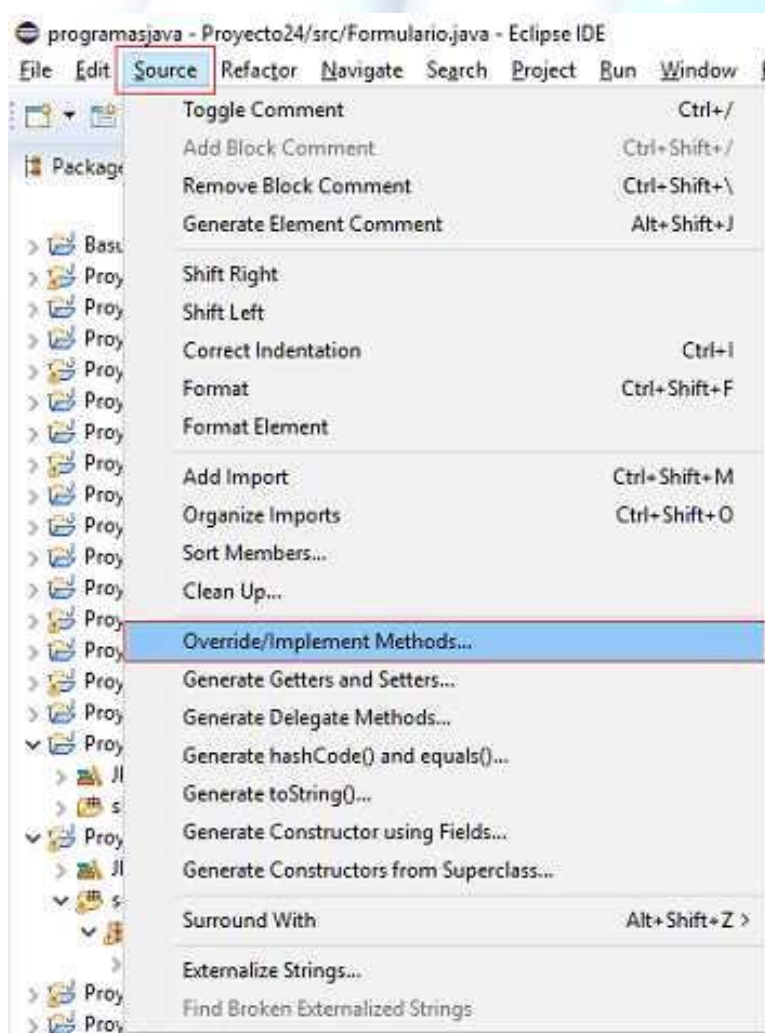


Curso de Java a distancia

Clase 34: Generación automático de métodos de las superclases con Eclipse

Es muy común que tengamos que sobrescribir otros métodos heredados a parte del equals, toString y hashCode. Eclipse nos permite ver todos los métodos heredados accediendo a la opción Source -> Override/Implement Methods...



Problema:

Implementar una clase que herede de JFrame. Luego sobrescribir el método paint y dibujar unas líneas.

Generar el método paint en forma automática con Eclipse.

El primer paso es crear el esqueleto básico para que muestre el formulario.

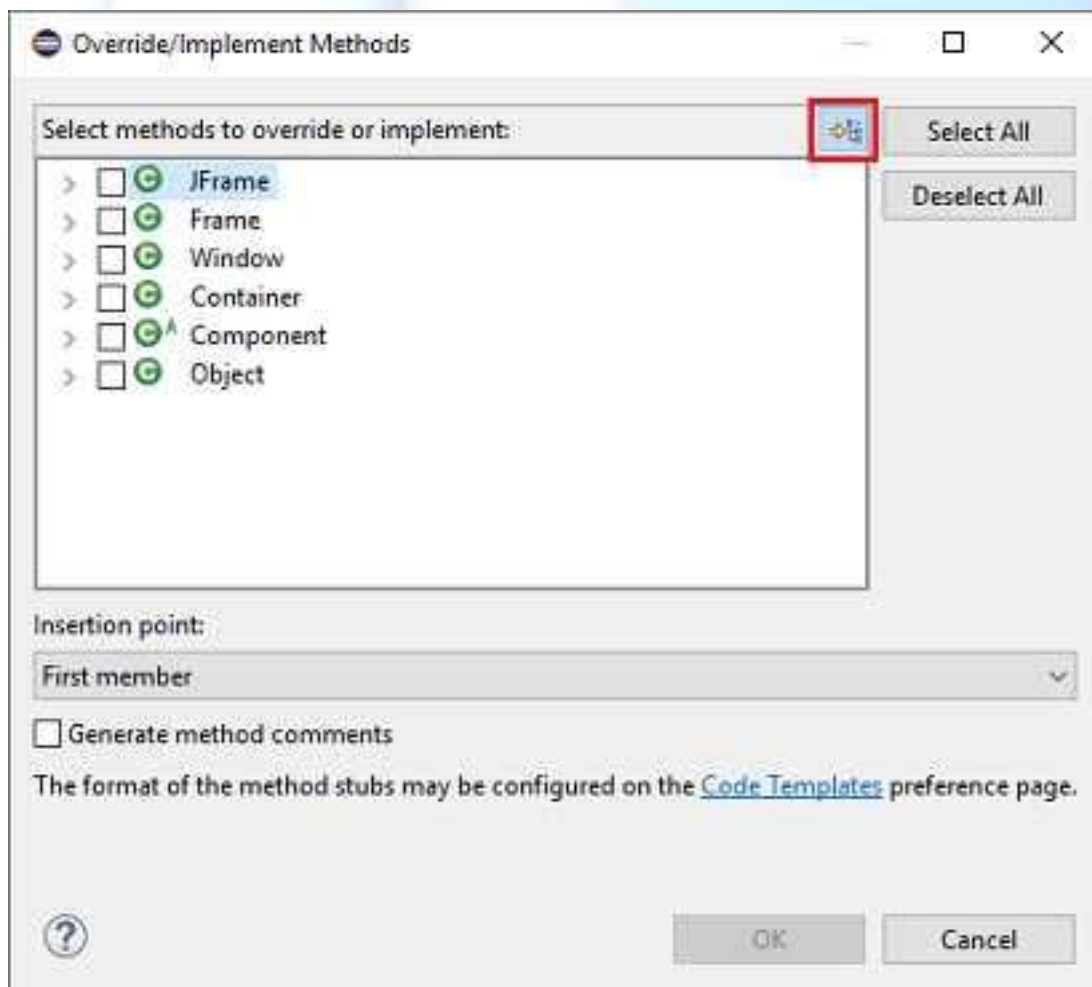
Clase: Formulario

```
import javax.swing.JFrame;
```

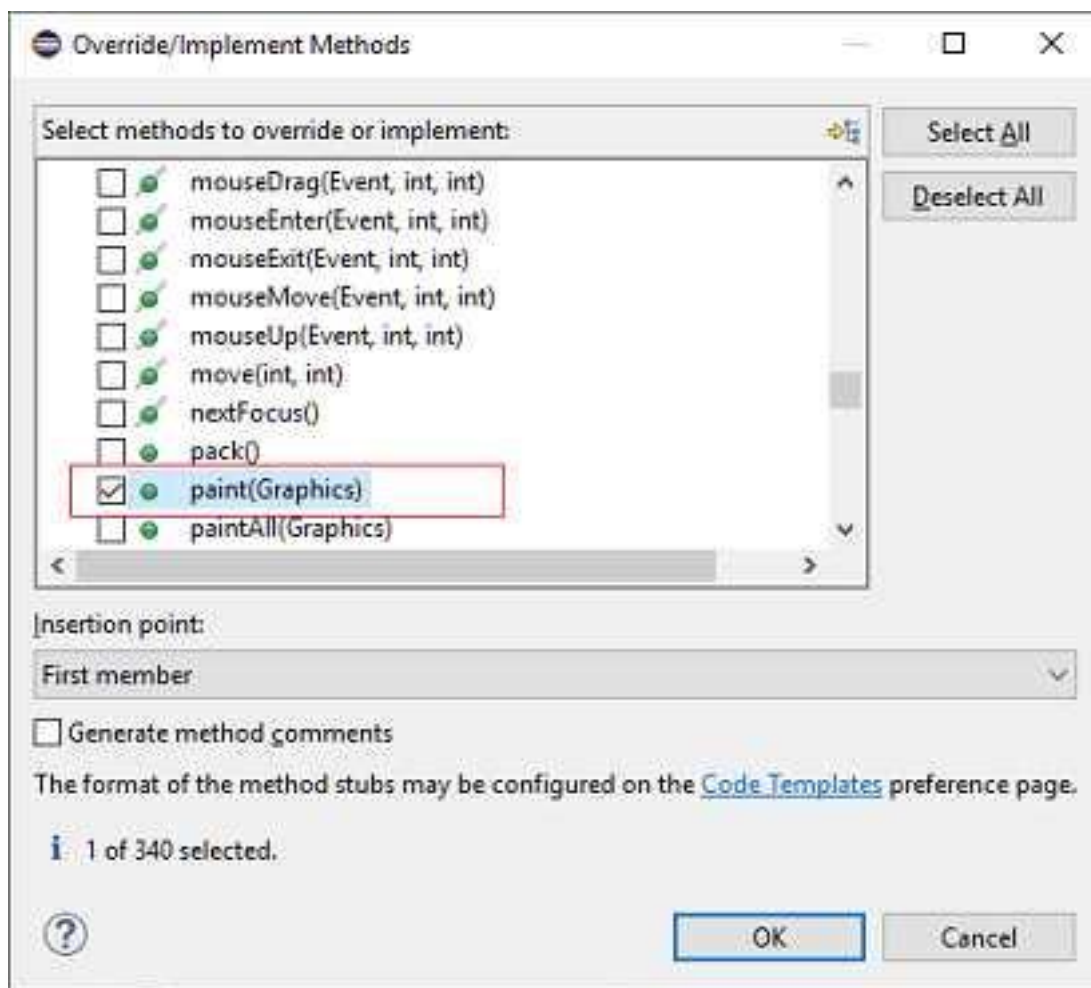
```
public class Formulario extends JFrame {
```

```
    public static void main(String[] args) {  
        Formulario formulario1=new Formulario();  
        formulario1.setBounds(0,0,800,600);  
        formulario1.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);  
        formulario1.setVisible(true);  
    }  
}
```

Ahora estando el cursor dentro de la clase Formulario procedemos a elegir la opción del menú de Eclipse "Source -> Override/Implement Methods...", la misma nos muestra un diálogo con todas las superclases de la clase Formulario:

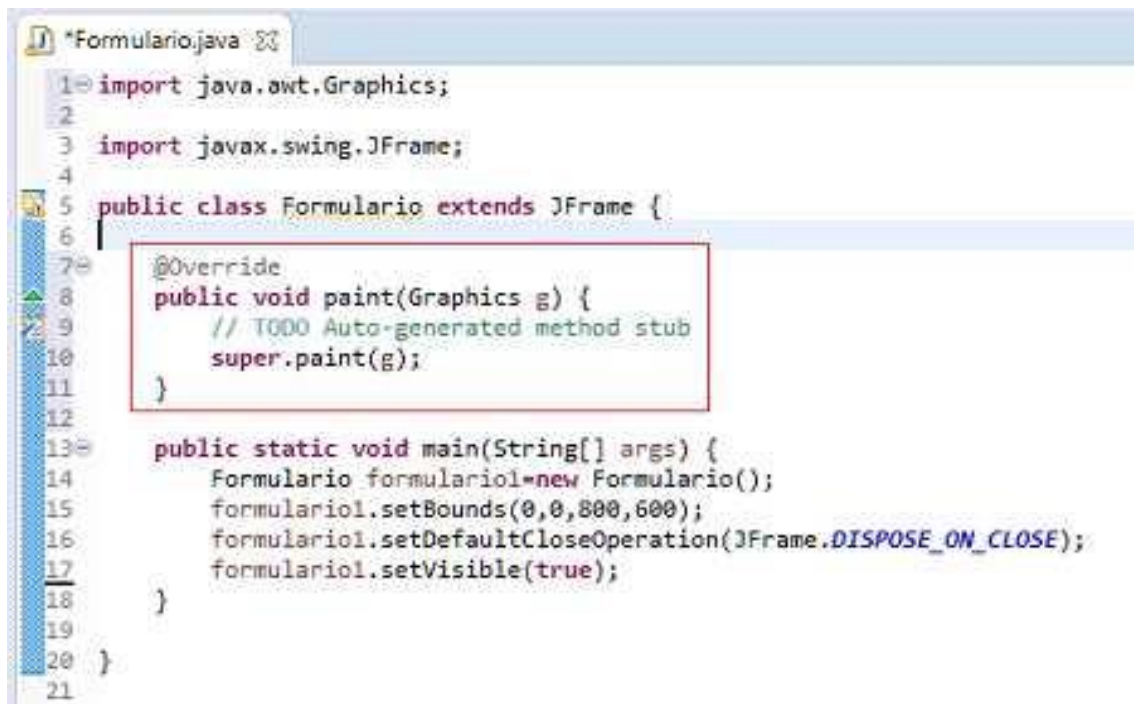


Si queremos ver los métodos en lugar de las clases debemos presionar el ícono marcado en la imagen anterior, luego se despliegan todos los métodos que podemos sobrescribir:



Podemos empezar a escribir el nombre del método y posteriormente seleccionarlo.

Una vez que aceptamos tendremos codificado el esqueleto de los métodos seleccionados:



```
1 import java.awt.Graphics;
2
3 import javax.swing.JFrame;
4
5 public class Formulario extends JFrame {
6
7     @Override
8     public void paint(Graphics g) {
9         // TODO Auto-generated method stub
10        super.paint(g);
11    }
12
13    public static void main(String[] args) {
14        Formulario formulario1 = new Formulario();
15        formulario1.setBounds(0,0,800,600);
16        formulario1.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
17        formulario1.setVisible(true);
18    }
19
20 }
21
```

Finalmente nos queda implementar la lógica para dibujar las líneas dentro del método paint:

```
import java.awt.Color;
import java.awt.Graphics;

import javax.swing.JFrame;

public class Formulario extends JFrame {

    @Override
    public void paint(Graphics g) {
        // TODO Auto-generated method stub
        super.paint(g);
        g.setColor(Color.BLUE);
        g.drawLine(0, 0, getWidth(), getHeight());
        g.drawLine(getWidth(), 0, 0, getHeight());
    }

    public static void main(String[] args) {
        Formulario formulario1 = new Formulario();
        formulario1.setBounds(0, 0, 800, 600);
        formulario1.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        formulario1.setVisible(true);
    }

}
```

Hilos en Java - clase Thread

Un sistema operativo puede realizar multiprogramación al reasignar rápidamente tiempos de la CPU entre muchos programas, dando el aspecto de paralelismo, al ejecutarse concurrentemente. Aunque el verdadero paralelismo se logra con una computadora con varias CPU.

Java soporta varios hilos de ejecución. Un proceso de Java puede crear y manejar, dentro de sí mismo, varias secuencias de ejecución concurrentes o paralelos. Cada una de estas secuencias es un hilo independiente y todos ellos comparten tanto el espacio de dirección como los recursos del sistema

operativo. Por lo tanto, cada hilo puede acceder a todos los datos y procedimientos del proceso, pero tiene su propio contador de programa y su pila de llamadas a métodos.

Todos los programas que hemos aprendido a desarrollar se ejecutan en forma secuencial, por ejemplo cuando llamamos a un método desde la main hasta que esta no finalice no continúan ejecutándose las instrucciones de la main.

Esta forma de resolver los problemas en muchas situaciones no será la más eficiente. Imaginemos si implementamos un algoritmo que busca en el disco duro la cantidad de archivos con extensión java, éste proceso requiere de mucho tiempo ya que el acceso a disco es costoso. Mientras esta búsqueda no finalice nuestro programa no puede desarrollar otras actividades, por ejemplo no podría estar accediendo a un servidor de internet, procesando tablas de una base de datos etc.

En el lenguaje Java se ha creado el concepto de hilo para poder ejecutar algoritmos en forma concurrente, es decir que comience la ejecución de la función pero continúe con la ejecución de la función main o la función desde donde se llamó al hilo.

Con los hilos podremos sacar ventajas en seguir la ejecución del programa y que no se bloquee en algoritmos complejos o que accedan a recursos lentos.

Otra gran ventaja con el empleo de los hilos es que los computadores actuales tienen múltiples procesadores y podremos ejecutar en esos casos hilos en forma paralela, con esto nuestros programas se ejecutarán mucho más rápido.

Los primeros problemas que resolvamos con hilos tienen por objetivo entender su creación y uso, más allá de su utilidad real.

Problema:

Mostrar el número "0" mil veces y el número "1" mil veces. Utilizar la programación secuencial vista hasta ahora.

Clase: MostrarCeroUno

```
public class MostrarCeroUno {  
  
    public void mostrar0() {  
        for (int f = 1; f <= 1000; f++)  
            System.out.print("0-");  
    }  
  
    public void mostrar1() {  
        for (int f = 1; f <= 1000; f++)  
            System.out.print("1-");  
    }  
  
    public static void main(String[] args) {  
        MostrarCeroUno m=new MostrarCeroUno();  
        m.mostrar0();  
        m.mostrar1();  
    }  
}
```

Si ejecutamos el programa no hay duda que primero se muestran los 1000 ceros y seguidamente se muestran los 1000 unos, ya que hasta que no finalice el método 'mostrar0' no comienza el método 'mostrar1':

Es muy importante tener en cuenta que para arrancar el hilo debemos llamar al método 'start' de la clase 'Thread' y no llamar directamente al método 'run'. El método 'start' llama posteriormente al método 'run'.

Segunda forma de crear un hilo.

Recién vimos que podemos crear un hilo planteando una clase que herede de la clase 'Thread'. Disponemos de una segunda forma de crear hilos en Java, debemos definir un objeto de la clase Thread y una clase que implemente la interface 'Runnable'. El programa anterior con esta otra metodología queda:

```
public class MostrarCeroUnoHilo {  
  
    public static void main(String[] args) {  
        HiloMostrarCero h1 = new HiloMostrarCero();  
        HiloMostrarUno h2 = new HiloMostrarUno();  
    }  
}
```

```
class HiloMostrarCero implements Runnable {  
    private Thread t;  
  
    public HiloMostrarCero() {  
        t = new Thread(this);  
        t.start();  
    }  
  
    @Override  
    public void run() {  
        for (int f = 1; f <= 1000; f++)  
            System.out.print("0-");  
    }  
}
```

```
class HiloMostrarUno implements Runnable {  
    private Thread t;  
  
    public HiloMostrarUno() {  
        t = new Thread(this);  
        t.start();  
    }  
  
    @Override  
    public void run() {  
        for (int f = 1; f <= 1000; f++)  
            System.out.print("1-");  
    }  
}
```

Ahora la clase 'HiloMostrarCero' no hereda de la clase 'Thread', en cambio implementa la interfaz 'Runnable':

```
class HiloMostrarCero implements Runnable {
```

Definimos como atributo un objeto de la clase 'Thread':

```
    private Thread t;
```

En el constructor creamos el hilo y le pasamos como referencia 'this' que representa el objeto que implementa la interfaz 'Runnable':

```
    public HiloMostrarCero() {  
        t = new Thread(this);  
        t.start();  
    }
```


Como la clase 'HiloMostrarCero' especifica que implementa la interfaz 'Runnable' estamos obligados a codificar el método 'run':

```
@Override
public void run() {
    for (int f = 1; f <= 1000; f++)
        System.out.print("0-");
}
```

Si ejecutamos la aplicación podemos comprobar la salida de unos y ceros no se hace en forma secuencial:

```

1  public class MostrarCeroUnoHilo {
2
3  public static void main(String[] args) {
4      HiloMostrarCero h1 = new HiloMostrarCero();
5      HiloMostrarUno h2 = new HiloMostrarUno();
6  }
7  }
8
9  class HiloMostrarCero implements Runnable {
10     private Thread t;
11
12     public HiloMostrarCero() {
13         t = new Thread(this);
14         t.start();
15     }
16
17     @Override
18     public void run() {
19         for (int f = 1; f <= 1000; f++)
20             System.out.print("0-");
21     }
22 }
23
24 class HiloMostrarUno implements Runnable {
25     private Thread t;
26
27     public HiloMostrarUno() {
28         t = new Thread(this);
29         t.start();
30     }
31
32     @Override
33     public void run() {
34         for (int f = 1; f <= 1000; f++)
35             System.out.print("1-");
36     }
37 }
38

```

Problems
 Javadoc
 Declaration
 Console

<terminated> MostrarCeroUnoHilo [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\javaw.exe (13 abr. 2019 11:43:18)

En la main podemos ver que se muestran 'Warnings' para las variables h1 y h2. Esto ocurre debido a que no hacemos uso de las mismas en las siguientes líneas.

En Java podemos crear un objeto de una clase y no asignarla a una variable si no vamos a hacer uso de la misma. Luego podemos codificar la main para que no aparezcan los "Warnings" con la sintaxis:

```
public static void main(String[] args) {
    new HiloMostrarCero();
    new HiloMostrarUno();
}
```

Hemos visto la sintaxis de como podemos crear hilos en Java, en el concepto siguiente lo emplearemos en problemas reales donde tiene sentido su empleo.

Hilos en Java - problemas de aplicación

En el concepto anterior vimos para que sirven los hilos y como se implementan los mismos en Java. Ahora veremos algunos ejemplos que se ven favorecidos empleando hilos.

Problema:

Desarrollar un programa que cargue en un objeto de la clase 'JTextArea' todos los directorios y archivos de la unidad "C". Como podemos imaginar esta es una actividad larga y lenta, por lo que si no la hacemos dentro de un hilo nuestro programa no responderá a las instrucciones del operador durante varios minutos.

Dispondremos de un botón para finalizar el programa en cualquier momento.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class FormularioBusqueda extends JFrame implements ActionListener {
    JTextArea textarea1;
    JScrollPane scrollpane1;
    JButton boton1;

    FormularioBusqueda() {
        setLayout(null);
        textarea1 = new JTextArea();
        scrollpane1 = new JScrollPane(textarea1);
        scrollpane1.setBounds(10, 30, 760, 300);
        add(scrollpane1);

        boton1 = new JButton("Salir");
        boton1.addActionListener(this);
        boton1.setBounds(320, 350, 100, 30);
        add(boton1);

        textarea1.setText("");
        HiloBusqueda hb = new HiloBusqueda("c:\\", textarea1);
        hb.start();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == boton1)
            System.exit(0);
    }

    public static void main(String[] arguments) {
        FormularioBusqueda fb;
        fb = new FormularioBusqueda();
        fb.setBounds(0, 0, 800, 640);
    }
}
```

```

        fb.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        fb.setVisible(true);
    }
}

class HiloBusqueda extends Thread {
    String directorio;
    JTextArea ta;

    public HiloBusqueda(String directorio, JTextArea ta) {
        this.directorio = directorio;
        this.ta = ta;
    }

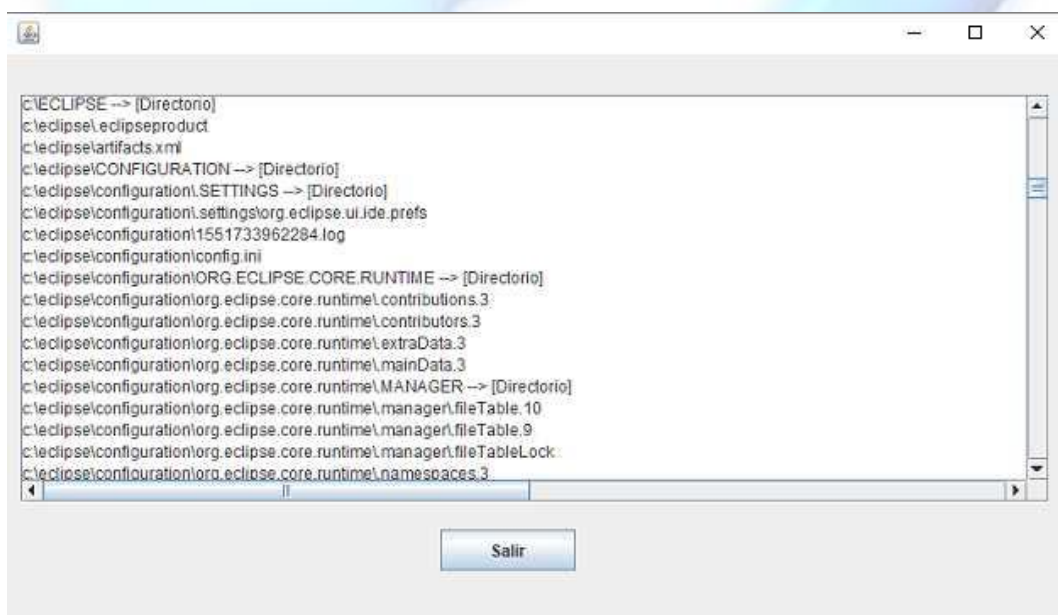
    @Override
    public void run() {
        leer(directorio);
    }

    private void leer(String inicio) {

        File ar = new File(inicio);
        String[] dir = ar.list();
        if (dir != null)
            for (int f = 0; f < dir.length; f++) {
                File ar2 = new File(inicio + dir[f]);
                if (ar2.isFile())
                    ta.append(inicio + dir[f] + "\n");
                if (ar2.isDirectory()) {
                    ta.append(inicio + dir[f].toUpperCase() + " --> [Directorio]\n");
                    leer(inicio + dir[f] + "\\");
                }
            }
    }
}

```

Cuando ejecutamos el programa podemos ver como se van cargando todos los archivos y directorios de la unidad "C:\", pero en cualquier momento podemos presionar el botón de "Salir" ya que la búsqueda se hace en otro hilo al principal de la aplicación:



Desde el constructor de la clase 'FormularioBusqueda' creamos un objeto de la clase 'HiloBusqueda', pasando la carpeta del disco duro a recorrer y el objeto de la clase JTextArea donde mostrar los datos:

```
HiloBusqueda hb = new HiloBusqueda("c:\\", textarea1);
```

Finalmente llamamos al método 'start' para inicializar el hilo:

```
hb.start();
```

La clase 'HiloBusqueda' define como atributos el directorio a recorrer y el JTextArea donde mostrar los datos, dichos atributos se inicializan en el constructor:

```
class HiloBusqueda extends Thread {  
    String directorio;  
    JTextArea ta;  
  
    public HiloBusqueda(String directorio, JTextArea ta) {  
        this.directorio = directorio;  
        this.ta = ta;  
    }  
}
```

El método 'run' se llama luego que desde la otra clase llamamos al método 'start'. En el método 'run' llamamos al método recursivo 'leer' para visitar todas las carpetas y archivos que hay en el directorio indicado en el atributo 'directorio':

```
@Override  
public void run() {  
    leer(directorio);  
}
```

Dentro del método leer lo primero que hacemos es crear un objeto de la clase 'File' y mediante el metodo 'list' recuperamos todos los archivos y carpetas del directorio donde nos encontramos posicionados:

```
private void leer(String inicio) {  
    File ar = new File(inicio);  
    String[] dir = ar.list();
```

Si la variable dir almacena un null significa que se trata de un directorio protegido y no tenemos acceso al mismo:

```
if (dir != null)
```

Ahora mediante un for recorreremos el vector con todos los nombres de archivos y carpetas:

```
for (int f = 0; f < dir.length; f++) {
```

Por cada archivo y carpeta creamos un objeto de la clase File para verificar de que tipo de recurso se trata (archivo o carpeta):

```
File ar2 = new File(inicio + dir[f]);
```

Si es un archivo lo agregamos al 'JTextArea':

```
if (ar2.isFile())  
    ta.append(inicio + dir[f] + "\n");
```

Si es un directorio además de agregarlo al 'JTextArea' procedemos a llamar en forma recursiva para recorrer la subcarpeta:

```
if (ar2.isDirectory()) {  
    ta.append(inicio + dir[f].toUpperCase() + " --> [Directorio]\n");
```

```

        leer(inicio + dir[f] + "\\");
    }

```

Si codificamos el mismo programa sin emplear el concepto de hilos podremos comprobar que el JFrame no aparece en pantalla hasta que finaliza el recorrido completo del disco duro (recorremos la carpeta 'c:\windows' en lugar de 'c:\' para que no demore tanto tiempo):

```

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

public class FormularioBusqueda extends JFrame implements ActionListener {
    JTextArea textarea1;
    JScrollPane scrollpane1;
    JButton boton1;

    FormularioBusqueda() {
        setLayout(null);
        textarea1 = new JTextArea();
        scrollpane1 = new JScrollPane(textarea1);
        scrollpane1.setBounds(10, 30, 760, 300);
        add(scrollpane1);

        boton1 = new JButton("Salir");
        boton1.addActionListener(this);
        boton1.setBounds(320, 350, 100, 30);
        add(boton1);

        textarea1.setText("");
        Busqueda hb = new Busqueda("c:\\windows\\", textarea1);
        hb.iniciar();
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == boton1)
            System.exit(0);
    }

    public static void main(String[] arguments) {
        FormularioBusqueda fb;
        fb = new FormularioBusqueda();
        fb.setBounds(0, 0, 800, 640);
        fb.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        fb.setVisible(true);
    }
}

class Busqueda {
    String directorio;
    JTextArea ta;

    public Busqueda(String directorio, JTextArea ta) {
        this.directorio = directorio;
        this.ta = ta;
    }
}

```



```

    }

    public void iniciar() {
        leer(directorio);
    }

    private void leer(String inicio) {

        File ar = new File(inicio);
        String[] dir = ar.list();
        if (dir != null)
            for (int f = 0; f < dir.length; f++) {
                File ar2 = new File(inicio + dir[f]);
                if (ar2.isFile())
                    ta.append(inicio + dir[f] + "\n");
                if (ar2.isDirectory()) {
                    ta.append(inicio + dir[f].toUpperCase() + " --> [Directorio]\n");
                    leer(inicio + dir[f] + "\\");
                }
            }
    }
}

```

Luego de ejecutar el programa anterior sin hilos podremos afirmar que el empleo de hilos para recorrer el disco duro es una herramienta fundamental.

Problema:

Crear un vector con quinientos millones de elementos con valores aleatorios. Proceder a buscar el mayor sin el empleo de hilos, luego buscar el mayor pero utilizando dos hilos, uno que busque el mayor en la primera parte del vector y otro que busque en la segunda parte.
Mostrar la cantidad de tiempo consumido en la búsqueda del mayor con hilos y sin hilos.

```

import java.util.Date;

public class MayorVector {

    public static void main(String[] args) {
        System.out.println("Cantidad de núcleos del procesador:" +
            Runtime.getRuntime().availableProcessors());
        int[] v = new int[500_000_000];
        System.out.println("Inicio de la carga del vector.");
        for (int f = 0; f < v.length; f++)
            v[f] = (int) (Math.random() * 2_000_000_000);
        System.out.println("Fin de la carga del vector.");
        Date d1 = new Date();

        HiloMayor hilo1 = new HiloMayor();
        hilo1.fijarRango(0, v.length / 2, v);
        HiloMayor hilo2 = new HiloMayor();
        hilo2.fijarRango(v.length / 2 + 1, v.length - 1, v);
        hilo1.start();
        hilo2.start();

        while (hilo1.isAlive() || hilo2.isAlive()) ;

        System.out.print("Mayor elemento del vector:");
        if (hilo1.may > hilo2.may)
            System.out.println(hilo1.may);
        else

```

```

        System.out.println(hilo2.may);
        Date d2 = new Date();
        long milisegundos = (d2.getTime() - d1.getTime());

        System.out.println("Milisegundos requeridos con 2 hilos:" + milisegundos);

        d1 = new Date();
        int may = v[0];
        for (int f = 1; f < v.length; f++) {
            if (v[f] > may)
                may = v[f];
        }
        System.out.println("Mayor elemento del vector:" + may);
        d2 = new Date();
        milisegundos = (d2.getTime() - d1.getTime());
        System.out.println("Milisegundos requeridos sin hilos:" + milisegundos);
    }
}

class HiloMayor extends Thread {
    int[] v;
    int ini, fin;
    int may;

    void fijarRango(int i, int f, int[] v) {
        this.ini = i;
        this.fin = f;
        this.v = v;
    }

    public void run() {
        may = v[ini];
        for (int f = ini + 1; f < fin; f++) {
            if (v[f] > may)
                may = v[f];
        }
    }
}

```

Si la computadora tiene más de un núcleo podemos comprobar que la búsqueda del mayor requiere menos tiempo al emplear dos hilos que se ejecutan en forma paralela:

```

1  import java.util.Date;
2
3  public class MayorVector {
4
5      public static void main(String[] args) {
6          System.out.println("Cantidad de núcleos del procesador:" + Runtime.getRuntime().availableProcessors());
7          int[] v = new int[500_000_000];
8          System.out.println("Inicio de la carga del vector.");
9          for (int f = 0; f < v.length; f++)
10             v[f] = (int) (Math.random() * 2_000_000_000);
11          System.out.println("Fin de la carga del vector.");
12          Date d1 = new Date();
13
14          HiloMayor hilo1 = new HiloMayor();
15          hilo1.fijarRango(0, v.length / 2, v);
16          HiloMayor hilo2 = new HiloMayor();
17          hilo2.fijarRango(v.length / 2 + 1, v.length - 1, v);
18          hilo1.start();
19          hilo2.start();
20
21          while (hilo1.isAlive() || hilo2.isAlive())
22              ;
23
24          System.out.print("Mayor elemento del vector:");
25      }
26  }

```

Problems | Javadoc | Declaration | Console

```

<terminated> MayorVector [Java Application] C:\Program Files\Java\jdk-11.0.2\bin\javaw.exe [14 abr. 2019 11:34:47]
Cantidad de núcleos del procesador:8
Inicio de la carga del vector.
Fin de la carga del vector.
Mayor elemento del vector:1999999999
Milisegundos requeridos con 2 hilos:160
Mayor elemento del vector:1999999999
Milisegundos requeridos sin hilos:251

```

Podemos saber la cantidad de núcleos de nuestra computadora llamando al método estático 'getRuntime' el cual retorna un objeto de la clase 'Runtime' y mediante este llamamos al método 'availableProcessors':

```
System.out.println("Cantidad de núcleos del procesador:" +  
    Runtime.getRuntime().availableProcessors());
```

Creamos dos objetos de la clase HiloMayor y le pasamos los rangos del vector 'v' que deben obtener el mayor:

```
HiloMayor hilo1 = new HiloMayor();  
hilo1.fijarRango(0, v.length / 2, v);  
HiloMayor hilo2 = new HiloMayor();  
hilo2.fijarRango(v.length / 2 + 1, v.length - 1, v);  
hilo1.start();  
hilo2.start();
```

Luego que llamamos al método 'start' para cada hilo procedemos a quedarnos dentro de un while vacío (por eso es fundamental el punto y coma al final del while):

```
while (hilo1.isAlive() || hilo2.isAlive()) ;
```

El método 'isAlive' retorna true mientras el hilo no ha terminado.

Cuando los dos hilos hay finalizado procedemos a buscar el mayor en el hilo principal del programa y calcular también la cantidad de milisegundos requeridos:

```
d1 = new Date();  
int may = v[0];  
for (int f = 1; f < v.length; f++) {  
    if (v[f] > may)  
        may = v[f];  
}  
System.out.println("Mayor elemento del vector:" + may);  
d2 = new Date();  
milisegundos = (d2.getTime() - d1.getTime());  
System.out.println("Milisegundos requeridos sin hilos:" + milisegundos);
```

Como el equipo donde ejecuté el programa tiene más de un núcleo el tiempo requerido con dos hilos paralelos es menor que la búsqueda del mayor en un único hilo.

Si ejecutamos el programa en una computadora con un único núcleo el empleo de hilos hará más ineficiente la búsqueda del mayor del vector.

También podríamos haber utilizado varios hilos para la carga de los valores aleatorios y hacer más rápido su almacenamiento.

Muchas gracias hasta la próxima clase.

Alsina 16 [B1642FNB] San Isidro | Pcia. De Buenos Aires |Argentina |

TEL.: [011] 4742-1532 o [011] 4742-1665 |

www.institutosanisidro.com.ar info@institutosanisidro.com.ar