Resumen secciones 16.6 a 16.10 del libro Cómo programar en Java de Deitel & Deitel, 5ª. Ed.

Relación productor/ consumidor sin sincronización

En una relación productor /consumidor, la parte de una aplicación corresponde al productor genera datos y los almacena en un objeto compartido, al consumidor lee los datos del objeto compartido. Un subproceso productor genera datos y los coloca en un objeto compartido, conocido como búfer. Un consumidor lee los datos del búfer. Si el productor que espera a colocar los siguientes datos en el búfer examina una condición y determina que el consumidor no ha leído todavía los datos anteriores del búfer, el subproceso productor debe llamar a wait. Cuando el consumidor lee los datos debe llamar a notify para permitir que un productor en espera almacene el siguiente valor. Si un consumidor encuentra el búfer vació o descubre que los datos anteriores ya han sido leídos, debe llamar a wait. Cuando el productor coloca los siguientes datos llama a notify.

En un búfer compartido: una ubicación de memoria compartida entre dos subprocesos.

Cada valor que el subproceso productor escribe en el búfer compartido debe ser consumido exactamente una vez por el consumidor. Sin embargo los subprocesos en este ejemplo no están sincronizados. Entonces se dice que el subproceso productor se ejecuta primero y el consumidor consumirá cada valor producido por el productor exactamente una vez.

Código de búfer compartido

```
//fi 16.4
//la interfaz bufer especifica los metodos llamdos por el productor y consumidor
public interface Bufer{
   public void establecer(int valor);//coloacr valro en bufer
   public int obtener();//devolver valor de bufer
}

//fig 16.5
  //el metodo run de productor conrola un subproceso que almacena los valores de 1 a 5 en ubicacioncompartida
public class Productor extends Thread
{
   private Bufer ubicacionCompartida;//referencia al objeto compartido
   //cosntuctor

   /** Creates a new instance of Producto */
   public Productor(Bufer compartido)
   {
      super("Productor");
      ubicacionCompartida=compartido;
         }
   //almacenar valores de 1 a 4 en ubicacionComaprtida
   public void run()
   {
      for(int cuenta=1;cuenta<=4;cuenta++){
         //estar incativo de0 a 3 segundo y lugo colocar valro en Bufer
         try{
            Thread.sleep((int)(Math.random()*3001));
```

```
            ubicacionCompartida.establecer(cuenta);
          }
          //si se interrumpio el subproceso inactivo, imprimir ratero de pila
          catch(InterruptedException excepcion){
            excepcion.printStackTrace();
          }
        }//fin de intriccion for
        System.err.println(getName()+"Termino de producir."+"\nTerminado"+getName()+".");
    }//fin de matodo run
    }//fin de clase productor

//fi 16.6
//el metodo run de consumidor controlar un subproceso que itera cuantro veces y lee un valro de
ubicacion compartida cada vez.
public class Consumidor extends Thread {
    private Bufer ubicacionCompartida;//referencia al objeto compartir

    //constructor
    public Consumidor(Bufer compartido)
    {
    super("Consumidor");
    ubicacionCompartida=compartido;
    }

    //leer el valor de ubicacionCompartida cuatro veces y sumar los valores
     public void run()
     {
        int suma=0;
        for(int cuenta=1;cuenta<=4;cuenta++)
        {
          //estar inactivo de 0 s3 , leer un valor de Bufer y agregarlo
          try {
            Thread.sleep((int)(Math.random()*3001));
            suma += ubicacionCompartida.obtener();
              }

          //swi se interrumpio el subproceso incativo, imprimir rastreo de la pila
          catch (InterruptedException excepcion){
            excepcion.printStackTrace();
          }
          }
        System.err.println(getName()+" Leyo valores, dado un total de:"+ suma +".\nTerminado"+
getName()+".");
     }//fin del metodo run
    }//fin de la clase Consumidor

//fig 16.7
//bufer no sincronizado repesenta a un solo entero comparrtido
public class BuferNoSincronizado implements Bufer
{
    private int bufer=-1;//compartido por los subproceso productor y cosntatres
```

```
   //colocar valor en bufer
   public void establecer(int valor)
   {
      System.err.println(Thread.currentThread().getName()+" Escribe:"+valor);
      bufer=valor;
   }
//devolver valor de bufer
   public int obtener()
   {
      System.err.println(Thread.currentThread().getName()+" Lee:"+ bufer);

         return bufer;
   }
   } //fin de lace bufernosintrocnicado

//fig. 16.8
//prueba de bufer comparrtdo cre alos subprocesos productor  y consumidor
public class PruebaBuferCompartido {
   public static void main(String [] args) {
      //crear el objeto compartido utilizando por los subprocesos
      Bufer ubicacionCompartida=new BuferNoSincronizado();

//crear objetos productor y consumidor
      Productor productor=new Productor( ubicacionCompartida);
      Consumidor consumidor =new Consumidor( ubicacionCompartida);

      productor.start();//iniciar subproceso productor
      consumidor.start();//inicar subproceso consumidor
   }
   }
```

Ejecución
compile:
run:
Consumidor Lee:-1
Productor Escribe:1
Consumidor Lee:1
Productor Escribe:2
Consumidor Lee:2
Productor Escribe:3
Productor Escribe:4
ProductorTermino de producir.
TerminadoProductor.
Consumidor Lee:4
Consumidor Leyo valores, dado un total de:6.
TerminadoConsumidor.
GENERACIÓN CORRECTA (tiempo total: 26 segundos)

Relación productor/ consumidor con sincronización

El consumidor consume solo hasta que el productor produce el valor y el productor produce un nuevo valor solo hasta que el consumidor consume el valor anterior producido. Este método nos permite demostrar que los subprocesos que utilizan el objeto compartido no saben que esta siendo sincronizados.
Código búfer sincronizado

Con base al ejemplo anterior solo se cambian dos archivos que sus códigos son:

```
/// Fig. 16.9:
// BuferSincronizado sincroniza el acceso a un solo entero compartido.

public class BuferSincronizado implements Bufer
{
  private int bufer = -1; // compartido por los subprocesos productor y consumidor
  private int cuentaBuferOcupado = 0; // cuenta de búferes ocupados

  // colocar valor en búfer
  public synchronized void establecer( int valor )
  {
    // obtener nombre del subproceso que llamó a este método, para mostrarlo en pantalla
    String nombre = Thread.currentThread().getName();

    // mientras no haya ubicaciones vacías, colocar subproceso en estado de espera
    while ( cuentaBuferOcupado == 1 ) {

      // mostrar información del subproceso y del búfer, después esperar
      try {
        System.err.println( nombre + " trata de escribir." );
        mostrarEstado( "Bufer lleno. " + nombre + " espera." );
        wait();
      }

      // si se interrumpió el subproceso en espera, imprimir el rastreo de la pila
      catch ( InterruptedException excepcion ) {
        excepcion.printStackTrace();
      }

    } // fin de instrucción while

    bufer = valor; // establecer nuevo valor de bufer

    // indicar que el productor no puede almacenar otro valor
    // sino hasta que el consumidor recupere el valor actual del búfer
    ++cuentaBuferOcupado;

    mostrarEstado( nombre + " escribe " + bufer );

    notify(); // indicar al subproceso en espera que entre al estado listo

  } // fin del método establecer; se libera el bloqueo en BuferSincronizado

  // devolver valor de bufer
```

```
  public synchronized int obtener()
  {
    // obtener nombre del subproceso que llamó a este método, para mostrarlo en pantalla
    String nombre = Thread.currentThread().getName();

    // mientras no haya datos que leer, colocar subproceso en estado de espera
    while ( cuentaBuferOcupado == 0 ) {

      // mostrar información del subproceso y del búfer, después esperar
      try {
        System.err.println( nombre + " trata de leer." );
        mostrarEstado( "Bufer vacio. " + nombre + " espera." );
        wait();
      }

      // si se interrumpió el subproceso en espera, imprimir el rastreo de la pila
      catch ( InterruptedException excepcion ) {
        excepcion.printStackTrace();
      }

    } // fin de instrucción while

    // indicar que el productor puede almacenar otro valor
    // ya que el consumidor acaba de recuperar el valor de bufer
    --cuentaBuferOcupado;

    mostrarEstado( nombre + " lee " + bufer );

    notify(); // indicar al subproceso en espera que esté listo para ejecutarse

    return bufer;

  } // fin del método obtener; libera bloqueo en BuferSincronizado

  // mostrar la operación actual y el estado del búfer
  public void mostrarEstado( String operacion )
  {
    StringBuffer lineaSalida = new StringBuffer( operacion );
    lineaSalida.setLength( 40 );
    lineaSalida.append( bufer + "\t\t" + cuentaBuferOcupado );
    System.err.println( lineaSalida );
    System.err.println();
  }

} // fin de la clase BuferSincronizado

// Fig. 16.10: PruebaBuferCompartido2.java
// PruebaBuferCompartido2 crea los subprocesos productor y consumidor.
public class PruebaBuferCompartido2
{
  public static void main( String [] args )
```

```
  {
    // crear objeto compartido utilizado por los subprocesos; usar una referencia
    // BuferSincronizado en vez de una referencia Bufer, para que podamos invocar
    // al método mostrarEstado de BuferSincronizado desde main
    BuferSincronizado ubicacionCompartida = new BuferSincronizado();
    // Mostrar encabezados de columna para los resultados
    StringBuffer encabezadosColumna = new StringBuffer( "Operacion" );
    encabezadosColumna.setLength( 40 );
    encabezadosColumna.append( "Bufer\t\tCuenta ocupado" );
    System.err.println( encabezadosColumna );
    System.err.println();
    ubicacionCompartida.mostrarEstado( "Estado inicial" );
    // crear objetos productor y consumidor
    Productor productor = new Productor( ubicacionCompartida );
    Consumidor consumidor = new Consumidor( ubicacionCompartida );
    productor.start();  // iniciar subproceso productor
    consumidor.start();  // iniciar subproceso consumidor
  } // fin de main
} // fin de la clase PruebaBuferCompartido2
```

Ejecución

```
compile:
run:
Operacion□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□Bufer           Cuenta ocupado

Estado inicial□□□□□□□□□□□□□□□□□□□□□□□□□□□-1              0

Productor escribe 1□□□□□□□□□□□□□□□□□□□□□□1              1

Consumidor lee 1□□□□□□□□□□□□□□□□□□□□□□□□1              0

Consumidor trata de leer.
Bufer vacio. Consumidor espera.□□□□□□□□□1              0

Productor escribe 2□□□□□□□□□□□□□□□□□□□□□□2              1

Consumidor lee 2□□□□□□□□□□□□□□□□□□□□□□□2              0

Productor escribe 3□□□□□□□□□□□□□□□□□□□□□3              1

Consumidor lee 3□□□□□□□□□□□□□□□□□□□□□□□3              0

Consumidor trata de leer.
Bufer vacio. Consumidor espera.□□□□□□□□□3              0

Productor escribe 4□□□□□□□□□□□□□□□□□□□□□4              1

Consumidor lee 4□□□□□□□□□□□□□□□□□□□□□□□4              0

Consumidor Leyo valores, dado un total de:10.
TerminadoConsumidor.
ProductorTermino de producir.
TerminadoProductor.
GENERACIÓN CORRECTA (tiempo total: 7 segundos)
```

Relación producto /consumidor Búfer circular

Si los dos subprocesos operan a distintas velocidades, uno de ellos invertirá más tiempo esperado. Si el subproceso productor produce los valores con más rapidez de las ubicaciones en memoria para colocar el siguiente valor.
El tiempo de espera para los subprocesos que comparten recursos y operan a las mismas velocidades promedió, podemos implementar un búfer circular que proporcione un espacio de búfer adicional en el cual el productor puede colocar valores y desde el cual el consumidor puede recuperar esos valores, el búfer se implementa como arreglo.

Codigo

```
// Fig. 16.4: Bufer.java
// La interfaz Bufer especifica los métodos llamados por el Productor y el Consumidor.

public interface Bufer {
   public void establecer( int valor );  // colocar valor en Bufer
   public int obtener();              // devolver valor de Bufer
}


// Fig. 16.11: SalidaRunnable.java
// La clase SalidaRunnable actualiza el objeto JTextArea con los resultados
import javax.swing.*;

public class SalidaRunnable implements Runnable {
   private JTextArea areaSalida;
   private String mensajeParaAnexar;

   // inicializar areaSalida y mensaje
   public SalidaRunnable( JTextArea salida, String mensaje )
   {
      areaSalida = salida;
      mensajeParaAnexar = mensaje;
   }

   // método llamado por SwingUtilities.invokeLater para actualizar areaSalida
   public void run()
   {
      areaSalida.append( mensajeParaAnexar );
   }

} // fin de la clase SalidaRunnable


// Fig. 16.12: Productor.java
// El método run de Productor controla un subproceso que
// almacena valores de 11 a 20 en ubicacionCompartida.
import javax.swing.*;

public class Productor extends Thread {
   private Bufer ubicacionCompartida;
```

```
  private JTextArea areaSalida;

  // constructor
  public Productor( Bufer compartido, JTextArea salida )
  {
    super( "Productor" );
    ubicacionCompartida = compartido;
    areaSalida = salida;
  }

  // almacenar valores de 11 a 20 en el búfer de ubicacionCompartida
  public void run()
  {
    for ( int cuenta = 11; cuenta <= 20; cuenta ++ ) {

      // estar inactivo de 0 a 3 segundos, después colocar valor en Bufer
      try {
        Thread.sleep( ( int ) ( Math.random() * 3000 ) );
        ubicacionCompartida.establecer( cuenta );
      }

      // si se interrumpió el subproceso inactivo, imprimir el rastreo de la pila
      catch ( InterruptedException excepcion ) {
        excepcion.printStackTrace();
      }
    }

    String nombre = getName();
    SwingUtilities.invokeLater( new SalidaRunnable( areaSalida, "\n" +
      nombre + " terminó de producir.\n" + nombre + " terminado.\n" ) );

  } // fin del método run

} // fin de la clase Productor


// Fig. 16.13: Consumidor.java
// El método run de Consumidor controla un subproceso que itera diez
// veces y lee un valor de ubicacionCompartida cada vez.
import javax.swing.*;

public class Consumidor extends Thread {
  private Bufer ubicacionCompartida; // referencia al objeto compartido
  private JTextArea areaSalida;

  // constructor
  public Consumidor( Bufer compartido, JTextArea salida )
  {
    super( "Consumidor" );
    ubicacionCompartida = compartido;
    areaSalida = salida;
```

```
  }

  // leer el valor de ubicacionCompartida diez veces y sumar los valores
  public void run()
  {
    int suma = 0;

    for ( int cuenta = 1; cuenta <= 10; cuenta++ ) {

      // estar inactivo de 0 a 3 segundos, leer el valor de Bufer y sumarlo a suma
      try {
        Thread.sleep( ( int ) ( Math.random() * 3001 ) );
        suma += ubicacionCompartida.obtener();
      }

      // si se interrumpió el subproceso inactivo, imprimir el rastreo de la pila
      catch ( InterruptedException excepcion ) {
        excepcion.printStackTrace();
      }
    }

    String nombre = getName();
    SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
      "\nTotal que consumió " + nombre + ": " + suma + ".\n" +
      nombre + " terminado.\n ") );

  } // fin del método run

} // fin de la clase Consumidor


// Fig. 16.14: BuferCircular.java
// BuferCircular sincroniza el acceso a un arreglo de búferes compartidos.
import javax.swing.*;

public class BuferCircular implements Bufer {

  // cada elemento del arreglo es un búfer
  private int buferes[] = { -1, -1, -1 };

  // cuentaBuferesOcupados mantiene la cuenta de búferes ocupados
  private int cuentaBuferesOcupados = 0;

  // variables que mantienen las ubicaciones de lectura y escritura en el búfer
  private int ubicacionLectura = 0, ubicacionEscritura = 0;

  // referencia al componente de la GUI que muestra la salida
  private JTextArea areaSalida;

  // constructor
  public BuferCircular( JTextArea salida )
```

```
{
  areaSalida = salida;
}

// colocar valor en bufer
public synchronized void establecer( int valor )
{
  // obtener nombre del subproceso que llamó a este método, para mostrarlo en pantalla
  String nombre = Thread.currentThread().getName();

  // mientras no haya ubicaciones vacías, colocar subproceso en estado de espera
  while ( cuentaBuferesOcupados == buferes.length ) {

    // mostrar información de subproceso y de bufer, después esperar
    try {
      SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
        "\nTodos los búferes llenos. " + nombre + " espera." ) );
      wait();
    }

    // si se interrumpió el proceso en espera, imprimir el rastreo de la pila
    catch ( InterruptedException excepcion )
    {
      excepcion.printStackTrace();
    }

  } // fin de  instrucción while

  // colocar valor en ubicacionEscritura de búferes
  buferes[ ubicacionEscritura ] = valor;

  // actualizar componente de la GUI de Swing con el valor producido
  SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
    "\n" + nombre + " escribe " + buferes[ ubicacionEscritura ] + " ") );

  // acaba de producir un valor, por lo que se incrementa el número de búferes ocupados
  ++cuentaBuferesOcupados;

  // actualizar ubicacionEscritura para la siguiente operación de escritura
  ubicacionEscritura = ( ubicacionEscritura + 1 ) % buferes.length;

  // mostrar contenido de búferes compartidos
  SwingUtilities.invokeLater( new SalidaRunnable(
    areaSalida, crearSalidaEstado() ) );

  notify(); // regresar el subproceso en espera (si hay uno) al estado listo

} // fin del método establecer

// devolver valor de bufer
public synchronized int obtener()
```

```
{
  // obtener nombre del subproceso que llamó a este método, para mostrarlo en pantalla
  String nombre = Thread.currentThread().getName();

  // mientras no haya datos que leer, colocar el subproceso en estado de espera
  while ( cuentaBuferesOcupados == 0 ) {

    // mostrar información de subproceso y de bufer, después esperar
    try {
      SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
        "\nTodos los búferes vacios. " + nombre + " espera.") );
      wait();
    }

    // si se interrumpió el subproceso en espera, imprimir el rastreo de la pila
    catch ( InterruptedException excepcion ) {
      excepcion.printStackTrace();
    }

  } // fin de instrucción while

  // obtener valor en ubicacionLectura actual
  int valorLectura = buferes[ ubicacionLectura ];

  // actualizar componente de la GUI de Swing con el valor consumido
  SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
    "\n" + nombre + " lee " + valorLectura + " ") );

  // acaba de consumir un valor, por lo que se decrementa el número de búferes ocupados
  --cuentaBuferesOcupados;

  // actualizar ubicacionLectura para la siguiente operación de lectura
  ubicacionLectura = ( ubicacionLectura + 1 ) % buferes.length;

  // mostrar contenido de búferes compartidos
  SwingUtilities.invokeLater( new SalidaRunnable(
    areaSalida, crearSalidaEstado() ) );

  notify(); // regresar el subproceso en espera (si hay uno) al estado listo

  return valorLectura;

} // fin del método obtener

// crear salida de estado
public String crearSalidaEstado()
{
  // primera línea de información de estado
  String salida =
    "(búferes ocupados: " + cuentaBuferesOcupados + ")\nbúferes: ";
```

```
      for ( int i = 0; i < buferes.length; i++ )
        salida += " " + buferes[ i ] + "  ";

      // segunda línea de información de estado
      salida += "\n         ";

      for ( int i = 0; i < buferes.length; i++ )
        salida += "---- ";

      // tercera línea de información de estado
      salida += "\n         ";

      // anexar indicadores de ubicacionLectura (L) y ubicacionEscritura (E)
      // debajo de las ubicaciones de búfer apropiadas
      for ( int i = 0; i < buferes.length; i++ )

        if ( i == ubicacionEscritura && ubicacionEscritura == ubicacionLectura )
          salida += " EL  ";
        else if ( i == ubicacionEscritura )
          salida += " E   ";
        else if ( i == ubicacionLectura )
          salida += "  L  ";
        else
          salida += "     ";

      salida += "\n";

      return salida;

    } // fin del método crearSalidaEstado

} // fin de la clase BuferCircular


// Fig. 16.15: PruebaBuferCircular.java
// PruebaBuferCircular muestra a dos subprocesos manipulando un búfer circular.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// establecer los subprocesos productor y consumidor e iniciarlos
public class PruebaBuferCircular extends JFrame {
  JTextArea areaSalida;

  // configurar la GUI
  public PruebaBuferCircular()
  {
    super( "Demostración de sincronización de subprocesos" );

    areaSalida = new JTextArea( 20,30 );
    areaSalida.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
```

```
    getContentPane().add( new JScrollPane( areaSalida ) );

    setSize( 350, 500 );
    setVisible( true );

    // crear objeto compartido utilizado por los subprocesos; utilizamos una referencia
    // BuferCircular en vez de una referencia Bufer, para poder invocar al
    // método crearSalidaEstado de BuferCircular
    BuferCircular ubicacionCompartida = new BuferCircular( areaSalida );

    // mostrar el estado inicial de los búferes en BuferCircular
    SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
      ubicacionCompartida.crearSalidaEstado() ) );

    // establecer subprocesos
    Productor productor = new Productor( ubicacionCompartida, areaSalida );
    Consumidor consumidor = new Consumidor( ubicacionCompartida, areaSalida );

    productor.start();  // iniciar subproceso productor
    consumidor.start();  // iniciar subproceso consumidor

  } // fin del constructor

  public static void main ( String args[] )
  {
    PruebaBuferCircular aplicacion = new PruebaBuferCircular();
    aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
  }

} // fin de la clase PruebaBuferCircular
```

```
Ejecución
(búferes ocupados: 0)
búferes:  -1   -1   -1
        ---- ---- ----
         EL

Todos los búferes vacios. Consumidor espera.
Productor escribe 11 (búferes ocupados: 1)
búferes:  11   -1   -1
        ---- ---- ----
          L   E

Consumidor lee 11 (búferes ocupados: 0)
búferes:  11   -1   -1
        ---- ---- ----
            EL

Todos los búferes vacios. Consumidor espera.
Productor escribe 12 (búferes ocupados: 1)
búferes:  11   12   -1
```

```
      ---- ---- ----
           L   E

Consumidor lee 12 (búferes ocupados: 0)
búferes:  11   12   -1
      ---- ---- ----
             EL

Todos los búferes vacios. Consumidor espera.
Productor escribe 13 (búferes ocupados: 1)
búferes:  11   12   13
      ---- ---- ----
       E         L

Consumidor lee 13 (búferes ocupados: 0)
búferes:  11   12   13
      ---- ---- ----
       EL

Todos los búferes vacios. Consumidor espera.
Productor escribe 14 (búferes ocupados: 1)
búferes:  14   12   13
      ---- ---- ----
        L   E

Consumidor lee 14 (búferes ocupados: 0)
búferes:  14   12   13
      ---- ---- ----
           EL

Todos los búferes vacios. Consumidor espera.
Productor escribe 15 (búferes ocupados: 1)
búferes:  14   15   13
      ---- ---- ----
           L   E

Consumidor lee 15 (búferes ocupados: 0)
búferes:  14   15   13
      ---- ---- ----
             EL

Productor escribe 16 (búferes ocupados: 1)
búferes:  14   15   16
      ---- ---- ----
       E         L

Consumidor lee 16 (búferes ocupados: 0)
búferes:  14   15   16
      ---- ---- ----
       EL
```

Productor escribe 17 (búferes ocupados: 1)
búferes: 17 15 16
---- ---- ----
L E

Productor escribe 18 (búferes ocupados: 2)
búferes: 17 18 16
---- ---- ----
L E

Consumidor lee 17 (búferes ocupados: 1)
búferes: 17 18 16
---- ---- ----
L E

Productor escribe 19 (búferes ocupados: 2)
búferes: 17 18 19
---- ---- ----
E L

Consumidor lee 18 (búferes ocupados: 1)
búferes: 17 18 19
---- ---- ----
E L

Productor escribe 20 (búferes ocupados: 2)
búferes: 20 18 19
---- ---- ----
E L

Productor terminó de producir.
Productor terminado.

Consumidor lee 19 (búferes ocupados: 1)
búferes: 20 18 19
---- ---- ----
L E

Consumidor lee 20 (búferes ocupados: 0)
búferes: 20 18 19
---- ---- ----
EL

Total que consumió Consumidor: 155.
Consumidor terminado.

Subproceso tipo “daemon”
Un subproceso daemon es un subproceso que se ejecuta para beneficio de otros subprocesos. A direncia de los subprocesos de usuario convencionales. Los subprocesos daemon no evitan que un programa termine. Algunas implementaciones del mecanismo de recolección de basura de java utilizan subprocesos daemon. Podemos designar un subproceso como daemon, llamando a su método

setDaemon con un argumento true. Si un subproceso va a ser daemon, deben establecerse, como tal antes de que su método start sea llamado, o antes de que se enlace una excepción IllegalThreadException. El método isDaemon devuelve true si un subproceso es daemon y false en caso contrario.

Interfaz Runnable

Soportar el subprocesamiento múltiple en una clase que ya extiende a otra clase distinta de Thread, debemos implementar la función Runnable en esa clase, ya que Java no permite que una clase extienda a mas de una clase a la vez. La clase Thread implementa la interfaz Runnable, según lo expresado en el encabezado de la clase:
Public class Thread extends Object implements Runnable

Al implementar la interfaz Runnable en una clase, un programa puede manipular objetos de esa clase como objetos Runnable.
Un programa que utiliza un objeto Runnable para controlar un subproceso crea un objeto Thread y asocia al objeto Runnable con ese objeto Thread. La clase Thread proporciona cinco constructores que pueden recibir referencias a objetos Runnable como argumentos. Por ejemplo, el constructor:

Public Thread (Runnable objectoRunnable)

Especifica que el método run de objetoRunnable es el método que deberá invocarse cuando el subproceso empiece a ejecutarse. El constructor:

Public Thread (Runnable objetoRunnable, String nombreSubproceso).

Se demuestra el uso de un subproceso con dos clases internas en donde cada una de ellas implementa la interfaz Runnable; uno de ellas controla los subprocesos creados en el subprograma y la otra se utiliza con el método invokeLater de SwingUtilities para asegurar que la GUI se actualice de manera correcta. En este ejemplo también demostraremos como suspender un subproceso (es decir cómo se ejecute temporalmente), como reanudar un subproceso suspendido y como terminar un subproceso en ejecución hasta que una condición se haga falsa.

NOTA: la clase Thread ya cuenta con los métodos suspend, resume, stop; sin embargo, estos métodos están desaprovechados y no deberán utilizarse más en los programas de Java.

La clase de subprograma CaracteresAleatorios muestra tres objetos JLabel y tres objetos JCheckBox. Un subproceso separado de ejecución se asocia con cada par de objetos JLabel y JCheckBox. Cada subproceso muestra letras del alfabeto en forma aleatoria, en su correspondiente objeto JLabel.

```
// Fig. 16.16: CaracteresAleatorios.java
// La clase CaracteresAleatorios demuestra el uso de la interfaz Runnable
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CaracteresAleatorios extends JApplet implements ActionListener {
   private String alfabeto = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

```
private final static int TAMANIO = 3;
private JLabel salidas[];
private JCheckBox casillasVerificacion[];
private Thread subprocesos[];
private boolean suspendido[];

// configurar la GUI y establecer arreglos
public void init()
{
  salidas = new JLabel[ TAMANIO ];
  casillasVerificacion = new JCheckBox[ TAMANIO ];
  subprocesos = new Thread[ TAMANIO ];
  suspendido = new boolean[ TAMANIO ];

  Container contenedor = getContentPane();
  contenedor.setLayout( new GridLayout( TAMANIO, 2, 5, 5 ) );

  // crear componentes de la GUI, registrar componentes de escucha y adjuntar
  // componentes al panel de contenido
  for ( int cuenta = 0; cuenta < TAMANIO; cuenta++ ) {
    salidas[ cuenta ] = new JLabel();
    salidas[ cuenta ].setBackground( Color.GREEN );
    salidas[ cuenta ].setOpaque( true );
    contenedor.add( salidas[ cuenta ] );

    casillasVerificacion[ cuenta ] = new JCheckBox( "Suspendido" );
    casillasVerificacion[ cuenta ].addActionListener( this );
    contenedor.add( casillasVerificacion[ cuenta ] );
  }

} // fin del método init

// crear e iniciar subprocesos cada vez que se hace una llamada a start (es decir, después de
// init y cuando el usuario vuelve a visitar la página Web que contiene a este subprograma)
public void start()
{
  for ( int cuenta = 0; cuenta < subprocesos.length; cuenta++ ) {

    // crear objeto Thread; inicializar objeto que implementa a Runnable
    subprocesos[ cuenta ] =
      new Thread( new ObjetoRunnable(), "Subproceso " + ( cuenta + 1 ) );

    subprocesos[ cuenta ].start(); // empezar la ejecución del objeto Thread
  }
}

// determinar ubicación del subproceso en el arreglo subprocesos
private int obtenerIndice( Thread actual )
{
  for ( int cuenta = 0; cuenta < subprocesos.length; cuenta++ )
    if ( actual == subprocesos[ cuenta ] )
```

```
        return cuenta;

    return -1;
  }

  // este método se llama cuando el usuario cambia de páginas Web; detiene todos los subprocesos
  public synchronized void stop()
  {
    // establecer referencias en null para terminar el método run de cada subproceso
    for ( int cuenta = 0; cuenta < subprocesos.length; cuenta++ )
      subprocesos[ cuenta ] = null;

    notifyAll(); // notificar a todos los subprocesos en espera, para que puedan terminar
  }

  // manejar eventos de botón
  public synchronized void actionPerformed( ActionEvent evento )
  {
    for ( int cuenta = 0; cuenta < casillasVerificacion.length; cuenta++ ) {

      if ( evento.getSource() == casillasVerificacion[ cuenta ] ) {
        suspendido[ cuenta ] = !suspendido[ cuenta ];

        // cambiar color de la etiqueta al suspender/reanudar
        salidas[ cuenta ].setBackground(
          suspendido[ cuenta ] ? Color.RED : Color.GREEN );

        // si el subproceso se reanudó, asegurarse de que empiece a ejecutarse
        if ( !suspendido[ cuenta ] )
          notifyAll();

        return;
      }
    }

  } // fin del método actionPerformed

  // clase interna privada que implementa a Runnable para controlar los subprocesos
  private class ObjetoRunnable implements Runnable {

    // colocar caracteres aleatorios en la GUI, las variables subprocesoActual e
    // indice son finales, para poder usarlas en una clase interna anónima
    public void run()
    {
      // obtener referencia al subproceso en ejecución
      final Thread subprocesoActual = Thread.currentThread();

      // determinar la posición del subproceso en el arreglo
      final int indice = obtenerIndice( subprocesoActual );

      // la condición del ciclo determina cuándo debe detenerse el subproceso; el ciclo
```

```
      // termina cuando la referencia subprocesos[ indice ] se vuelve null
      while ( subprocesos[ indice ] == subprocesoActual ) {

        // estar inactivo de 0 a 1 segundo
        try {
          Thread.sleep( ( int ) ( Math.random() * 1000 ) );

          // determinar si el subproceso debe suspender su ejecución;
          // sincronizar con el objeto de subprograma CaracteresAleatorios
          synchronized( CaracteresAleatorios.this ) {

            while ( suspendido[ indice ] &&
              subprocesos[ indice ] == subprocesoActual ) {

              // suspender temporalmente la ejecución del subproceso
              CaracteresAleatorios.this.wait();
            }
          } // fin del bloque synchronized

        } // fin de bloque try

        // si el subproceso se interrumpió durante su espera/inactividad, imprimir el rastreo de la pila
        catch ( InterruptedException excepcion ) {
          excepcion.printStackTrace();
        }

        // mostrar caracter en objeto JLabel correspondiente
        SwingUtilities.invokeLater(
          new Runnable() {

            // elegir caracter aleatorio y mostrarlo
            public void run()
            {
              char mostrarChar =
                alfabeto.charAt( ( int ) ( Math.random() * 26 ) );

              salidas[ indice ].setText(
                subprocesoActual.getName()  + ": " + mostrarChar );
            }

          } // fin de la clase interna

        ); // fin de la llamada a SwingUtilities.invokeLater

      } // fin de instrucción while

      System.err.println( subprocesoActual.getName() + " terminando" );

    } // fin del método run

  } // fin de la clase interna privada ObjetoRunnable
```

} // fin de la clase CaracteresAleatorios