# Task1

## Alice:

Alice extends AbstractActor and overrides the createReceive method. What this method does is that if the message received is an actorRef, it first sends a new Wheat() to Charles and tell Charles that Tim wants to send it to him, then it sends another message to itself by calling “tell” method on “self()”. This time it tells itself that Tim wants wheat. In this process 2 messages are sent. One is sent to Charles and the other is sent to itself. we have to do everything through message passing because actors can only do things by processing message. Using a loop here to keep giving wheat to Charles would break the actors because we want to be able to terminate the actor when some criteria is satisfied. For example we can send “PoisonPill”, which is a predefined behavior that actually terminates the actor.

## Bob:

Bob extends AbstractActor and overrides the createReceive method. What this method does is that if the message received is an actorRef, it first sends a new Sugar() to Charles and tells Charles that Tim wants to send it to him, then it sends another message to itself by calling “tell” method on “self()”. This time it tells itself that Tim wants sugar. In this process 2 messages are sent.

## Charles:

Charles extends AbstractActor and overrides the createReceive method.

It has a list of sugar and a list of wheat.

In the createReceive method if Charles receives a wheat, it checks if the list of sugar is empty. If it is empty, add this wheat to the list of wheat and return. If it is not empty, send a cake to Tim saying that it is Charles who sent it. If Charles receives a sugar, it checks if the list of wheat is empty. If it is empty, add this sugar to the list of sugar and return. If it is not empty, send a cake to Time saying that it is Charles who sent it.

In this way no synchronization problem (race conditions) is guaranteed because every actor has single-threaded execution. In other words, every actor processes messages sequentially and only process one message at a time. For example. Alice and Bob are can do things only if Tim sends them message and Charles only need to know about Tim.

## Tim:

Tim extends AbstractActor and overrides the createReceive method.

Tim has 3 fields, an integer hunger, a Boolean running and an ActorRef that represents the original sender.

In the createReceive method, Tim checks if the message received is a GiftRequest. If the originalSender is null, set the originalSender as sender(). Then if running is true and the message is Cake.class, hunger is decremented by 1 and a message is printed out saying Yummy but I am still hungry. Then, if hunger is still > 0, return. If hunger <= 0, running is set to false and a gift is sent to the originalSender, which would be the temporary actor created by “Patterns.ask” in the computeGift method.

If running is set to false, Tim stops receiving cakes and since the gift has been sent, the gift.join unblocks and all actors are killed before system is terminated.

## OpenAkka:

OpenAkka is just a class containing a main method which throws InterruptedException. In the main method an ActorSystem is created using AkkaConfig.newSystem. the input values “openAkka” specifies the name, 2500 specifies the port number and Collections.emptyMap() is an empty map indicating that we are running locally.

Then AkkaConfig.keybordClose throws InterruptedException. It waits for user input to throw InterruptedException and close the system.

## AkkaConfig:

Two classes in AkkaConfig: TerminatorSupervisor and AkkaConfig.

### In AkkaConfig:

makeMap method makes a map that maps a string to an ip address.

guessMyIp() loops through all networkInterfaces and for each network interface it loops through all InetAddresses and for each InetAddress, get the host address of it. If the host address is valid, it is added into a list. After some filtering a list containing some host addresses is returned.

NewSystem method first choses the first ip address got by the guessMyIp() method. Then there is a set of configurations.

akka.actor.guardian-supervisor-strategy = TerminatorSupervisor

akka.actor.provider = remote

akka.remote.enabled-transports = ["akka.remote.netty.tcp"]

akka.remote.netty.tcp.hostname = MyIp

akka.remote.netty.tcp.port = port

akka.actor.deployment."/Actor1.Name".remote = "akka.tcp://OpenAkka@"Actor1.Ip:2500"

...

akka.actor.deployment."/Actorn.Name".remote = "akka.tcp://OpenAkka@"Actorn.Ip:2500"

Basically the first 5 lines say that guardian-supervisor-strategy is the TerminatorSupervisor, actor provider can be remote, enabled-transport is netty.tcp, the hostname used is Myip and the port is the input port.

The for each loop loops through the input map and attaches the name of the actor and its ip address to the config.

The akka.actor.deplyments say that for example. the actor called: “Actor1.Name” will be run on another machine with Ip address and port: Actorn.Ip:2500. These enable that actors can be run on multiple machines.

Finally the config

### TerminatorSupervisor

TerminatorSupervisor implements akka.actor.SupervisorStrategyConfigurator with all behaviors delegated to Supervisor.stoppingStrategy but additionally in the processFailure method, it terminates the system. So if any exception is thrown in the process of the actorSytem, the system is terminated.

## Cakes

In the Cakes class there is a computeGift method and a main method.

The compute gift method creates an ActorSystem using AkkaConfig.newSystem which has name “cakes”, port num 2501 and an empty map of name to ip address.

Then four ActorRef object are created, namely alice, bob, Charles and Tim.

After the actorRefs are created, some initialization messages are sent: alice is like an agent that sets up a message-passing connection between Tim and Charles. Bob also tells Charles that Tim wants something from him.

The CompletableFuture Patterns.ask creates a temporary actor whose only goal is to receive the message and put it into the completable future. In this case the temporary actor is used to send message to Tim and let him know that it requests a “gift” from Tim so that the gift.join() is unlocked only if Tim sends back the gift. Duration.ofMillis(10000) means that it waits 10000 ms for Tim to send back the gift. If the gift is not received, an exception is thrown.

Finally after the gift.join() is unlocked, all actorRefs have to “die” before the system s is terminated. In the finally block this is done by using “PoisonPill.getInstance()”. What it does is that instead of calling a method that kills the actors, we send a message “PoisonPill” through actorRef to instruct the actors to die. The s.terminate() method waits until all actors are dead before it shuts down the system.

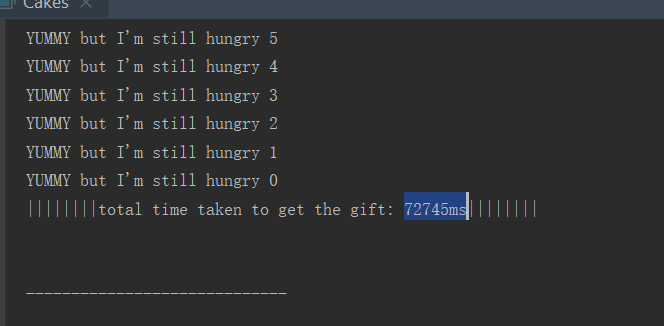
# Task2

Submitted as task2.jar

# Task3

Submitted as task3.jar

Performance:

Running with 1 bob takes more than 200,000ms while running with 4 bobs takes:

4 bobs takes much less time because my CompletableFuture implementation is non-blocking. The bobs can make sugar separately without blocking.

I was expecting that running 4 bobs would take more than 200,000/4 = 50,000 milliseconds because four bobs split up the workload. The extra time taken may be due to the inaccuracy of Thread.sleep method and the System.out.println method and object creations, etc.