

Universidad Nacional Autónoma de México

Facultad de ingeniería



Java Avanzado

**Edgar E. García Cano
Jorge A. Solano**

Temario

- 1. Excepciones**
- 2. Flujos de datos**
- 3. Hilos**
- 4. Clases internas**
- 5. Colecciones**
- 6. Archivo jar**
- 7. Interfaz gráfica de usuario (GUI)**
- 8. Sockets**



1

Excepciones



1. Excepciones

1.1 Jerarquía de excepciones

1.2 Estructura try-catch

1.3 Bloque finally

1.4 Propagación de excepciones

1.5 Declaración de excepciones



Excepciones

Durante la ejecución de un programa es posible que se presenten condiciones anómalas (errores). Las excepciones permiten controlar los errores producidos. Las excepciones son objetos que contienen información del error que se ha producido.

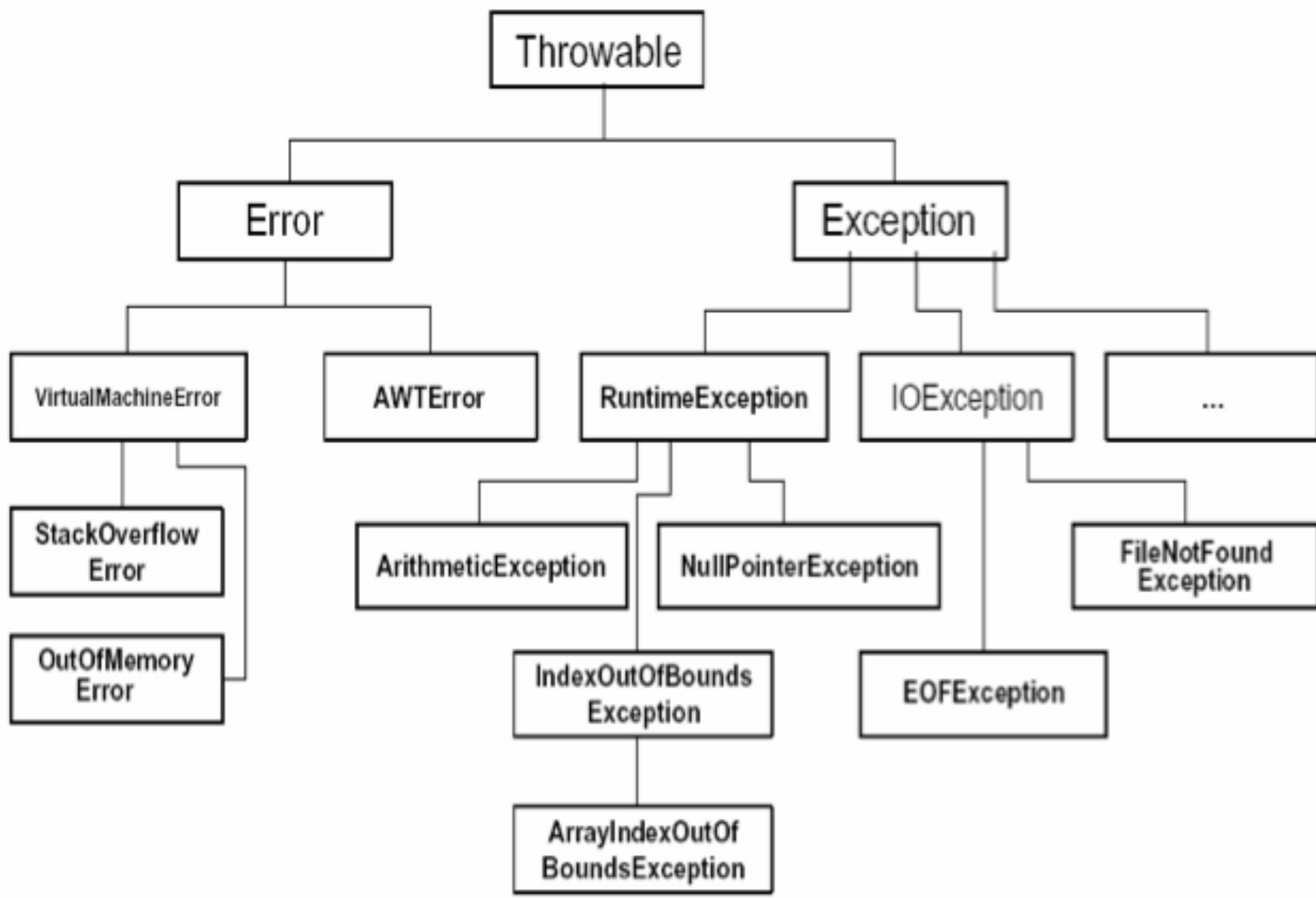
Los errores que produce una aplicación pueden ser generados por la lógica propia del programa (como un índice fuera del rango de un arreglo, una división entre cero, etc.) o errores generados por los propios objetos que denuncian algún tipo de estado no previsto o condición que no pueden manejar.

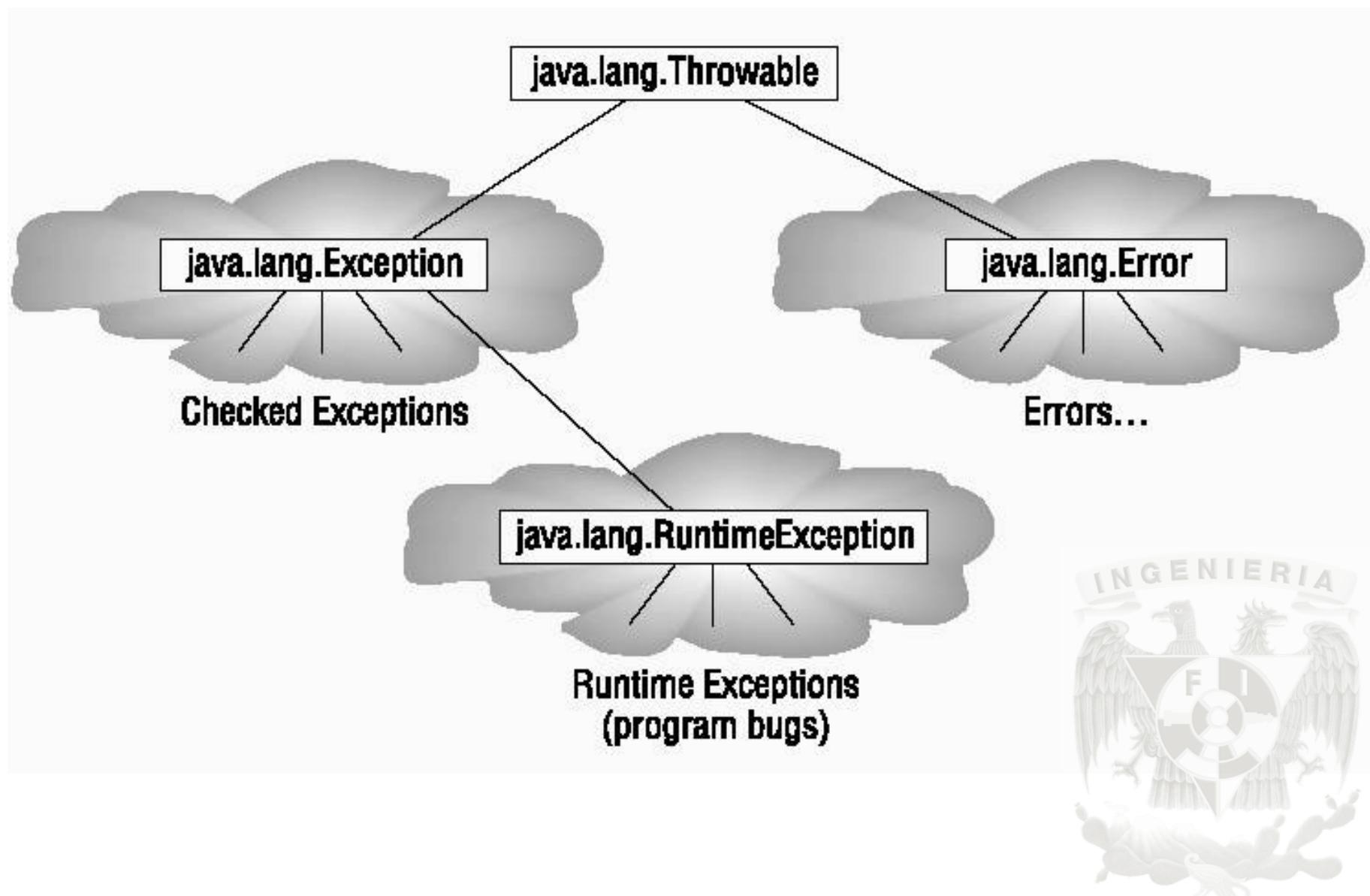
1.1 Jerarquía de excepciones

Las excepciones también manejan jerarquías, es decir, se parte de una excepción general hacia excepciones de un tipo más específico.

Todas las excepciones provienen de la clase Exception que, a su vez, deriva de la clase Throwable y ésta, a su vez, deriva de la clase Object.







Como se puede observar en la figura anterior, existen dos subclases que derivan de la clase base *Throwable*: *Error* y *Exception*.

Las clases que derivan de *Error* representan situaciones inusuales que no son causadas por errores de programación e indican cosas que normalmente no suceden durante la ejecución de un programa.

Generalmente las aplicaciones no son capaces de recuperarse de un *Error* y, por tal motivo, no se requiere que el programador los maneje.



1.2 Estructura try-catch

Cuando se presenta una excepción (Condición excepcional) el flujo normal del programa se ve afectado. Para capturar/manejar estos errores se cuenta con la estructura de control **try-catch**.

La estructura ***try-catch*** está compuesta por dos partes principales: el **bloque try** y el **bloque catch**.

En el **bloque try** se coloca el código que se quiere ejecutar y que podría contener un error (división entre cero, índice fuera del arreglo, comparación entre tipos diferentes, etc.).



El bloque *catch* permite capturar excepciones, es decir, permite manejar los errores que genere el código del bloque try en tiempo de ejecución impidiendo así que el programa deje de ejecutarse y posibilitando al desarrollador enviar el error generado.

El bloque *catch* es opcional y se pueden tener más de un bloque catch para capturar excepciones.

```
catch (ClassNotFoundException e) {...}  
catch (IOException e) {...}  
catch (Exception e) {...}
```



1.3 Bloque finally

Existe un tercer bloque de código dentro de la estructura try-catch. Este bloque es opcional y es conocido como bloque finally.

El código que se implementa en el bloque *finally* siempre se ejecuta, independientemente de que se ejecute o no una excepción.



La sintaxis completa de esta estructura de control (try-catch-finally) es:

```
try {
    // bloque de código a ejecutarse
    // dentro del flujo normal del programa
} catch (Excepcion e) {
    // bloque de código a ejecutarse
    // si ocurre un error
} finally {
    // bloque de código a ejecutarse al final
}
```



Los siguientes códigos son válidos para la estructura try-catch-finally.

```
try {  
    // código  
} catch (AlgunaException ae) {  
    // código  
}
```

```
try {  
    // código  
} catch (ExcepcionEspecífica ae) {  
    // código  
} catch (ExceptionGeneral ae) {  
    // código  
}
```



```
try {  
    // código  
} catch (AlgunaException ae) {  
    // código  
} finally {  
    // código  
}
```

```
try {  
    // código  
} finally {  
    // código  
}
```



Los siguientes códigos NO son válidos para la estructura try-catch-finally.

```
try {  
    // código  
}
```

```
try {  
    // código  
}  
System.out.println("Fuera del bloque try");  
catch(Exception ex) {}
```



Ejemplo 1.1

```
public class TryCatchFinally {  
    public static void main(String[] args) {  
        try {  
            String mensajes[] = {"Primero", "Segundo", "Tercero" };  
            for (int i=0; i<=3; i++)  
                System.out.println(mensajes[i]);  
        } catch ( ArrayIndexOutOfBoundsException e ) {  
            System.out.println("Error: apuntador fuera del rango  
                del arreglo");  
        } finally {  
            System.out.println("A pesar de todo, se ejecuta el  
                bloque finally");  
        }  
    }  
}
```



1.4 Propagación de excepciones

Cuando ocurre una excepción, es posible arrojarla de un método a otro método.

Una excepción puede ocurrir en un método ‘C’ y, si ésta no es manejada inmediatamente, puede ser lanzada al método ‘B’ (método que invocó a C). Si la excepción tampoco es manejada en éste método (B), se sigue lanzando la excepción hasta que llega al método main() y, si éste tampoco maneja la excepción, el programa termina anormalmente.



Ejemplo 1.2

```
class PropagacionExcepciones {  
    public static void main(String[] args){  
        try{  
            throw new Exception();  
        } catch(Exception e) {  
            try{  
                throw new Exception();  
            } catch(Exception e2) {  
                System.out.println("Excepción interna");  
            }  
            System.out.println("Excepción media");  
        }  
        System.out.println("Excepción externa");  
    }  
}
```



Java permite capturar varias tipos de excepciones, por lo tanto, es posible declarar diferentes bloques para capturar excepciones (catch).

Cuando una excepción es lanzada, JVM tratará de encontrar un bloque catch para ese tipo de excepciones. Si no encuentra alguno tratará buscar un tipo de excepción más genérico (supertipo), hasta encontrar el más adecuado.



Ejemplo 1.3

```
import java.io.*;
public class LeerInfo {
    public static void main(String args[]) {
        try {
            RandomAccessFile raf =
                new RandomAccessFile("archivo.txt", "r");
            byte b[] = new byte[1000];
            raf.readFully(b, 0, 1000);
        } catch(FileNotFoundException e) {
            System.err.println("Archivo no encontrado");
            System.err.println(e.getMessage());
            e.printStackTrace();
        } catch(IOException e) {
            System.err.println("Error de entrada/salida");
            System.err.println(e.toString());
            e.printStackTrace();
        }
    }
}
```

El programa anterior intenta abrir un archivo para leer sus datos. La lectura y escritura de archivos puede generar muchas excepciones de tipo entrada o salida de datos (*IOException*), además existen una excepción dedicada para cuando no existe el archivo (*FileNotFoundException*), que es una excepción más específica.

En el código anterior se puede ver como se manejan, de manera separada, las excepciones *IOException* y *FileNotFoundException*. Si se generara otras excepciones no manejadas, como *EOFException*, éstas caen dentro del bloque de excepción *IOException*.



Las subclases de una excepción tienen que colocarse antes de las superclases, de lo contrario el programa no compilará. El siguiente código no compila:

```
try {  
    // Código para leer escribir archivo  
} catch (IOException e) {  
    // Manejo general de IOExceptions  
} catch (FileNotFoundException ex) {  
    // Manejo solo de FileNotFoundException  
}
```

El código anterior devuelve el siguiente error al compilar:

Exception java.io.FileNotFoundException has already been caught



1.5 Declaración de excepciones

Para que un método sea capaz de lanzar una excepción es necesario que, dentro de la firma (declaración) del método, se especifique el tipo de excepción que puede lanzar. Para especificar que un método puede lanzar una excepción se utiliza la palabra reservada `throws`:

```
void miMetodo() throws MiExcepcion1, MiExcepcion2
{
    // Código a ejecutar
}
```



El método “miMetodo” tiene como valor de retorno void, no acepta ningún parámetro como argumento y especifica que puede lanzar dos tipos de excepciones, tanto MyException1 como MyException2.

La cláusula *throws* ayuda a lanzar la excepción hacia otro método, evitando hacer uso del manejo de excepciones.

Es posible crear excepciones propias. Para ello es solo se tiene que heredar de la clase *Exception*.



Ejemplo 1.4

```
public class ExcepcionPropia extends  
Exception {  
    public ExcepcionPropia() {  
        super();  
    }  
  
    public ExcepcionPropia(String msj) {  
        super(msj);  
    }  
}
```



Ejemplo 1.4

```
public class UnaClase{  
    public UnaClase() {  
    }  
    public void revisar (String s) throws ExcepcionPropia {  
        System.out.println(s);  
        throw new ExcepcionPropia ("Error:" +s);  
    }  
    public void revisar2 (String s) throws ExcepcionPropia {  
        revisar (s);  
    }  
    public static void main(String[] a){  
        UnaClase uc= new UnaClase();  
        try{  
            uc.revisar2("¿Se puede?");  
        } catch(ExcepcionPropia ep) {  
            ep.printStackTrace();  
            System.out.println("No se pudo");  
        }  
    }  
}
```

Excepciones típicas

Excepción	Descripción	Generalmente arrojada por:
ArrayIndexOutOfBoundsException	Arroja cuando se intenta acceder a un valor fuera del arreglo.	JVM
ClassCastException	Cuando se intenta realizar convertir una variable de referencia a otro tipo.	JVM
IllegalArgumentException	Ocurre cuando un método recibe un argumento con formato diferente al esperado por el método	Programación
IllegalStateException	Se arroja cuando el contexto no encuadra con la operación que se intenta. Ej.: Usar Scanner cuando ya ha sido cerrado.	Programación
NullPointerException	Se arroja cuando se intenta acceder a una referencia que cuyo valor es nulo (NULL).	JVM

Excepción	Descripción	Generalmente arrojada por:
NumberFormatException	Se produce cuando un método intenta convertir una cadena de caracteres a entero y recibe una cadena de caracteres que no se puede convertir.	Programación
AssertionError	Se arroja cuando la prueba de un valor booleano regresa falso (false).	Programación
ExceptionInInitializerError	Se arroja cuando se intenta inicializar una variable estática o un bloque de inicialización	JVM
StackOverflowError	Se arroja cuando la llamada a un método recursivo es muy profunda (cada invocación se agrega a la pila)	JVM
NoClassDefFoundError	Se arroja cuando la JVM no puede encontrar la clase que necesita debido a un error en la línea de comandos, detalle en el Classpath o la omisión de un archivo .class.	JVM

2

Flujos de datos



2. Flujos de datos

2.1 Clase File

2.2 Lectura y escritura de datos

2.2.1 Clase FileOutputStream

2.2.2 Clase FileInputStream

2.2.3 Clase FileWriter

2.2.4 Clase FileReader

2.2.5 BufferedReader

2.2.6 Scanner

2.2.7 Console

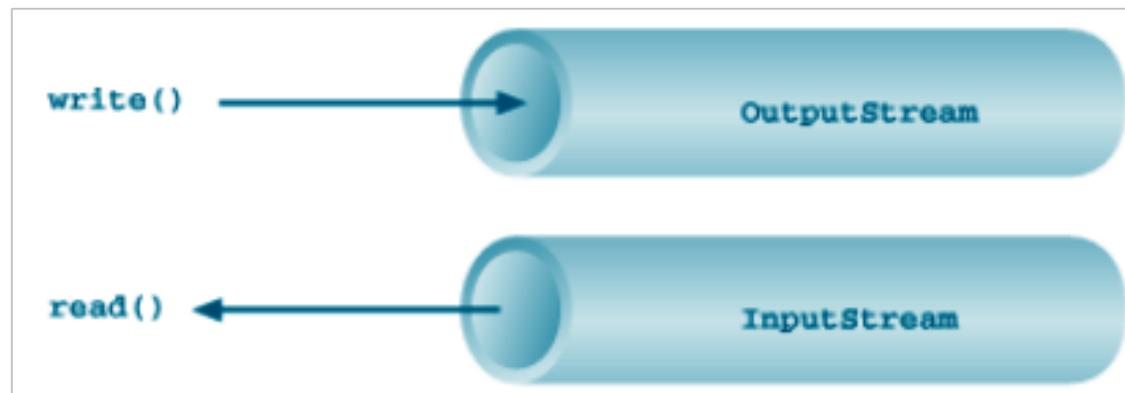
2.2.8 StringTokenizer

2.3 Serialización



Flujos de datos

Las entradas y las salidas de datos en java se manejan mediante flujos de datos (streams). Un stream es una conexión entre el programa (código fuente) y la fuente (archivo de lectura) o el destino (archivo de escritura) de los datos.



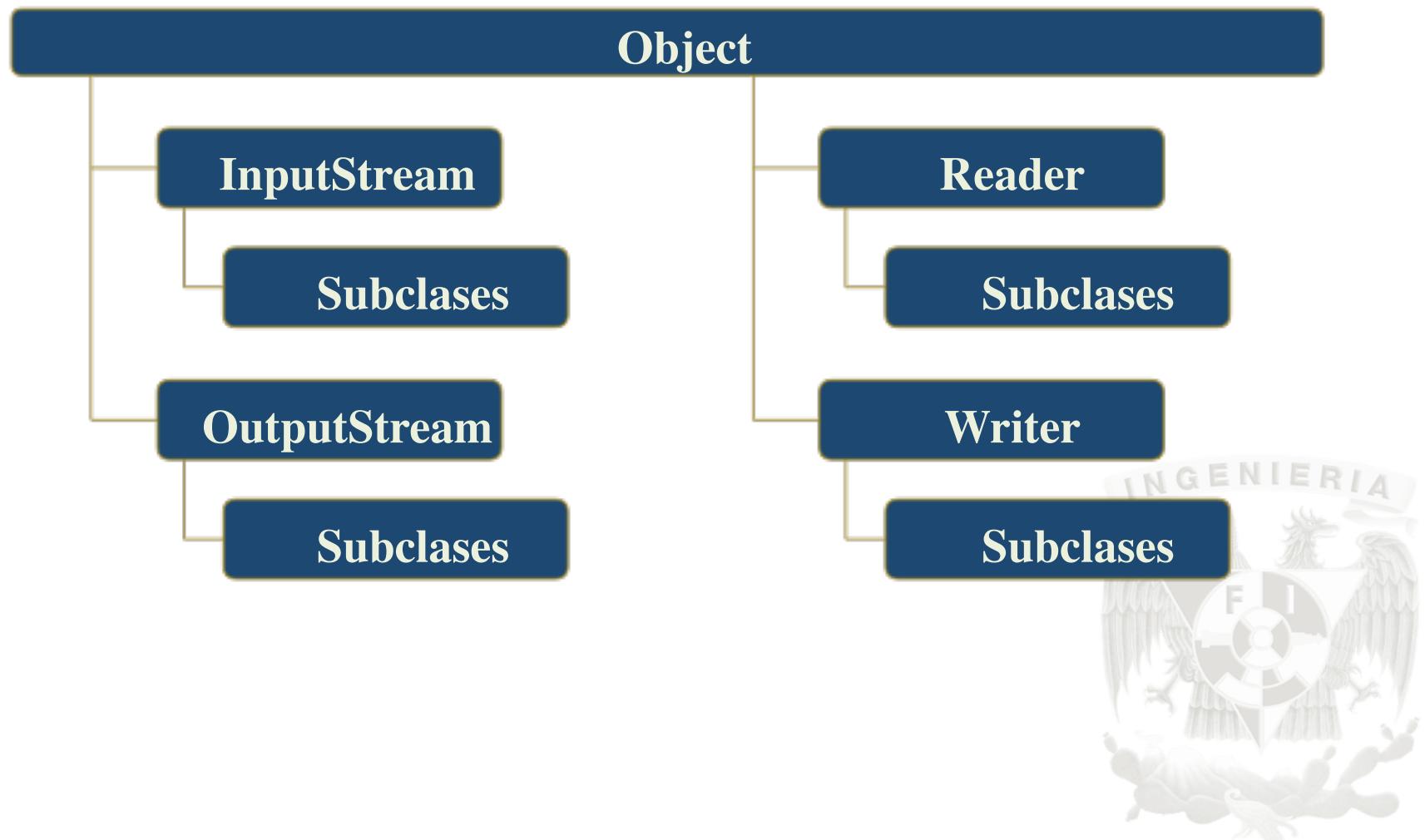
Existen 4 jerarquías de clases relacionadas con la entrada y salida de datos.

- **Las clases derivadas de InputStream (para lectura) y de OutputStream (para escritura) manejan streams de bytes.**
- **Las clases derivadas de Reader (para lectura) y Writer (para escritura) manejan caracteres en vez de bytes.**

Las clases que permiten manejar flujos de datos se encuentran dentro del paquete java.io



Jerarquía de clases de los flujos de E/S



2.1 Clase File

La clase file permite manejar archivos o carpetas, es decir, crear y borrar tanto archivos como carpetas.

Es importante hacer notar que cuando se crea un objeto de la clase File no se crea ningún archivo o directorio, solo se instancia un objeto de este tipo. La creación de archivos o carpetas se realizan al momento de invocar el método respectivo, por ejemplo: createNewFile() o mkdir().



Algunos métodos que posee la clase File son los siguientes:

- **exists()**
- **createNewFile()**
- **mkdir()**
- **delete()**
- **renameTo()**
- **list()**



Ejemplo 2.1

```
import java.io.*;
class Escribir {
    public static void main(String [] args) {
        try {
            File archivo = new File("archivo.txt");
            System.out.println(archivo.exists());
            boolean seCrea = archivo.createNewFile();
            System.out.println(seCrea);
            System.out.println(archivo.exists());
        } catch(IOException e) { }
    }
}
```



2.2 Lectura y escritura de datos

Para leer o escribir flujos de bytes desde o hacia un archivo se utilizan las clases *FileOutputStream* y *FileInputStream*, que son subclases de *OutputStream* e *InputStream*, respectivamente.

Para leer o escribir flujos de caracteres desde o hacia un archivo se utilizan las clases *FileWriter* y *FileReader* que son subclases de *Writer* y *Reader*, respectivamente.



2.2.1 Clase FileOutputStream

La clase *FileOutputStream* permite escribir bytes en un archivo. Esta clase hereda los métodos de la clase *OutputStream* y, además, posee sobrecarga de constructores:

FileOutputStream (String nombre)

FileOutputStream (String nombre, boolean añadir)

FileOutputStream (File archivo)

El primer constructor abre un flujo de salida hacia el archivo especificado (nombre). El segundo constructor también abre un flujo de salida hacia el archivo especificado y, si el archivo existe, permite agregar datos al mismo (añadir = true). El tercer constructor abre un flujo de salida a partir de un objeto File.

Ejemplo 2.2

```
import java.io.FileOutputStream;
import java.io.IOException;

public class ClaseOutputStream {
    public static void main (String [] args){
        FileOutputStream fos = null;
        byte[] buffer = new byte[81];
        int nBytes;
        try {
            System.out.println("Escribir el texto a guardar en el
archivo:");
            nBytes = System.in.read(buffer);
            fos = new FileOutputStream("fos.txt");
            fos.write(buffer,0,nBytes);
        } catch (IOException ioe){
            System.out.println("Error: " + ioe.toString());
        }
    }
}
```



En el ejemplo anterior se define un arreglo llamado buffer de 81 bytes donde se va a almacenar lo que se capture del teclado.

Después se crea el flujo de bytes mediante la clase FileOutputStream y se redirige al archivo de nombre fos.txt.

Al final, se hace uso del método write de la clase FileOutputStream para escribir los datos del buffer en el archivo, que recibe como primer parámetro la referencia a la matriz que contiene los bytes a escribir, como segundo parámetro la posición de la matriz del primer dato y como último parámetro el número de datos tipo bytes a escribir.

El método write es un método sobrecargado. Los otros usos del mismo se pueden consultar en el API de java.



Es una buena costumbre cerrar el flujo de datos cuando éste ya no se vaya a utilizar. Para ello se puede ocupar el bloque finally:

```
finally {
    try {
        if (fos != null)
            fos.close();
    } catch (IOException ioe){
        System.out.println("Error : " + ioe.toString());
    }
}
```



2.2.2 Clase FileInputStream

FileInputStream permite leer flujos de bytes desde un archivo. Hereda de la clase *InputStream* y, además, posee diferentes constructores (sobrecarga):

FileInputStream(String nombre)

FileInputStream(File archivo)

donde, el primer constructor abre un flujo de entrada desde el archivo especificado por nombre y el segundo constructor abre un flujo de entrada a partir de un objeto del tipo File.



Ejemplo 2.3

```
import java.io.FileInputStream;
import java.io.IOException;

public class ClaseInputStream {
    public static void main (String [] args){
        FileInputStream fis = null;
        byte[] buffer = new byte[81];
        int nbytes;
        try {
            fis = new FileInputStream("leer.txt");
            nbytes = fis.read(buffer, 0, 81);
            String texto = new String(buffer, 0, nbytes);
            System.out.println(texto);
        }
    }
}
```



Ejemplo 2.3

```
catch (IOException ioe) {  
    System.out.println("Error: " + ioe.toString());  
} finally {  
    try {  
        if (fis != null) fis.close();  
    } catch (IOException ioe) {  
        System.out.println("Error al cerrar el archivo.");  
    }  
}  
}
```



Para el ejemplo anterior, se define un flujo que va a leer desde el archivo leer.txt (si no existe el archivo se genera una excepción). También se crea un buffer de 81 bytes.

FileInputStream lee el texto desde el archivo y lo almacena en el buffer creado. Se lee el archivo hasta que se terminen los caracteres del mismo o hasta que se llene el buffer (desde la posición 0 hasta la 81), lo primero que ocurra.

El método `read` devuelve el número de bytes leídos o -1 si se finalizó de leer el archivo.



2.2.3 Clase *FileWriter*

La clase *FileWriter* hereda de *Writer* y permite escribir caracteres en un archivo.

La clase *BufferedWriter* también deriva de la clase *Writer* y permite añadir un buffer para realizar una escritura eficiente de caracteres.

La clase *PrintWriter*, que también deriva de *Writer*, permite escribir de forma sencilla en un archivo de texto, ya que posee los métodos *print* y *println*, idénticos a los de *System.out*. El método *close()* cierra el stream.



Para escribir una cadena de caracteres en un archivo de texto se puede utilizar el siguiente código:

```
// Se instancia la clase que permite crear/leer archivos  
FileWriter fw = new FileWriter("archivo.txt");  
  
// Se crea un buffer de escritura de caracteres hacia el archivo  
BufferedWriter bw = new BufferedWriter(fw);  
  
// Se instancia la clase que permite escribir en el archivo  
PrintWriter salida = new PrintWriter(bw);  
  
// Se escribe en el archivo  
salida.println(texto);  
  
// Se cierra el buffer y el archivo  
salida.close();
```



Ejemplo 2.4

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.io.IOException;

public class ClaseFileWriter{
    public static void main (String [] leer){
        String texto = "";
        try{
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir texto:");
            texto = br.readLine();
            FileWriter fw = new FileWriter("archivo.txt");
            BufferedWriter bw = new BufferedWriter(fw);
```

Ejemplo 2.4

```
PrintWriter salida = new PrintWriter(bw);
salida.println(texto);
salida.close();
} catch (IOException ioe){
System.out.println("\n\nError al abrir o guardar el archivo:");
ioe.printStackTrace();
} catch (Exception e){
System.out.println("\n\nError al leer de teclado:");
e.printStackTrace();
}
}
```



2.2.4 Clase FileReader

Las clases *Reader* se utilizan para obtener los caracteres ingresados desde la entrada estándar.

La clase *FileReader* hereda de *Reader* y permite leer caracteres de un archivo.

InputStreamReader es una clase derivada de *Reader* que convierte los streams de bytes a streams de caracteres, es decir, lee bytes y los convierte en caracteres. *System.in* es el objeto de la clase *InputStream* para recibir datos desde la entrada estándar del sistema (el teclado).



Ejemplo 2.5

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ClaseFileReader{
    public static void main (String [] escribir){
        String texto = "";
        try {
            BufferedReader br;
            FileReader fr = new FileReader("leer.txt");
            br = new BufferedReader(fr);
            System.out.println("El texto contenido en el archivo leer.txt es:");
            String linea = br.readLine();
            while (linea != null ) {
                System.out.println(linea);
                linea = br.readLine();
            }
        }
    }
}
```

Ejemplo 2.5

```
    catch (IOException ioe){
        System.out.println("\n\nError al abrir o guardar el archivo:");
        ioe.printStackTrace();
    } catch (Exception e){
        System.out.println("\n\nError al leer de teclado:");
        e.printStackTrace();
    }
}
```



2.2.5 BufferedReader

La clase ***BufferedReader***, que también deriva de la clase ***Reader***, añade un buffer para realizar una lectura eficiente de caracteres. Dispone del método **readLine** que permite leer una línea de texto y tiene como valor de retorno un **String**.

System.in es el objeto de la clase **InputStream** para recibir datos desde la entrada estándar del sistema (el teclado).



Para leer una cadena de texto utilizando esta clase se realiza lo siguiente:

```
String texto = “”;  
BufferedReader br;  
br = new BufferedReader(new InputStreamReader(System.in));  
texto = br.readLine();
```



Ejemplo 2.6

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class LeerTeclado {
    public static void main (String [] args){
        try {
            String texto = "";
            BufferedReader br;
                br = new BufferedReader(new
InputStreamReader(System.in));
            System.out.println("Escribir el texto deseado:");
            texto = br.readLine();
            System.out.println("El texto escrito fue: " + texto);
        } catch (IOException ioe){
            System.out.println("Error al leer caracteres: \n" + ioe);
        }
    }
}
```

2.2.6 StringTokenizer

La clase *StringTokenizer* permite separar una cadena de texto por palabras (espacios) o por algún otro carácter. La clase *StringTokenizer* pertenece al paquete *java.util*.

El código para separar una cadena de texto es el siguiente:

```
StringTokenizer      st      =      new
StringTokenizer(texto);
while(st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
```

El método *countTokens()* devuelve el número de tokens que se pueden extraer de la frase.



Ejemplo 2.7

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.PrintWriter;
import java.util.StringTokenizer;

public class LeerTecladoCompleto {
    public static void main (String [] leer){
        String texto = "";
        try{
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Escribir texto:");
            texto = br.readLine();
```

Ejemplo 2.7

```
System.out.println("\n\nEl texto separado por espacios es:");
StringTokenizer st = new StringTokenizer(texto);
while(st.hasMoreTokens()) {
    System.out.println(st.nextToken());
}
br.readLine();
System.out.println("\n\nEl texto completo es:");
System.out.println(texto);

FileWriter fw = new FileWriter("archivo.txt");
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter salida = new PrintWriter(bw);
salida.println(texto);
salida.close();
System.out.println("\n\nEl texto se guardó en archivo.txt");
```

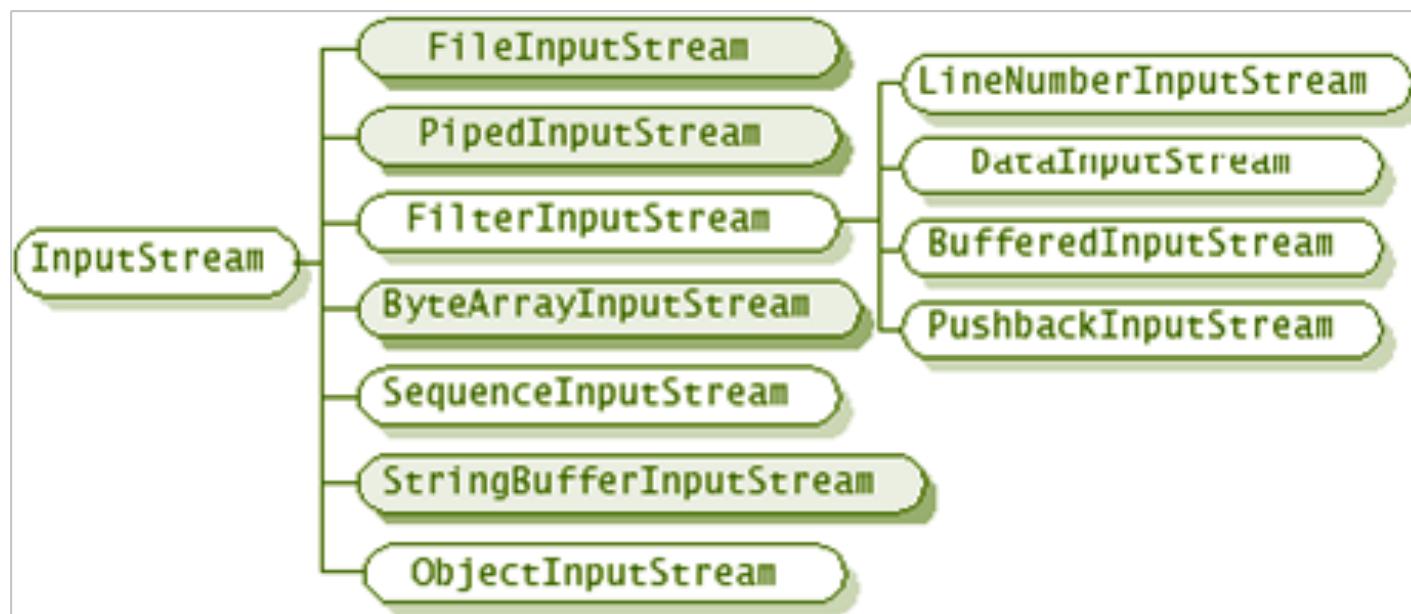


Ejemplo 2.7

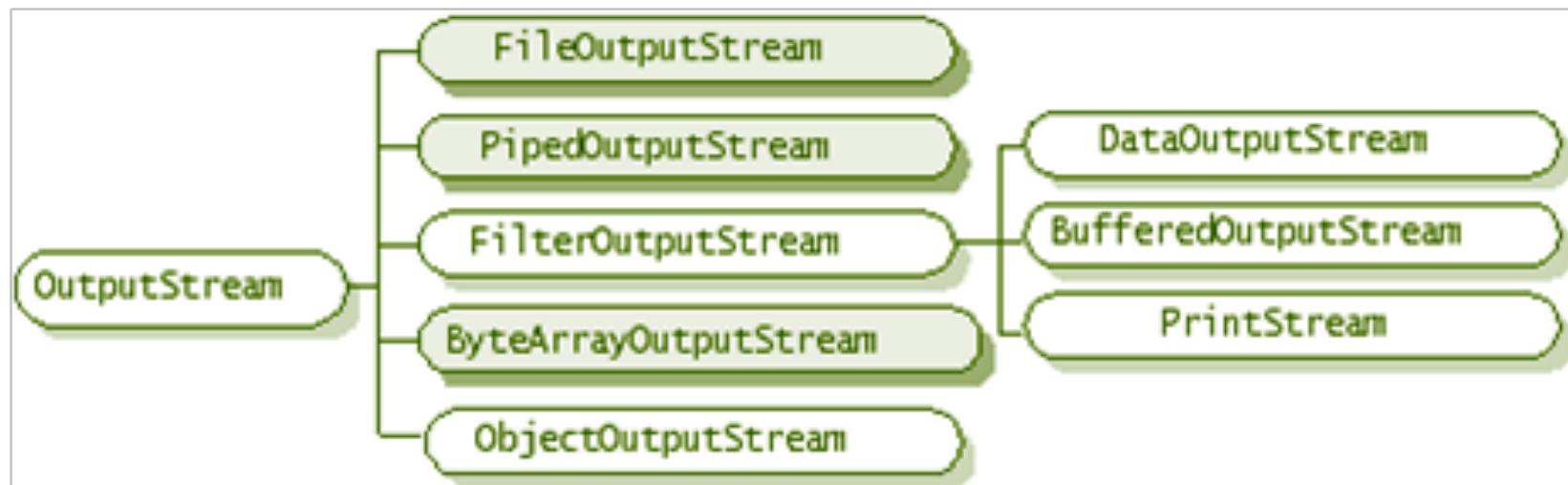
```
    } catch (IOException ioe){
        System.out.println("\n\nError al abrir o guardar el archivo:");
        ioe.printStackTrace();
    } catch (Exception e){
        System.out.println("\n\nError al leer de teclado:");
        e.printStackTrace();
    }
}
```



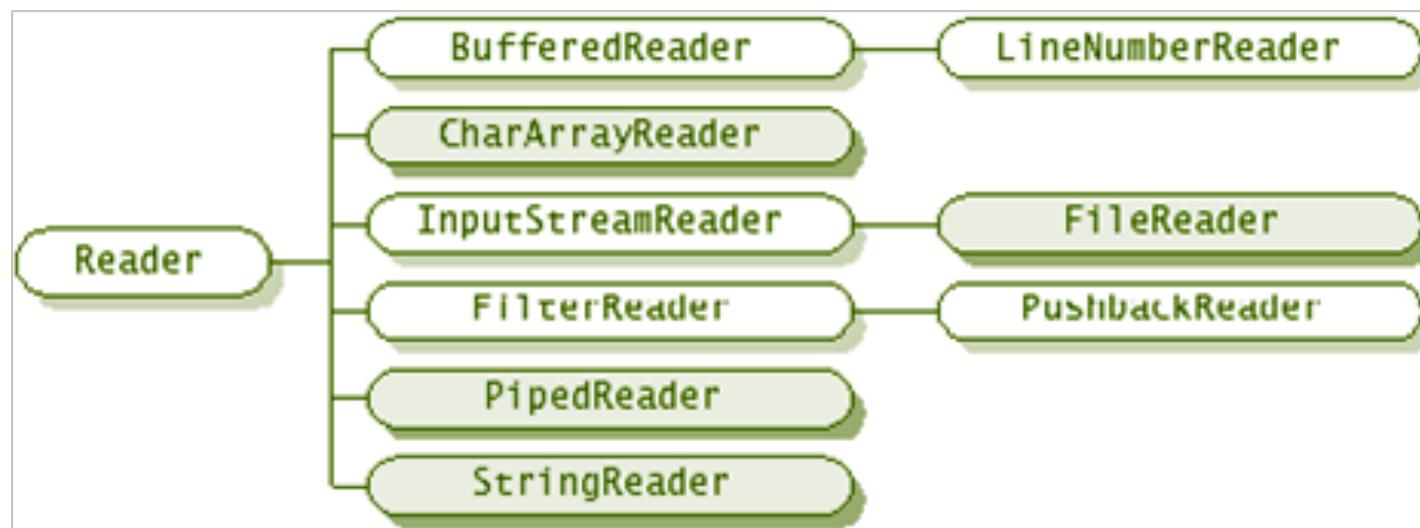
Jerarquía de clases para lectura de datos



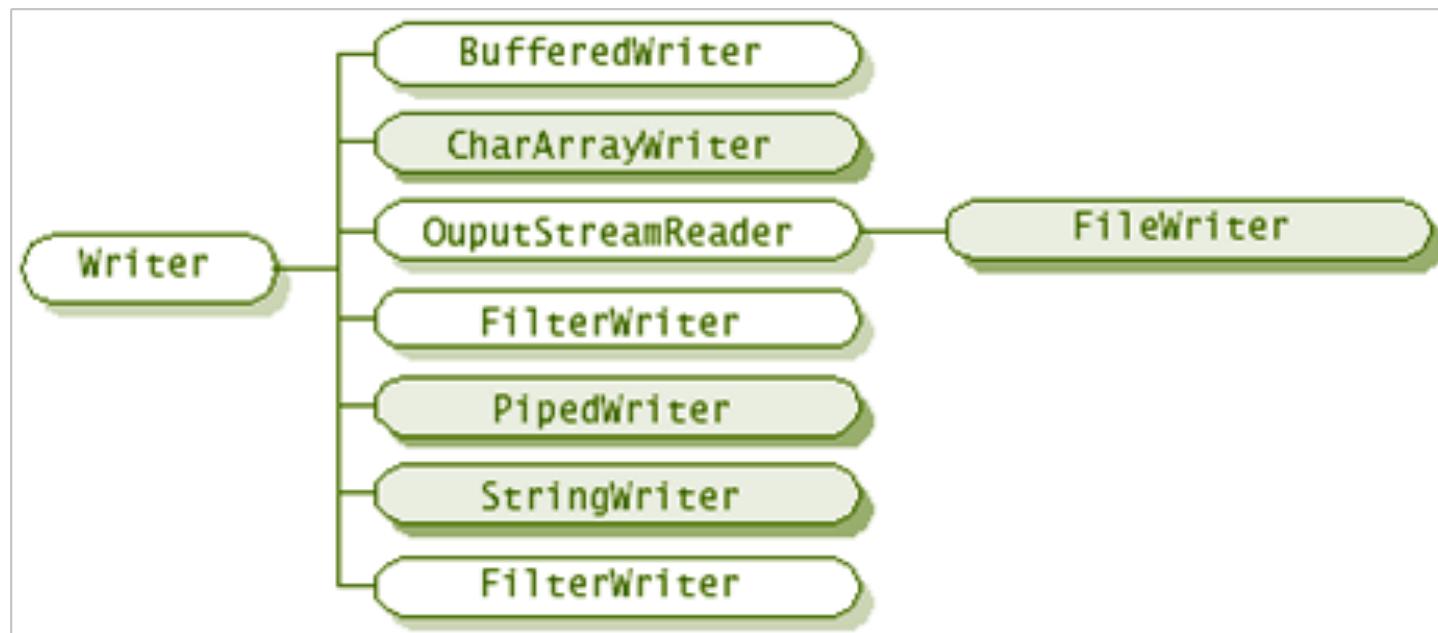
Jerarquía de clases para escritura de datos



Jerarquía de clases para lectura de datos



Jerarquía de clases para escritura de datos



2.2.7 Scanner

La clase Scanner permite leer de la entrada estándar. Pertenece al paquete `java.util`.

Los métodos principales de esta clase son `next()` y `hasNext()`. El método `next()` obtiene el siguiente elemento del flujo de datos. El método `hasNext()` verifica si el flujo de datos todavía posee elementos, en caso afirmativo regresa `true`, de lo contrario regresa `false`.

El delimitador de la clase scanner, por defecto, es el espacio en blanco. Es posible cambiar el delimitador utilizando el método:

`useDelimiter(String cadena);`



Ejemplo 2.8

```
import java.util.Scanner;

public class EjScanner {
    public static void main(String [] args) {
        try {
            String cad = "";
            Scanner s = new Scanner(System.in);
            System.out.println( s.nextLine() );
            s.close();
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```



2.2.8 Consola

La clase *Console* permite recibir datos de la línea de comandos (entrada estándar). Se encuentra dentro del paquete *java.io*.

Entre los métodos importantes que posee la clase *Console* están:

- *readLine()*: lee una cadena de caracteres hasta que encuentra el salto de línea (enter).
- *readPassword()*: lee una cadena de caracteres hasta que encuentra el salto de línea (enter), ocultando los caracteres.



Ejemplo 2.9

```
import java.io.Console;

public class EjConsole {
    public static void main(String [] args){
        Console con = System.console();
        System.out.print("Usuario: ");
        String s = con.readLine();
        System.out.println(s);
        System.out.print("Contraseña: ");
        char [] s2 = con.readPassword();
        System.out.println(s2);
    }
}
```



Clases básicas del paquete java.io

Clase	Hereda de	Argumentos de los constructores claves	Métodos claves
File	Object	File, String String String, String	createNewFile() delete() exists() isDirectory() list() mkdir() renameTo()
FileWriter	Writer	File String	close() flush() write()
BufferedWriter	Writer	Writer	close() flush() newLine() write()

Clases básicas del paquete java.io

Clase	Hereda de	Argumentos de los constructores claves	Métodos claves
PrintWriter	Writer	File String OutputStream Writer	close() flush() format() print() printf() println() write()
FileReader	Reader	File String	read()
BufferedReader	Reader	Reader	read() readLine()

2.3 Serialización

La serialización consiste en transformar un objeto en una cadena de bytes y almacenar esta última. La clase solo debería tener un método que serializa el objeto, mientras que el almacenamiento quedaría en la capa de acceso a datos. Además, es necesario proveer también la función inversa a la serialización, a menudo denominada "hidratación".

Esto es particularmente útil cuando se necesita guardar el estado de un objeto, es decir, cada variable de instancia de el o los objetos que se deseé.



Java ofrece la serialización en forma estándar y esta se logra mediante la interfaz Serializable del paquete java.io, de modo que toda clase que implemente la interfaz podrá transformar sus objetos a cadenas de bytes y viceversa.

Para serializar un dato es necesario utilizar en la clase los objetos ObjectOutputStream y ObjectInputStream para escribir los objetos (writeObject) al disco o leer objetos (readObject) del disco.



Ejemplo 2.10

Para crear una clase que permita serializar es necesario implementar la interfaz Serializable en dicha clase:

```
import java.io.*;  
class Gato implements Serializable {  
}
```

Ahora, en la clase en la que se deseé serializar, es necesario tener una instanciar de la clase que puede serializar (en este caso la clase Gato):

```
public class SerializaGato {  
    public static void main(String[] args) {  
        Gato g = new Gato();
```



Ejemplo 2.10

El siguiente paso sería serializar el objeto Gato invocando el método `writeObject()`. Una vez serializado el objeto, se almacena dentro de un archivo, en este caso, con el nombre `testSer.ser`

```
try {  
    FileOutputStream fs = new FileOutputStream("testSer.ser");  
    ObjectOutputStream os = new ObjectOutputStream(fs);  
    os.writeObject(g);  
    os.close();  
} catch (Exception e) {  
    e.printStackTrace();  
}
```



Ejemplo 2.10

Para recuperar la información se tiene que abrir un flujo de datos desde el archivo donde se almacenó el objeto e invocar al método `readObject()` de la clase Gato:

```
try {  
    FileInputStream fis = new FileInputStream("testSer.ser");  
    ObjectInputStream ois = new ObjectInputStream(fis);  
    g = (Gato) ois.readObject();  
    ois.close();  
} catch (Exception e) {  
    e.printStackTrace();  
}  
}
```



Ejemplo 2.11

Cuando los datos que se quieren guardar incluyen la referencia a otra clase que no está serializada, Java no puede guardar estas referencias. Por ejemplo, se tiene la clase Perro:

```
import java.io.Serializable;
class Perro implements Serializable{
    private Collar elCollar;
    private int tamCollar;

    public Perro(Collar collar, int tam) {
        elCollar = collar;
        tamCollar = tam;
    }

    public Collar getCollar() {
        return elCollar;
    }
}
```



Ejemplo 2.11

Como se pudo observar, la clase Perro es serializable y hace referencia a la clase Collar:

```
class Collar {  
    private int tamCollar;  
  
    public Collar(int tam) {  
        tamCollar = tam;  
    }  
  
    public int getCollarSize() {  
        return tamCollar;  
    }  
}
```

Empero, la clase Collar no es serializable.



Ejemplo 2.11

```
// Se serializa la clase Perro:  
import java.io.*;  
Public class SerializarPerro {  
    public static void main(String[] args) {  
        Collar c = new Collar(3);  
        Perro p = new Perro(c, 8);  
        System.out.println("Guardar: tamaNo del collar "  
                           + p.getCollar().getCollarSize());  
        try {  
            FileOutputStream fs = new FileOutputStream("testSer.ser");  
            ObjectOutputStream os = new ObjectOutputStream(fs);  
            os.writeObject(p);  
            os.close();  
        } catch (Exception e) {  
            System.out.println("Error al serializar");  
            e.printStackTrace();  
        }  
    }  
}
```

Ejemplo 2.11

```
try {
    FileInputStream fis = new FileInputStream("testSer.ser");
    ObjectInputStream ois = new ObjectInputStream(fis);
    p = (Perro) ois.readObject();
    ois.close();
    System.out.println("Dato recuperado: tamaNo del collar "
        + p.getCollar().getCollarSize());
} catch (Exception e) {
    System.out.println("Error al hidratar");
    e.printStackTrace();
}
```



Ejemplo 2.11

Cuando se intenta serializar la clase Perro, como ésta tiene una referencia a la clase Collar, se intenta también serializar Collar, sin embargo, como la clase Collar no es serializable se genera un error al ejecutar el programa:

```
java SerializarPerro
```

Guardar: tamaNo del collar 3

Error al serializar

java.io.NotSerializableException: Collar

Error al hidratar

java.io.WriteAbortedException: writing

aborted;

java.io.NotSerializableException: Collar



Ejemplo 2.11

Para poder ejecutar el programa anterior es necesario hacer serializable la clase Collar:

```
import java.io.Serializable;
class Collar implements
Serializable{
    private int tamCollar;

    public Collar(int tam) {
        tamCollar = tam;
    }

    public int getCollarSize() {
        return tamCollar;
    }
}
```



Ejemplo 2.11

La salida del programa corregido es:

```
java SerializarPerro
Guardar: tamaNo del collar 3
Dato recuperado: tamaNo del collar 3
```



Modificador transient

En la práctica, se puede presentar la situación de que no se posea acceso a alguna de las clases que se deseen serializar. Para estas situaciones java proporciona el modificador *transient* que evita que la referencia de una clase no serializable se intente serializar.

Por ejemplo, para evitar que se intente serializar la clase Collar, al momento de crear la referencia se agrega la palabra reservada *transient* de la siguiente manera:

private transient Collar el collar;



Ejemplo 2.12

```
import java.io.Serializable;

class Perro implements Serializable{
    private transient Collar elCollar;
    private int tamCollar;

    public Perro(Collar collar, int tam) {
        elCollar = collar;
        tamCollar = tam;
    }

    public Collar getCollar() {
        return elCollar;
    }
}
```



Ejemplo 2.12

```
class Collar {  
    private int tamCollar;  
  
    public Collar(int tam) {  
        tamCollar = tam;  
    }  
  
    public int getCollarSize() {  
        return tamCollar;  
    }  
}
```



Cuando se ejecuta el programa anterior se obtiene la siguiente salida:

```
java SerializarPerro
Guardar: tamaNo del collar 3
Error al hidratar
java.lang.NullPointerException at
  SerializarPerro.main(SerializarPerro.java:22)
```

Se puede observar que el objeto se serializa de manera correcta, pero, cuando se intenta recuperar, se presenta una excepción.



Para solventar el error anterior se deben implementar los métodos *writeObject* y *readObject* dentro de la clase que instancia una clase no serializable, en este caso dentro de la clase Perro.

```
private void writeObject(ObjectOutputStream os) {  
    // Código para guardar el objeto Collar  
}
```

```
private void readObject(ObjectInputStream os) {  
    // Código para leer el objeto Collar  
}
```



Ejemplo 2.12

El código del método para guardar el elemento Collar es el siguiente:

```
private void writeObject(ObjectOutputStream os) {  
    try {  
        os.defaultWriteObject();  
        os.writeInt(elCollar.getCollarSize());  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Se invoca el método **defaultWriteObject()**, para que realizar la serialización del objeto Collar de forma natural. Después se obtiene el tamaño del collar para guardarlo.



El código del método para leer el elemento Collar es:

```
private void readObject(ObjectInputStream is) {  
    try {  
        is.defaultReadObject();  
        elCollar = new Collar(is.readInt());  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

Para deserializar (hidratar) el objeto se crea una nueva referencia hacia la clase Collar asignandole el valor guardado del mismo.



Serialización en clases derivadas (herencia)

Para el caso de herencia, si la superclase es serializable entonces todas las clases derivadas también son serializables.

Si, por el contrario, una subclase se establece como serializable, la superclase regresará a su estado inicial por defecto, debido a que el constructor de la super clase se vuelve a ejecutar.



Dada la siguiente jerarquía de clases:

```
class Perro2 extends Animal implements Serializable
{
    String nombre;
    Perro2(int w, String n) {
        peso = w; // heredado
        nombre = n; // no heredado
    }
}

class Animal { // Clase no serializable
    int peso = 42;
}
```



Ejemplo 2.13

Si se intenta serializar y después, recuperar la instancia de Perro2:

```
import java.io.*;
class SuperNoSerial {
    public static void main(String [] args) {
        Perro2 p = new Perro2(35, "Fido");
        System.out.println("Antes: " + p.nombre + " " + p.peso);
        try {
            FileOutputStream fs =
                new FileOutputStream("testSer.ser");
            ObjectOutputStream os =
                new ObjectOutputStream(fs);
            os.writeObject(p);
            os.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Ejemplo 2.13

```
try {  
    FileInputStream fis = new FileInputStream("testSer.ser");  
    ObjectInputStream ois = new ObjectInputStream(fis);  
    p = (Perro2) ois.readObject();  
    ois.close();  
    System.out.println("after: " + p.nombre + " " + p.peso);  
} catch (Exception e) {  
    e.printStackTrace();  
}  
}  
}
```



Se obtendría la siguiente salida:

```
java  
SuperNoSerial  
Antes: Fido 35  
after: Fido 42
```

Como se puede observar, la serialización no genera ningún problema, sin embargo, al recuperar los datos, el valor que se hereda de la clase Animal regresa a su valor inicial.

Para finalizar, debido a que una variable estática le pertenece a la clase y no al objeto (instancia), las variables estáticas no se pueden serializar.



Ejercicio 2.1

Mejorar la serialización / deserialización de la clase de la clase Perro2 de tal manera que los atributos que hereda de la clase Animal se recuperen de manera íntegra.



3

Hilos



3. Hilos

- 3.1 Clase Thread**
- 3.2 Interfaz Runnable**
- 3.3 Clase ThreadGroup**
- 3.4 Ciclo de vida del hilo**
- 3.5 Planificador**
- 3.6 Prioridad**
- 3.7 Métodos de la clase Thread**
 - 3.7.1 Método sleep**
 - 3.7.2 Método yield**
 - 3.7.3 Método join**
- 3.8 Sincronización**
- 3.9 Bloqueos**
- 3.10 Clase object**



Hilos

Un hilo es un único flujo de ejecución dentro de un proceso. Un proceso es un programa en ejecución dentro de su propio espacio de direcciones.

Por lo tanto, un hilo es una secuencia de código en ejecución dentro del contexto de un proceso, esto se presenta debido a que los hilos no pueden ejecutarse solos y requieren la supervisión de un proceso.



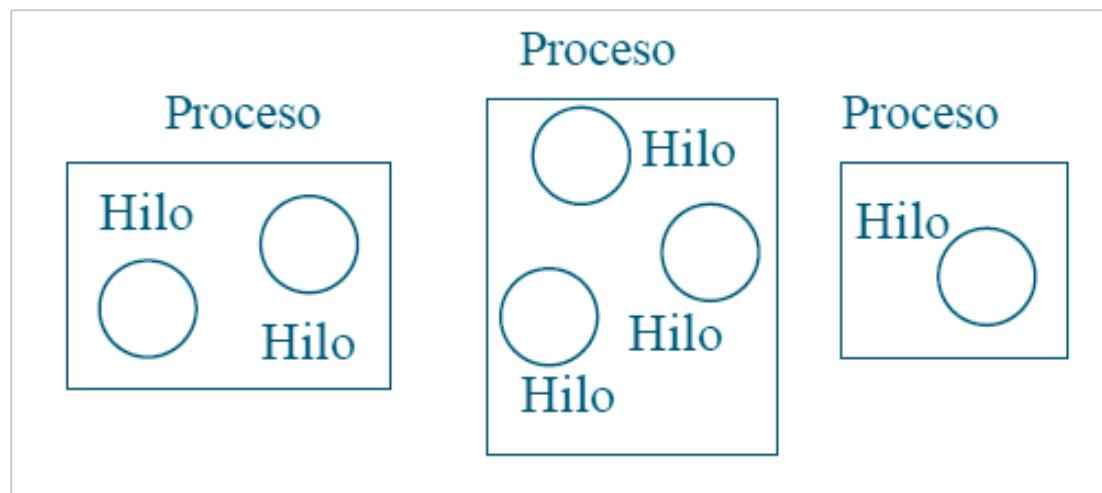
La Máquina Virtual Java (JVM) es capaz de manejar multihilos, es decir, es posible crear varios flujos de ejecución de manera simultánea.

La JVM gestiona detalles como asignación de tiempos de ejecución o prioridades de los hilos, de forma similar a como gestiona un Sistema Operativo múltiples procesos.

La diferencia básica entre un proceso del Sistema Operativo y un hilo de Java es que estos últimos se ejecutan dentro de la JVM, que es, a su vez, un proceso del Sistema Operativo y, por tanto, los hilos que se ejecutan dentro de la máquina virtual comparten todos los recursos asignados a la JVM (memoria, variables y objetos).



A los procesos donde se comparte los recursos se les llama procesos ligeros (lightweight process). La siguiente figura muestra la relación entre hilos y procesos, es decir, el Sistema Operativo ejecuta varios procesos y estos procesos a su vez ejecutan varios hilos.



Java proporciona soporte para hilos a través de una interfaz y un conjunto de clases:

- Thread
- Runnable (interfaz)
- ThreadGroup
- Object

Todas las clases mencionadas, así como la interfaz, son parte del paquete Java.lang.



3.1 Clase Thread

Es la clase responsable de producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase solo se hereda de esta clase.

El método run está definido en la clase Thread y determina el comportamiento de un hilo y, por lo tanto, se conoce como el cuerpo del hilo.

La clase Thread también define los métodos start y stop, los cuales permiten iniciar y detener la ejecución del hilo, entre otros métodos útiles.



Por lo tanto, para añadir la funcionalidad deseada a cada hilo creado es necesario sobre-escribir el método *run()*.

El método *run()* es invocado cuando se inicia el hilo (mediante una llamada al método *start()*) y el hilo termina cuando este método llega a su fin.

Para crear un hilo solo se tiene que crear una intancia de la clase que hereda de Thread. La clase Thread se encuentra dentro del paquete *java.lang*.



Ejemplo 3.1

```
public class EjConThread extends Thread {  
    public EjConThread(String nombre) {  
        super(nombre);  
    }  
    public void run() {  
        for(int i = 0 ; i < 5 ; i++) {  
            System.out.println("IteraciOn " + (i+1) + " de " +  
getName());  
        }  
        System.out.println("Termina el " + getName());  
    }  
    public static void main(String[] args) {  
        new EjConThread("Primer hilo").start();  
        new EjConThread("Segundo Hilo").start();  
        System.out.println("Termina el hilo principal");  
    }  
}
```

Del ejemplo anterior podemos observar que el método run contiene el bloque de ejecución del hilo. El método main crea dos objetos del tipo EjConThread mandando llamar al método start (en cada caso). El método start inicia un nuevo hilo y manda llamar al método run.

Se observa que la ejecución de los tres hilos (el método principal y los hilos generados) es asíncrona.



3.2 Interfaz Runnable

La interfaz *Runnable* permite producir hilos funcionales para otras clases. Para añadir la funcionalidad de hilo a una clase por medio de *Runnable*, solo es necesario implementar la interfaz.

La interface *Runnable* proporciona un método alternativo al uso de la clase *Thread*, para aquellos casos en los que no sea posible hacer que la clase definida herede de la clase *Thread*, es decir, cuando la clase definida hereda de alguna otra clase.



Debido a que no existe herencia múltiple, una clase derivada no podría hacer uso de la clase Thread. En este caso, la clase debe implementar la interface Runnable.

Las clases que implementan la interfaz *Runnable* proporcionan un método run que es ejecutado por un objeto hilo asociado que es creado aparte. Esta es una herramienta muy útil y, a menudo, es la única salida que tenemos para incorporar multihilos dentro de las clases.



Ejemplo 3.2

```
public class EjConRunnable implements Runnable {  
  
    public void run() {  
        for(int i = 0 ; i < 5 ; i++) {  
            System.out.println("IteraciOn " + (i+1) + " de " +  
                Thread.currentThread().getName());  
        }  
        System.out.println("Termina el " +  
            Thread.currentThread().getName());  
    }  
  
    public static void main(String[] args) {  
        new Thread(new EjConRunnable(), "Primer hilo").start();  
        new Thread(new EjConRunnable(), "Segundo  
Hilo").start();  
        System.out.println("Termina el hilo principal");  
    }  
}
```



Cuando se utiliza la interfaz *Runnable* los hilos se instancian de manera distinta. Primero se crea un objeto de la clase que implementa *Runnable* y después se crea una instancia de la clase *Thread*, pasando como parámetros el objeto creado y el nombre que tendrá el hilo.

Igual que con la clase *Thread*, para ejecutar la acción del hilo es necesario llamar al método *start* de la clase *Thread* y, a su vez, *start* mandará llamar al método *run()* del objeto enviado como parámetro.



Por último, en el método run para obtener el nombre del hilo que se está ejecutando se hace una llamada al método estático *currentThread()*, que devuelve el hilo que se está ejecutando y, a su vez, se invoca el método *getName()* que obtiene el nombre del hilo en ejecución.

Es importante aclarar que una vez que se ha iniciado el hilo (mediante la llamada al método *start()*) ya no se puede volver a iniciar, es decir, ya no es posible mandar llamar, otra vez, al método *start()*. Si se intentase hacer esto se obtendría una excepción del tipo *IllegalThreadStateException*.



3.3 Clase ThreadGroup

ThreadGroup se utiliza para manejar un grupo de hilos de manera unida (en conjunto). Esto proporciona una manera de controlar de modo eficiente la ejecución de una serie de hilos.

Esta clase proporciona métodos como stop, suspend y resume para controlar la ejecución del grupo (todos los hilos del grupo).

Los hilos de un grupo pueden, a su vez, contener otros grupos de hilos permitiendo una jerarquía anidada de hilos. Los hilos individuales tienen acceso al grupo pero no al padre del grupo.



```
public class EjThreadGroup {  
  
    public static void listarHilos(ThreadGroup grupoActual) {  
        int numHilos;  
        Thread [] listaDeHilos;  
  
        numHilos = grupoActual.activeCount();  
        listaDeHilos = new Thread[numHilos];  
        grupoActual.enumerate(listaDeHilos);  
        System.out.println("Numero de hilos = " + numHilos);  
        for (int i = 0 ; i < numHilos ; i++) {  
            System.out.println("Hilo " + (i+1) + " = " +  
listaDeHilos[i].getName());  
        }  
    }  
}
```

Ejemplo 3.3

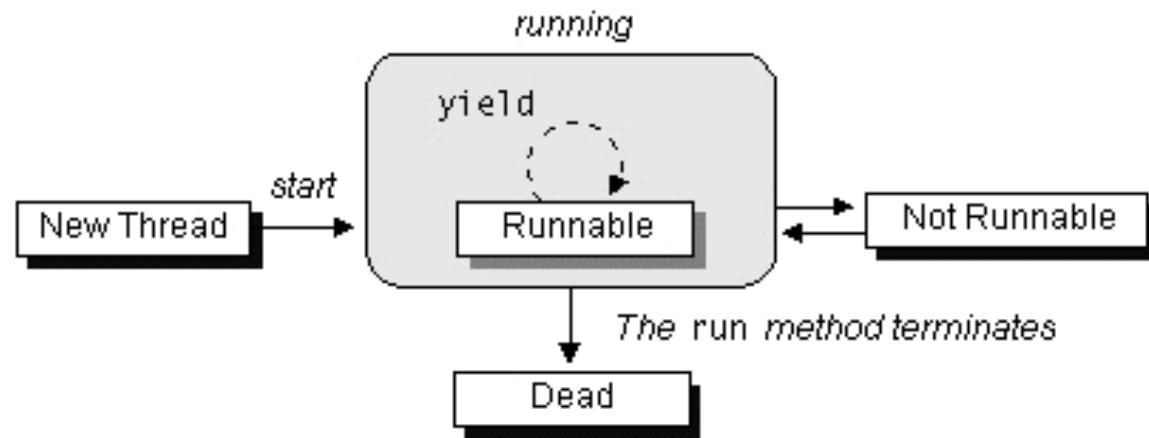
```
public static void main(String[] args) {  
    ThreadGroup grupoHilos =  
        new ThreadGroup("Grupo con prioridad normal");  
    Thread hilo1 = new Thread(grupoHilos, "Hilo 1 con prioridad  
mAxima");  
    Thread hilo2 = new Thread(grupoHilos, "Hilo 2 con prioridad  
normal");  
    Thread hilo3 = new Thread(grupoHilos, "Hilo 3 con prioridad  
normal");  
    Thread hilo4 = new Thread(grupoHilos, "Hilo 4 con prioridad  
normal");  
    Thread hilo5 = new Thread(grupoHilos, "Hilo 5 con prioridad  
normal");  
    hilo1.setPriority(Thread.MAX_PRIORITY);  
    grupoHilos.setMaxPriority(Thread.NORM_PRIORITY);
```



Ejemplo 3.3

```
System.out.println("Máxima prioridad del grupo = " +  
    grupoHilos.getMaxPriority());  
    System.out.println("Prioridad del Thread = " +  
hilo1.getPriority());  
    System.out.println("Prioridad del Thread = " +  
hilo2.getPriority());  
    System.out.println("Prioridad del Thread = " +  
hilo3.getPriority());  
    System.out.println("Prioridad del Thread = " +  
hilo4.getPriority());  
    System.out.println("Prioridad del Thread = " +  
hilo5.getPriority());  
  
    hilo1.start();  
    hilo2.start();  
    hilo3.start();  
    hilo4.start();  
    hilo5.start();  
  
    listarHilos(grupoHilos);  
}  
}
```

3.4 Ciclo de vida del hilo



En la figura anterior se puede observar las etapas de un hilo. Del diagrama es claro que el campo de acción de un hilo lo compone la etapa **Runnable**, es decir, cuando se está ejecutando (corriendo) el proceso ligero.



- Estado new

Un hilo esta en el estado new la primera vez que se crea y hasta que el método start es llamado. Los hilos en estado new ya han sido inicializados y están listos para empezar a trabajar, pero aún no han sido notificados para que empiecen a realizar su trabajo.



- **Estado runnable**

Cuando se llama al método start de un hilo nuevo, el método run es invocado y el hilo entra en el estado runnable. Cuando el hilo entra en este estado se dice que está en ejecución.

Es importante tener en cuenta que los hilos manejan diferentes prioridades y, dependiendo de ellas, el hilo se ejecutará o estará en espera hasta que se libere un recurso.



- **Estados running y not running**

Cuando un hilo invoca al método start() entra en el estado runnable y se dice que está corriendo (estado running).

Cuando se detiene la ejecución de un hilo por alguna razón se dice que el hilo está en estado not running. Cuando un hilo está en este estado, se encuentra listo para ser usado y es capaz de volver al estado runnable en cualquier momento dado.

Los hilos pueden pasar al estado not running por diferentes métodos (suspend, sleep y wait) o por algún bloqueo de I/O.



Dependiendo de la manera en que el hilo pasó al estado not running, se puede regresar al estado runnable:

- Cuando un hilo está suspendido, se invoca al método resume.
- Cuando un hilo está durmiendo, se mantendrá así el número de milisegundo especificado.
- Cuando un hilo está en espera, se activará cuando se haga una llamada a los métodos notify o notifyAll.
- Cuando un hilo está bloqueado por I/O, regresará al estado runnable cuando la operación I/O sea completada.



- **Estado dead**

Un hilo entra en estado dead cuando ya no es un objeto necesario. Los hilos en estado dead no pueden ser resucitados, es decir, ejecutados de nuevo.

Un hilo puede entrar en estado dead por dos causas:

- **El método run termina su ejecución.**
- **Se realiza una llamada al método stop.**



3.5 Planificador

Java tiene un Planificador (Scheduler) que controla todos los hilos que se están ejecutando en todos sus programas y decide cuáles deben ejecutarse (por tener una prioridad más alta) y cuáles deben encontrarse preparados para su ejecución (en estado wait o suspend).

La decisión de qué hilos se ejecutan primero y cuales tienen que esperar la realiza java con base en la prioridad que tiene asignado cada hilo.



3.6 Prioridad

Cada hilo tiene una prioridad, que no es otra cosa que un valor entero entre 1 y 10, de modo que cuanto mayor el valor, mayor es la prioridad.

El planificador determina que hilo debe ejecutarse en función de la prioridad asignada a cada uno de ellos. Cuando se crea un hilo en Java, éste hereda la prioridad de su padre. Una vez creado el hilo es posible modificar su prioridad en cualquier momento utilizando el método `setPriority`.



Las prioridades de un hilo varían en un rango de enteros comprendido entre MIN_PRIORITY y MAX_PRIORITY (estas variables están definidas en la clase Thread). La prioridad por defecto es 5 (NORM_PRIORITY).

El planificador ejecuta primero el hilo de prioridad superior (en estado runnable) y sólo cuando éste se detiene, comienza la ejecución de en hilo de prioridad inferior. Si dos hilos tienen la misma prioridad, el programador elige entre ellos de manera alternativa (forma de competición).



El hilo seleccionado se ejecutará hasta que:

- Un hilo con prioridad mayor pase a ser “Ejecutable”.
- En sistemas que soportan tiempo-compartido, termina su tiempo.
- Abandone o termine su método run.

Luego, un segundo hilo puede ejecutarse, y así continuamente hasta que el intérprete abandone.

El algoritmo del sistema de ejecución de hilos que sigue Java es de tipo preventivo. Si en un momento dado un hilo que tiene una prioridad mayor a cualquier otro hilo que se está ejecutando pasa a ser “Ejecutable”, entonces el sistema elige a este nuevo hilo.

3.7 Métodos de la clase Thread

La clase Thread posee varios métodos importantes para manejar hilos:

- **public static void sleep(long miliseg) throws InterruptedException**
- **public static void yield()**
- **public final void join() throws InterruptedException**
- **public final void setPriority(int newPriority)**

Así mismo, un hilo puede estar en tres diferentes estados: dormido, esperando o bloqueado.



3.7.1 Método sleep

Este método permite dormir a un hilo por un tiempo (detiene su ejecución el tiempo deseado), una vez transcurrido el tiempo definido el hilo regresa al estado runnable.

sleep() es un método estático y, si algo llegara a fallar, lanza una excepción del tipo InterruptedException.



Ejemplo 3.4

```
class EjSleep implements Runnable {  
    public void run() {  
        for (int x = 1; x < 4; x++) {  
            System.out.println("Ejecutado por: "  
                + Thread.currentThread().getName());  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException ex) {}  
        }  
    }  
}
```



Ejemplo 3.4

```
public class MainSleep {  
    public static void main (String [] args) {  
        EjSleep es = new EjSleep();  
        Thread uno = new Thread(es);  
        uno.setName("Alfredo");  
        Thread dos = new Thread(es);  
        dos.setName("Lucy");  
        Thread tres = new Thread(es);  
        tres.setName("Rogelio");  
        uno.start();  
        dos.start();  
        tres.start();  
    }  
}
```



3.7.2 Método yield

El método estático `yield()` provoca que un hilo pase de estar ejecutándose (`running`) a estar ejecutable (`runnable`), esto con el fin de permitir que otros hilos con la misma prioridad tomen su turno.

Sin embargo, el funcionamiento de este método (al igual que de los hilos en general) no está garantizado, es decir, puede que después de que se establezca un hilo por medio del método `yield()` a su estado en ejecución(`runnable`), éste vuelva a ser elegido para ejecutarse.



No se recomienda el uso de yield debido a que no todas las JVM tienen implementado este método. Además, algunas máquinas virtuales no son capaces de reconocer los 10 valores asignados a la prioridad de los hilos, lo que causa un problema al momento de la ejecución



3.7.3 Método join

Este método hace que un hilo se una a otro, es decir, permite al hilo agregado "formarse en la cola de espera" del otro hilo. Por lo tanto, el hilo que se une va justo al final del otro y no se va a ejecutar hasta que el hilo al que está unido termine de ejecutarse.



A continuación se muestra el código que utiliza:

```
Thread t = new Thread();  
t.start();  
t.join();
```

En el código anterior crea un hilo t, lo inicia y le indica al método principal que se una a él, así, cuando t termine su ejecución, main podrá pasar al estado runnable de nuevo.



Ejemplo 3.5

```
public class Join extends Thread {  
    public void run() {  
        for (int i = 0; i < 8; i++)  
            System.out.println("Mi hilo");  
    }  
  
    public static void main(String [] args){  
        Join t = new Join();  
        t.start();  
        for (int i = 0; i < 8; i++){  
            System.out.println("Hilo de main");  
            if(i == 4){  
                try{  
                    t.join();  
                }catch(Exception e){}  
            }  
        }  
    }  
}
```

3.8 Sincronización

La sincronización es un mecanismo mediante el cual se puede controlar el acceso a datos por parte de los hilos. Este control de acceso se lleva a cabo haciendo que los métodos se sincronizan.

Los métodos sincronizables (synchronized) son aquellos a los que es imposible acceder si otro objeto está haciendo uso de ellos, es decir, dos objetos no pueden acceder a un método sincronizable al mismo tiempo.

Estos tipos de métodos son muy utilizados con hilos para evitar la pérdida de información o ambigüedades en la misma.



Los hilos se ejecutan como procesos paralelos (independientes). Cuando se ejecutan varios hilos de manera simultanea, estos pueden intentar acceder al mismo tiempo a un método, para, por ejemplo, obtener o modificar el valor de un atributo.

Como los hilos se ejecutan en paralelo, el acceso al recurso no es controlado y, por lo tanto, puede haber pérdida de información.



Por ejemplo, si se tiene un hilo que está guardando información en un arreglo por medio de un método y, al mismo tiempo, otro hilo ejecuta el mismo método y comienza a escribir en el mismo arreglo, esto generaría pérdida de datos y ambigüedad en la información para el primer hilo.



Estas complicaciones se pueden evitar bloqueando el acceso al método mientras algún proceso lo esté ejecutando. Lo anterior se puede lograr haciendo que el método en cuestión sea sincronizable (synchronizable).

Así, al estar sincronizados los métodos de una clase, si un recurso está ocupando un método, éste es inaccesible hasta que sea liberado, es decir, dos hilos no pueden acceder a un mismo método al mismo tiempo.



Ejemplo 3.6

```
public class Cuenta extends Thread {  
    private static long saldo = 0;  
    public Cuenta (String nombre){  
        super(nombre);  
    }  
  
    public void run() {  
        if (getName().equals("Deposito 1") ||  
            getName().equals("Deposito 2")) {  
            this.depositarDinero(100);  
        } else {  
            this.extraerDinero(50);  
        }  
        System.out.println("Termina el " + getName());  
    }  
}
```



Ejemplo 3.6

```
public synchronized void extraerDinero(int cantidad) {  
    try {  
        if (saldo <= 0){  
            System.out.println(getName() + " espera depOsito.  
            \nSaldo = " + saldo);  
            sleep(5000);  
        }  
    } catch (InterruptedException e) {  
        System.out.println(e);  
    }  
    saldo -= cantidad;  
    System.out.println(getName() + " extrajo " + cantidad + "  
    pesos.\nSaldo restante = "+ saldo);  
    notifyAll();  
}
```



Ejemplo 3.6

```
public synchronized void depositarDinero(int cantidad) {  
    saldo += cantidad;  
    System.out.println("Se depositaron " + cantidad + "  
pesos");  
    notifyAll();  
}  
  
public static void main(String[] args) {  
    new Cuenta("Acceso 1").start();  
    new Cuenta("Acceso 2").start();  
    new Cuenta("Deposito 1").start();  
    new Cuenta("Deposito 2").start();  
    System.out.println("Termina el hilo principal");  
}
```



3.9 Bloqueos

Todos los objetos que se crean en Java tienen una bandera asociada llamada lock (bloqueo). Cuando se utiliza la palabra synchronized se habilita en automático el uso de la bandera lock y proporciona un acceso exclusivo para el código donde se comparten datos, es decir, informa si el código está siendo utilizado o está disponible.

Cuando un hilo encuentra la palabra synchronized intenta acceder a la bandera lock antes de continuar la ejecución, debido a que si la bandera no está presente, no puede continuar su ejecución.



Una clase puede tener tanto métodos sincronizados, como no sincronizados. Es posible sincronizar tanto métodos como pequeños fragmentos de código.

```
public class PruebaSincro {  
    public void hacerAlgo() {  
        System.out.println("No sincronizado");  
        synchronized(this) {  
            System.out.println("Sincronizado");  
        }  
    }  
}
```



Como se sabe, los métodos estáticos le pertenecen a la clase, por lo tanto, si se quiere sincronizar un método estático se debe sincronizar la clase donde está definido, esto debido a que sólo hay una copia de los datos estáticos y esta copia le pertenece a la clase.

Toda clase cargada en Java tiene posee una instancia de java.lang.Class, por lo tanto, esa instancia es la que se debe proteger mediante la sincronización.



**Existen dos maneras de sincronizar un método estático:
sincronizando todo el método o parte del código.**

Para sincronizar todo el método estático únicamente es necesario hacer uso de la palabra reservada synchronized en la firma del método, es decir:

```
public static synchronized int mostrarSaldo()
{
    // Código a ejecutar
}
```



Para sincronizando parte del código se usa la palabra **synchronized** y, entre llaves, se encierra el método a sincronizar, pasando como parámetro la clase a la que pertenece el método:

```
public static int mostrarSaldo() {  
    synchronized(MiClase.class) {  
        // Código a ejecutar  
    }  
}  
  
public static int mostrarSaldo() {  
    Class cl = Class.forName("MiClase");  
    synchronized (cl) {  
        // Código a ejecutar  
    }  
}
```



3.10 Clase Object

Esta clase proporciona métodos cruciales dentro de la arquitectura multihilos de Java. Estos métodos son wait, notify y notifyAll.

El método wait hace que el hilo de ejecución espere en estado dormido (sleep) un tiempo indefinido.

El método notify informa a un hilo que está en espera (estado wait) que puede continuar con su ejecución.



El método notifyAll es similar a notify excepto que se aplica a todos los hilos en espera.

Los métodos anteriores (wait, notify y notifyAll) solo pueden ser llamados desde un método o bloque sincronizado (o bloque de sincronización).



Ejemplo 3.7

```
// Ejemplo de notify

class HiloA {
    public static void main(String [] args) {
        HiloB b = new HiloB();
        b.start();
        synchronized(b) {
            try {
                System.out.println("Esperando a b");
                b.wait();
            } catch (InterruptedException e) {}
            System.out.println("Total: " + b.total);
        }
    }
}
```

Ejemplo 3.7

```
class HiloB extends Thread {  
    int total;  
    public void run() {  
        synchronized(this) {  
            System.out.println("Inicia B...");  
            for(int i=0;i<100;i++) {  
                total += i;  
            }  
            System.out.println("Se notifica B terminO.");  
            notify();  
        }  
    }  
}
```



Ejemplo 3.8

```
// Ejemplo de notifyAll

class Leer extends Thread {
    Calcular c;
    public Leer(Calcular calc) {
        c = calc;
    }
    public void run() {
        synchronized(c) {
            try {
                System.out.println("Esperando...");
                c.wait();
            } catch (InterruptedException e) {}
            System.out.println("Total: " + c.total);
        }
    }
}
```

Ejemplo 3.8

```
public static void main(String [] args) {  
    Calcular calcular = new Calcular();  
    new Leer(calcular).start();  
    new Leer(calcular).start();  
    new Leer(calcular).start();  
    calcular.start();  
}  
}
```



Ejemplo 3.8

```
class Calcular extends Thread {  
    int total;  
    public void run() {  
        synchronized(this) {  
            for(int i=0;i<100;i++) {  
                total += i;  
            }  
            notifyAll();  
        }  
    }  
}
```



4

Clases internas



4. Clases internas

- 4.1 Clase interna estática**
- 4.2 Clase interna miembro**
- 4.3 Clase interna local**
- 4.4 Clase interna anónima**



Clases internas

Una clase puede ser etiquetada como privada o protegida únicamente si es un miembro anidado de otra clase, es decir, una clase interna.

La clase interna es un miembro más de la clase que la declara (clase externa) y, por tanto, puede tener el modificador de acceso privado.

Una clase externa solo se puede crear si posee un modificador de acceso publico (public) o si no posee modificador.



Una clase interna (inner class) es un clase definida dentro de otra clase es decir, una clase que es, a su vez, miembro de otra clase. La sintaxis de este tipo de clases es la siguiente:

```
class Externa {  
    // Atributos  
    // Métodos  
    class Interna {  
        // Atributos  
        // Métodos  
    }  
}
```

Este estilo de programación se utiliza cuando las relaciones entre clases es muy estrecha.



Para estas clases, un objeto de la clase interna está siempre relacionado con un objeto de la clase externa, es decir, las instancias de la clase interna deben ser creadas a partir de instancias de la clase externa.

Una instancia de la clase interna tiene acceso a todos los datos miembro de la clase que lo contiene (clase externa) sin utilizar un calificador de acceso especial, es decir, como si le pertenecieran.



Las clases internas se clasifican en:

- **Clases anidadas de alto nivel (clases internas estáticas)**
- **Clases internas miembro**
- **Clases internas locales**
- **Clases internas anónimas**



4.1 Clase interna estática

Las clases internas estáticas son clases miembro de otra clase, es decir, están embebidas dentro de otra clase.

Para acceder a las clases internas estáticas no es necesario crear una referencia (instancia) de la clase externa, sólo hay que especificar donde se encuentra la clase interna estática, es decir, especificar la clase que la contiene.



Desde la clase estática interna se puede acceder, únicamente, a los miembros estáticos de la clase externa. Para acceder a los miembros no estáticos de la clase externa es necesario crear una referencia dentro de la clase interna hacia la clase externa.



Ejemplo 4.1

```
public class ClaseExterna {  
    static int i = 5;  
    void metodoClaseExterna() {  
        System.out.println("Dentro de metodo de la clase externa");  
    }  
  
    // Inicia la clase interna estática  
    public static class InternaEstatica {  
        public static void main(String args[]) {  
            System.out.println("Dentro de la clase interna " + i);  
            ClaseExterna cie = new ClaseExterna();  
            cie.metodoClaseExterna();  
        }  
    } // Fin de la clase interna estática  
} // Fin de la clase Externa
```

La clase anterior se ejecuta de la siguiente manera en Linux:

java ClaseExterna\$InternaEstatica

Y en Windows:

java ClaseExterna\$InternaEstatica



4.2 Clase interna miembro

Esta clase se define como un miembro (no estático) de la clase contenedora. Se pueden declarar, incluso, como privadas o protegidas.

A cada instancia de la clase externa (contenedora) se asocia una instancia de la clase interna miembro. La clase interna miembro tiene acceso a todos los atributos y métodos de la clase externa, incluyendo los privados.



Una clase interna miembro no puede tener miembros estáticos. Tampoco puede tener nombres comunes con la clase externa.

La única forma para acceder a la clase interna es creando una referencia a la clase externa.



Ejemplo 4.2

```
class ExternaMiembro {  
    private int x = 7;  
    public void crearInterna () {  
        InternaMiembro in = new InternaMiembro();  
        in.verExterna();  
    }  
  
    class InternaMiembro {  
        int x = 3;  
        public void verExterna() {  
            System.out.println("X externa: "  
                + ExternaMiembro.this.x);  
            System.out.println("X interna: " + x);  
            System.out.println("Clase interna: " + this);  
            System.out.println("Clase externa: "  
                + ExternaMiembro.this);  
        }  
    }  
}
```

Ejemplo 4.2

```
public static void main (String[] args) {  
    ExternaMiembro.InternaMiembro interna =  
        new ExternaMiembro().new InternaMiembro();  
    interna.verExterna();  
}
```



Ejemplo 4.3

```
public class ConjuntoObjetos {  
    Object arreglo[];  
    int cima = 0;  
    ConjuntoObjetos(int tam){  
        arreglo = new Object[tam];  
    }  
    public void setObject(Object ob){  
        arreglo[cima++] = ob;  
    }  
    public Object getObject(){  
        return arreglo[--cima];  
    }  
    public boolean vaciar(){  
        return cima == 0;  
    }  
}
```



Ejemplo 4.3

```
// clase interna miembro
class Enumerador implements java.util.Enumeration {
    int cont = cima;
    public boolean hasMoreElements() {
        return cont > 0;
    }
    public Object nextElement() {
        if (cont == 0) {
            throw new java.util.NoSuchElementException("Error");
        }
        return arreglo[--cont];
    }
} // Fin de la clase interna

public java.util.Enumeration elementos(){
    return new Enumerador();
}
```

Ejemplo 4.3

```
public class ConjuntoObjetosPrueba {  
    public static void main (String [] args){  
        ConjuntoObjetos pila = new ConjuntoObjetos(5);  
  
        for (int i = 0 ; i < pila.arreglo.length ; i++){  
            String txt = "Objeto " + i;  
            pila.setObject(txt);  
        }  
  
        java.util.Enumeration enumerador;  
        enumerador = pila.elementos();  
  
        while(enumerador.hasMoreElements()){  
            String txt = (String) enumerador.nextElement();  
            System.out.println(txt);  
        }  
    }  
}
```

Para crear una instancia de una clase interna miembro se utiliza la clase externa: InstanciaClaseExterna.newInstanceClaseInterna().

Si se hereda de una clase interna miembro y se quiere acceder a la clase base, es necesario hacer referencia a la instancia de la clase externa: InstanciaClaseExterna.super().



Ejemplo 4.4

```
public class ConjuntoObjetosPrueba2 {  
  
    public static void main (String [] args) {  
        ConjuntoObjetos pila = new ConjuntoObjetos(5);  
        for (int i = 0 ; i < pila.arreglo.length ; i++){  
            String txt = "Objeto " + i;  
            pila.setObject(txt);  
        }  
        ConjuntoObjetos.Enumerador enumerador = pila.new Enumerador();  
        while(enumerador.hasMoreElements()){  
            String txt = (String) enumerador.nextElement();  
            System.out.println(txt);  
        }  
    }  
}
```

Para las clases internas, la palabra reservada this hace referencia a sí misma.

Si desde una clase interna se quiere hacer referencia a la clase contenedora (clase externa) se realiza invocando el nombre de la clase externa seguido de la palabra reservada this: ClaseExterna.this.



Los modificadores que se pueden aplicar sobre las distintas clases internas son los siguientes:

- **final (No se puede heredar)**
- **abstract (clase abstracta)**
- **public (clase pública)**
- **private (miembro solo de la clase)**
- **protected (miembro de la clase y clases derivadas)**
- **static (clase estática)**



4.3 Clase interna local

Estas clases, también conocidas como clases internas de métodos locales, se definen dentro del bloque de código de un método, por lo tanto, solo se pueden utilizar dentro del código donde están declaradas.

Las clases internas locales pueden hacer uso de las variables locales y los parámetros del método declarados como final, únicamente. No utilizan ningún modificador de acceso. No pueden ser estáticas.



Estas clases no están disponibles de manera pública, por lo tanto son inaccesibles.

Los modificadores de acceso que se pueden aplicar a las clases internas locales son abstract y final, pero no al mismo tiempo.



Ejemplo 4.5

```
public class ClaseInternalLocal {  
    Object arreglo[];  
    int cima = 0;  
    ClaseInternalLocal(int tam){  
        arreglo = new Object[tam];  
    }  
    public void setObject(Object ob){  
        arreglo[cima++] = ob;  
    }  
    public Object getObject(){  
        return arreglo[--cima];  
    }  
    public boolean vaciar(){  
        return cima == 0;  
    }  
}
```



Ejemplo 4.5

```
// método con clase interna local
java.util.Enumeration enumeradorPropio(final Object objetos[]) {
    class ClaseLocal implements java.util.Enumeration {
        int cont = 0;
        public boolean hasMoreElements(){
            return cont < objetos.length;
        }
        public Object nextElement(){
            if (cont == objetos.length) {
                throw new java.util.NoSuchElementException("Error");
            }
            return objetos[cont++];
        }
    } // Fin de la clase Enumerador
    return new ClaseLocal();
} // Fin del método que implementa clase interna local
public java.util.Enumeration elementos(){
    return enumeradorPropio(arreglo);
}
}
```

Implementar el método principal para ejecutar la clase anterior.

```
public class ClaseInternaLocalPrueba {  
    public static void main (String [] args){  
        // Implementar clase principal para comprobar  
        // el funcionamiento de la ClaseInternaLocal  
    }  
}
```



4.4 Clase interna anónima

En esencia, las clases anónimas son clases internas locales sin nombre, por eso se les llama anónimas.

Se define al mismo tiempo que se instancia y, por tanto, sólo puede existir una instancia de una clase anónima.

Los constructores de estas clases deben ser sencillos (sin argumentos), para evitar anidar demasiado código en una sola línea.



Ejemplo 4.6

```
public class ClaseInternaAnonima {  
    Object arreglo[];  
    int cima = 0;  
    ClaseInternaAnonima(int tam){  
        arreglo = new Object[tam];  
    }  
    public void setObject(Object ob){  
        arreglo[cima++] = ob;  
    }  
    public Object getObject(){  
        return arreglo[--cima];  
    }  
    public boolean vaciar(){  
        return cima == 0;  
    }  
}
```



Ejemplo 4.6

```
// metodo con clase interna anonima
public java.util.Enumeration elementos() {
    return new java.util.Enumeration() {
        int cont = cima;
        public boolean hasMoreElements(){
            return cont > 0;
        }
        public Object nextElement(){
            if (cont == 0) {
                throw new
                java.util.NoSuchElementException("Error");
            }
            return arreglo[--cont];
        }
    }; // Fin de la clase (una sola linea de código)
} // Fin del metodo que implementa clase interna anonima
}
```



Implementar el método principal para ejecutar la clase anterior.

```
public class ClaseInternaAnonimaPrueba {  
    public static void main (String [] args){  
        // Implementar clase principal para comprobar  
        // el funcionamiento de la ClaseInternaLocal  
    }  
}
```



5

Colecciones



5. Colecciones

- 5.1 Conjunto**
- 5.2 Lista**
- 5.3 Mapa**
- 5.4 Interfaz Comparable**
- 5.5 Interfaz Comparator**

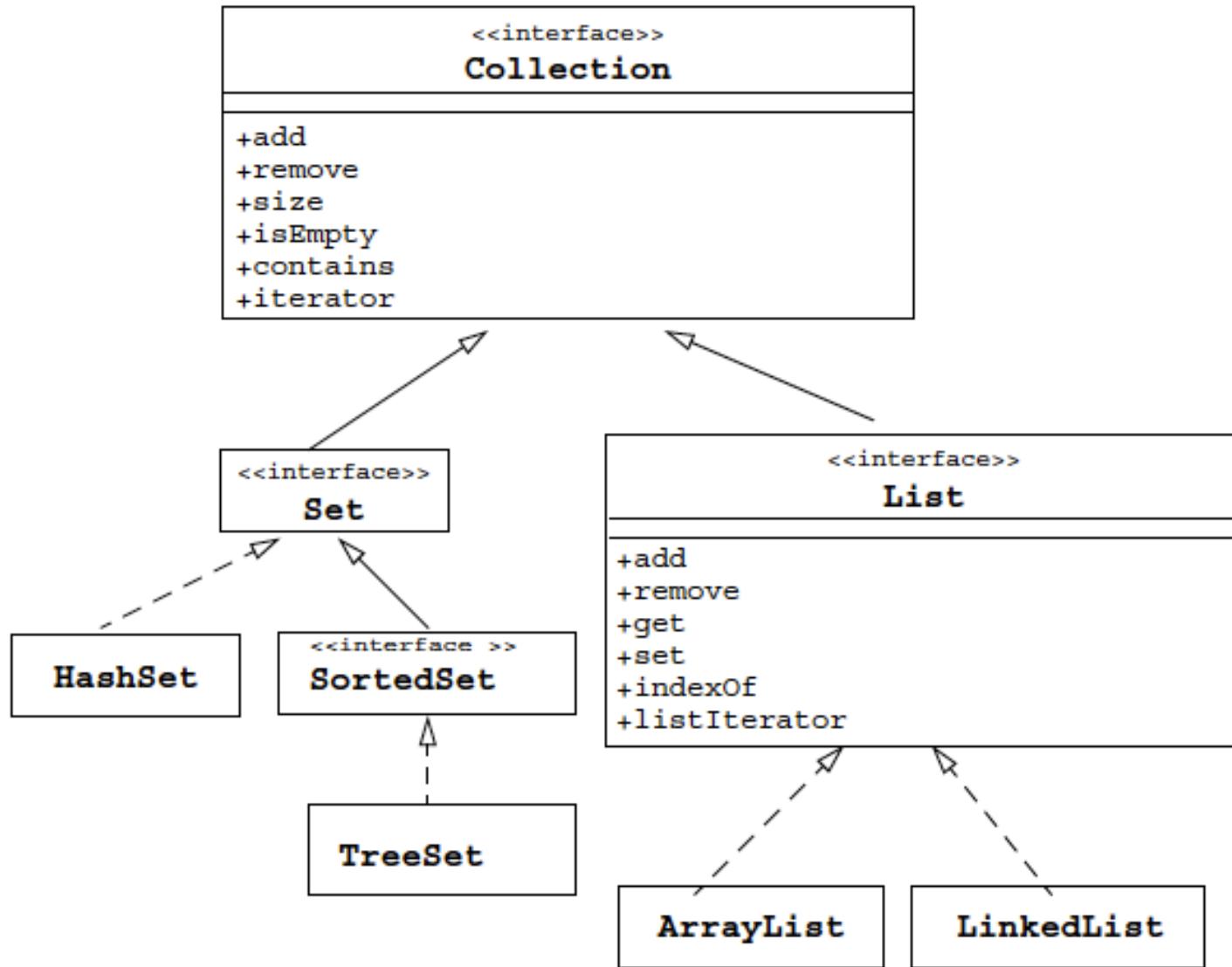


Colecciones

Una colección es un conjunto de datos, es decir, una referencia que gestiona un grupo de objetos, a este grupo de objetos se les conoce como elementos de la colección.



Jerarquía de clases de las colecciones



5.1 Conjunto

Un conjunto (Set) es una colección desordenada (no mantiene un orden de inserción) y no permite elementos duplicados.



Ejemplo 5.1

```
import java.util.*;  
  
public class Conjunto {  
    public static void main(String[] args) {  
        Set c = new HashSet();  
        c.add("uno");  
        c.add("segundo");  
        c.add("3ro");  
        c.add(new Integer(4));  
        c.add(new Float(5.0F));  
        c.add("segundo"); // duplicado  
        c.add(new Integer(4)); // duplicado  
        System.out.println(c);  
    }  
}
```



5.2 Lista

Una lista (List) es una colección ordenada (debido a que mantiene el orden de inserción) pero permite elementos duplicados.



Ejemplo 5.2

```
import java.util.*;  
  
public class Lista {  
    public static void main(String[] args) {  
        List l = new ArrayList();  
        l.add("uno");  
        l.add("segundo");  
        l.add("3ro");  
        l.add(new Integer(4));  
        l.add(new Float(5.0F));  
        l.add("segundo"); // duplicado  
        l.add(new Integer(4)); // duplicado  
        System.out.println(l);  
    }  
}
```



5.3 Mapa

Un mapa (también llamado arreglo asociativo) es un conjunto de elementos agrupados con una llave y un valor:

<llave, valor>

donde las llaves no pueden ser repetidas y a cada valor le corresponde una llave. La columna de valores sí puede repetir elementos.

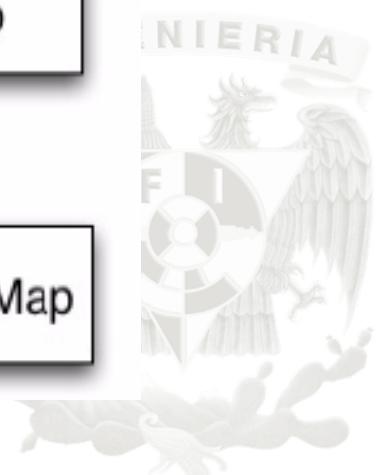
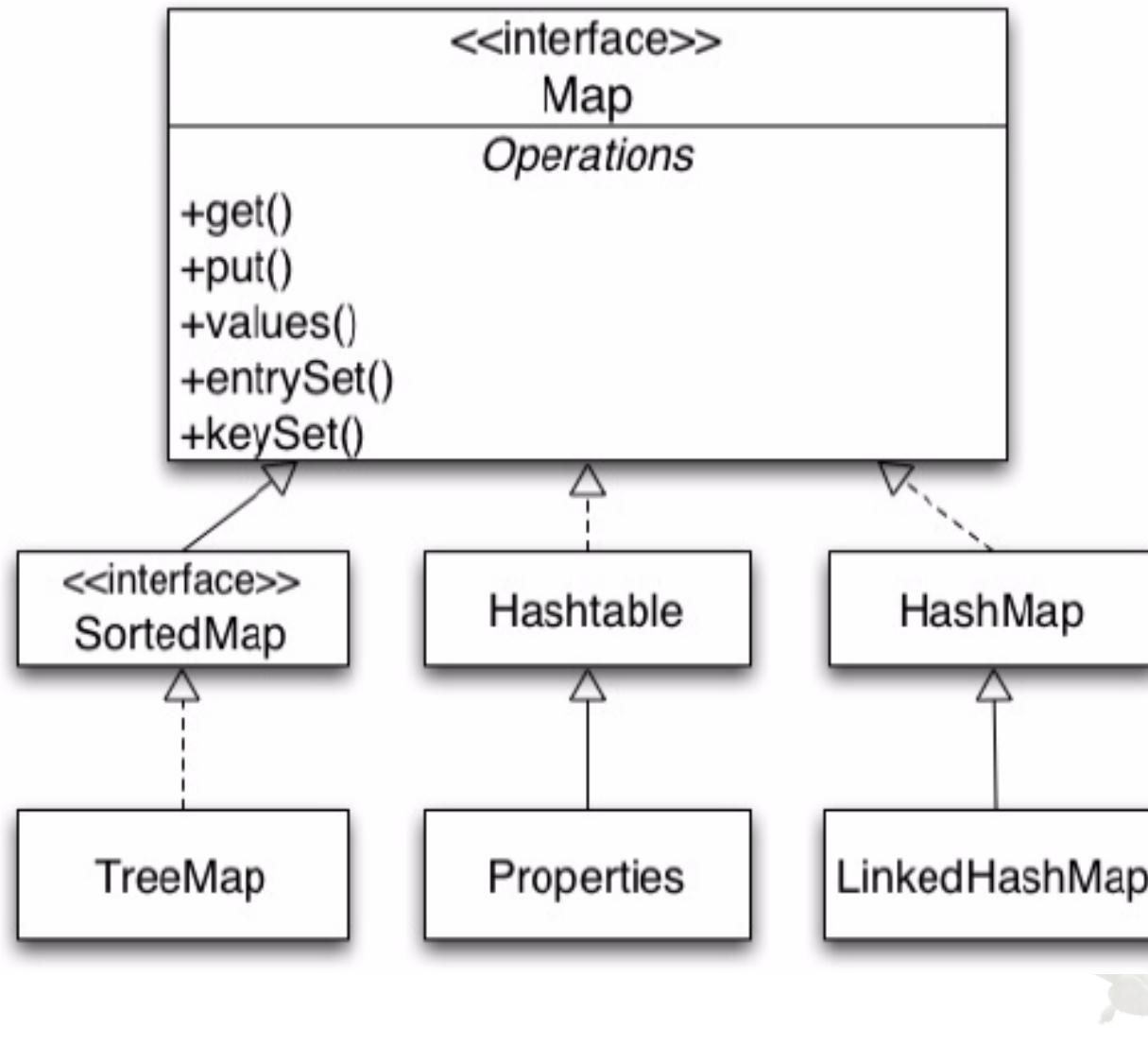


El contenido de un mapa (ya sea su valor o su llave) puede ser manipulado como un conjunto gracias a unos métodos que posee la interfaz Map:

- **entrySet():** regresa un conjunto con todos los pares <llave-valor>.
- **keySet():** regresa un conjunto con todas las llaves del mapa.
- **values():** regresa una colección con todos los valores en el mapa.



Jerarquía de clases de los mapas



Ejemplo 5.4

```
import java.util.*;
public class Mapa {
    public static void main(String [] args) {
        Map m = new HashMap();
        m.put("uno","1ro");
        m.put("segundo", new Integer(2));
        m.put("tercero","3rd");
        // Se sobre-escribe un valor
        m.put("tercero","III");
        // Regresa un conjunto con las llaves del mapa
        Set llaves = m.keySet();
        // Regresa una colección con los valores del mapa
        Collection valores = m.values();
        // Regresa un conjunto con los pares <llave, valor>
        Set pares = m.entrySet();
        System.out.println("Llaves: \n " + llaves);
        System.out.println("Valores: \n " + valores);
        System.out.println("Pares: \n " + pares);
    }
}
```

5.4 Interfaz *Comparable*

La intefaz *Comparable* se utiliza para ordenar colecciones. Provee un orden natural a las clases que la implementan. Es utilizada para ordenar.

Solo posee un método que se debe sobre-escribir para proveer a la clase un parámetro de comparación:

```
public int compareTo(Object ob)
```



Como se puede observar, el método `compareTo` regresa un valor entero, el cuál puede ser negativo, positivo o cero:

- Cero: significa que los elementos comparados son iguales.
- Positivo (> 0): significa que el objeto que compara (`this`) es más grande y, por tanto, se debe insertar después del objeto con el que se compara (`Object`).
- Negativo (< 0): significa que el objeto que compara (`this`) es menor y, por tanto, se debe insertar antes del objeto con el que se compara (`Object`).



Ejemplo 5.5

```
public class PersonaComparable implements Comparable {  
    String nombre;  
    int edad;  
  
    PersonaComparable (String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    public int compareTo(Object obj){  
        return this.nombre.compareTo  
            (((PersonaComparable)obj).nombre);  
    }  
  
    public String toString() {  
        return nombre + ":" + edad + " ";  
    }  
}
```

Ejemplo 5.5

```
import java.util.TreeSet;

public class PruebaComparable {
    public static void main (String [] args) {
        TreeSet grupo = new TreeSet();
        grupo.add(new PersonaComparable("Eduardo",18));
        grupo.add(new PersonaComparable("Jaime",19));
        grupo.add(new PersonaComparable("Alan",17));
        grupo.add(new PersonaComparable("Samuel",18));
        grupo.add(new PersonaComparable("adriana",18));
        System.out.println(grupo);
    }
}
```

Ejercicio 5.1

Modificar el método `compareTo` de la clase `PersonaComparable` de tal modo que ordene por edad en lugar de ordenar por nombre.



5.5 Interfaz *Comparator*

La interfaz *Comparator* representa una relación de orden. Es utilizada para ordenar colecciones.

Se utiliza cuando los objetos con los que se trabaja no tienen implementada la interfaz *Comparable*.

Solo posee un método que se debe sobre-escribir para proveer un parámetro de comparación:

```
public int compare(Object ob1, Object ob2)
```



Ejemplo 5.6

```
public class Alumno {  
    String nombre;  
    int numCuenta;  
  
    Alumno (String nombre, int numCuenta){  
        this.nombre = nombre;  
        this.numCuenta = numCuenta;  
    }  
  
    public String toString(){  
        return nombre + "-" + numCuenta;  
    }  
}
```



Ejemplo 5.6

```
import java.util.Comparator;

public class ComparadorNombreAlumno implements Comparator
{
    public int compare(Object obj1, Object obj2){
        Alumno a1 = (Alumno)obj1;
        Alumno a2 = (Alumno)obj2;
        return a1.nombre.compareTo(a2.nombre);
    }
}
```



Ejemplo 5.6

```
import java.util.Comparator;
import java.util.TreeSet;

public class PruebaComparador {
    public static void main (String [] args) {
        Comparator c = new ComparadorNombreAlumno();
        TreeSet grupo = new TreeSet(c);
        grupo.add(new Alumno("Eduardo",3018));
        grupo.add(new Alumno("Jaime",3019));
        grupo.add(new Alumno("Alan",3017));
        grupo.add(new Alumno("Samuel",3022));
        grupo.add(new Alumno("adriana",3015));
        System.out.println(grupo);
    }
}
```



Ejercicio 5.1

Realizar una clase que implemente la interfaz *Comparator* que permita comparar Alumnos por sus números de cuenta.

Así mismo, realizar la clase `PruebaComparador2.java` que implemente el Comparador que realizaron.



6

Archivo jar



6. Archivo jar

6.1 Comando jar

6.2 Archivo de manifiesto



Archivo jar

Una archivo JAR es un archivo comprimido que incluye una estructura de directorios con Clases. Estos archivos se utilizan para distribuir y/o utilizar las clases (programas) de una manera eficiente a través de un solo archivo, esto es, agrega todas las clases de nuestro programa en un paquete.



Además, Java permite ejecutar las clases del archivo .jar sin tener que desempaquetarlo. Esto es muy útil cuando se quiere distribuir la aplicación.

El JDK y/o J2SE se incluye el comando jar el cual permite generar, observar y descomprimir archivos .jar.



6.1 Comando jar

Para crear un archivo ejecutable es necesario tener los archivos clase que se van a insertar dentro del archivo, es decir, los bytecode (archivo .class) del código fuente (archivo .java).

Si las clases no pertenecen a ningún paquete, simplemente se insertan dentro del archivo .jar con la siguiente sintaxis:

```
jar -cvf nombreArchJar.jar archivo1.class archivo2.class
```



```
jar -cvf nombreArchJar.jar archivo1.class archivo2.class
```

De donde:

- c crear archivo de almacenamiento**
- v generar salida detallada de los datos de salida estándar**
- f especificar nombre del archivo de almacenamiento**

Al final de la instrucción se escriben los nombres de las clases que va a incluir el archivo jar.



Para ejecutar el archivo jar se utiliza el comando java de la siguiente manera:

```
java -cp ./nombreArchJar.jar ClaseMain
```

Donde ClaseMain es el nombre de la clase que contiene el método main, es decir, es necesario especificar qué clase del paquete jar ejecutar.



Si las clases que van a pertenecer al archivo jar pertenecen a algún paquete, hay que especificar el directorio donde se encuentran las clases del paquete:

jar -cvf nombreArchJar.jar nombreDirectorio

jar -cvf nombreArchJar.jar RutaAbsoluta



Para ver el contenido de un archivo .jar se utiliza la bandera –t (crear la tabla de contenido del archivo de almacenamiento):

jar –tf nombreArchJar.jar

Para agregar y/o modificar (reemplazar) una clase de un archivo .jar se utiliza la bandera –u (actualizar archivo de almacenamiento existente):

jar –uf nombreArchJar.jar directorio/nombreClase.class



El comando jar tiene diferentes banderas para comprimir. Se puede consultar todas las banderas y el uso del comando mediante la ayuda del mismo:

```
jar --help
```



6.2 Archivo de manifiesto

Para ejecutar un elemento .jar, se puede crear un archivo que indique cuál es la clase principal que se debe ejecutar. Este archivo se conoce como archivo de manifiesto.

El archivo de manifiesto debe contener el nombre de la clase a ejecutar y puede ser un archivo de texto:

Main-Class: Instrumento



El comprimido .jar con archivo de manifiesto se genera de la siguiente manera:

```
jar -cmvf manifiesto.txt nombreArchJar.jar *.class
```

donde

-m incluye la información de un archivo de manifiesto específico

Para ejecutarlo se escribe: java -jar nombreArchJar.jar, donde -jar le indica a la máquina virtual que se va a ejecutar un archivo jar.



Para realizar un archivo jar primero se debe generar el archivo de manifiesto, en este caso, manifiesto.txt cuyo contenido es el siguiente:

Main-Class: PanelDivisor

Ya que se posee el archivo de manifiesto y los archivos que se desean ingresar al comprimido (imágenes, .class, .java, etc), se genera el archivo ejecutable (.jar):

```
$ jar -cmvf manifiesto.txt PanelDivisor.jar PanelDivisor.class
```

Una vez generado el comprimido es posible ejecutarlo:

```
$ java -jar PanelDivisor.jar
```

7

Interfaz gráfica de usuario (GUI)



7. Interfaz gráfica de usuario (GUI)

7.1 AWT (Abstract Window Toolkit)

7.2 Swing

7.3 Diferencias entre AWT y Swing

7.4 Componentes del paquete Swing

7.4.1 Contenedores básicos

7.4.2 Componentes básicos

7.4.3 Componentes especializados

7.4.4 Componentes básicos no interactivos

7.5 Aspecto de la aplicación

7.6 Distribución de los componentes

7.7 Eventos

7.7.1 Adaptadores

7.8 Menú



Interfaz gráfica de usuario (GUI)

El objetivo de la librería de interfaz gráfica de usuario (GUI) de Java es permitir al programador construir una aplicación gráfica que se vea bien en diferentes plataformas.



Para construir una Interfaz Gráfica de Usuario (GUI) es necesario seguir los siguientes pasos:

- **Crear un contenedor superior y obtener su contenedor intermedio.**
- **Seleccionar un gestor de esquemas para el contenedor intermedio.**
- **Crear los componentes adecuados.**
- **Agregarlos al contenedor intermedio.**
- **Dimensionar el contenedor superior.**
- **Mostrar el contenedor superior.**



7.1 AWT (Abstract Window Toolkit)

Las JFC (Java Foundation Classes) son parte de la API de Java, compuesto por clases que sirven para crear interfaces gráficas visuales para las aplicaciones y applets de Java. AWT es un paquete gráfico contenido en las JFC.

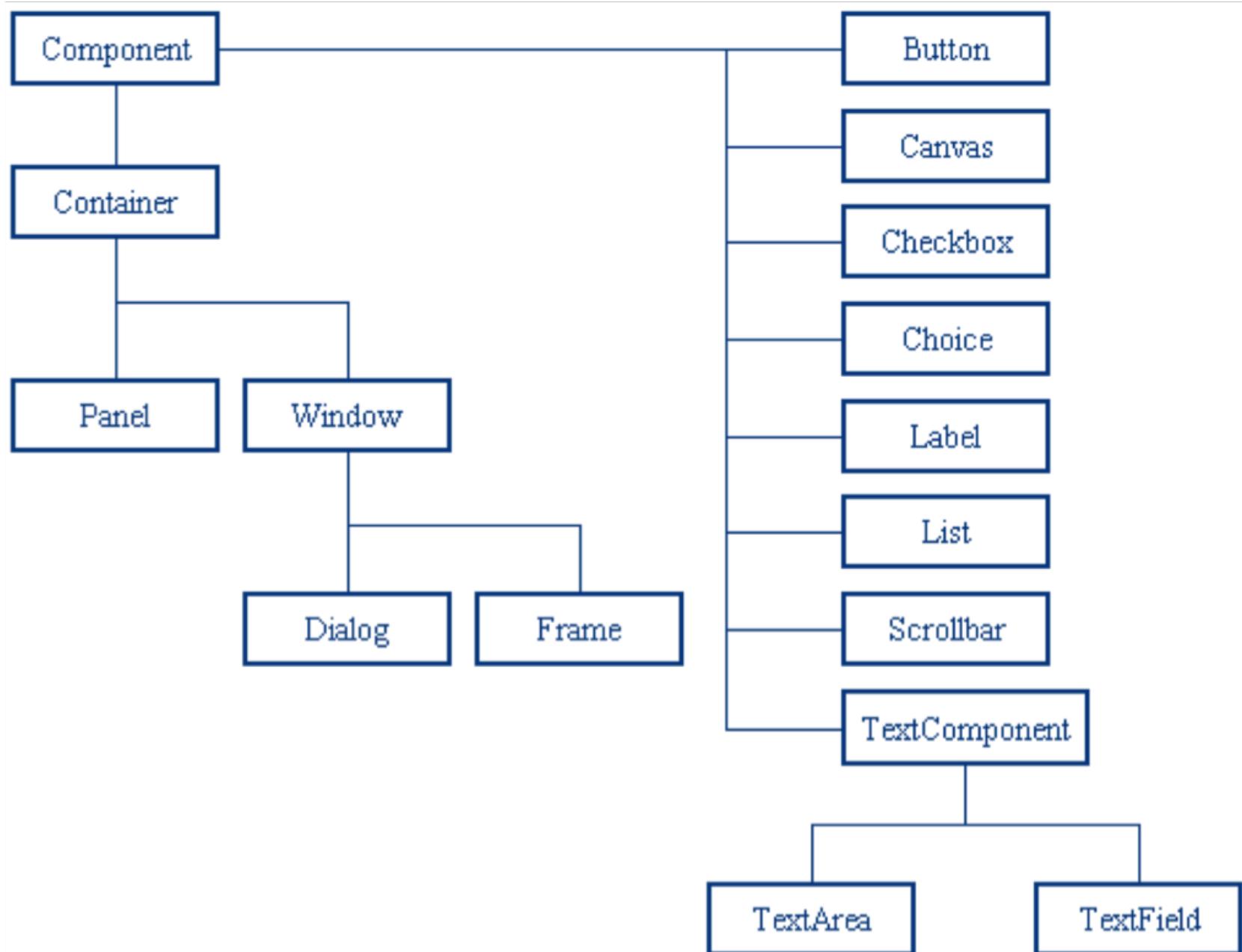


AWT es un conjunto de herramientas GUI diseñadas para trabajar con múltiples plataformas. Este paquete viene incluido en la API de Java como `java.awt` desde la primera versión, por lo que las interfaces generadas con esta biblioteca funcionan en todos los entornos Java disponibles.

AWT también funciona en navegadores que soportan Java lo que lo hace muy eficiente para la creación de applets.

A continuación se muestran las principales clases de AWT así como la relación que existe entre ellas.





7.2 Swing

El paquete Swing es un paquete gráfico que apareció en la versión 1.2 de Java. Esta compuesto por un amplio conjunto de componentes de interfaces de usuario y que pretenden funcionar en el mayor número posible de plataformas.

Las clases de Swing se encuentran disponibles en el paquete *javax.swing*.



Cada uno de los componentes de paquete Swing puede presentar diversos aspectos y comportamientos en función de una biblioteca de clases (bibliotecas de aspecto y de comportamiento):

- **metal.jar:** Aspecto y comportamiento independiente de la plataforma.
- **motif.jar:** Basado en la interfaz Sun Motif.
- **windows.jar:** Muy similar a las interfaces Microsoft Windows 95.



Modelo-Vista-Controlador (MVC)

Los componentes de Swing están basados en la arquitectura Modelo-Vista-Controlador (MVC). De acuerdo con la arquitectura MVC, un componente puede ser modelado por tres partes separadas:

- **Modelo:** almacena la información utilizada para definir al componente.
- **Vista:** representa la visualización del componente. La visualización está definida por el modelo (la información).
- **Controlador:** maneja el comportamiento del componente cuando un usuario interactúa con él.



Para construir una Interfaz Gráfica de Usuario con Swing se necesitan:

- **Los contenedores:** entes donde se situarán los componentes y/o donde se realizarán los dibujos: **JFrame, JDialog, JWindow, y JApplet.**
- **Los componentes:** menús, botones de comando, barras de desplazamiento, cajas y áreas de texto, botones de opción y selección, etc.
- **Administradores de posición:** permiten agrupar los componentes en un orden (disposición) específica.



7.3 Diferencias entre AWT y Swing

Una las principales diferencias entre los componentes de `java.awt` con los de `javax.swing` es que los primeros están enlazados directamente a las herramientas de la interfaz gráfica de usuario de la plataforma local.

Por otro lado, a los componentes Swing se les conoce como componentes puros de Java (o componentes ligeros), debido a que su escritura, manipulación y despliegue son completamente en Java y, por tanto, ofrecen mayor portabilidad y flexibilidad.



Por lo anterior, un programa realizado con AWT de Java que se ejecuta en distintas plataformas tiene una apariencia distinta y, en ocasiones, hasta las interacciones con el usuario cambian dependiendo de la plataforma donde se ejecute.

Por otro lado, los componentes Swing permiten al programador especificar una apariencia visual distinta para cada plataforma, una apariencia visual uniforme entre todas las plataformas o, incluso, puede cambiar la apariencia visual mientras el programa se ejecuta.



En general, tiene más ventajas utilizar Swing que AWT al momento de crear GUI's. A continuación se listan algunas de estas ventajas:

- **Variedad de componentes**
- **Aspecto modifiable**
- **Arquitectura Modelo-Vista-Controlador**
- **Objetos de acción**
- **Contenedores anidados**
- **Escritorios virtuales**
- **Bordes complejos**
- **Componentes para tablas y árboles de datos**
- **Potentes manipuladores de texto**



7.4 Componentes del paquete Swing

Los elementos del paquete swing son los componentes y los contenedores.

Los componentes son los elementos visibles en la interfaz gráfica, como lo son los botones, las etiquetas, los campos de texto, etc. Estos componentes se sitúan dentro de algún contenedor.

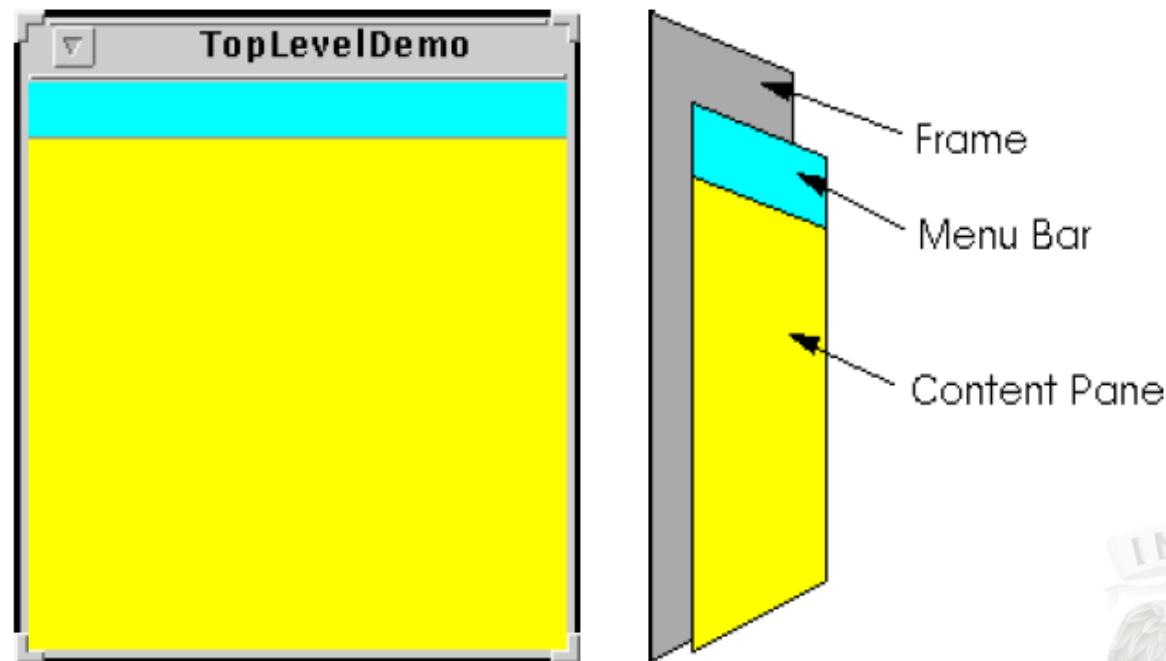


Los contenedores son los elementos que permiten almacenar componentes u otros contenedores. Existen dos tipos de contenedores:

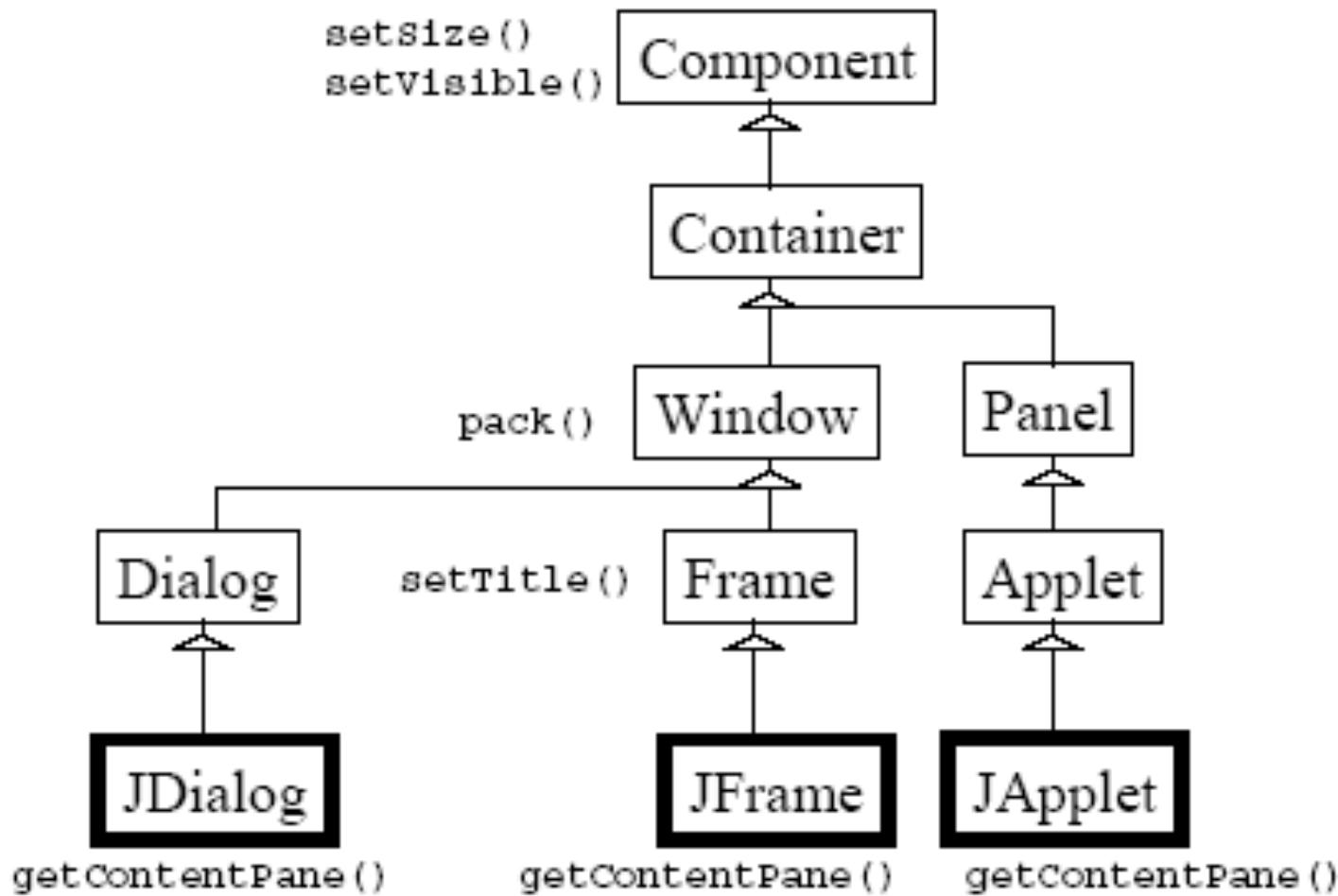
- **Superiores: como los JApplet, JFrame, JDialog, etc.**
- **Intermedios: JPanel, JScrollPane, JSplitPane, JTabbedPane, JToolBar, etc.**



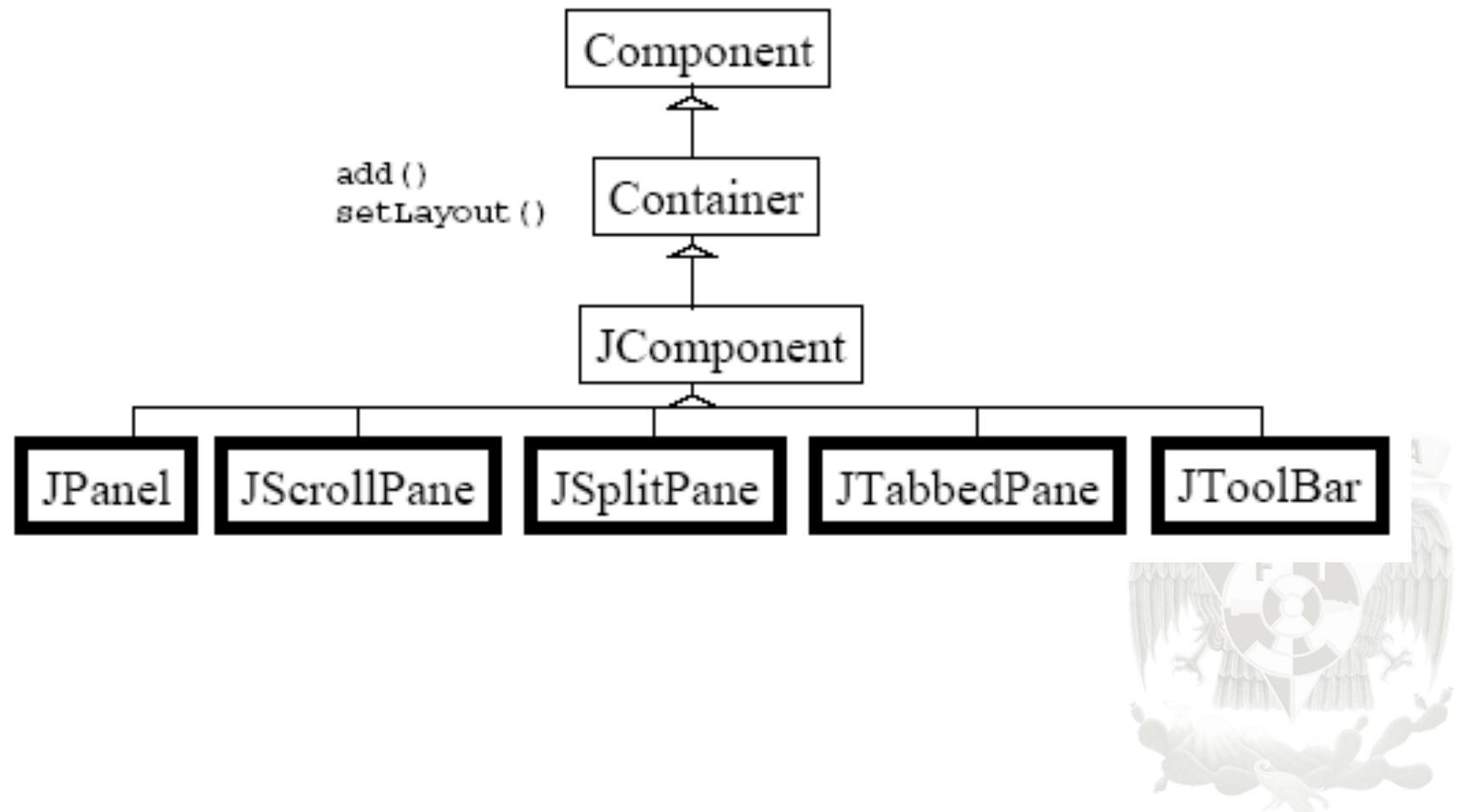
Contenedores superiores



Jerarquía de los contenedores superiores

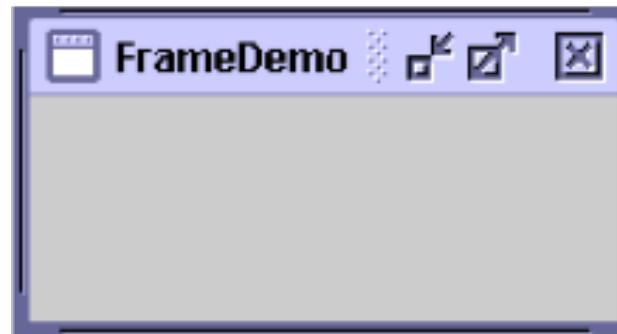


Contenedores intermedios

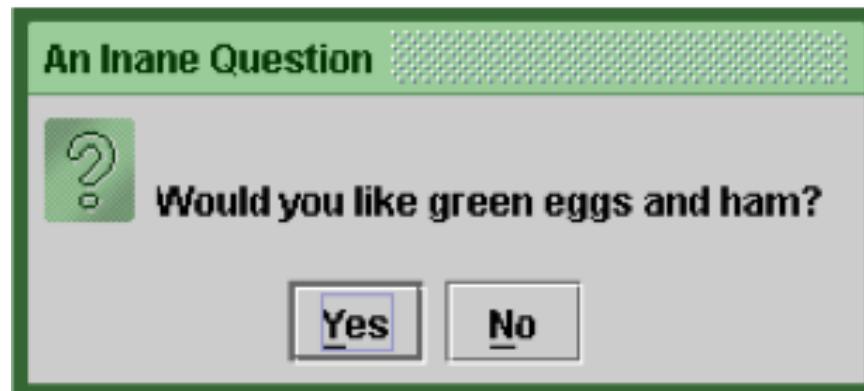


7.4.1 Contenedores básicos

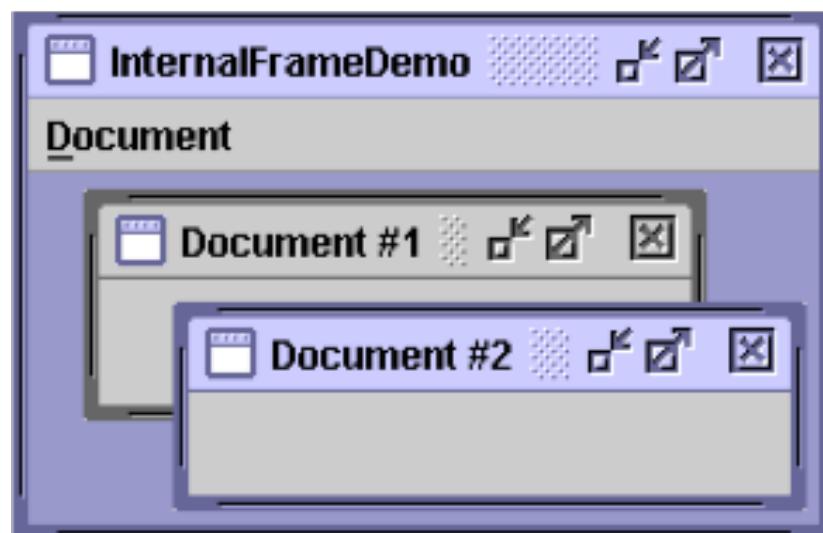
- **JFrame:** Representa una ventana básica, capaz de contener otros componentes. Casi todas las aplicaciones construyen al menos un marco de este tipo.



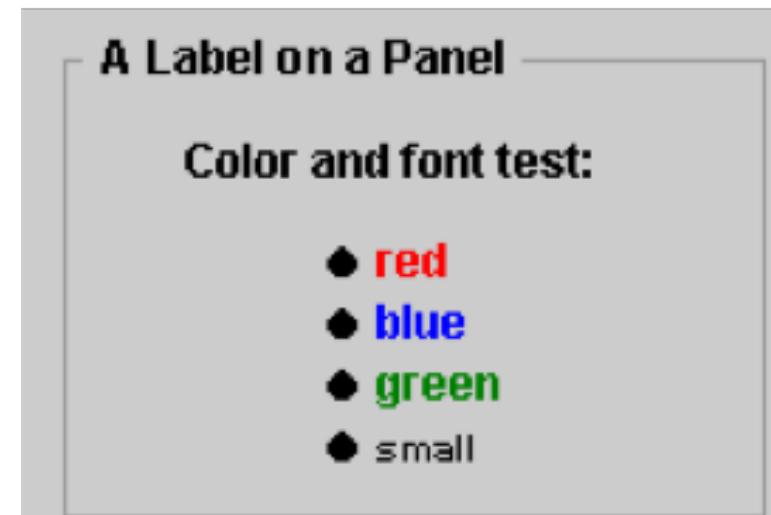
- **JDialog:** Los cuadros de diálogo son marcos restringidos, dependientes de un marco principal. Son cuadros de diálogo generales normalmente utilizados para solicitar datos.
- **JOptionPane:** son cuadros de diálogo sencillos predefinidos para solicitar confirmación, realizar advertencias o notificar errores.



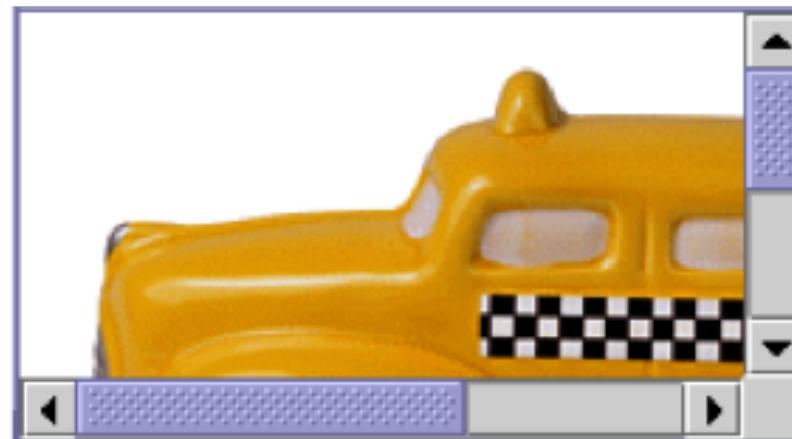
- **JInternalFrame:** es una sub-ventana, que no puede salir de los límites marcados por la ventana principal. Es muy común en aplicaciones que permiten tener varios documentos abiertos simultáneamente.



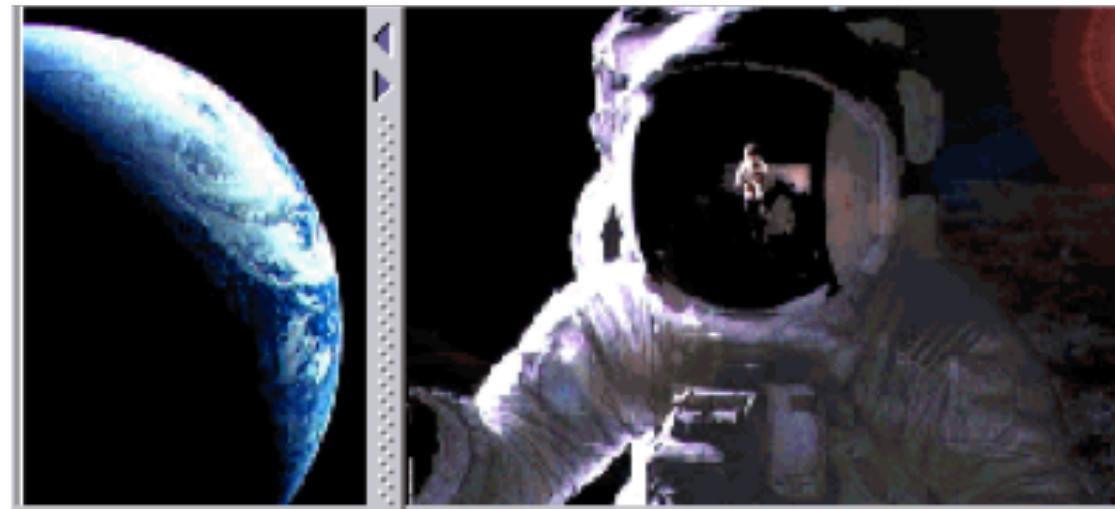
- JPanel: Un panel sirve para agrupar y organizar otros componentes. Puede estar decorado por un borde y una etiqueta.



- **JScrollPane:** Es un panel que permite visualizar un componente de un tamaño mayor que el disponible mediante el uso de barras de desplazamiento.



- **JSplitPane:** Permite visualizar dos componentes en un marco, ya sea de manera horizontal o vertical, así como, modificar la personalizar la distribución.



- **JTabbedPane:** Permite definir varias hojas con pestañas que pueden albergar otros componentes. El usuario puede seleccionar la hoja que desea ver mediante las pestañas.



- **JToolBar:** Es un contenedor que permite agrupar otros componentes, normalmente botones con iconos, en una fila o columna. Las barras de herramientas tienen la particularidad de que el usuario puede situarlas en distintas posiciones sobre el marco principal.

