

University of Wolverhampton

School of Mathematics and Computer Science

5CS022 Distribute and Cloud Systems Programming

Workshop 3 The Akka Framework

Overview

Akka (<https://akka.io/>) is an Actor software framework. It is compatible with both the Scala and Java programming languages.

This workshop shows you how to import an existing simple Akka program into either the Eclipse or IntelliJ IDE to get you started with a working setup.

If you are working with Eclipse, follow Part A. If you are working with IntelliJ, jump to Part B.

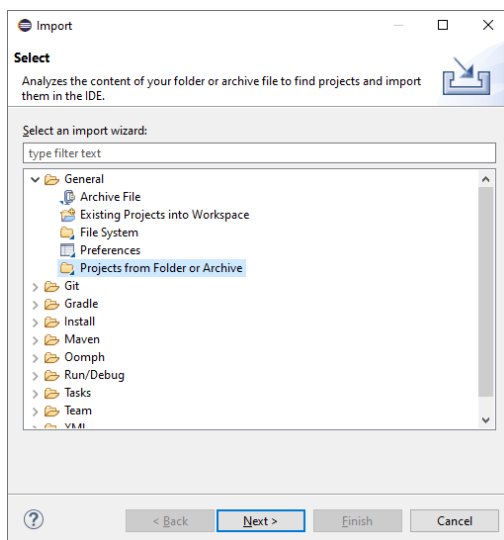
Part A. Working with Eclipse

1. Download the sample project

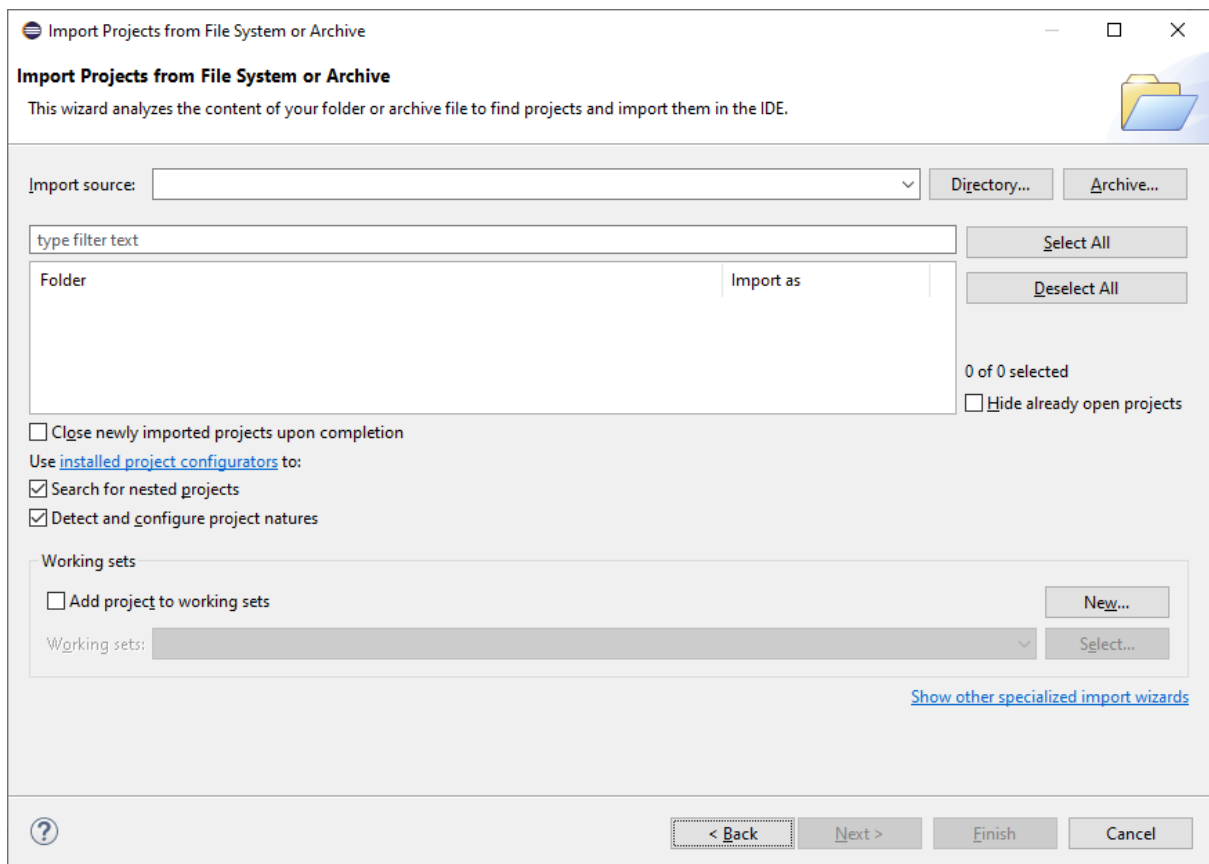
Go to the Canvas topic for 5CS022 Distributed and Cloud Systems Programming, and download the zip file “akka-example(Eclipse).zip” (https://canvas.wlv.ac.uk/courses/18227/files/2532668/download?download_frd=1) and unzip the zip file into a folder.

2. Import the project into Eclipse

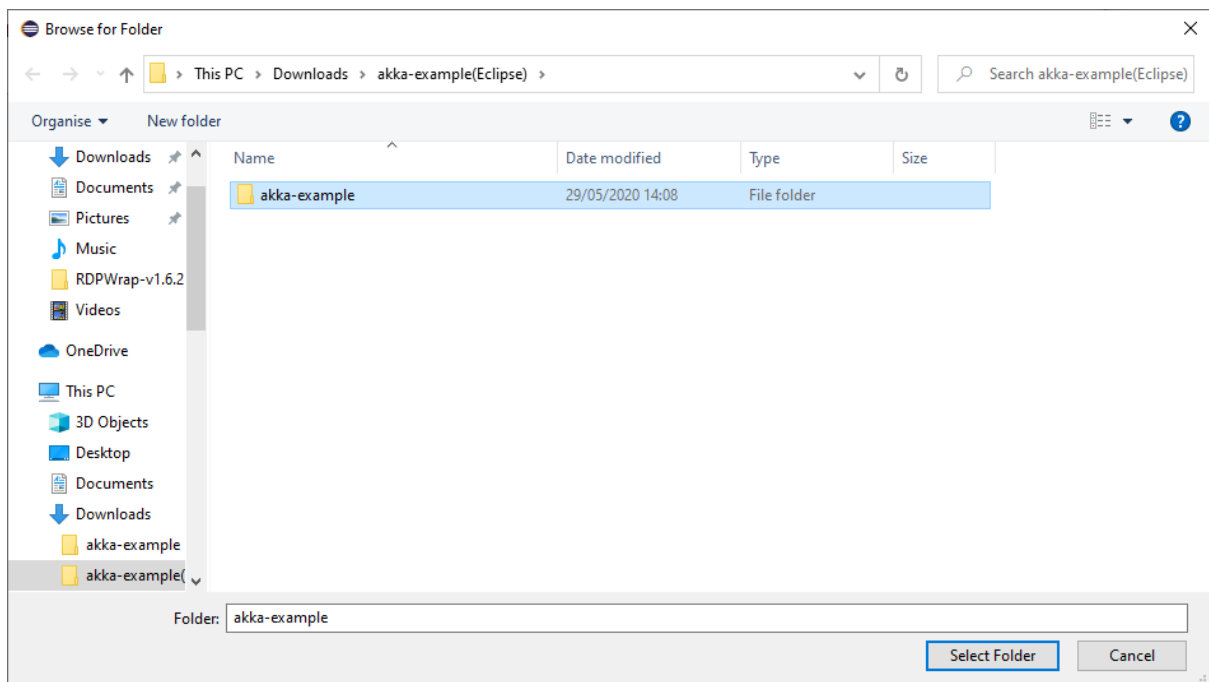
Start Eclipse. Then go to “File, “Import...” and expand “General” and then select “Projects from Folder or Archive”:



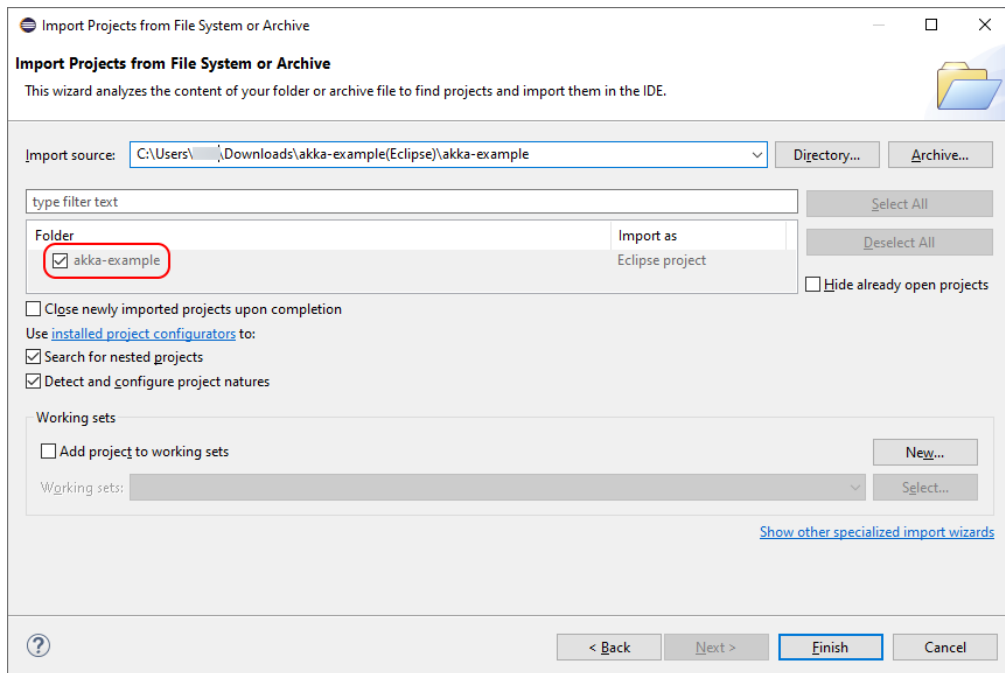
Click Next. Then on the next screen, click “Directory”



Browse to the unzipped folder and select the project folder inside:



One the next screen, make sure that the “akka-example” project has been correctly detected:

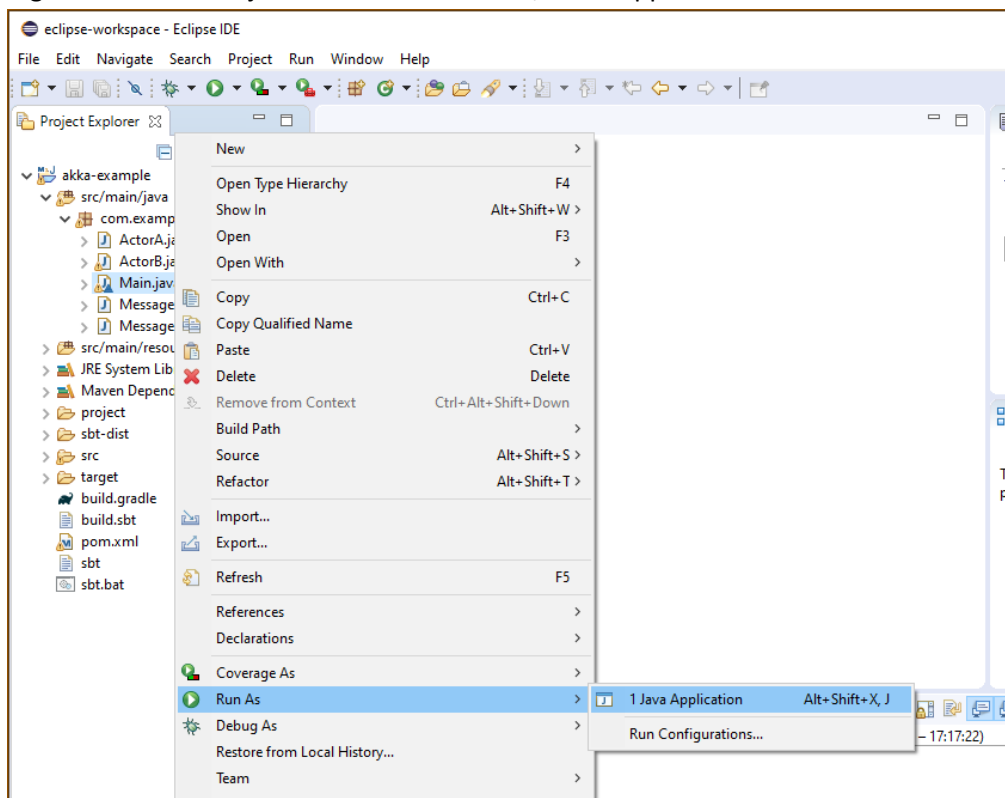


If so, click “Finish”

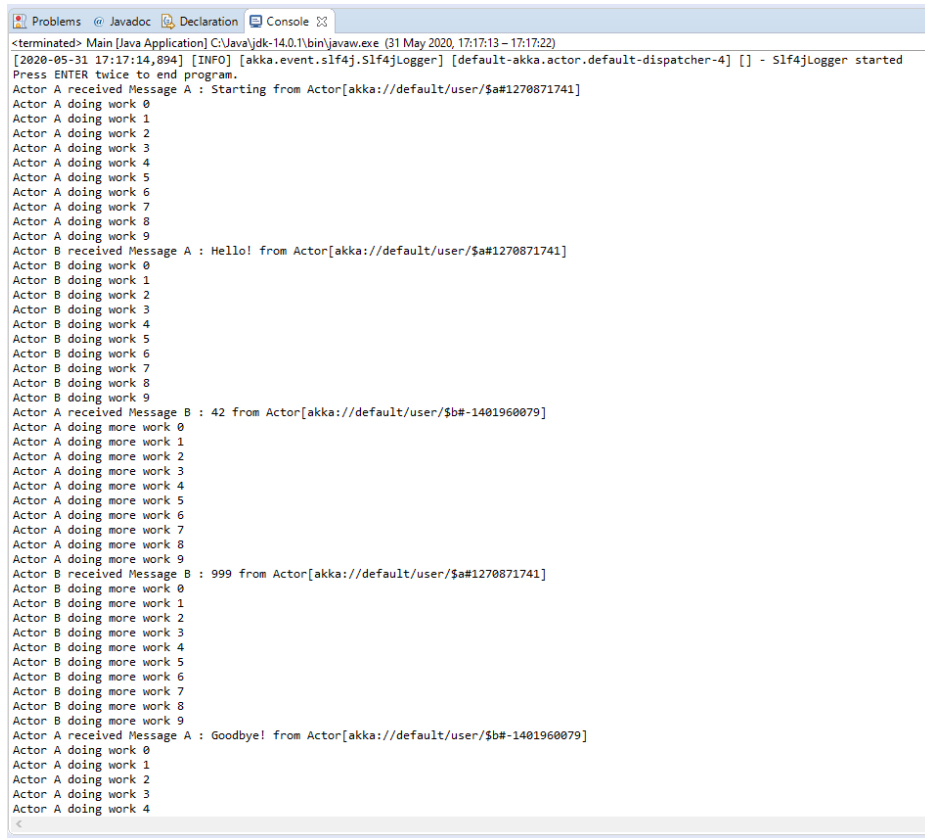
3. Building and running the example Akka program

When the project has been successfully imported, expand the “akka-example” folder inside Eclipse, until you can see “Main.java”.

Right -click on Main.java and Seelct “RunAs”, “Java Application:

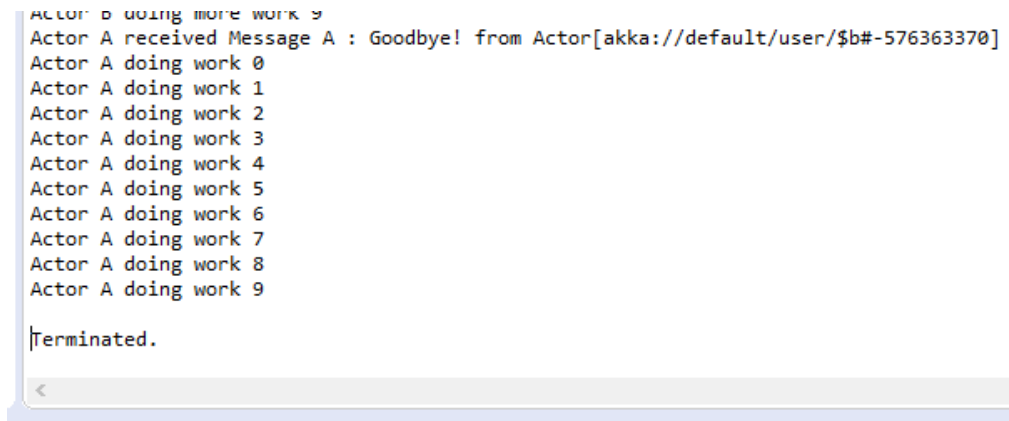


If it all went well, you should see this in the Console window:



```
<terminated> Main [Java Application] C:\Java\jdk-14.0.1\bin\javaw.exe (31 May 2020, 17:17:13 - 17:17:22)
[2020-05-31 17:17:14,894] [INFO] [akka.event.slf4j.Slf4jLogger] [default-akka.actor.default-dispatcher-4] [] - Slf4jLogger started
Press ENTER twice to end program.
Actor A received Message A : Starting from Actor[akka://default/user/$a#1270871741]
Actor A doing work 0
Actor A doing work 1
Actor A doing work 2
Actor A doing work 3
Actor A doing work 4
Actor A doing work 5
Actor A doing work 6
Actor A doing work 7
Actor A doing work 8
Actor A doing work 9
Actor B received Message A : Hello! from Actor[akka://default/user/$a#1270871741]
Actor B doing work 0
Actor B doing work 1
Actor B doing work 2
Actor B doing work 3
Actor B doing work 4
Actor B doing work 5
Actor B doing work 6
Actor B doing work 7
Actor B doing work 8
Actor B doing work 9
Actor A received Message B : 42 from Actor[akka://default/user/$b#-1401960079]
Actor A doing more work 0
Actor A doing more work 1
Actor A doing more work 2
Actor A doing more work 3
Actor A doing more work 4
Actor A doing more work 5
Actor A doing more work 6
Actor A doing more work 7
Actor A doing more work 8
Actor A doing more work 9
Actor B received Message B : 999 from Actor[akka://default/user/$a#1270871741]
Actor B doing more work 0
Actor B doing more work 1
Actor B doing more work 2
Actor B doing more work 3
Actor B doing more work 4
Actor B doing more work 5
Actor B doing more work 6
Actor B doing more work 7
Actor B doing more work 8
Actor B doing more work 9
Actor A received Message A : Goodbye! from Actor[akka://default/user/$b#-1401960079]
Actor A doing work 0
Actor A doing work 1
Actor A doing work 2
Actor A doing work 3
Actor A doing work 4
```

Click on the Console window and press the “Enter” key a couple of times and you should see the program output “Terminated.” and end:



```
Actor B doing more work 9
Actor A received Message A : Goodbye! from Actor[akka://default/user/$b#-576363370]
Actor A doing work 0
Actor A doing work 1
Actor A doing work 2
Actor A doing work 3
Actor A doing work 4
Actor A doing work 5
Actor A doing work 6
Actor A doing work 7
Actor A doing work 8
Actor A doing work 9

Terminated.
```

Now skip the following IntelliJ part, and jump to Part C.

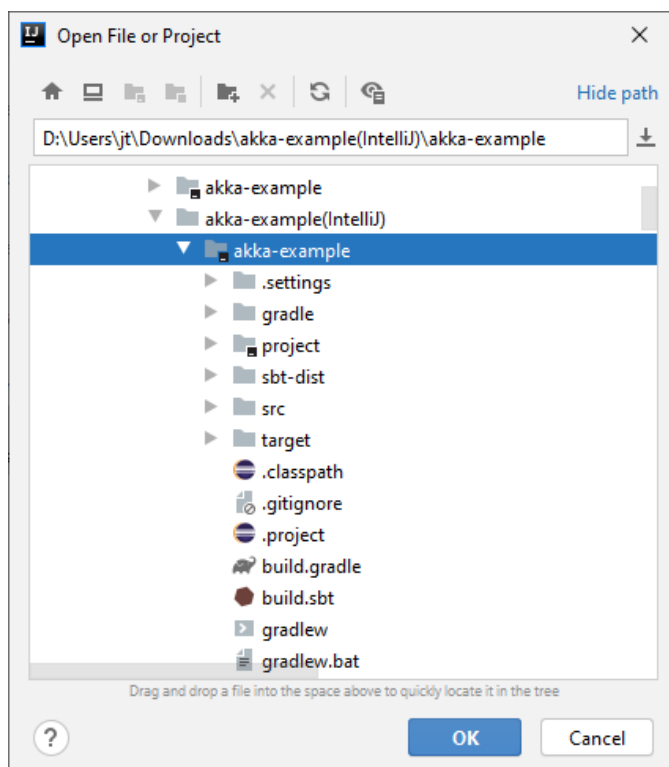
Part B. Working with IntelliJ

1. Download the sample project

Go to the Canvas topic for 5CS022 Distributed and Cloud Systems Programming, and download the zip file “akka-example(IntelliJ).zip” (https://canvas.wlv.ac.uk/courses/18227/files/2535512/download?download_frd=1) and unzip the zip file into a folder.

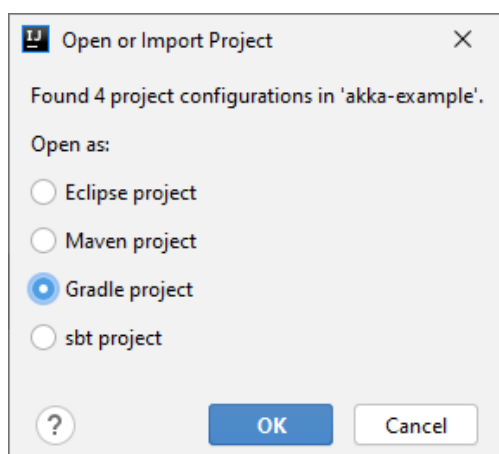
2. Import the project into IntelliJ

Start the IntelliJ IDEA. Then go to “Open or Import” and browse to your unzipped project folder:



And click OK.

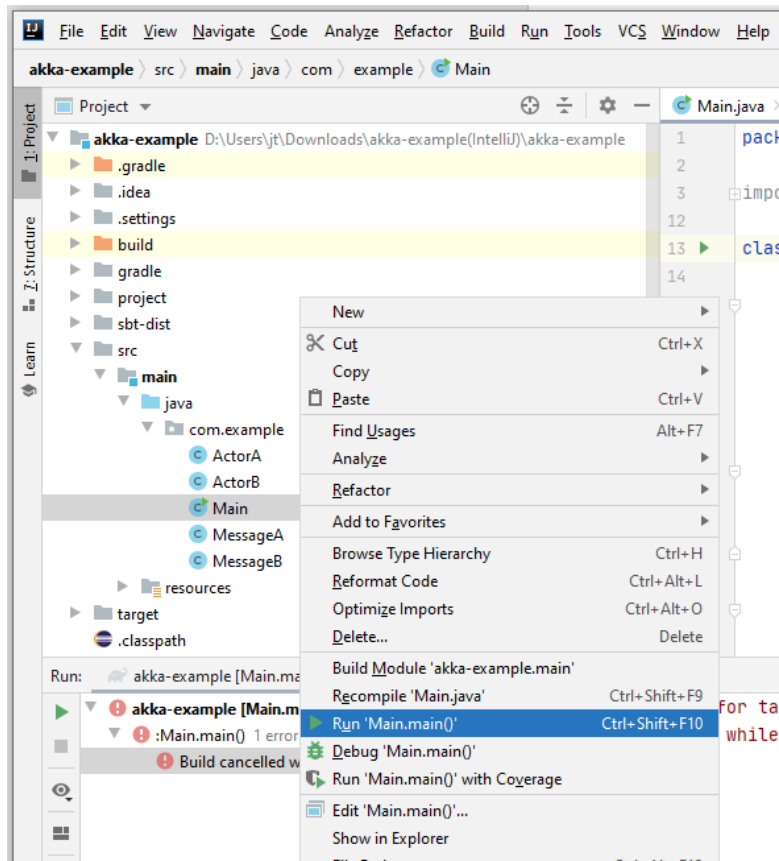
Then select Gradle project:



And click OK. The project will load and be built automatically in IntelliJ.

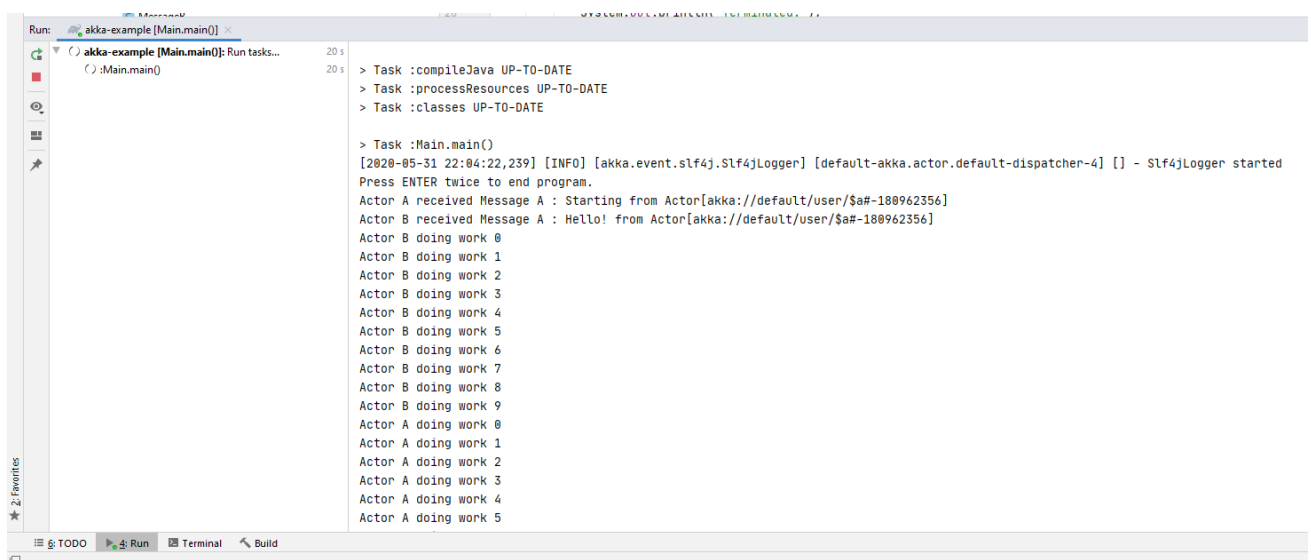
3. Building and running the example Akka program

When it is ready, in the project window, expand the “src” folder until you get to the “Main” class:

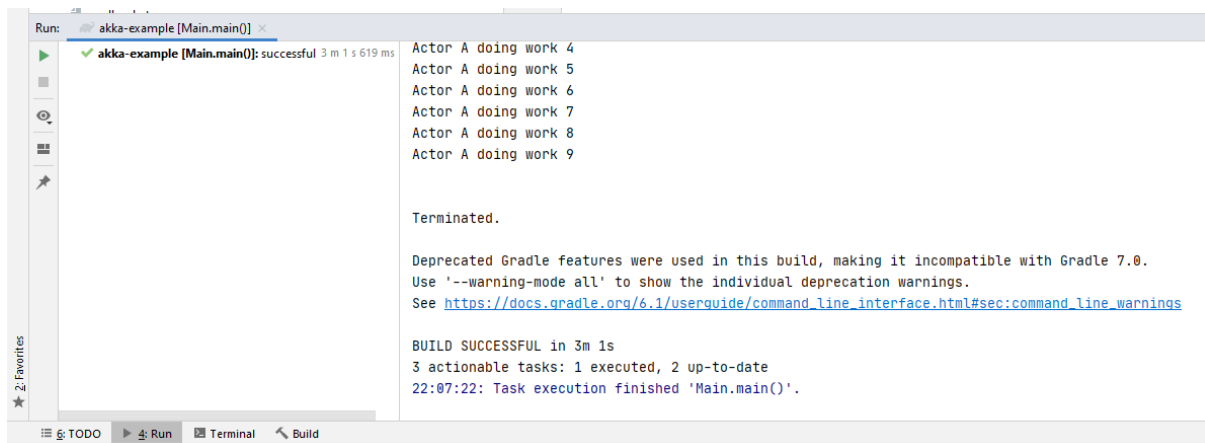


Then right-click on it and select “Run ‘Main.main()’”

The Akka-example application will run and you will the program outputting something to the following in the “Run” window:



Click in the “run” window and press the “Enter” key twice and the program will output the following and end:



```
Run: akka-example [Main.main()] x
✓ akka-example [Main.main()]: successful 3 m 1 s 619 ms

Actor A doing work 4
Actor A doing work 5
Actor A doing work 6
Actor A doing work 7
Actor A doing work 8
Actor A doing work 9

Terminated.

Deprecated Gradle features were used in this build, making it incompatible with Gradle 7.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle.org/6.1/userguide/command\_line\_interface.html#sec:command\_line\_warnings

BUILD SUCCESSFUL in 3m 1s
3 actionable tasks: 1 executed, 2 up-to-date
22:07:22: Task execution finished 'Main.main()'.

2 Favorites
★
TODO Run Terminal Build
```

Part C. The Akka Example Program

Main.java

This is the program's starting point. Its main purpose is to create the Actor system and the first actor, and then start it running:

```
package com.example;

import akka.actor.ActorRef;
import akka.actor.ActorSystem;
import akka.actor.Props;
import java.io.IOException;

class Main {

    public static void main(String[] args) {
        ActorSystem system = ActorSystem.create();
        ActorRef actorARef = system.actorOf(Props.create(ActorA.class));
        actorARef.tell(new MessageA("Starting"), actorARef);
        try {
            System.out.println("Press ENTER twice to end program.");
            System.in.read();
        }
        catch (IOException ignored) { }
        finally {
            system.terminate();
            System.out.println("Terminated.");
        }
    }
}
```

The Akka Actor system is created via:

```
ActorSystem.create();
```

This has to be done first in any Akka program.

Then the first actor object is created via :

```
actorARef = system.actorOf(Props.create(ActorA.class));
```

And it is then set off doing something by send it a message of “Starting”, using the method tell() :

```
actorARef.tell(new MessageA("Starting"),actorARef);
```

Then it just waits for the user to press the Enter key before it terminates the Akka system via :

```
system.terminate();
```

And then it ends the program.

Why does it need to wait for the user? Try commenting out the line:

```
System.in.read();
```

And then re-running it and see what happens.

Basically, the program might end before the actors managed to get anything work done, and the main program will just go right through to the end without waiting.

ActorA.java

```
package com.example;

import akka.actor.AbstractActor;
import akka.actor.ActorRef;
import akka.actor.Props;

public class ActorA extends AbstractActor {

    public static Props props() { return Props.create(ActorA.class, ActorA::new); }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
            .match(MessageA.class, this::onMessageA)
            .match(MessageB.class, this::onMessageB)
            .build();
    }

    private void onMessageA(MessageA msg) {
        System.out.println("Actor A received Message A : " + msg.text + " from " + getSender());
        if(msg.text.equalsIgnoreCase("Goodbye!")) {
            getContext().getSystem().terminate();
        }
        else {
            ActorRef actorBRef = getContext().getSystem().actorOf(Props.create(ActorB.class));
            actorBRef.tell(new MessageA("Hello!"), getSelf());
        }
        for (int i=0; i<10; i++){
            System.out.println("Actor A doing work "+i);
        }
    }

    private void onMessageB(MessageB msg) {
        System.out.println("Actor A received Message B : " + msg.number + " from " + getSender());
        getSender().tell(new MessageB(999),getSelf());
        for (int i=0; i<10; i++){
            System.out.println("Actor A doing more work "+i);
        }
    }
}
```

This is a very simple Actor that responds to two different types of messages; MessageA and MessageB.

For each different type of message that you need to send and receive, you need to create a new class, as this is how Akka differentiate between messages. In this example, we have the MessageA class which we are using to send text string messages, and MessageB which we use for numbers.

For message classes can be complex composite object containing combinations of different types of values or collections, but that is dependent on your application design.

When a message is received by an actor, the createReceive() method is called, and match() method is used to match the type of message then the message handler is invoked:

```
.match(MessageA.class, this::onMessageA)
```

In this case, for a message of the type `MessageA`, the message handler `onMessageA()` is called to process the message.

In our example, if the message contains the string “Goodbye!”, it will tell the Akka system to terminate.

Otherwise it will create another Akka actor object of the type `ActorB`, and then send it a message, and carry on to do its own work.

The method `onMessageB()` will be invoked if `ActorA` receives a message of the type `MessageB`, and in response it will just send the sender of the message another `MessageB` message with a number of 999, and then gets on with its work.

Summary

Thus, this is what the program does:

1. Creates the Akka actor system.
2. Create the actor `ActorA`.
3. Sends `ActorA` a `MessageA` of “Starting”.
4. `ActorA` receives the message and creates `ActorB`.
5. `ActorA` sends `ActorB` a `MessageA` of “Hello!”.
6. `ActorB` receives the `MessageA`.
7. `ActorB` replies with a `MessageB` of value 42.
8. `ActorA` receives the `MessageB` and replies a `MessageB` of 999.
9. `ActorB` receives the `MessageB` of 999 and replies with a `MessageA` of “Goodbye!”.
10. `ActorA` receives the `MessageA` of “Goodbye!” and tells the Akka actor system to terminate.

Tasks

1. Change the “work” done to add up numbers. Make the actors do the work before responding to their message senders. Send back the result of the work done to their senders.
2. Rewrite the “prime numbers” task from Week 1 MPI workshop to use Actors instead.