

# 1 - Introducción a FIPA

*FIPA = Foundation for Intelligent Physical Agents*

## 1.1 - Estructura mensaje FIPA

- **Performative** → Tipo de acto comunicativo del mensaje
- **Sender** → Identidad del emisor del mensaje
- **Receiver** → Identidad de los destinatarios del mensaje
- **Reply-to** → A qué agente dirigir los mensajes posteriores dentro de un hilo de conversación
- **Content** → Contenido del mensaje
- **Language** → Idioma en el que se expresa el parámetro de contenido
- **Encoding** → Codificación específica del contenido del mensaje
- **Ontology** → Referencia a una ontología para dar significado a los símbolos del contenido del mensaje
- **Protocol** → Protocolo de interacción utilizado para estructurar una conversación
- **Conversation-id** → Identidad única de un hilo de conversación
- **Reply-with** → Expresión que utilizará el agente que responde para identificar el mensaje
- **In-reply-to** → Referencia a una acción anterior de la que el mensaje es una respuesta.
- **Reply-by** → Hora/Fecha en la que debe recibirse una respuesta

## 1.2 - Acto Comunicativo

La FIPA define la comunicación en términos de una función o acción denominada Acto Comunicativo (acto de comunicar). Esta norma ofrece una biblioteca de todos los actos comunicativos especificados por la FIPA y que describiremos a continuación:

SENTENCIA	DEFINICIÓN
Accept proposal	La acción de aceptar una propuesta presentada previamente para realizar una acción
Agree	Acción de comprometerse a realizar una acción, posiblemente en el futuro
Cancel	La acción de un agente de informar a otro agente de que el primer agente ya no tiene la intención de que el segundo agente realice alguna acción
Call for Proposal (CFP)	La acción de solicitar propuestas para realizar una acción determinada
Confirm	El emisor informa al receptor de que una proposición dada es verdadera, cuando se sabe que el receptor no está seguro de la proposición.
Disconfirm	El emisor informa al receptor de que una proposición dada es falsa, cuando se sabe que el receptor no está seguro de la proposición
Failure	Acción de comunicar a otro agente que se ha intentado realizar una acción pero que el intento ha fracasado.
Inform	El emisor informa al receptor de que una proposición dada es verdadera.
Inform if	Una macroacción para que el agente de la acción informe al destinatario de si una proposición es cierta o no.
Inform Ref	Una macroacción que permite al emisor informar al receptor de un objeto que el emisor cree que corresponde a un descriptor específico, por ejemplo, un nombre.
No Understood	El emisor del acto (por ejemplo, i) informa al receptor (por ejemplo, j) de que ha percibido que j ha realizado alguna acción, pero que i no ha intentado lo que j acaba de hacer. Un caso muy habitual es que i comunique a j que no ha entendido el mensaje que j acaba de enviarle.
Propagate	El emisor pretende que el receptor trate el mensaje incrustado como enviado directamente al receptor, y quiere que el receptor identifique a los agentes indicados por el descriptor dado y les envíe el mensaje de propagación recibido.
Propose	Acción de presentar una propuesta para realizar una determinada acción, dadas ciertas condiciones previas
Proxy	El emisor desea que el receptor seleccione agentes objetivo indicados por una descripción dada y les envíe un mensaje incrustado.
Query If	Acción de preguntar a otro agente si una proposición dada es verdadera o no
Query Ref	Acción de pedir a otro agente el objeto al que se refiere una expresión referencial

Refuse	La acción de negarse a realizar una acción determinada y explicar el motivo de la negativa
Reject Proposal	Acción de rechazar una propuesta para realizar alguna acción durante una negociación
Request	El emisor solicita al receptor que realice alguna acción. Una clase importante de usos del acto de petición (Request) es solicitar al receptor que realice otro acto comunicativo.
Request When	El emisor desea que el receptor realice alguna acción cuando una proposición determinada se convierte en verdadera
Request Whenever	El emisor quiere que el receptor realice alguna acción tan pronto como alguna proposición se convierta en verdadera y, a partir de entonces, cada vez que la proposición vuelva a ser verdadera.
Subscribe	El acto de solicitar a una intención persistente que notifique al remitente el valor de una referencia y que vuelva a notificar cada vez que cambie el objeto identificado por la referencia

## 2 - Comunicación básica entre agentes

En una comunicación intervienen 2 o más agentes:

- El agente que envía el mensaje (emisor)
- El agente que recibe el mensaje (receptor)

De este modo, un agente puede enviar un mensaje haciendo uso del método **send** que se hereda al extender la clase **Agent** de **JADE**. Para enviar un mensaje, podemos usar el siguiente código:

- El AID es el identificador del receptor y viene dado en los comandos de inicio:

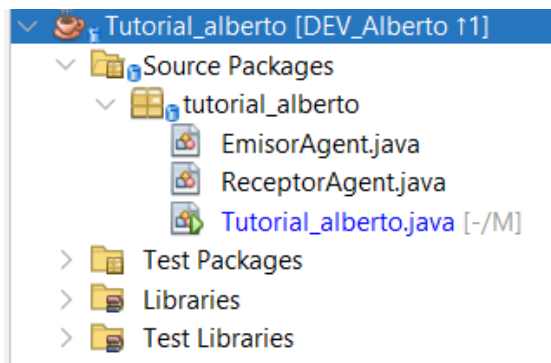
```
ACLMessage msg = new ACLMessage();
msg.addReceiver(new AID("mjco-bo-receiver", AID.ISLOCALNAME));
msg.setContent("Hola agente");
send(msg);
```

Para recibir un mensaje, podemos usar el siguiente código:

```
ACLMessage msg = blockingReceive();
```

Realmente, para recibir mensajes existe el método no bloqueante **receive()**, pero para estas prácticas usaremos el método bloqueante **blockingReceive()**, ya que no podemos avanzar hasta que recibamos un mensaje.

## 2.1 - Tutorial\_alberto



Esta es la estructura de clases que he seguido.

También he tenido que meterme en las propiedades del proyecto y tocar algo de la librería jade pero bueno se hace rápido si vas comparando con el proyecto que tenemos de la p2.

He creado esta clase main (hay que importar 150 paquetes) copiando y pegando prácticamente de la práctica individual.

El nombre del emisor es alberto-emisor y el del receptor, alberto-receptor.

Obviamente el className de cada agente es el de su clase correspondiente (EmisorAgent o ReceptorAgent)

```
public class Tutorial_alberto {  
  
    public static void main(String[] args) throws StaleProxyException {  
        System.out.println(x: "Hola mundo");  
  
        // Obtener la instancia del entorno de ejecución:  
        Runtime rt = Runtime.instance();  
  
        // Configurar el perfil para el contenedor principal:  
        Profile p = new ProfileImpl();  
        p.setParameter(string: Profile.MAIN_HOST, string1: "localhost");  
        p.setParameter(string: Profile.CONTAINER_NAME, string1: "1099");  
  
        // Crear el contenedor de agentes:  
        ContainerController cc = rt.createMainContainer(p);  
  
        AgentController emisor = cc.createNewAgent(nickname: "alberto-emisor", className: EmisorAgent.class.getCanonicalName(), args: null);  
        emisor.start();  
  
        AgentController receptor = cc.createNewAgent(nickname: "alberto-receptor", className: ReceptorAgent.class.getCanonicalName(), args: null);  
        receptor.start();  
    }  
}
```

### EMISOR

```
public class EmisorAgent extends Agent {  
    @Override  
    protected void setup() {  
        ACLMessage msg = new ACLMessage();  
  
        msg.addReceiver(new AID(name: "alberto-receptor", isGUID: AID.ISLOCALNAME));  
        msg.setContent(content: "Hola agente");  
  
        send(msg);  
  
        doDelete();  
    }  
  
    @Override  
    public void takeDown() {  
        System.out.println("\n-Emisor: El agente " + getLocalName() + " ha finalizado. \n");  
    }  
}
```

## RECEPTOR

Lo de que el mensaje se muestre con “*System.out.println(msg)*” es un triple que me he pegado pero funciona.

```
public class ReceptorAgent extends Agent{
    @Override
    protected void setup(){
        ACLMessage msg = blockingReceive();
        System.out.println(x: msg);
        doDelete();
    }

    @Override
    public void takeDown(){
        System.out.println("\n-Receptor: El agente " + getLocalName() + " ha finalizado. \n");
    }
}
```

## SALIDA

```
run:
Hola mundo
dic 06, 2024 12:43:46 P.M. jade.core.Runtime beginContainer
INFO: -----
This is JADE 4.6.0 - revision 6869 of 30-11-2022 14:47:03
downloaded in Open Source, under LGPL restrictions,
at http://jade.tilab.com/
-----
dic 06, 2024 12:43:47 P.M. jade.imtp.leap.LEAPIMTPManager initialize
INFO: Listening for intra-platform commands on address:
- jicp://192.168.56.1:1099

dic 06, 2024 12:43:47 P.M. jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
dic 06, 2024 12:43:47 P.M. jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
dic 06, 2024 12:43:47 P.M. jade.core.BaseService init
INFO: Service jade.core.resource.ResourceManagement initialized
dic 06, 2024 12:43:47 P.M. jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
dic 06, 2024 12:43:47 P.M. jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
dic 06, 2024 12:43:51 P.M. jade.mtp.http.HTTPServer <init>
INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser
dic 06, 2024 12:43:51 P.M. jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://LAPTOP-6BNFKKI8:7778/acc
dic 06, 2024 12:43:51 P.M. jade.core.AgentContainerImpl joinPlatform
INFO: -----
Agent container 1099@192.168.56.1 is ready.
-----
(NOT-UNDERSTOOD
:sender ( agent-identifier :name alberto-emisor@192.168.56.1:1099/JADE :addresses (sequence http://LAPTOP-6BNFKKI8:7778/acc ))
:receiver (set ( agent-identifier :name alberto-receptor@192.168.56.1:1099/JADE ) )
:content "Hola agente"
)

-Receptor: El agente alberto-receptor ha finalizado.

-Emisor: El agente alberto-emisor ha finalizado.
```

La salida parece ser la que se espera, lo que no entiendo muy bien es el orden de finalizar de los agentes y el por qué hay que terminar la ejecución manualmente.

```
-----  
(NOT-UNDERSTOOD  
:sender ( agent-identifier :name alberto-  
:receiver (set ( agent-identifier :name a.  
:content "Hola agente"  
)
```

Esto de **NOT-UNDERSTOOD** es la performativa del mensaje recibido por el agente, lo que significa que el receptor no entendió el mensaje enviado por el emisor.

Esto se debe a que el mensaje enviado no especifica explícitamente la performativa del acto comunicativo, ya que el mensaje se crea con **new ACLMessage()** sin especificar nada, por lo que utiliza el constructor por defecto, que crea el mensaje sin especificar.

No hay protocolo de comunicación definido explícitamente para interpretar el mensaje. En algunos casos los agentes pueden requerir información adicional, como un **conversation-id** o una performativa clara para interpretar el mensaje.

### VARIANTE

Si en vez de seguir literalmente el tutorial y crear el mensaje a partir de la performativa por defecto lo creamos con una performativa como **INFORM**:

```
public class EmisorAgent extends Agent{  
    @Override  
    protected void setup() {  
        ACLMessage msg = new ACLMessage(perf: ACLMessage.INFORM);  
    }  
}
```

La salida es la misma, pero al menos ya aparece esto →

```
(INFORM  
:sender ( agent-identifier :n  
:receiver (set ( agent-identi  
:content "Hola agente"  
)
```

También podemos hacer algún tipo de filtrado para asegurarnos de que el mensaje que recibimos es de la performativa que esperamos, e incluso no mostrar toda esa información (:sender, :receiver...) y limitarnos a mostrar el contenido. **Cambiamos el receptor:**

```
public class ReceptorAgent extends Agent{  
    @Override  
    protected void setup() {  
        ACLMessage msg = blockingReceive();  
        //System.out.println(msg);  
  
        if(msg != null && msg.getPerformative() == ACLMessage.INFORM){  
            System.out.println("-Receptor: Mensaje recibido: '" + msg.getContent() + "'");  
        }else{  
            System.out.println(x: "Mensaje no entendido o no esperado");  
        }  
  
        doDelete();  
    }  
}
```

```
Agent container 1099@192.168.56.1 is ready.  
-----  
-Receptor: Mensaje recibido: 'Hola agente'  
  
-Receptor: El agente alberto-receptor ha finalizado.  
  
-Emisor: El agente alberto-emisor ha finalizado.
```

### 3 - Comunicación entre agentes con Performativas

En el ejemplo hemos realizado una conversación muy básica. Como ya he explicado en la variante, el constructor por defecto del `ACLMessage` está obsoleto.

Un mensaje se tiene que crear o bien en réplica a otro mensaje, o bien indicando la performativa en un mensaje nuevo.

Es posible que tras recibir un mensaje el agente receptor quiera enviar un mensaje de vuelta al agente emisor. El mensaje deberá ir en la misma secuencia de mensajes y no como un mensaje nuevo. Esto hace que se pueda seguir la traza de toda la conversación.

Para ello usaremos el método **`createReply()`** del mensaje.

Cuando establecemos una comunicación entre unos agentes, los agentes pueden estar en estados diferentes, y por lo tanto, pueden estar a la espera de recibir un tipo de mensaje en particular. Esto podemos hacerlo comprobando los diferentes campos del mensaje recibido. De modo que, si se recibe un mensaje incorrecto o no esperado, se puede terminar la ejecución del agente, o informar del error.

Para modelar los estados del agente, se puede emplear una estructura del tipo **`switch-case`**.

Imaginaremos el siguiente comportamiento:

*“El agente emisor envía un saludo al agente receptor, y este segundo le contesta agradeciendo dicho saludo.”*

Por un lado, tendremos al agente emisor. Este tendrá varios estados internos:

- 1) Emitir el mensaje
- 2) Recibir el mensaje de respuesta → Nótese que en este estado se comprueba que el mensaje sea continuación del mensaje iniciado proveniente mediante el uso del **`conversation-id`**.

## 3.1 - Tutorial\_alberto

Según el guión del tutorial, las clases quedarían así:

### EMISOR

```
public class EmisorAgent extends Agent{
    private int step=0;
    private boolean finishConversation=false;

    @Override
    protected void setup(){
        switch (this.step){
            case 0 -> {
                ACLMessage msg = new ACLMessage();
                msg.addReceiver(new AID(name: "alberto-receptor",isGUID:AID.ISLOCALNAME));
                msg.setContent(content: "E - Hola agente. Como esta el tio? :P");
                msg.setConversationId(str: "conversacion-agradecimiento");
                send(msg);
                this.step=1;
            }

            case 1 -> {
                ACLMessage msg = blockingReceive();

                if(msg.getConversationId().equals(anObject: "conversacion-agradecimiento")){
                    System.out.println("El emisor recibe este mensaje como respuesta: '" + msg.getContent() + "'");
                    this.finishConversation = true;
                    System.out.println(x: "\nFin de la conversacion");
                }else{
                    System.out.println(x: "\nError en el protocolo de conversacion");
                    doDelete();
                }
            }

            default -> {
                System.out.println(x: "\nError en el protocolo de conversacion");
                doDelete();
            }
        }
    }

    @Override
    public void takeDown(){
        System.out.println("\n-Emisor: El agente " + getLocalName() + " ha finalizado. \n");
    }
}
```

### RECEPTOR

```
public class ReceptorAgent extends Agent{
    @Override
    protected void setup(){
        ACLMessage msg = blockingReceive();

        System.out.println("\nEl receptor recibe este mensaje: '" + msg.getContent() + "'");

        ACLMessage respuesta = msg.createReply();

        respuesta.setContent(content: "\nBien!!, que alegria verte :D");
        send(msg: respuesta);

        doDelete();
    }

    @Override
    public void takeDown(){
        System.out.println("\n-Receptor: El agente " + getLocalName() + " ha finalizado. \n");
    }
}
```



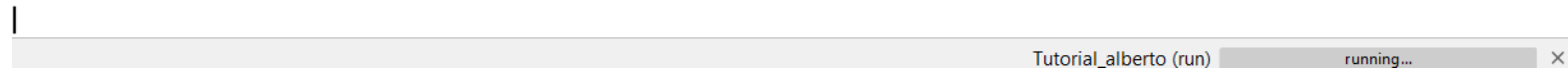
## SALIDA

No obstante la salida que obtengo es la siguiente:

```
Agent container 1099@192.168.56.1 is ready.
-----

El receptor recibe este mensaje: 'E - Hola agente. Como esta el tio? :P'

-Receptor: El agente alberto-receptor ha finalizado.
```



(de ahí no pasa).

Según ChatGPT hay que hacer cosas con el tema de los Behaviours aunque parece un cigarro lo que me ha puesto que haga pero bueno.

## 4 - Comunicación mediante un protocolo previamente establecido

La conversación entre los agentes tiene que estar perfectamente protocolizada, para que así, se puedan entender entre ellos de forma autónoma.

Imaginemos el siguiente ejemplo:

El agente (emisor) solicita mediante **REQUEST** que otro agente le salude.

El segundo agente (receptor) le responde con un **AGREE**, indicando que está dispuesto a saludarle.

El primer agente (emisor) entonces envía un saludo mediante un **INFORM** y el segundo agente (receptor) le responde con otro **INFORM**.

Una vez llegados a este punto, la conversación termina.