BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN

PROGRAMACIÓN CONCURRENTE Y PARALELA

DR. MARIO ROSSAINZ LÓPEZ

Guía de programación de aplicaciones paralelas con paso de mensajes en Java a través de la librería Jpmi

El problema del Emisor-Receptor

OTOÑO 2022 – NRC: 60898

PROCEDIMIENTO DE USO DE LA LIBRERÍA Jpmi de JAVA

1. Leer los apuntes de la materia cuyo título es: *"Mecanismos de Comunicación usando Paso de Mensajes"*, los cuales se encuentran en la siguiente liga: https://www.cs.buap.mx/~rossainz/PCyP_Maestria/1_Apuntes/9_SistemasPasoMensajes.pdf

2. Complementar la lectura con las diapositivas que llevan por título: *"Programación Paralela Usando Paso de Mensajes"* y que se encuentran en el siguiente URL: https://www.cs.buap.mx/~rossainz/PCyP_Maestria/2_Diapositivas/4_PasoDeMensajes.pdf

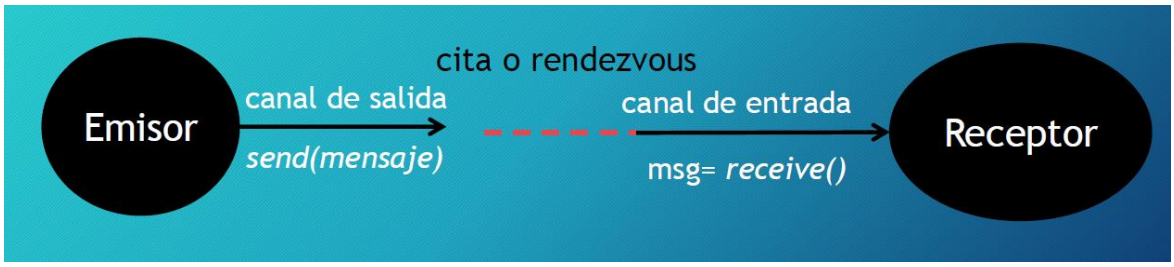
3. Bajar la librería Jpmi de la siguiente liga: https://www.cs.buap.mx/~rossainz/PCyP_Maestria/3_PMI_Java/JpmiRossainz.jar y guardarla en cualquier parte del disco duro de su computadora

4. Con ayuda del tutorial de instalación que pueden bajar de la siguiente dirección: https://www.cs.buap.mx/~rossainz/PCyP_Maestria/3_PMI_Java/1_Instalacion_Jpmi.zip instale la librería siguiendo los pasos del manual de su elección, ya sea para hacer la instalación en consola, en JCreator o en NetBeans según su preferencia

5. Leer el tutorial de uso de la librería Jpmi, el cual se encuentra en la siguiente liga: https://www.cs.buap.mx/~rossainz/PCyP_Maestria/3_PMI_Java/2_Tutorial_Jpmi_actualizado.pdf

6. Pruebe que la librería funciona y esta bien instalada llevando paso a paso la implementación del problema Emisor-Receptor de esta guía de programación comprobando que lo que dice la guía se lleva a cabo tal cual en la codificación y ejecución de cada código mostrado en ella.

7. Revise y estudie los códigos de ejemplos relacionados con el problema Emisor-Receptor que se encuentran en la siguiente liga: https://www.cs.buap.mx/~rossainz/CyP_Maestria/5_EjemplosProgramados/4_PMI_Java.zip Compílelos, ejecútelos y analice como es que están implementadas cada una de las clases de éstos ejemplos



Supongamos que queremos implementar un programa en Java utilizando la librería Jpmi de Paso de Mensajes en donde un proceso llamado Emisor envíe por medio de su canal de salida un mensaje con dos datos, una cadena que representa un ID y un número entero. Este mensaje se desea que sea recibido por un proceso llamado Receptor a través de su canal de entrada y cuando lo reciba, imprima en pantalla los datos del mensaje para comprobar dicha recepción y a continuación el programa termine.

El PROCESO EMISOR

Escribamos primero el código que represente al Mensaje que será enviado y recibido por los procesos Emisor y Receptor respectivamente.

```
1
2  public class Mensaje
3  {
4   public String id;
5   public int dato;
6
7   public Mensaje()
8      {
9       id="";
10      dato=0;
11     }
12
13  public Mensaje(String id,int dato)
14     {
15      this.id=id;
16      this.dato=dato;
17     }
18 }
19
```

Como se muestra en el código de la imagen, la clase Mensaje no tiene nada en especial. Es una clase como cualquier otra dentro del paradigma de la Programación Orientada a Objetos en JAVA. En las líneas 4 y 5 del código se definen los dos datos que contendrá un objeto Mensaje. En la línea 7 se define un primer constructor que muestra la manera de crear un objeto de tipo Mensaje sin parámetros y en la línea 13 se muestra la segunda forma de construir un objeto Mensaje pasando dos parámetros: un String que representa el ID del mensaje y un int o número entero como segundo dato del mensaje. NOTA: Por un canal sólo puede viajar un objeto a la vez. Si queremos enviar varios datos al mismo tiempo deberemos incluir dichos datos dentro de una clase, generar el objeto de esa clase y enviar dicho objeto a través del canal que corresponda con la información que queremos.

NOTA IMPORTANTE: Cualquier mensaje que queramos recibir o enviar por medio de un canal de comunicación de un proceso utilizando la librería Jpmi, deberá ser un objeto de una clase definida por el usuario. No se permite enviar por un canal de comunicación ningún tipo primitivo de dato, es decir, los canales de comunicación que se crearán en los códigos siguientes no pueden recibir o enviar como mensajes datos de tipo int, boolean, char, float, double, etc… Si sólo deseáramos enviar o recibir en un proceso un dato int por medio de su canal, deberemos crear por ejemplo la clase MiEntero y en esa clase definir una variable de instancia de tipo int con su correspondiente constructor para "camuflajear el dato int" que queremos enviar o recibir.

Veamos ahora el código de la clase Emisor que será un proceso que tenga asociado un canal de salida por el que se envié un objeto de tipo Mensaje hacia otro proceso que ahorita no sabemos quién es.


```
2  import Jpmi.*;
3
4  public class Emisor implements Proceso
5    {
6     CanalSimple canal[];
7     Mensaje msg;
8
9     public Emisor(Mensaje msg, CanalSimple... canal)
10      {
11       this.msg=msg;
12       this.canal=canal;
13      }
14
15    public void run()
16      {
17       for(int i=0;i<canal.length;i++)
18       {
19        System.out.println("Emisor "+msg.id+" enviando "+msg.dato+" al receptor...");
20        canal[i].send(msg);
21       }
22      }
23   }
```

Línea 2: Se importa la librería para paso de mensajes en java Jpmi.*

Línea 4: Para que nuestra clase Emisor se defina como un proceso, en su declaración habrá que implementar la interface Proceso que esta incluida en la librería Jpmi. NOTA IMPORTANTE: Note que no se hereda de Thread, ni se implementa Runnable como se hace en la programación con memoria compartida.

Linea 6: Se utiliza el tipo o clase CanalSimple que se incluye en la librería Jpmi para declarar variables que representen los canales de comunicación que pueda tener en este caso el proceso Emisor. NOTA IMPORTANTE: Como un proceso en Jpmi puede contener la cantidad de canales que uno desee y utilizarlos algunos como de entrada y otros como de salida según la lógica del problema, en este ejemplo estamos utilizando un arreglo de canales. Sin embargo, el problema es muy simple y el proceso Emisor sólo tendrá un canal, en este caso de salida. En su momento cuando se le asigne a Emisor su canal, el arreglo canal[ ] será de una sola casilla. Si Emisor tuviera por ejemplo 10 canales, entonces el arreglo canal[ ] tendría un tamaño de 10 casillas en donde en cada casilla estaría guardado un canal de comunicación ya sea de entrada o de salida. No omito comentar que se podrían crear tantas variables como canales tuviéramos, sin embargo, el uso del arreglo resulta ser un elemento de mucha ayuda y muy ahorrativo en el código.

Línea 7: Se define la variable msg que representa el objeto Mensaje que el Emisor enviará a otro proceso por su canal de salida.

Linea 9: se define el constructor de la clase Emisor que es nuestro Proceso. Tiene dos argumentos: el mensaje que el usuario le asociará al Emisor para ser enviado por medio de su canal de salida y el parámetro canal que representa el canal de salida que el usuario le asociará al proceso Emisor. Como se muestra en el código, el parámetro canal es de tipo CanalSimple y los tres puntos (. . .) que aparecen después del tipo de dato indica a Java que el parámetro canal es en realidad un arreglo cuyo tamaño se definirá una vez que el usuario en el programa principal haya definido el o los canales que quiere asociar al Emisor. Esto también es de mucha ayuda porque en este código del Emisor no sabemos cuantos canales de comunicación le va a asignar el usuario y por lo tanto con los tres puntos le estamos indicando al Emisor que una vez que el usuario los proporcione entonces ya sabrá de qué tamaño será el arreglo de canales y almacenará automáticamente cada canal definido por el usuario dentro del arreglo canal… (una regla sintáctica es que cuando se utilizan los tres puntos para definir un arreglo, esta declaración dentro de un método, en este caso, en el constructor, deberá ser el último parámetro que se defina siempre, sino, causará un error de sintaxis cuando se compile el programa).

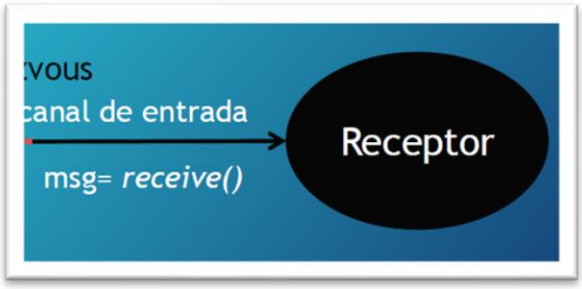
En las líneas 11 y 12 se hacen las asignaciones correspondientes a las variables de instancia del proceso Emisor.

Línea 15: se define el método Run() del proceso Emisor. Recordar que método Run( ) representa algo así como el programa principal del Proceso Emisor. Este método será ejecutado automáticamente cuando se haya creado un objeto Emisor en el Programa Principal de la aplicación o en alguna otra clase y sea lanzado para ejecutarse. Dicho de otra manera: En el método Run( ) se encuentra todo el código que el proceso Emisor ejecutará cunado sea lanzado o invocado. No omito señalar que podemos hacer uso de la modularización, es decir, escribir tantos métodos como se requieran pero para que estos sean ejecutados deberán llamarse en algún momento dentro del método Run( ) del proceso.

Si se observa el código del Run ( ) desde la línea 17 a la 20, se verá que lo que ejecuta el Emisor es un ciclo que va de 0 a la longitud del arreglo canal[ ] . Es decir, si al Emisor se le asociaron 3 canales de comunicación, su ciclo se repetirá 3 veces, una repetición por cada canal asignado. En cada repetición del ciclo el Emisor manda un letrero a pantalla indicando que va a enviar el ID del mensaje y el dato del mensaje (en este ejemplo, el número entero que el usuario defina) y a continuación lo envía por medio del canal[i] utilizando una operación de los canales llamada send(msg). E nuestro ejemplo como el Emisor sólo va a tener un canal de comunicación de salida, el ciclo se ejecutará una sola vez.

EL PROCESO RECEPTOR

El código de la clase Receptor será un proceso que tenga asociado un canal de entrada por el que recibirá un objeto de tipo Mensaje de otro proceso que ahorita no sabemos quién es.




```
2  import Jpmi.*;
3
4  public class Receptor implements Proceso
5    {
6     CanalSimple canal[];
7     Mensaje msg;
8     String name;
9
10    public Receptor(String name,CanalSimple... canal)
11    {
12     this.name=name;
13     this.canal=canal;
14    }
15
16    public void run()
17      {
18       for(int i=0;i<canal.length;i++)
19       {
20         msg=(Mensaje)canal[i].receive();
21         System.out.println("RECEPTOR "+name+" recibiendo "+msg.dato+" del emisor "+msg.id);
22       }
23      }
24   }
```

Línea 2: Se importa la librería para paso de mensajes en java Jpmi.*

Línea 4: Para que nuestra clase Receptor se defina como un proceso, en su declaración habrá que implementar la interface Proceso que está incluida en la librería Jpmi. NOTA

IMPORTANTE: Note que no se hereda de Thread, ni se implementa Runnable como se hace en la programación con memoria compartida.

Línea 6: Se utiliza el tipo o clase CanalSimple que se incluye en la librería Jpmi para declarar variables que representen los canales de comunicación que pueda tener en este caso el proceso Receptor. NOTA IMPORTANTE: Como un proceso en Jpmi puede contener la cantidad de canales que uno desee y utilizarlos algunos como de entrada y otros como de salida según la lógica del problema, en este ejemplo estamos utilizando un arreglo de canales. Sin embargo, el problema es muy simple y el proceso Recepto sólo tendrá un canal, en este caso de entrada. En su momento cuando se le asigne a Recepto su canal, el arreglo canal[ ] será de una sola casilla. Si Recepto tuviera por ejemplo 10 canales, entonces el arreglo canal[ ] tendría un tamaño de 10 casillas en donde en cada casilla estaría guardado un canal de comunicación ya sea de entrada o de salida. No omito comentar que se podrían crear tantas variables como canales tuviéramos, sin embargo, el uso del arreglo resulta ser un elemento de mucha ayuda y muy ahorrativo en el código.

Línea 7: Se define la variable msg de tipo Mensaje que utilizará el Receptor para almacenar el objeto Mensaje que reciba por su canal de entrada de otro proceso que esté conectado a él.

Línea 8: Se declara la variable name de tipo String que representa el nombre del proceso Receptor que le asigne el usuario al momento de crearlo en el programa principal o en otra clase donde se utilice.

Línea 10: se define el constructor de la clase Recepto que es nuestro Proceso. Tiene dos argumentos: el nombre que el usuario le asociará cuando lo construya y el parámetro canal que representa el canal de entrada que el usuario le asociará al proceso Receptor. Como se muestra en el código, el parámetro canal es de tipo CanalSimple y los tres puntos (. . .) que aparecen después del tipo de dato indica a Java que el parámetro canal es en realidad un arreglo cuyo tamaño se definirá una vez que el usuario en el programa principal haya definido el o los canales que quiere asociar al Receptor. Esto también es de mucha ayuda porque en este código del Receptor no sabemos cuántos canales de comunicación le va a asignar el usuario y por lo tanto con los tres puntos le estamos indicando al Receptor que una vez que el usuario los proporcione entonces ya sabrá de qué tamaño será el arreglo de canales y almacenará automáticamente cada canal definido por el usuario dentro del arreglo canal… (una regla sintáctica es que cuando se utilizan los tres puntos para definir un arreglo, esta declaración dentro de un método, en este caso, en el constructor, deberá ser el último parámetro que se defina siempre, sino, causará un error de sintaxis cuando se compile el programa).

En las líneas 12 y 13 se hacen las asignaciones correspondientes a las variables de instancia del proceso Receptor.

Línea 16: se define el método Run() del proceso Receptor. Recordar que método Run( ) representa algo así como el programa principal del Proceso Receptor. Este método será ejecutado automáticamente cuando se haya creado un objeto Receptor en el Programa Principal de la aplicación o en alguna otra clase y sea lanzado para ejecutarse. Dicho de otra manera: En el método Run( ) se encuentra todo el código que el proceso Receptor ejecutará cunado sea lanzado o invocado. No omito señalar que podemos hacer uso de la modularización, es decir, escribir tantos métodos como se requieran pero para que estos sean ejecutados deberán llamarse en algún momento dentro del método Run( ) del proceso.

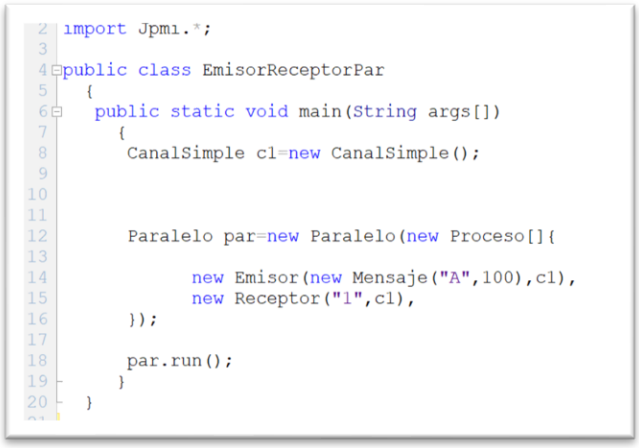
Si se observa el código del Run ( ) desde la línea 18 a la 21, se verá que lo que ejecuta el Receptor es un ciclo que va de 0 a la longitud del arreglo canal[ ] . Es decir, si al Receptor se le asociaron 3 canales de comunicación, su ciclo se repetirá 3 veces, una repetición por cada canal asignado. En cada repetición del ciclo el Receptor ejecutará el método receive( ) del canal[i] que en ese momento esté utilizando para recibir el mensaje que otro proceso le envió por ese canal de comunicación y lo almacena en la variable msg (de tipo Mensaje) y a continuación escribe en pantalla tanto el ID como el DATO del objeto Mensaje recibido. En nuestro ejemplo como el Receptor sólo va a tener un canal de comunicación de entrada, el ciclo se ejecutará una sola vez.

LA COMPOSICIÓN PARALELA DE PROCESOS

Se muestra a continuación el código del programa principal (PP) de la aplicación que muestra la comunicación del Proceso Emisor, con el proceso Receptor, conectando el canal de salida del primero con el canal de entrada del segundo a través de un mismo canal definido en el PP. A esto se le llama COMPOSICION de PROCESOS. En la librería de clase Jpmi se tienen 3 formas de hacer composiciones de procesos: Composición Paralela, Composición Secuencial y Composición Alternativa.

Es importante entender que para que la comunicación entre dos o más procesos se pueda llevar a cabo, se debe cumplir el concepto de CITA (rendezvous). Una cita se da cuando un proceso, en este caso el proceso EMISOR a la hora de que se ejecuta, se encuentra listo para enviar el mensaje y al mismo tiempo el RECEPTOR cuando se ejecuta está listo para recibir el mensaje. En ese momento se establece la cita y se efectúa la comunicación. Si el proceso EMISOR se encuentra listo para enviar el mensaje, pero el RECEPTOR no se encuentra listo para recibirlo porque esta haciendo otras cosas o bien por alguna circunstancia no lo está, entonces el sistema se bloquea hasta que el RECEPTOR se encuentre listo para la recepción

y a la inversa pasa lo mismo. ¡Si alguno de los dos procesos o ningún proceso se encuentra listo para recibir o para enviar entonces el programa se bloquea y habrá que abortarlo!...



```
2  import Jpmi.*;
3
4  public class EmisorReceptorPar
5    {
6     public static void main(String args[])
7       {
8        CanalSimple c1=new CanalSimple();
9
10
11
12       Paralelo par=new Paralelo(new Proceso[]{
13
14             new Emisor(new Mensaje("A",100),c1),
15             new Receptor("1",c1),
16       });
17
18       par.run();
19      }
20   }
```

LINEA 2: Se importa la librería para paso de mensajes en java Jpmi.*

LINEA 4: Se define la clase del programa principal de la aplicación. En este caso la clase será como cualquier otra de Java, no será un proceso y por lo tanto no hay que implementar de Proceso, mucho menos de Thread o de Runnable.

LINEA 6: se define el método main( ) o programa principal

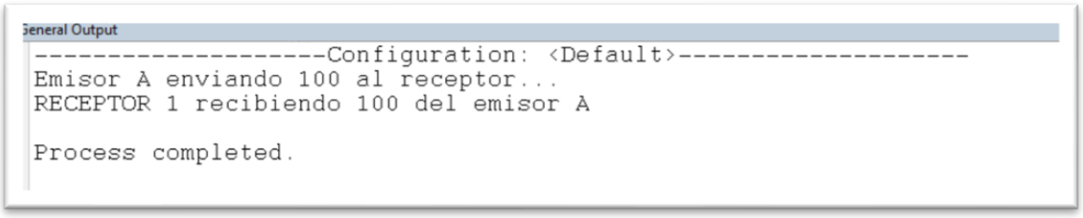
Línea 8: Se declara el canal que comunicará al proceso Emisor con el proceso Receptor, es decir, el CanalSimple c1 definido en esta línea de código conectará el canal de salida definido en la clase Emisor con el canal de Entrada definido en la clase Receptor en el nivel más alto de la aplicación.

Línea 12: Se utiliza la clase Paralelo definida en la librería Jpmi para crear una composición Paralela de dos procesos: el Proceso Emisor y el Proceso Receptor. El constructor de la clase Paralelo recibe un arreglo de procesos, por eso en el código se define un arreglo de procesos implícito en donde la línea 14 muestra que en la casilla cero de ese arreglo se almacena el proceso Emisor donde al construirlo se le pasa el nuevo mensaje y el canal c1 con el cual se va a comunicar. De la misma la línea 15 representa la casilla uno de ese arreglo y en ella se almacena el proceso Receptor donde al construirlo se le asigna un nombre como una cadena seguido del canal c1 con el cual se va a comunicar.

Línea 18. Se lanza a ejecución la composición paralela creada. Automáticamente el run( ) del objeto par en esta línea de código llama en una ejecución paralela al run( ) del Emisor y al run( ) del Receptor.

Ahora bien, como la composición es paralela se supone que tanto el proceso Emisor como el proceso Receptor se lanzan en ejecución simultánea. Ya sea que el Emisor esté listo primero para enviar o que el Receptor esté listo primero para recibir, la comunicación entre

ellos se va a dar y se llevará a cabo el concepto de cita porque los dos están en ejecución simultánea; de forma que el mensaje se enviará con éxito y también será recibido con éxito dando el siguiente resultado de ejecución:



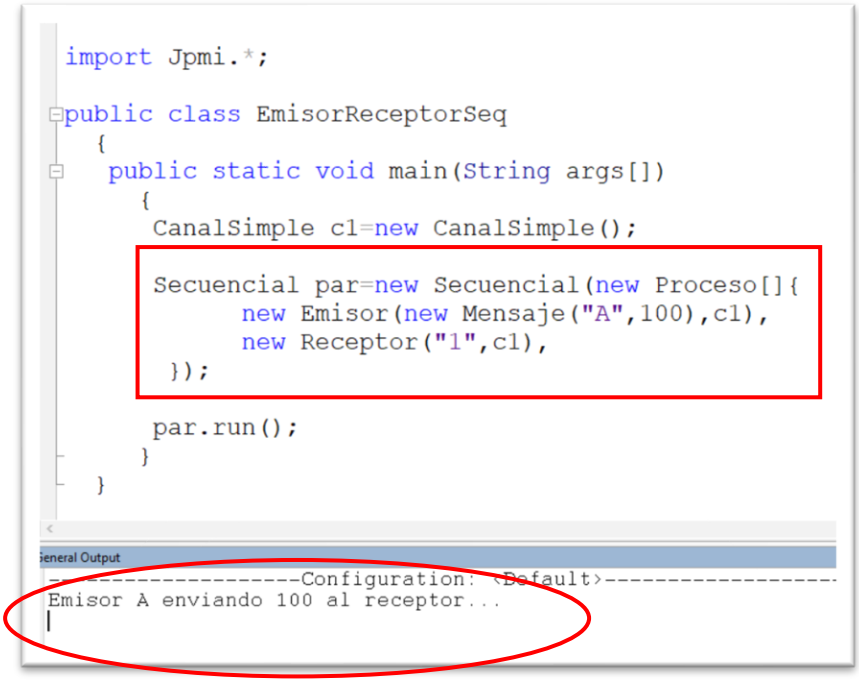
Respecto a los otros dos tipos de composición de procesos: la Composición Secuencial y la Composición Alternativa… habrá que tener cuidado en la forma en que se utilizan y dependiendo del problema saber cuándo resulta ser útil utilizarlas.

LA COMPOSICIÓN SECUENCIAL DE PROCESOS

Si en este mismo problema en el código del programa principal, cambiáramos la composición de procesos y en lugar de generar una composición paralela (como esta escrita en la línea 12 del código anterior) se generara una composición alternativa como lo muestra el código siguiente; el resultado en la ejecución sería que el Sistema se encontraría bloqueado. Porqué?. Recuérdese que para que la comunicación entre el Emisor y el Receptor exista a través de un canal, estos procesos tienen que estar activos para poder llevar a cabo la cita. Si el proceso Emisor se lanza en una ejecución secuencial con el proceso Receptor o a la inversa, sólo uno de ellos se encontraría activo en memoria mientras que el otro proceso todavía no se estaría ejecutando por la secuencialidad definida. Esto pasa porque en una composición secuencial los procesos son lanzados en secuencia de la forma en como hayan sido escritos, es decir, primero se lanza un proceso y después en secuencia se lanza el segundo que se definió dentro de la composición secuencial y después el tercer proceso y así sucesivamente, pero el segundo proceso no puede iniciar su ejecución hasta que el anterior no haya terminado por completo y se haya eliminado de memoria. Por tanto, se puede conectar el proceso Emisor con el proceso Receptor a través del canal común de comunicación que comparten en la composición secuencial hablando en términos de codificación, sin embargo la cita nunca se llevaría a cabo porque si la secuencia es lanzar primero el Emisor y luego el Receptor, el proceso Emisor se ejecuta, imprime en pantalla que va a enviar el mensaje fulanito y lo intenta enviar por su canal de salida. En ese momento el proceso Emisor se bloquea y no continua su ejecución hasta que el proceso Receptor reciba el mensaje (establecimiento de cita), pero eso nunca pasará porque el proceso Receptor no ha sido lanzado a ejecución. Para que esto suceda el Emisor tendría que terminar su ejecución por completo, pero se encuentra bloqueado a la espera de que el Receptor reciba el mensaje y se produce el el bloqueo de todo el sistema. A la inversa pasa algo similar. Si el proceso Receptor se lanza primero y después de lanza el proceso Emisor, el Receptor se encuentra listo para recibir por su canal el mensaje que le será

enviado por el Emisor, pero como el Emisor no se esta ejecutando porque para que se ejecute se requiere que el Receptor termine por completo su ejecución, entonces el sistema queda bloqueado por la espera infinita del Receptor. En conclusión, una composición secuencial para comunicar el proceso Emisor con el Receptor en este ejemplo, provoca un bloqueo del sistema y habrá que abortarlo.

Primer caso: Composición secuencial de Procesos donde la secuencia es lanzar primero el Proceso Emisor y luego el proceso Receptor, en ese orden.



Segundo Caso: Composición secuencial de Procesos donde la secuencia es lanzar primero el Proceso Receptor y luego el proceso Emisor, en ese orden.

```
import Jpmi.*;

public class EmisorReceptorSeq
   {
    public static void main(String args[])
      {
       CanalSimple c1=new CanalSimple();

       Secuencial par=new Secuencial(new Proceso[]{
             new Receptor("1",c1),
             new Emisor(new Mensaje("A",100),c1),
        });

       par.run();
      }
   }
```

```
General Output
--------------------Configuration: <Default>-------------------
```

LA COMPOSICIÓN ALTERNATIVA DE PROCESOS

Esta tercera forma de generar y lanzar a ejecución una composición de procesos es una tanto más especial y diferente que las anteriores, tanto en su codificación como en la lógica de su ejecución. Los códigos de las clases Mensaje, Emisor y Receptor, en este ejemplo se mantienen sin cambio. Nuevamente el único código que cambia es el del programa principal.

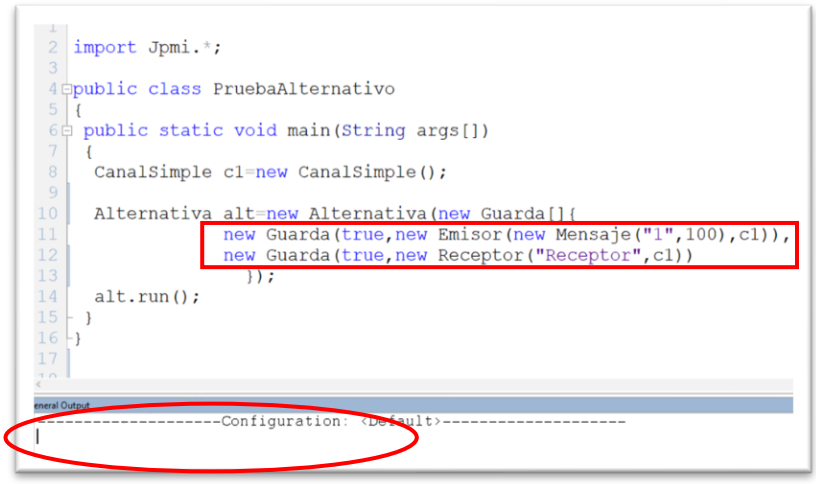
```
1
2  import Jpmi.*;
3
4  public class PruebaAlternativo
5  {
6   public static void main(String args[])
7   {
8    CanalSimple c1=new CanalSimple();
9
10   Alternativa alt=new Alternativa(new Guarda[]{
11               new Guarda(true,new Emisor(new Mensaje("1",100),c1)),
12               new Guarda(true,new Receptor("Receptor",c1))
13                 });
14   alt.run();
15  }
16 }
```

En la línea 10 se utiliza ahora la clase Alternativa que se encuentra dentro de la librería Jpmi y que sirve para definir una composición alternativa que significa elección (choice). El constructor de la clase Alternativa a diferencia de las clases Paralelo y Secuencial, requiere de un arreglo de Guardas. Un Guarda se define como un centinela que monitorea en la elección o choice si el proceso asociado al Guarda puede ser ejecutado o no por la

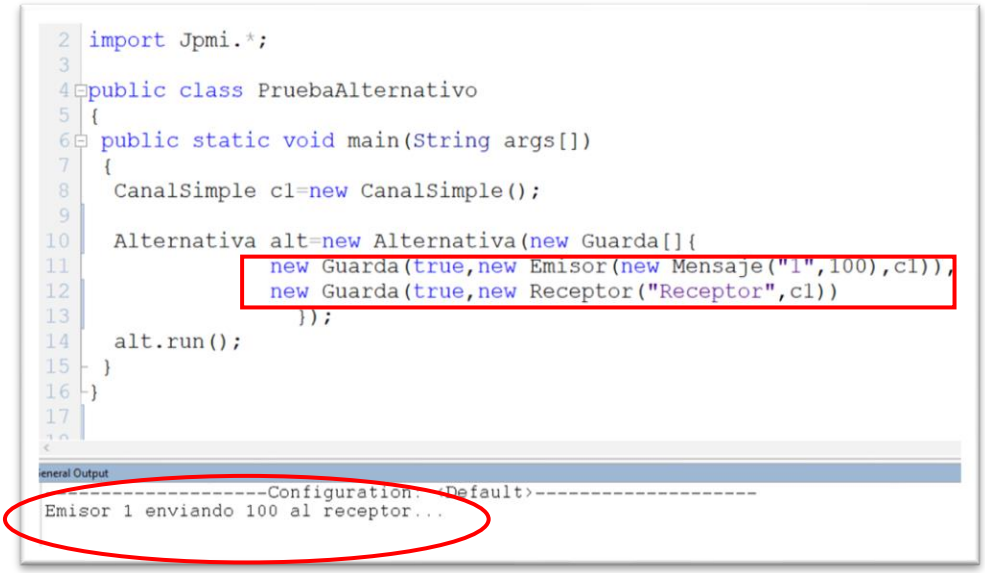
composición alternativa a la hora de elegirlo. Como se observa en el código, Guarda es también una clase de la librería Jpmi la cual en su constructor habrá que pasarle dos parámetros: una bandera y el proceso que se quiere ejecutar asociado al Guarda (líneas 11 y 12). Si la bandera es “true” entonces se le indica al Guarda que el proceso asociado, si es elegido por la composición Alternativa para ejecutarlo, podría lanzarse a ejecución. Si la bandera es “false” se le indica al Guarda que el proceso asociado no estará activo de forma que la composición Alternativa si lo elige no podrá lanzarlo a ejecución y deberá buscar otro proceso cuya bandera en el guarda esta puesta a “true”.

En nuestro ejemplo en el código de las líneas 11 y 12 los dos Guardas, uno para el Emisor y otro para el Receptor, están puestos a “true”. Esto significa que la composición alternativa podría elegir a cualquiera de ellos para lanzarlos a ejecución. El problema es que como es una elección sólo se ejecutará un proceso: o se ejecuta el proceso Emisor, o se ejecuta el proceso Receptor, pero no ambos. Nuevamente tenemos el problema del bloque en el sistema porque para que se comuniquen Emisor y Receptor los dos procesos deberían estar en ejecución y no sucede eso. ¿Cómo sabemos qué proceso va a ser elegido por la composición Alternativa para ser ejecutado? Como los dos Guardas tienen la bandera puesta a “true” la elección de qué proceso se ejecutará la realiza el sistema o “la computadora”, es decir, es una elección no determinística. El usuario no sabrá que proceso es el elegido hasta que no ejecute el programa.

CASO 1: Ejecución del programa principal del problema Emisor-Receptor en donde la composición Alternativa elije de forma no-deterministica ejecutar el proceso Receptor provocando el bloqueo del sistema porque la cita nunca se da debido a que el Emisor nunca se ejecutará y el Receptor se queda esperándolo para recibir su mensaje.

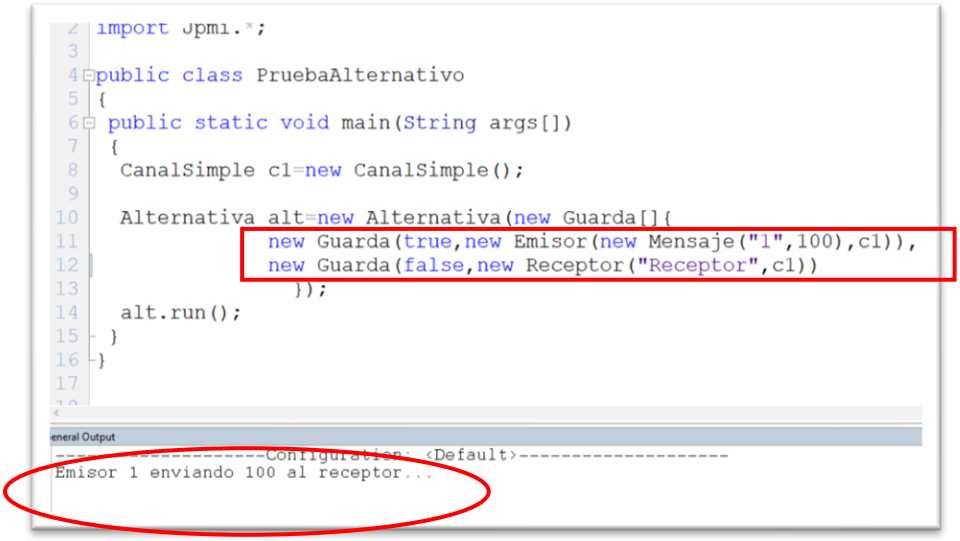


CASO 2: Ejecución del programa principal del problema Emisor-Receptor en donde la composición Alternativa elije de forma no-deterministica ejecutar el proceso Emisor provocando el bloqueo del sistema porque la cita nunca se da debido a que el Receptor nunca se ejecutará y el Emisor se queda esperándolo para enviar su mensaje.

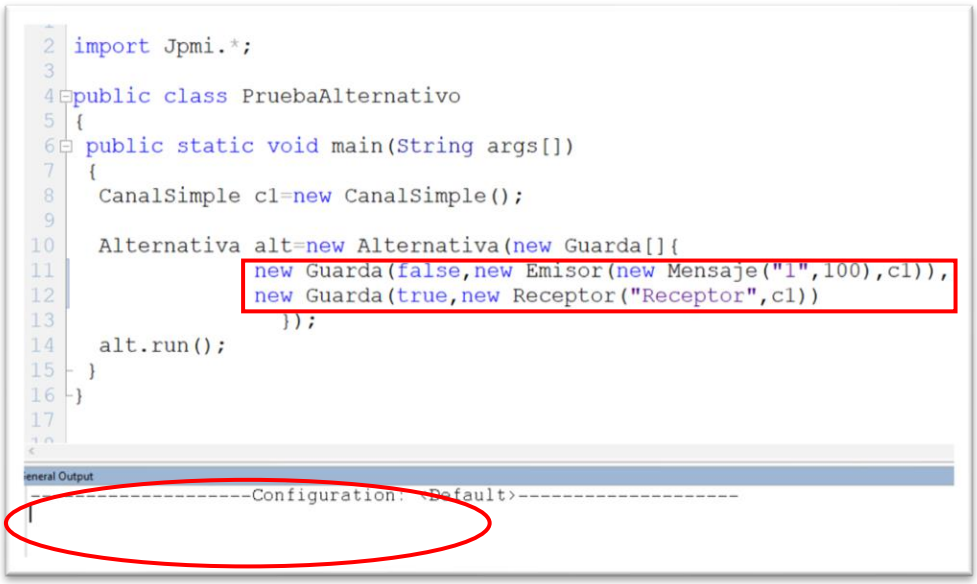


Si queremos una elección determinística, es decir, que el usuario indique qué procesos son los que la composición Alternativa debiera considerar para su ejecución y que procesos no, entonces bastará con colocar las banderas de cada Guarda en “true” o “false” según se desee. Aquí los casos determinísticos:

CASO 3: Ejecución del programa principal del problema Emisor-Receptor en donde la composición Alternativa es determinística a través del usuario al elegir ejecutar siempre el Emisor poniendo la bandera de su Guarda asociado a “true” y deshabilitar la elección y ejecución del Receptor colocando la bandera de su Guarda asociado a “false”. De esta manera al ejecutar el programa el bloqueo se sigue dando porque sólo se ejecutará el proceso Emisor y no el Receptor y la cita nunca se dará.

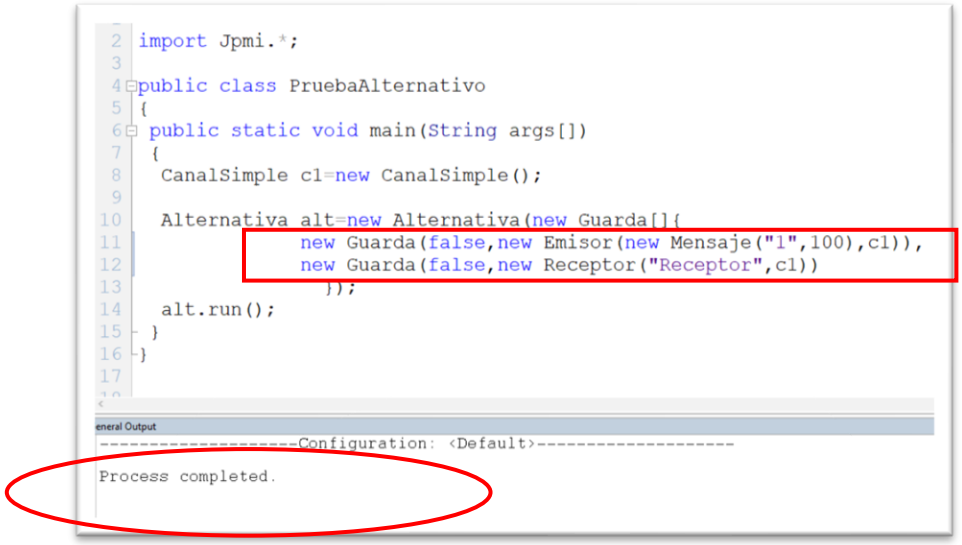


CASO 4: Ejecución del programa principal del problema Emisor-Receptor en donde la composición Alternativa es determinística a través del usuario al elegir ejecutar siempre el Receptor poniendo la bandera de su Guarda asociado a "true" y deshabilitar la elección y ejecución del Emisor colocando la bandera de su Guarda asociado a "false". De esta manera al ejecutar el programa el bloqueo se sigue dando porque sólo se ejecutará el proceso Receptor y no el Emisor y la cita nunca se dará.



CASO 5: Ejecución del programa principal del problema Emisor-Receptor en donde la composición Alternativa es determinística a través del usuario al indicarle a la composición Alternativa que no ejecute ninguno de los dos procesos, Emisor y Receptor. Esto se consigue poniendo las banderas asociadas a ambos Guardas de los procesos, a "false". En este caso

la ejecución del programa no se bloquea, simplemente termina el programa por completo sin haber ejecutado ninguno de los dos procesos.



De cualquier manera, como puede concluirse, se defina como se defina la composición Alternativa con los Guardas, en este ejemplo particular o se ejecuta el Emisor, o se ejecuta el Receptor o no se ejecuta ninguno y en todos los casos posibles salvo en el último el resultado es “un sistema bloqueado”.