut = get | put
val route = path("order" / IntNumber) { id =>
⊠    getOrPut {
        extractMethod { m =>⊠
            complete(s"""Received ${m.name}
                        request for order $id")⊠
        }⊠
    }
}
```

Combining it all together

```
val getOrPut = get | put
val route =
  (path("order" / IntNumber)
    & getOrPut & extractMethod) {
      (id, m) =>
         complete(s"""Received ${m.name}
                      request for order $id""")
      }
```

Various Directives/Extractors

- There are various extractors and directives
  - Security
  - Handling Requests
  - Logging
  - Cookies
  - More...

Marshallers/Unmarshallers

- Marshalling converts a custom object into raw wire data
- Unmarshalling converts raw wire data into a custom object

`Marshaller` deep dive

- A `Marshaller` is actually: `A ⇒ Future[List[Marshalling[B]]]`
  - `Future`: `Marshallers` are not required to synchronously produce a result, so instead they return a `Future`
  - `List`: `Marshallers` can offer several ones. Which one will be rendered onto the wire in the end is decided by content negotiation. For example, the Marshaller[OrderConfirmation, MessageEntity] might offer a JSON as well as an XML representation. The client can decide through the addition of an Accept request header which one is preferred. If the client doesn't express a preference the first representation is picked.
  - `Marshalling[B]`: Rather than returning an instance of `B` directly marshallers first produce a Marshalling[B]. This allows for querying the `MediaType` and potentially the `HttpCharset` that the marshaller will produce before the actual marshalling is triggered. Apart from enabling content negotiation this design allows for delaying the actual construction of the marshalling target instance to the very last moment when it is really needed.

Source: https://doc.akka.io/docs/akka-http/current/common/marshalling.html

http://doc.akka.io/docs/akka-http/current/scala/http/routing-dsl/directives/alphabetically.html

Lab: Trying out Akka-HTTP

Step 1: In *src/main/scala* and in the package `com.akkatraining.scala.http` locate the `BasicIntroServer`

Step 2: Discuss the organization of `BasicIntroServer`

Step 3: Run `BasicIntroServer` in either your IDE or in SBT using the following, and select `BasicIntroServer`

```
> run
```

Step 4: In your web browser go to, http://localhost:8080/hello

Lab: Trying out Akka-HTTP get with an `Actor`

Step 1: In *src/main/scala* and in the package `com.akkatraining.scala.http` locate the `BasicGetServer`

Step 2: Discuss the organization of `BasicGetServer` and how it uses the `EmployeeActor` and `Employee` entity

Step 3: Run `BasicGetServer` in either your IDE or in SBT using the following, and select `BasicGetServer`

```
> run
```

Step 4: In your web browser go to, http://localhost:8080/employee/1 (up to 4)

Step 5: You can also use `curl` on MacOSX and Linux

```
curl -X GET http://localhost:8080/employee/1
```

Lab: Trying out Akka-HTTP get and post with an `Actor`

Step 1: In *src/main/scala* and in the package `com.akkatraining.scala.http` locate the `BasicGetPostServer`

Step 2: Discuss the organization of `BasicGetPostServer` and how it uses the `EmployeeActor` and `Employee` entity. Also discuss this time the Marshaller and how that works

Step 3: Run `BasicGetPostServer` in either your IDE or in SBT using the following, and select `BasicGetPostServer`

```
> run
```

Step 4: Since we are now dealing with `POST`, use `curl` on MacOSX or Linux, or find an alternate.

```
curl -X POST -d "firstName=Beyonce&lastName=Knowles"
http://localhost:8080/employee
```

Step 5: In your web browser go to, http://localhost:8080/employee/5 (or whatever the latest one would be)

Lab: Trying out Akka-HTTP get, post, and put with an `Actor`

Step 1: In *src/main/scala* and in the package `com.akkatraining.scala.http` locate the `BasicGetPostPutServer`

Step 2: Discuss the organization of `BasicGetPostPutServer` particularly the DSL and how it is organized

Step 3: Run `BasicGetPostPutServer` in either your IDE or in SBT using the following, and select `BasicGetPostPutServer`

```
> run
```

Step 4: Since we are dealing with `PUT`, use `curl` on MacOSX or Linux, or find an alternate.

```
curl -X PUT -d "firstName=Nancy&lastName=Sinatra"
http://localhost:8080/employee/1
```

Step 5: In your web browser go to, http://localhost:8080/employee/1 and verify that `Frank` is now `Nancy`

Lab: Trying out Akka-HTTP get, post, and put with an `Actor` as JSON

Step 1: In *src/main/scala* and in the package `com.akkatraining.scala.http` locate the `JSONGetPostPutServer`

Step 2: Discuss the organization of `JSONGetPostPutServer` particularly the `implicit` that affects that behavior

Step 3: Run `JSONGetPostPutServer` in either your IDE or in SBT using the following, and select `JSONGetPostPutServer`

```
> run
```

Step 4: Go through all the HTTP methods, be sure to use JSON for the payload since, that's what we have setup, for example here is a post

```
curl -X POST -d '{"firstName":"Beyonce","lastName":"Knowles"}'
     -H "Content-Type: application/json"
      http://localhost:8082/employee
```

Lab: Trying out Akka-HTTP get, post, and put with an `Actor` as JSON, Securely

Step 1: In *src/main/scala* and in the package `com.akkatraining.scala.http` locate the `SecureJSONGetPostPutServer`

Step 2: Discuss the organization of `SecureJSONGetPostPutServer` particularly the security aspect of the server

Step 3: Run `SecureJSONGetPostPutServer` in either your IDE or in SBT using the following, and select `SecureJSONGetPostPutServer`

```
> run
```

Step 4: In your web browser go to, http://localhost:8080/employee/1 (up to 4), and enter whatever username but with the password: `AkkaTime`