

# Lenguaje de Programación I

# Faltan créditos

# ÍNDICE

Presentación	5
Red de contenidos	6
<b>Unidad de Aprendizaje 1</b>	
SEMANA 1 : Gestión de excepciones. Excepciones y errores comunes en Java	7
SEMANA 2 : Uso de Assertions en Java	25
<b>Unidad de Aprendizaje 2</b>	
SEMANA 3 : La clase String. Definición y gestión de memoria. Las Clases StringBuffer y StringBuilder. Principales métodos	37
SEMANA 4 : Principales clases del paquete java.io y serialización de objetos	47
SEMANA 5 : Clases Calendar y Date. Formateo de datos. Uso de regular Expressions, Tokens y delimitadores en Java	61
<b>Unidad de Aprendizaje 3</b>	
SEMANA 6 : Framework Collections y estrategias de ordenamiento	83
SEMANA 7 : EXAMEN PARCIAL DE TEORÍA	
SEMANA 8 : Examen Parcial de Laboratorio	
SEMANA 9 : Tipos Genéricos en Java	103
<b>Unidad de Aprendizaje 4</b>	
SEMANA 10 : Clases internas estándares y locales a un método	113
SEMANA 11 : Clases internas anónimas y estáticas	123
SEMANA 12 : Threads en java: definición, estados y priorización	133
SEMANA 13 : Sincronización y bloqueo de código	153
<b>Unidad de Aprendizaje 5</b>	
SEMANA 14 : Compilación y ejecución de aplicaciones java. Uso de Sentencias import estáticas.	163
SEMANA 15 : Taller Java SE	
SEMANA 16 : EXAMEN FINAL DE LABORATORIO	
SEMANA 17 : EXAMEN FINAL DE TEORÍA	



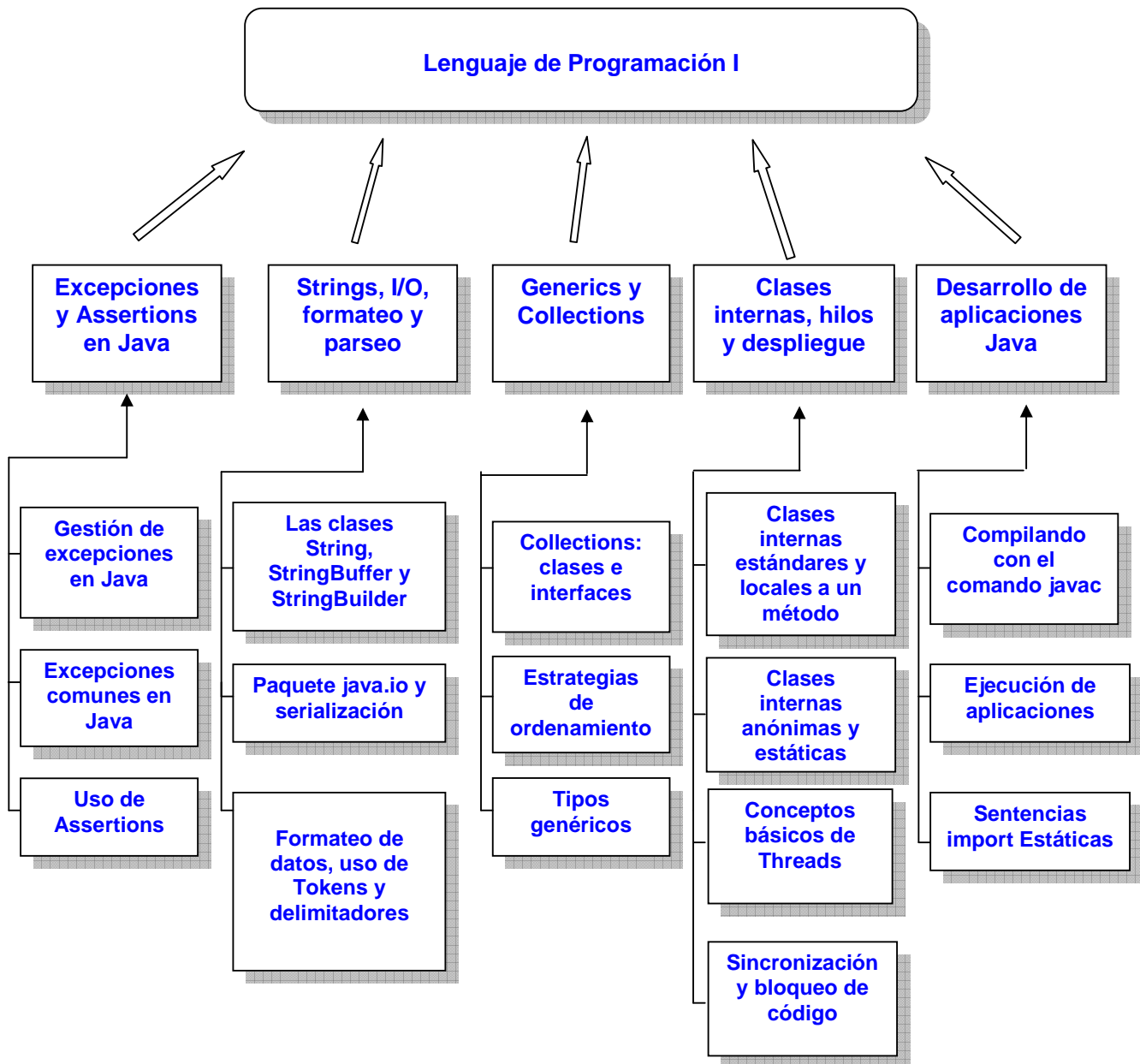
# PRESENTACIÓN

***Lenguaje de Programación I*** pertenece a la línea de Programación y Desarrollo de Aplicaciones y se dicta en la carrera de Computación e Informática de la institución. El curso brinda un conjunto de conceptos, técnicas y herramientas que permiten al alumno alcanzar un nivel avanzado de conocimiento en el lenguaje Java Standard Edition (JSE), implementando una aplicación Stand Alone, utilizando Java como lenguaje de programación.

El manual para el curso ha sido diseñado bajo la modalidad de unidades de aprendizaje, las que se desarrollan durante semanas determinadas. En cada una de ellas, hallará los logros, que debe alcanzar al final de la unidad; el tema tratado, el cual será ampliamente desarrollado; y los contenidos, que debe desarrollar, es decir, los subtemas. Por último, encontrará las actividades que deberá desarrollar en cada sesión, que le permitirán reforzar lo aprendido en la clase.

El curso es eminentemente práctico: consiste de una sesión semanal de laboratorio. En primer lugar, se inicia con la definición de conceptos básicos para el manejo de Excepciones y Assertions en Java, enfatizándose en las buenas prácticas existentes para manejar ambos aspectos. Continúa con la implementación de aplicaciones que apliquen conceptos avanzados de manejo de Strings, flujos de entrada y salida, serialización de objetos, formateo y parseo de datos, así como el uso del Framework Collections de Java. Luego, se desarrollan aplicaciones que utilicen Clases internas e hilos. Se concluye con tópicos avanzados de instalación y despliegue de aplicaciones Java Stand Alone a través de la implementación de una aplicación que incluye todos los conceptos aprendidos en el curso.

## RED DE CONTENIDOS



**UNIDAD DE  
APRENDIZAJE**

**1**

**SEMANA**

**1**

## **GESTIÓN DE EXCEPCIONES Y ASSERTIONS EN JAVA**

### **LOGRO DE LA UNIDAD DE APRENDIZAJE**

- Al término de la unidad, los alumnos crearán aplicaciones que utilizan assertions y gestionan los diferentes tipos de excepciones en Java, haciendo uso de sentencias assert, manejadores de excepciones, declarando y sobrescribiendo métodos que atrapan y lanzan excepciones.

### **TEMARIO**

- Manejo de excepciones usando las sentencias Try, Catch y Finally
- Propagación de excepciones
- Definición de excepciones y jerarquía de Excepciones
- Declaración de excepciones e interfaces públicas
- Errores y excepciones comunes
- Assertions - Definición
- Configuración del uso de Assertion
- Buenas prácticas del uso de Assertions

### **ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucran la declaración y gestión de excepciones.
- Los alumnos implementan una aplicación java swing que gestiona excepciones usando bloques TRY – CATCH – FINALLY.

## 1. Gestión de excepciones en Java

En muchos lenguajes, escribir código que verifique y gestione errores suele ser tedioso; incluso, puede prestarse a confusión si la codificación se hace poco entendible, esto es lo que se conoce como un código "spaghetti". Java proporciona un mecanismo eficiente para gestionar errores y excepciones: el gestor de excepciones de java.

Este mecanismo facilitará la detección de errores sin tener que escribir código especial para verificar los valores retornados. El código que gestiona las excepciones se encuentra claramente separado del código que las genera, esto permite utilizar el mismo código de gestión de excepciones con un amplio rango de posibles excepciones.

### 1.1 Sentencias TRY - CATCH

El término "exception" significa "condición excepcional" y es una ocurrencia que altera el flujo normal de un programa. Son múltiples los factores que pueden generar una excepción, incluyendo fallas de hardware, agotamiento de recursos, entre otros.

Cuando un evento excepcional ocurre en Java, se dice que una excepción ha sido lanzada. El código responsable por hacer algo con respecto a la excepción generada es llamado el código gestor de excepciones y atrapa la excepción lanzada.

La palabra clave TRY es usada para definir un bloque de código dentro del cual puede ocurrir una excepción. Una o más cláusulas CATCH están asociadas a una excepción específica o grupo de excepciones. A continuación se muestra un ejemplo:

```
try {  
    // Aquí colocamos código que podría lanzar alguna excepción.  
  
} catch(MiPrimeraExcepcion) {  
    // Aquí colocamos código que gestiona la excepción  
  
} catch(MiSegundaExcepcion) {  
    // Aquí colocamos código que gestiona la excepción  
  
}  
  
// Aquí colocamos código que no es riesgoso.
```



En el ejemplo, las líneas que definamos dentro de la cláusula TRY constituyen la zona protegida. Las líneas dentro de la primera cláusula **CATCH** constituyen el gestor de excepciones para excepciones de tipo **MiPrimeraExcepcion**.

Imaginemos ahora que tenemos el siguiente código:

```
try {  
  
    obtenerArchivoDeLaRed  
    leerArchivoYCargarTabla  
  
}catch(NoPuedeObtenerArchivoDeLaRed) {  
    mostrarMensajeDeErrorDeRed  
  
}
```

En el ejemplo anterior, podemos observar cómo el código susceptible de una operación riesgosa, en este caso, cargar una tabla con los datos de un archivo, es agrupado dentro de un bloque TRY. Si la operación **obtenerArchivoDeLaRed** falla, no leeremos el archivo ni cargaremos la tabla. Esto se debe a que la operación previa ya lanzó una excepción.

## 1.2 La cláusula Finally

Las cláusulas TRY y CATCH proporcionan un mecanismo eficiente para atrapar y gestionar excepciones, sin embargo, aún no estamos controlando cómo inicializar variables o asignar valores a objetos que deben ser inicializados ocurra o no ocurra una excepción.

Un bloque **FINALLY** encierra código que es siempre ejecutado después del bloque TRY, sea que se produzca o no una excepción. Finally se ejecuta, incluso, si existe una sentencia return dentro del bloque TRY:

“El bloque finally se ejecuta después de que la sentencia return es encontrada y antes de que la sentencia return sea ejecutada”.

El bloque **FINALLY** es el lugar correcto para, por ejemplo, cerrar archivos, liberar sockets de conexión y ejecutar cualquier otra operación de limpieza que el código necesite. Si se generó una excepción, el bloque **FINALLY** se ejecutará después de que se haya completado el bloque **CATCH** respectivo.

La cláusula **FINALLY** no es obligatoria. Por otro lado, dado que el **compilador no requiere una cláusula CATCH**, podemos tener código con

un bloque TRY seguido inmediatamente por un bloque FINALLY. Este tipo de código es útil cuando la excepción será retornada al método que hizo la llamada. Usar un bloque FINALLY permite ejecutar código de inicialización así no exista la cláusula CATCH.

A continuación, se muestran algunos ejemplos de aplicación de las cláusulas TRY, CATCH y FINALLY.

**//El siguiente código muestra el uso de TRY y FINALLY:**

```
try {  
    // hacer algo  
} finally {  
    //clean up  
}
```

**//El siguiente código muestra el uso de TRY, CATCH y FINALLY:**

```
try {  
    // hacer algo  
} catch (SomeException ex) {  
    // do exception handling  
} finally {  
    // clean up  
}
```

**//El siguiente código es inválido:**

```
try {  
    // hacer algo  
}  
// Se necesita un CATCH o un FINALLY aquí  
System.out.println("fuera del bloque try");
```

**//El siguiente código es inválido:**

```
try {  
    // hacer algo  
}  
// No se puede tener código entre una TRY y un CATCH  
System.out.println("fuera del bloque try");  
catch(Exception ex) { }
```

## 1.3 Propagación de excepciones

### 1.1.1 Pila de llamadas

La mayoría de lenguajes manejan el concepto de pila de llamadas o pila de métodos. La pila de llamadas es la cadena de métodos que un programa ejecuta para obtener el método actual. Si nuestro programa inicia con el método `main()`, y éste invoca al método `a()`, el cual invoca al método `b()` y éste al método `c()`, **la pila de llamadas** tendría la siguiente apariencia:

c
b
a
main

Como se puede observar, el último método invocado se encuentra en la parte superior de la pila, mientras que el primer método llamado se encuentra en la parte inferior de la misma. El método en la parte más alta de la pila será el método que **actualmente estamos ejecutando**. Si retrocedemos en la pila, nos estaremos moviendo desde el método actual hacia el método previamente invocado.

La pila de llamadas mientras el método 3 está ejecutándose:

metodo3()	El método 2 invoca al método 3
metodo2()	El método 1 invoca al método 2
metodo1()	main invoca al método 1
main ()	main inicia la ejecución

**Orden en el cual los métodos son colocados en la pila de llamadas**

La pila de llamadas después de terminar la ejecución del método 3.

metodo2()	Se completará el método 2
metodo1()	Se completará el método 1
main ()	Se completará el método main y la JVM hará un exit

**Orden en el cual los métodos completan su ejecución**

### 1.1.1 Esquema de propagación de excepciones

Una excepción es primero lanzada por el método en la parte más alta de la pila. Si no es atrapada por dicho método, la excepción caerá al siguiente nivel de la pila, al siguiente método. Se seguirá este proceso hasta que la excepción sea atrapada o hasta que llegar a la parte más baja de la pila. Esto se conoce como **propagación de excepciones**.

Una excepción que nunca fue atrapada ocasionará que se detenga la ejecución de nuestro programa. Una descripción de la excepción será visualizada y el detalle de la pila de llamadas será visualizado. Esto nos ayudará a hacer un “**debug**” sobre nuestra aplicación indicándonos qué excepción fue lanzada y qué método la lanzó.

## 1.4 Definiendo excepciones

Cada excepción es una instancia de una clase que tiene a la clase `Exception` dentro de su jerarquía de herencia. Es decir, las excepciones son siempre subclases de la clase `java.lang.Exception`.

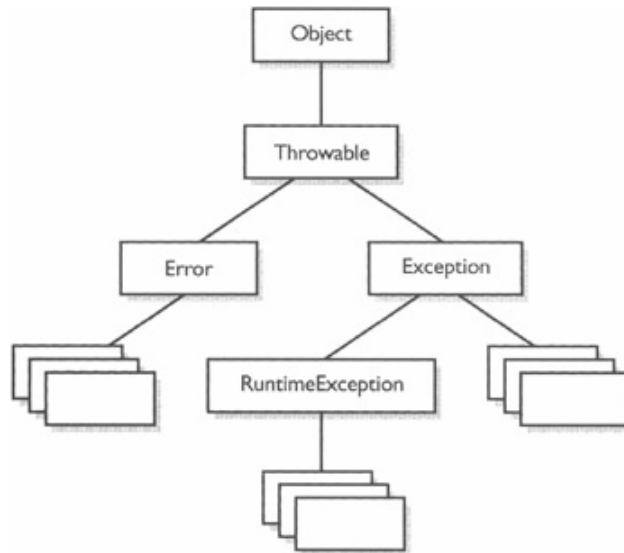
Cuando una excepción es lanzada, un objeto de un subtipo particular de excepción es instanciado y pasado al gestor de excepciones como un argumento de la cláusula `CATCH`. A continuación, se muestra un ejemplo:

```
try {  
    // colocamos algún código aquí  
}  
catch (ArrayIndexOutOfBoundsException e) {  
    e.printStackTrace();  
}
```

En el ejemplo, **e** es una instancia de la clase **ArrayIndexOutOfBoundsException**. Como cualquier otro objeto, podemos invocar a cualquiera de sus métodos.

## 1.5 Jerarquía de excepciones

Todas las clases de excepciones son subtipos de la clase Exception. Esta clase se deriva de la clase Throwable (la cual deriva de la clase Object).



Podemos observar que existen dos subclases que se derivan de la clase Throwable: **Exception** y **Error**. Las clases que se derivan de Error, representan situaciones inusuales que **no son causadas por errores del programa** e indican situaciones que normalmente no se darían durante la ejecución del programa, como por ejemplo, “JVM running out of memory”.

Generalmente, nuestra aplicación no estará capacitada para recuperarse de un error, por lo que no será necesario que los gestionemos. Los errores, técnicamente, **no son excepciones**, dado que no se derivan de la clase Exception.

En general, una excepción representa algo que sucedió no como resultado de un error de programación, sino, más bien, porque algún recurso no se encuentra disponible o alguna otra condición requerida para la correcta ejecución no está presente. Por ejemplo, si se supone que la aplicación debe comunicarse con otra aplicación o computadora que no está contestando, esta sería una excepción no causada por un bug. En la figura previa también se aprecia un subtipo de excepción llamado RuntimeException. Estas excepciones son un caso especial porque ellas, a veces, indican errores del programa. Ellas también representan condiciones excepcionales raras y/o difíciles de gestionar.

Existen dos maneras de obtener información relacionada con una excepción:

- La primera es del tipo de excepción en sí misma ( el parámetro en la cláusula CATCH).
- La segunda, es obteniéndola del mismo objeto Exception.

La clase Throwable, ubicada en la parte superior de la jerarquía del árbol de excepciones, proporciona a sus descendientes algunos métodos que son útiles para los gestores de transacciones. Uno de estos métodos es

**printStackTrace()**. Este método imprime primero una descripción de la excepción y el método invocado más recientemente, continua así imprimiendo el nombre de cada uno de los métodos en la pila de llamadas.

## 1.6 Gestionando una jerarquía completa de Excepciones

La palabra clave **catch** permite especificar un tipo particular de excepción a atrapar. Se puede atrapar más de un tipo de excepción en una cláusula **catch**. Si la clase de excepción especificada en la cláusula **catch** no tiene subclases, entonces sólo excepciones de ese tipo podrán ser atrapadas. Sin embargo, si la clase especificada tiene subclases, se podría atrapar cualquiera de esas clases.

A continuación, se muestra un ejemplo:

```
try {  
    // Algun codigo que pueda lanzar una boundary exception  
}  
catch (IndexOutOfBoundsException e) {  
    e.printStackTrace ();  
}
```

Se podrán atrapar excepciones de los tipos **ArrayIndexOutOfBoundsException** or **StringIndexOutOfBoundsException**.

En algunas situaciones, es conveniente trabajar con superclases para manejar excepciones, pero no se debe exagerar en el uso de este tipo de manejos, porque podríamos perder importante información relacionada con una excepción específica.

En la práctica, se recomienda evitar el uso de sentencias como la que se muestra a continuación:

```
try {  
    // some code  
}catch (Exception e) {  
    e.printStackTrace();  
}
```

Este código atraparé cada excepción generada, pero el gestor de excepciones no podrá manejar adecuadamente las excepciones generadas debido a que sólo contamos con una cláusula **CATCH**.

## 1.7 Jerarquía de excepciones

Si tenemos una jerarquía de excepciones compuesta de una superclase y varios subtipos y estamos interesados en gestionar uno de los subtipos en particular, necesitaremos escribir sólo dos cláusulas catch.

```
import java.io.*;
public class ReadData {
    public static void main(String args[]) {
        try {
            RandomAccessFile raf =
                new RandomAccessFile("myfile.txt", "r");
            byte b[] = new byte[1000];
            raf.readFully(b, 0, 1000);
        }
        catch(FileNotFoundException e) {
            System.err.println("File not found");
            System.err.println(e.getMessage());
            e.printStackTrace();
        }
        catch(IOException e) {
            System.err.println("IO Error");
            System.err.println(e.toString());
            e.printStackTrace();
        }
    }
}
```

En el ejemplo, se requiere gestionar la excepción `FileNotFoundException`. Si el código genera una excepción de tipo `FileNotFoundException`, ésta será gestionada por la primera cláusula CATCH. Si se genera otra excepción de tipo `IOException`, como por ejemplo `EOFException`, la cual es una subclase de `IOException`, ésta será gestionada por la segunda cláusula CATCH. Si alguna otra excepción es generada, ninguna cláusula CATCH será ejecutada y la excepción será **propagada hacia abajo** en la pila de llamadas.

Es importante notar que la cláusula catch para `FileNotFoundException` fue ubicada sobre la de `IOException`. Si lo hacemos al revés, el programa **no compilará**. A continuación se muestra un ejemplo:

```
try {
    // do risky IO things
} catch (IOException e) {
    // handle general IOExceptions
} catch (FileNotFoundException ex) {
    // handle just FileNotFoundException
}
```

Obtendremos un **error de compilación** similar al siguiente:

```
TestEx.java:15: exception java.io.FileNotFoundException has
already been caught
} catch (FileNotFoundException ex) {
^
```

### Ejercicios de aplicación:

1. Dado el siguiente código:

```
class Plane {
    static String s = "-";
    public static void main(String[] args) {
        new Plane().s1();
        System.out.println(s);
    }
    void s1() {
        try { s2();
        catch (Exception e) { s += "c"; }
    }
    void s2() throws Exception {
        s3(); s += "2";
        s3(); s += "2b";
    }
    void s3() throws Exception {
        throw new Exception();
    }
}
```

¿Cuál será el resultado de su ejecución?

- a) -
- b) -c
- c) -c2
- d) -2c
- e) -c22b
- f) -2c2b
- g) -2c2bc
- h) Compilation fails



2. Dada la siguiente sentencia:

```
try { int x = Integer.parseInt("two");}
```

Seleccione todas las opciones que permitirían implementar un bloque CATCH adecuadamente:

- a) ClassCastException
- b) IllegalStateException
- c) NumberFormatException
- d) IllegalArgumentException
- e) ExceptionInInitializerError
- f) ArrayIndexOutOfBoundsException

3. Dado el siguiente código:

```
class Swill {
    public static void main(String[] args) {
        String s = "-";
        switch(TimeZone.CST) {
            case EST: s += "e";
            case CST: s += "c";
            case MST: s += "m";
            default: s += "X";
            case PST: s += "p";
        }
        System.out.println(s);
    }
}
enum TimeZone {EST, CST, MST, PST }
```

¿Cuál será el resultado de su ejecución?

- a) -c
- b) -X
- c) -cm
- d) -cmp
- e) -cmXp
- f) Compilation fails.
- g) An exception is thrown at runtime.

## 2. Excepciones y errores comunes en Java

### 2.1 Origen de las excepciones

Es importante entender qué causan las excepciones y de dónde vienen éstas. Existen dos categorías básicas:

- **Excepciones de la JVM.** Son las excepciones y errores lanzadas por la Java Virtual Machine.
- **Excepciones programáticas.** Aquellas que son lanzadas explícitamente por la aplicación y/o APIs propias del desarrollador.

### 2.2 Excepciones lanzadas por la JVM

Son excepciones en las que no es posible que el compilador pueda detectarlas antes del tiempo de ejecución. A continuación, se muestra un ejemplo:

```
class NPE {  
    static String s;  
    public static void main(String [] args) {  
        System.out.println(s.length());  
    }  
}
```

En el ejemplo anterior, el código compilará satisfactoriamente y la JVM lanzará un **NullPointerException** cuando trate de invocar al método **length()**.

Otro tipo de excepción típica dentro de esta categoría es **StackOverflowError**. La manera más común de que esto ocurra es creando un método recursivo. Ejemplo:

```
void go() { //Aplicación errada de recursividad  
    go();  
}
```

Si cometemos el error de invocar al método **go()**, el programa caerá en un loop infinito, invocando al método **go()** una y otra vez hasta obtener la excepción **stackOverflowError**. Nuevamente, **sólo la JVM** sabe cuando ocurre este momento y la JVM será la fuente de este error.

### 2.3 Excepciones lanzadas programáticamente

Muchas clases pertenecientes a la API java tienen métodos que toman objetos **String** como argumentos y convierten estos **Strings** en valores numéricos primitivos.

Se muestra, a continuación, un ejemplo de excepción perteneciente a este contexto:

```
int parseInt (String s) throws NumberFormatException {
    boolean parseSuccess = false;
    int result = 0;
    // se implementa la lógica de la conversión
    if (!parseSuccess) // si la conversión falló
        throw new NumberFormatException();
    return result;
}
```

Otro ejemplo de excepción programática es `AssertionError` (que en realidad no es una excepción, pero es lanzada programáticamente). Se genera un `IllegalArgumentException`.

Es posible, también, crear nuestras propias excepciones. Estas excepciones caen dentro de la categoría de excepciones generadas programáticamente.

## 2.4 Resumen de excepciones y errores comunes

Se muestran, a continuación, los errores y excepciones más comunes generados por una aplicación java.

<i>Descripción y Fuentes de Excepciones comunes</i>		
Excepción	Descripción	Lanzada generalmente por:
<b>ArrayIndexOutOfBoundsException</b>	Lanzada cuando se intenta acceder a un arreglo con un valor de índice inválido (sea éste negativo o superior a la longitud del arreglo).	JVM
<b>ClassCastException</b>	Lanzada cuando intentamos convertir una referencia a variable a un tipo que falla la prueba de casteo IS-A.	JVM
<b>IllegalArgumentException</b>	Lanzada cuando un método recibe un argumento formateado de manera diferente a lo que el método esperaba.	Programáticamente
<b>IllegalStateException</b>	Lanzada cuando el estado del entorno no coincide con la operación que se intenta ejecutar. Por ejemplo, usar un objeto de la clase <code>Scanner</code> que ha sido cerrado previamente.	Programáticamente

<b>NullPointerException</b>	Lanzada cuando intentamos acceder a un objeto con una variable de referencia cuyo valor actual es null.	JVM
<b>NumberFormatException</b>	Lanzada cuando un método que convierte un String a un número recibe un String que no puede ser convertido.	Programáticamente
<b>AssertionError</b>	Lanzada cuando una sentencia Boolean retorna el valor falso después de ser evaluada.	Programáticamente
<b>ExceptionInInitializerError</b>	Lanzada cuando intentamos inicializar una variable estática o un bloque de inicialización.	JVM
<b>StackOverflowError</b>	Típicamente lanzada cuando un método es invocado demasiadas veces, por ejemplo, recursivamente.	JVM
<b>NoClassDefFoundError</b>	Lanzada cuando la JVM no puede ubicar una clase que necesita, por un error de línea de comando, problemas de classpath, o un archivo class perdido.	JVM

### Ejercicios de aplicación:

1. Dado la siguiente clase:

```
class Circus {
    public static void main(String[] args) {
        int x = 9;
        int y = 6;
        for(int z = 0; z < 6; z++, y--) {
            if(x > 2) x--;
            label:
            if(x > 5) {
                System.out.print(x + " ");
                --X;
                continue label;
            }
            X--;
        }
    }
}
```

¿Cuál será el resultado de su ejecución?

- a) 8
- b) 8 7
- c) 8 7 6
- d) Compilation fails.
- e) An exception is thrown at runtime.

2. Dada la siguiente clase:

```
class Mineral { }
class Gem extends Mineral { }
class Miner {
    static int x = 7;
    static String s = null;
    public static void getWeight(Mineral m) {
        int y = 0 / x;
        System.out.print(s + " ");
    }
    public static void main(String[] args) {
        Mineral[] ma = {new Mineral(), new Gem()};

        for(Object o : ma)
            getWeight((Mineral) o);
    }
}
```

Y la siguiente línea de comando: **java Miner.java**

¿Cuál será el resultado de su ejecución?

- a) null
- b) null null
- c) A ClassCastException is thrown.
- d) A NullPointerException is thrown.
- e) A NoClassDefFoundError is thrown.
- f) An ArithmeticException is thrown.
- g) An IllegalArgumentException is thrown.
- h) An ArrayIndexOutOfBoundsException is thrown

# Resumen

- 📖 Las excepciones pueden ser de dos tipos, controladas y no controladas (checked / unchecked).
- 📖 Las excepciones controladas incluyen todos los subtipos de la clase Exception, excluyendo las clases que heredan de RuntimeException.
- 📖 Las excepciones controladas son sujetas de gestión o declaración de reglas. Cualquier método que pudiese lanzar una excepción controlada (incluyendo métodos que invocan otros que pueden lanzar una excepción controlada), deben declarar la excepción usando la cláusula throws o gestionarla con la sentencia TRY – CATCH más adecuada.
- 📖 Los subtipos de las clases Error o RuntimeException no son controlables, por lo tanto el compilador no forzará la gestión o declaración de regla alguna.
- 📖 Podemos crear nuestras propias excepciones, normalmente heredando de la clase Exception o de cualquiera de sus subtipos. Este tipo de excepción será considerada una excepción controlada por lo que el compilador forzará su gestión o la declaración de alguna regla para la excepción.
- 📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.
  - 🔗 <http://today.java.net/pub/a/today/2003/12/04/exceptions.html>  
Aquí hallará importantes reglas básicas para la gestión de excepciones en java.
  - 🔗 <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Exception.html>  
En esta página, hallará documentación detallada de la clase base de todas las excepciones controladas: Exception







## GESTIÓN DE EXCEPCIONES Y ASSERTIONS EN JAVA

### LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos crearán aplicaciones que utilizan assertions y gestionan los diferentes tipos de excepciones en Java, haciendo uso de sentencias assert, manejadores de excepciones, declarando y sobrescribiendo métodos que atrapan y lanzan excepciones.

### TEMARIO

- Manejo de excepciones usando las sentencias Try, Catch y Finally
- Propagación de excepciones
- Definición de excepciones y jerarquía de excepciones
- Declaración de excepciones e interfaces públicas
- Errores y excepciones comunes
- Assertions - Definición
- Configuración del uso de Assertion
- Buenas prácticas del uso de Assertions

### ACTIVIDADES PROPUESTAS

- Los alumnos resuelven ejercicios que involucren la declaración de assertions de tipos simple y muy simple.
- Los alumnos implementan una aplicación java swing que utiliza assertions para validar argumentos de un método privado.

## 1. Assertions en Java

### 1.1 Conceptos básicos de Assertions

Supongamos que asumimos que un número enviado como parámetro a un método (metodoA()), nunca será negativo. Mientras probamos y depuramos, queremos validar nuestra asunción, sin tener la necesidad de crear sentencias del tipo `System.out.println()`, manejadores de excepciones o sentencias de tipo IF – ELSE en pleno desarrollo, que a la larga podrían implicar también afectar la performance de la aplicación.

Podremos alcanzar este objetivo utilizando Assertions. Revisemos el código mostrado a continuación:

```
private void metodoA(int num) {
    if (num >= 0) {
        useNum(num + x);
    } else { // el número debe ser < 0
        // Este código nunca debería ser alcanzado!
        System.out.println("Ups! num es un numero negativo! "
            + num);
    }
}
```

Dado que estamos seguros de nuestra asunción, no queremos invertir tiempo (o afectar la performance de la aplicación), escribiendo código que gestione errores.

Un assertion permite probar nuestras asunciones durante el desarrollo de una aplicación, mientras que en tiempo de despliegue, el código del assertion literalmente “se evapora”. Esto permite que no tengamos que preocuparnos de eliminar código de prueba o depuración. A continuación, reescribimos el código del metodoA() para validar que el argumento no es negativo:

```
private void metodoA(int num) {
    assert (num>=0); // lanza un AssertionError
    // Si el test no es verdadero
    useNum(num + x);
}
```

El uso de assertions no solo permite que nuestro código sea más limpio y confiable, sino también que se encuentran inactivos. A menos que específicamente los activemos, el código de ejemplo se ejecutaría tal como se muestra a continuación:

```
private void metodoA(int num) {
    useNum(num + x) ; // Hemos probado esto;
    // logica a ejecutarse, asumiendo que num es mayor que 0
}
```

Los assertions trabajan de manera muy simple. Siempre asumiremos que algo es verdadero. Si se cumple esta asunción, no tendremos problema alguno, el código se ejecutará con normalidad. Pero si nuestra asunción se hace falsa, entonces un objeto **AssertionError** será lanzado (no debemos tratar de gestionar este tipo de error). Esto permitirá corregir la lógica que generó el problema.

Los Assertions pueden ser: muy simples o simples:

#### Muy simples:

```
private void doStuff()
    assert (y > x);
    // mas codigo asumiendo que y es mayor que x
}
```

#### Simple:

```
private void doStuff() {
    assert (y > x): "y es " + y + " x es " + x;
    // mas codigo asumiendo que y es mayor que x
}
```

La diferencia entre los dos ejemplos previos es que la versión simple agrega una segunda expresión, separada de la primera (expresión booleana) por dos puntos. Esta cadena es agregada a la pila de seguimiento. Ambas versiones lanzan un **AssertionError**, pero la versión simple nos proporciona un poco más de ayuda, mientras que la versión muy simple, sólo nos dice que nuestra asunción falló.

#### Importante:

En la práctica, los assertions son habilitados, generalmente, cuando una aplicación está siendo probada o depurada, pero deshabilitados cuando la aplicación ha sido desplegada. Los assertions continuarán como parte del código, aunque serán ignorados por la JVM. Si tenemos una aplicación desplegada que empieza a tener un comportamiento anómalo, podremos siempre seleccionar habilitar los assertions para poder realizar pruebas adicionales.

#### 1.1.1 Reglas de expresión de Assertions

Los Assertions pueden tener una o dos expresiones, dependiendo de si usamos la “versión simple” o la “versión muy simple”. La primera expresión siempre debe corresponder a un valor booleano. El objetivo principal es asumir una prueba, lo cual significa que estamos asumiendo que una prueba es verdadera. Si esto se cumple, no tendremos ningún problema, de lo contrario, obtendremos un **AssertionError**.

La segunda expresión, usada solo con la versión simple de una sentencia **assert**, puede ser cualquier cosa que resulte en un valor.

Recordemos que la segunda expresión es usada para generar un mensaje de tipo String que mostrará en la pila de seguimiento información adicional de depuración. Esto funciona de manera similar a una sentencia `System.out.println()` a la cual le enviamos un tipo primitivo u `Object`, que será convertido a su representación String. De ser susceptible de ser convertido en un valor.

El código presentado a continuación muestra ejemplos de usos correctos e incorrectos de expresiones dentro un `assertion`. Recordemos que la segunda expresión es solo usada con la versión simple de `assertions`, siendo su objetivo proporcionarnos un mayor nivel de detalle para efectos de depuración:

```
void noReturn { }

int aReturn() { return 1; }

void go() {
    int x = 1;
    boolean b = true;

    // sentencias legales para el uso de assertions
    assert (x == 1);
    assert(b);
    assert true;
    assert(x == 1) : x;
    assert(x == 1) : aReturn();
    assert(x == 1) : new ValidAssert();

    // sentencias ilegales en el uso de assertions
    assert(x = 1); // ninguna es un valor booleano
    assert(x);
    assert 0;
    assert(x == 1) : ; // ninguna retorna un valor
    assert(x == 1) : noReturn();
    assert(x == 1) : ValidAssert va;
}
```

## 1.2 Habilitando el uso de Assertions

### Identificador vs. Palabra clave

Previamente a la versión 1.4, podríamos haber escrito código como el mostrado a continuación:

```
int assert = getInitialValue();
if (assert == getActualResult()) {
    // hacemos algo
}
```

En el ejemplo anterior, la palabra **assert** es utilizada como un identificador. Sin embargo, a partir de la **versión 1.4** `assert` es palabra clave; por lo tanto, ya no podrá ser utilizada normalmente como un identificador.

Podemos utilizar la palabra `assert` como palabra clave o identificador, pero no ambas a la vez.

#### Para tener en cuenta:

Si estamos utilizando un compilador java 1.4 y requerimos emplear `assert` como palabra clave (es decir, estamos tratando de **"afirmar"** algo en nuestro código), entonces, debemos habilitar explícitamente el uso de assertions en tiempo de compilación:

```
javac -source 1.4 com/geeksanonymous/TestClass.java
```

Podemos leer el código previo de la siguiente manera: "Compila la clase `TestClass`, en el directorio `com/geeksanonymous`, y hazlo bajo el esquema de java 1.4, donde `assert` es una palabra clave".

#### Java version 5 o superior y el commando `javac`

El compilador de la versión 5 de java usa `assert` como palabra clave por defecto. A menos que indiquemos lo contrario, el compilador generará un mensaje de error si encuentra la palabra `assert` utilizada como un identificador. Es posible indicar al compilador que compile un código específico bajo el esquema previo a la version 1.4. A continuación, un ejemplo:

```
javac -source 1.3 OldCode.java
```

El compilador generará **"warnings"** cuando descubra la palabra `assert` como identificador, pero el código compilará y se ejecutará correctamente.

Si indicamos al compilador que nuestro código corresponde a la versión 1.4 o superior:

```
javac -source 1.4 NotQuiteSoOldCode.java
```

En este caso, el compilador generará **errores** cuando detecte que la palabra `assert` ha sido usada como un identificador.

Si queremos indicar a java 5 que queremos usar assertions, podemos hacer una de las siguientes tres cosas: omitir la opción `-source`, la cual es la default, o agregar `-source 1.5` o `-source 5`.

Si queremos usar `assert` como identificador, debemos compilar usando la opción `-source 1.3`. A continuación, se muestra un resumen del comportamiento de Java 5 o superior cuando utilizamos `assert` como identificador o palabra clave.

Usando Java 5 o superior para compilar código que use assert como identificador y palabra clave		
Línea de comando	Si assert es un identificador	Si assert es una palabra clave
javac -source 1.3 TestAsserts.java	Código compila con warnings.	Falla de compilación.
javac -source 1.4 TestAsserts.java	Falla de compilación.	Código compila.
javac -source 1.5 TeotAsserts.Java	Falla de compilación.	Código compila.
javac -source 5 TestAsserts.java	Falla de compilación.	Código compila.
javac TestAsserts.java	Falla de compilación.	Código compila.

### Assertions en tiempo de ejecución

Una vez que hemos escrito nuestro código correctamente, utilizando assert como palabra clave, podemos seleccionar habilitar o deshabilitar nuestras assertions en tiempo de ejecución. Recordemos que el uso de assertions en tiempo de ejecución está deshabilitado por defecto.

### Habilitando assertions en tiempo de ejecución

Podemos hacerlo de la siguiente manera:

```
java -ea com.geelcsanonymous.TestClass

o

java -enableassertions com.geeksanonymous.TestClass
```

Las líneas previas le indicant a la JVM que se ejecute con el control de assertions habilitado.

### Deshabilitando assertions en tiempo de ejecución

Las líneas de comando que nos permiten deshabilitar el uso de assertions son las siguientes:

```
java -da com.geeksanonymous.TestClass

o

java -disableassertions com.geeksanonymous.TestClass
```

## Habilitando y deshabilitando assertions de manera selectiva

Para el ejemplo mostrado a continuación:

```
java -ea -da:com.geeksanonymous.Foo
```

Indicamos a la JVM habilitar el uso de assertions en general, pero deshabilitarlo en la clase `com.geeksanonymous.Foo`. Podemos hacer lo mismo para un paquete, tal como se muestra a continuación:

```
java -ea -da:com.geeksanonymous...
```

A continuación, se muestran algunos ejemplos de líneas de commando para el uso de assertions:

Ejemplo línea comando	Significado
java -ea	Habilita el uso de assertions.
java -enableassertions	
java -da	Deshabilita el uso de assertions.
Java -disableassertions	
java -ea:com.foo.Bar	Habilita assertions en la clase <code>com.foo.Bar</code> .
java -ea:com.foo...	Habilita assertions en el paquete <code>com.foo</code> y cualquiera de sus subpaquetes.
java -ea -dsa	Habilita assertions en general, pero deshabilita assertions en las clases del sistema.
java -ea -da:com.foo...	Habilita assertions en general, pero deshabilita assertions en el paquete <code>com.foo</code> y cualquiera de sus subpaquetes.

### 1.3 Uso apropiado de Assertions

No todos los usos legales de assertions son considerados apropiados. Por ejemplo, no es correcto gestionar usando TRY – CATCH la falla de un assertion. Sintácticamente, la clase `AssertionError` es una subclase de la clase `Throwable`, por lo tanto, podría ser atrapada. Sin embargo, no debemos hacerlo. Es por ello que la clase `AssertionError` no proporciona acceso alguno a la clase que generó el error.

Es la documentación oficial de SUN la que nos servirá de guía para determinar si estamos haciendo un uso apropiado o no de un assertion. Por ejemplo, no debemos usar Assertions para validar los argumentos de un método público:

A continuación se muestra un ejemplo de uso inapropiado de assertions:

```
public void doStuff(int x) {
    assert (x > 0);           // inapropiado !
    // do things with x
}
```

### Importante para el examen de Certificación:

Si vemos la palabra "apropiado" en el examen, no debemos confundirla con "legal". Apropiado siempre se refiere a la manera en la cual se supone algo será utilizado de acuerdo con las mejores prácticas oficialmente establecidas por SUN. Si vemos la palabra "correcto" dentro del contexto de los assertions, debemos también asumir que correcto se refiere a cómo los assertions deben ser usados y no a cómo, legalmente, éstos podrían ser válidos.

Un método público puede ser invocado por código que no controlamos o código que nunca hemos visto. Dado que los métodos públicos son parte de nuestra interface con el mundo exterior, no podremos realmente asumir o afirmar cierto comportamiento.

Si necesitamos validar los argumentos de un método público, lo más probable es que usemos excepciones para lanzar, por ejemplo, un objeto de tipo `IllegalArgumentException` si el valor enviado a un método público es inválido.

### Usemos assertions para validar argumentos de un método privado

Si escribimos un método privado, se entiende que controlamos el código que lo invoca. Bajo esta premisa, cuando asumimos que el código que invoca a este método es correcto, podemos comprobarlo utilizando un assertion:

```
private void doMore(int x) {  
    assert (x > 0);  
    // Hacemos cosas con x  
}
```

La única diferencia entre el ejemplo anterior y el precedente, es el modificador acceso (`private`). Por lo tanto, forzamos las restricciones sobre los argumentos en los métodos privados, pero no debemos hacerlo sobre métodos públicos. Sintácticamente, podríamos hacerlo; sin embargo, sabemos que las buenas prácticas nos indican que no debemos.

### No usar Assertions para validar argumentos de línea de comando

Si nuestra aplicación requiere argumentos de línea de comando, es muy probable que usemos el mecanismo de gestión de excepciones para validar estos argumentos.

Podemos usar Assertions, aún en métodos públicos, para verificar casos que sabemos nunca, supondremos que pasen.

Esto puede incluir bloques de código que sabemos nunca serán alcanzados, incluyendo la cláusula **default** de una sentencia `switch` tal como se muestra a continuación:



```
switch(x) {
  case 1: y = 3;
  case 2: y = 9;
  case 3: y = 27;
  default: assert false; // Nunca deberíamos llegar a
                        // ejecutar esta línea de código
}
```

Si asumimos que un código en particular no será alcanzado, como en el ejemplo previo, en el que x debería tomar como valores posibles 1, 2 ó 3, podemos, entonces, usar “assert false” para causar un AssertionError a ser lanzado inmediatamente si alguna vez logramos ejecutar dicho código. En el ejemplo que utiliza la cláusula switch, no estamos haciendo un test booleano, por el contrario, estamos asegurando que nunca se ejecutará la cláusula default; es por ello que, de ejecutarse default, automáticamente se genera un error.

**No usar assertions que pueden causar efectos secundarios.** A continuación se muestra un ejemplo:

```
public void doStuff() {
  assert (modifyThings());
  // continues on
}
public boolean modifyThings() {
  y = x++;
  return true;
}
```

### Importante:

Debemos aplicar la siguiente regla: un assertion debe dejar un programa en el mismo estado en que lo encontró antes de la ejecución del assertion. Recordemos que los assertions no siempre serán ejecutados, y nuestro código no debe de comportarse de manera distinta, dependiendo de si el uso de assertions está habilitado o no.

### Autoevaluación:

1. Seleccione TODAS las respuestas correctas:

- a) Es apropiado usar assertions para validar argumentos de métodos públicos.
- b) Es apropiado atrapar y gestionar errores generados por assertion.
- c) No es apropiado usar assertions para validar argumentos de línea de commando.
- d) Es apropiado usar assertions para generar alertas cuando alcanzamos código que nunca debería ser alcanzado.
- e) No es apropiado para un assertions cambiar el estado de un programa.

### Ejercicio de Aplicación:

1. Implemente la siguiente interface gráfica utilizando un método privado para la creación de las etiquetas sobre el Panel. El método privado debe recibir los nombres de ambas etiquetas y, utilizando un **assertion**, asumirá que los valores nunca serán nulos.



# Resumen

- 📖 Las **Assertions** nos proporcionan una manera adecuada de probar nuestras hipótesis durante el desarrollo y la depuración de nuestros programas.
- 📖 Las Assertions son habilitadas generalmente durante la etapa de pruebas pero deshabilitadas durante el despliegue definitivo de la aplicación.
- 📖 Podemos usar un assertion como palabra clave o un identificador, pero no ambos juntos. Para compilar código antiguo que use un assertion como identificador (por ejemplo, un nombre de método), debemos usar el flag `-source 1.3` para el comando `javac`.
- 📖 Los Assertions están deshabilitados en tiempo de ejecución por defecto. Para habilitarlos, debemos usar los flags de línea de comando: `-ea` o `-enableassertions`.
- 📖 Selectivamente podemos deshabilitar assertions usando los flags `-da` o `-disableassertions`.
- 📖 No debemos usar assertions para validar argumentos de un método público.
- 📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.
  - 🔗 <http://www.programacion.com/java/articulo/assertos/>  
Aquí encontrará un interesante artículo en castellano sobre el uso de assertions en java.
  - 🔗 <http://java.sun.com/developer/technicalArticles/JavaLP/assertions/>  
En esta página, hallará documentación detallada y oficial sobre las principales características y mejores prácticas para el uso de assertions en Java.



**UNIDAD DE  
APRENDIZAJE**

**2**

**SEMANA**

**3**

## **GESTIÓN DE CADENAS, FLUJOS DE ENTRADA Y SALIDA, FORMATEO Y PARSEO EN JAVA**

### **LOGRO DE LA UNIDAD DE APRENDIZAJE**

- Al término de la unidad, los alumnos crearán aplicaciones utilizando de manera individual y combinada las clases String, StringBuffer, StringBuilder, clases de manejo de flujos del paquete java.io, clases de formateo y parseo de datos, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo

### **TEMARIO**

- La clase String – Definición y gestión de memoria
- Principales métodos de la clase String
- Las clases StringBuffer y StringBuilder
- Principales métodos de las Clases StringBuffer y StringBuilder
- Creación y gestión de las clases:
  - File, FileReader, FileWriter
  - BufferedReader, BufferedWriter
  - PrintWriter
- Serialización de objetos:
  - ObjectInputStream, ObjectOutputStream
- Creación y gestión de las clases:
  - Date, Calendar
  - DateFormat, NumberFormat
  - Locale
- Regular Expression en Java, Tokens y delimitadores

### **ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucran el uso de las clases String, StringBuffer y StringBuilder.

## 1. La clase String

### 1.1 Métodos importantes de la clase String

Se muestran, a continuación, los métodos más usados en la clase String:

**charAt()**. Retorna el character indicado en el índice especificado.

**concat()** Agrega un String al final de otro. (También funciona +)

**equalsIgnoreCase()** Determina la igualdad de dos Strings sin importar las mayúsculas y minúsculas.

**length()** Retorna el número de caracteres de un String.

**replace()** Reemplaza las ocurrencias de un caracter por otro.

**substring()** Retorna parte de un String

**toLowerCase()** Retorna un String con los caracteres en mayúscula convertidos a minúscula.

**toString()** Retorna el valor de un String.

**trim()** Elimina los espacios en blanco al final de un String.

A continuación se muestran algunos ejemplos de aplicación:

```
1) String x = "airplane";  
System.out.println( x.charAt(2) );    // se obtiene 'r'
```

```
2) String x = "taxi";  
System.out.println( x.concat(" cab") ); // obtendremos "taxi cab"
```

```
3) String x = "library";  
System.out.println( x + " card" );    // obtenemos "library card"
```

```
4) String x = "Atlantic";  
x += " ocean"  
System.out.println( x );              // la salida es "Atlantic ocean"
```

En el ejemplo previo, el valor de x en realidad no cambia. El operador += es un operador de asignación, por lo que x+= " ocean" en realidad está creando una nueva cadena: "Atlantic ocean", la cual es asignada a la variable x. Después de la asignación, la cadena original de x : "Atlantic" es abandonada.

```
5) String x = "oxoxoxox";
```

```
System.out.println( x.replace('x', 'X') ); // obtenemos "oXoXoXoX"
```

### Importante:

Tenga en cuenta que los arreglos tienen un atributo (no método) llamado `length`. A continuación, se muestran dos ejemplos que generan errores de compilación:

```
String x = "test";
System.out.println( x.length ); // error
```

o

```
String[] x = new String[3];
System.out.println( x.length() ); // error
```

```
6) String x = "0123456789"; //En el ejemplo el valor de cada caracter
    // es el mismo de su índice
System.out.println( x.substring(5) ); // obtenemos "56789"
System.out.println( x.substring(5, 8)); // obtenemos "567"
```

## 1.2 La Clase String y el manejo de memoria

Uno de los objetivos de un buen lenguaje de programación es hacer un uso eficiente de la memoria. Java (la JVM), en atención a este objetivo cuenta con una zona especial de memoria llamada: "String constant pool." Cuando el compilador encuentra un literal String, verifica el pool para validar si ya existe una cadena idéntica, si la encuentra, la referencia al nuevo literal es dirigida a dicha cadena evitando la creación de un nuevo objeto String.

### Creación de nuevos Strings

A continuación, se muestran dos ejemplos en los que se asume no existen Strings idénticos en el pool.

```
String s = "abc"; // crea un objeto String y una variable de referencia.
```

En este ejemplo, el objeto "abc" irá al pool y s lo referenciará.

```
String s = new String("abc"); // crea dos objetos,
    // y una variable de referencia
```

En este otro ejemplo, dado que usamos la palabra clave `new`, Java crea un nuevo objeto String en la zona normal de memoria (no en el pool) y s la referencia. Adicionalmente, el literal "abc" será ubicado en el pool.

## 2. Las clases StringBuffer y StringBuilder

Las clases `java.lang.StringBuffer` y `java.lang.StringBuilder` deben ser usadas cuando sabemos haremos muchas modificaciones a las cadenas de caracteres. Los objetos de tipo `String` son inmutables; por lo tanto, cuando hacemos muchas manipulaciones sobre los objetos `String`, estamos generando demasiados objetos `String` abandonados en el pool de `Strings`. Por otro lado, los objetos de tipo `StringBuffer` y `StringBuilder` pueden ser modificados una y otra vez sin generar objetos abandonados de tipo `String`.

En la práctica, un uso común de las clases `StringBuffers` y `StringBuilders` es para operaciones de Lectura/Escritura de archivos, cuando grandes cantidades de caracteres deben ser gestionadas por el programa. En estos casos, los bloques de caracteres son manejados como unidades, los objetos de tipo `StringBuffer` son ideales para manejar bloques de data, transmitirla y reutilizar la misma zona de memoria para manejar el siguiente bloque de data.

### 2.1 StringBuffer versus StringBuilder

La clase `StringBuilder` fue agregada en la versión 5 de java. Tiene exactamente las mismas características que la clase `StringBuffer`, solo que no es un hilo de ejecución seguro (thread safe). En otras palabras, sus métodos no están sincronizados.

Sun recomienda que usemos `StringBuilder` en vez de `StringBuffer` siempre que sea posible, porque `StringBuilder` se ejecutará más rápido. Fuera del tema de sincronización, cualquier cosa que podamos decir de los métodos de la clase `StringBuilder` la podremos decir de `StringBuffer` y viceversa.

En el presente capítulo, hemos visto varios ejemplos que evidencian la inmutabilidad de los objetos de tipo `String`:

```
String x = "abc";
x.concat("def");
System.out.println("x = " + x); // la salida es "x = abc"
```

Dado que no hubo ninguna asignación usando el operador `new`, el Nuevo objeto `String` creado con el método `concat()` fue abandonada instantáneamente. Ahora veamos el siguiente ejemplo:

```
String x = "abc";
x = x.concat("def");
System.out.println("x = " + x); // la salida es "x = abcdef"
```

En el ejemplo previo, hemos creado un nuevo objeto `String`; sin embargo, el antiguo objeto `x` con el valor `abc`, ha sido perdido en el pool; por lo tanto, estamos desperdiciando memoria. Si estuviéramos usando un objeto de tipo `StringBuffer` en vez de un `String`, el código se vería de la siguiente manera:

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb); // la salida es "sb = abcdef"
```



Todos los métodos del objeto `StringBuffer` operan sobre el valor asignado al objeto del método que invocamos. Por ejemplo, una llamada a `sb.append("def");` implica que estamos agregando "def" al mismo objeto de tipo `StringBuffer sb`. A continuación, se muestra un ejemplo:

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb );           // la salida es "fed --- cba"
```

Debemos notar que en los dos ejemplos previos, solo hemos realizado una llamada al operador `new` y no hemos creado tampoco ningún objeto `String` adicional. Cada ejemplo solo necesitó ejecutar un único objeto `StringXXX`.

## 2.2 Principales métodos de las clases `StringBuffer` y `StringBuilder`

**public synchronized `StringBuffer` append(`String` s).** Este método actualiza el valor del objeto que lo invocó. El método puede tomar argumentos de diferentes tipos tales como `boolean`, `char`, `double`, `float`, `int`, `long`, entre otros, siendo el más usado el argumento de tipo `String`.

```
StringBuffer sb = new StringBuffer("set ");
sb.append("point");
System.out.println(sb);           // la salida es "set point"
StringBuffer sb2 = new StringBuffer("pi = ");
sb2.append(3.14159f);
System.out.println(sb2);         // la salida es "pi = 3.14159"
```

**public `StringBuilder` delete(int start, int end)** retorna un objeto `StringBuilder` y actualiza el valor del objeto `StringBuilder` que invocó al método. En ambos casos una subcadena es removida del objeto original. El índice de inicio de la subcadena a eliminar es definido por el primer argumento (el cual está basado en cero), y el índice final de la subcadena a ser removida es definido por el segundo argumento (el cual está basado en 1). A continuación, se muestra un ejemplo:

```
StringBuilder sb = new StringBuilder("0123456789");
System.out.println(sb.delete(4,6)); // la salida es "01236789"
```

### Tip:

Dado que los objetos `StringBuffer` son modificables, el código mostrado a continuación se comportará de manera diferente que un código similar aplicado sobre objetos `String`:

```
StringBuffer sb = new StringBuffer("abc");
```

```
sb.append("def");  
System.out.println( sb );
```

En este caso, la salida sera : "abcdef"

**public StringBuilder insert(int offset, String s)** retorna un objeto `StringBuilder` y actualiza el valor del objeto `StringBuilder` que invocó al método. En ambos casos el string pasado en el segundo argumento es insertado en el objeto original sobre la base del primer argumento (el desplazamiento está basado en cero). Podemos pasar otros tipos de datos como segundo argumento (boolean, char, double, float, int, long, etc; sin embargo, el tipo `String` es el más usado).

```
StringBuilder sb = new StringBuilder("01234567");  
sb.insert(4, "---");  
System.out.println( sb );      // la salida es "0123---4567"
```

**public synchronized StringBuffer reverse()** This method returns a `StringBuffer` object and updates the value of the `StringBuffer` object that invoked the method call. En ambos casos, los caracteres en el objeto `StringBuffer` son invertidos.

```
StringBuffer s = new StringBuffer("A man a plan a canal Panama");  
sb.reverse();  
System.out.println(sb); // salida : "amanaP lanac a nalp a nam A"
```

**public String toString()** Este método retorna el valor del objeto `StringBuffer` object que invocó al método como un `String`:

```
StringBuffer sb = new StringBuffer("test string");  
System.out.println( sb.toString() ); // la salida es "test string"
```

Esto se cumple para `StringBuffers` y `StringBuilders`: a diferencia de los objetos de tipo `String`, los objetos `StringBuffer` y `StringBuilder` pueden ser cambiados (no son inmutables).

### Importante:

Es común utilizar en Java métodos encadenados ("chained methods"). Una sentencia con métodos encadenados tiene la siguiente apariencia:

```
result = method1().method2().method3();
```

En teoría, cualquier número de métodos pueden estar encadenados tal como se muestra en el esquema previo, aunque de manera típica encontraremos como máximo tres.

Sea **x** un objeto invocando un segundo método. Si existen solo dos métodos encadenados, el resultado de la llamada al segundo método será la expresión final resultante.

Si hay un tercer método, el resultado de la llamada al segundo método será usado para invocar al tercer método, cuyo resultado será la expresión final resultante de toda la operación:

```
String x = "abc";
String y = x.concat("def").toUpperCase().replace('C','x');
//metodos encadenados
System.out.println("y = " + y); // el resultado es "y = ABxDEF"
```

El literal `def` fue concatenado con `abc`, creando un `String` temporal e intermedio (el cual pronto sera descartado), con el valor `abcdef`. El método `toUpperCase()` creó un nuevo `String` temporal (que también pronto sera perdido) con el valor `ABCDBF`. El método `replace()` creó un `String` final con el valor `ABxDEF` el cual es referenciado por la variable `y`.

### Autoevaluación:

1. Seleccione todas las afirmaciones válidas sobre las clases `String`, `StringBuilder`, y `StringBuffer`:

- a) Las tres clases tienen el método `length()`.
- b) Los objetos de tipo `StringBuffer` son thread-safe.
- c) Las tres clases han sobrecargado sus métodos `append()`.
- d) El operador `+` ha sido sobrecargado para las tres clases.
- e) En conformidad con la API java, `StringBuffer` será más rápido que `StringBuilder` para la mayoría de implementaciones.
- f) El valor de una instancia de cualquiera de estas tres clases puede ser modificado a través de de varios métodos de la API java.

**Ejercicio de aplicación:**

1. Dada la siguiente clase:

```
class Polish {  
    public static void main(String[] args) {  
        int x = 4 ;  
        StringBuffer sb = new StringBuffer("..fedcba");  
        sb.delete(3,6);  
        sb.insert(3, "az");  
        if(sb.length() > 6) x = sb.indexOf("b");  
        sb.delete((x-3), (x-2));  
        System.out.println(sb);  
    }  
}
```

¿Cuál será el resultado de su ejecución?

- a) .faza                      b) .fzba
- c) ..azba                    d) .fazba
- e) ..fezba                   f) Compilation fails.
- g) An exception is thrown at runtime.

# Resumen

- 📖 Los objetos String son inmutables; las variables que referencian a Strings, no.
- 📖 Si creamos un nuevo String sin asignarlo, este objeto se habrá perdido para nuestro programa.
- 📖 Los objetos String tienen un método llamado `length()`; los arreglos tienen un atributo llamado `length`.
- 📖 Los métodos de un objeto `StringBuilder` deben ejecutarse más rápido que los de un objeto `StringBuffer`.
- 📖 El método `equals()` de la clase `StringBuffer` no está sobrescrito; por lo tanto, no compara valores.
- 📖 Los métodos encadenados son evaluados de derecha a izquierda.
- 📖 Los métodos más usados de la clase String son:  
`charAt()`, `concat()`, `equalsIgnoreCase()`, `length()`, `replace()`, `substring()`, `toLowerCase()`, `toString()`, `toUpperCase()`, and `trim()`.
- 📖 Los métodos más usados de `StringBuffer` son `append()`, `delete()`, `insert()`, `reverse()`, and `toString()`.
- 📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.
  - 🔗 <http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/clases1/string.htm>  
Aquí encontrará un interesante artículo en castellano sobre el uso de Strings en java.
  - 🔗 <http://www.java-examples.com/java-stringbuffer-examples>  
En esta página, hallará documentación con ejemplos de uso de la clase `StringBuffer`.



**UNIDAD DE  
APRENDIZAJE**

**2**

**SEMANA**

**4**

## **GESTIÓN DE STRINGS, FLUJOS DE ENTRADA Y SALIDA, FORMATEO Y PARSEO EN JAVA**

### **LOGRO DE LA UNIDAD DE APRENDIZAJE**

- Al término de la unidad, los alumnos crearán aplicaciones utilizando de manera individual y combinada las clases String, StringBuffer, StringBuilder, clases de manejo de flujos del paquete java.io, clases de formateo y parseo de datos, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo

### **TEMARIO**

- La clase String – Definición y gestión de memoria
- Principales métodos de la clase String
- Las clases StringBuffer y StringBuilder
- Principales métodos de las clases StringBuffer y StringBuilder
- Creación y gestión de las clases:
  - File, FileReader, FileWriter
  - BufferedReader, BufferedWriter
  - PrintWriter
- Serialización de objetos
  - ObjectInputStream, ObjectOutputStream
- Creación y gestión de las clases:
  - Date, Calendar
  - DateFormat, NumberFormat
  - Locale
- Regular Expression en Java, Tokens y Delimitadores

### **ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucren el uso de las clases File, FileReader y FileWriter así como de serialización de objetos.

## 1. Archivos y flujos de entrada y salida

### 1.1 Creación de archivos usando la clase File

Los objetos de tipo File son usados para representar los archivos actuales (pero no los datos en cada archivo) o directorios que existen en un disco físico dentro de una computadora. A continuación, se muestra un ejemplo:

```
import java.io.*;

class Writer1 {
    public static void main(String [] args) {
        File file = new File("fileWrite1.txt"); // Aún no existe un archivo
    }
}
```

Si compilamos y ejecutamos este programa, cuando observemos el contenido de nuestro directorio, nos percataremos que no existe aún un archivo llamado fileWrite1.txt. Cuando creamos una instancia de la clase File, no estamos creando un archivo, sino sólo un nombre de archivo. Una vez que tenemos un objeto File, existen varias maneras de crear un archivo. A continuación, algunos ejemplos:

```
import java.io.*;

class Writer1 {
    public static void main(String [] args) {
        try { // advertencia: es posible se generen excepciones

            boolean newFile = false;
            File file = new File // es solo un objeto
                ("fileWritel.txt");
            System.out.println(file.exists()); // se busca por un archivo real
            newFile = file.createNewFile(); // es posible se cree un archivo
            System.out.println(newFile);
            System.out.println(file.exists()); // buscamos nuevamente por un
                // archivo
        } catch(IOException e) { }
    }
}
```

Obtendremos la siguiente salida:

```
false
true
true
```

y también crearemos un archivo vacío en nuestro directorio actual. Si ejecutamos el código por segunda vez, obtendremos:



```
true
false
true
```

Notemos que en el código previo hemos utilizado dos métodos de la clase File:

**boolean exists()**. Este método retorna verdadero si logra encontrar el archivo actual.

**boolean createNewfile()**. Este método crea un nuevo archivo siempre y cuando aún no exista.

## 1.2 Uso de las clases FileWriter y FileReader

En la práctica, es poco probable que utilicemos directamente las clases FileWriter y FileReader, sin embargo, mostraremos algunos ejercicios básicos:

```
import java.io.*;

class Writer2 {
    public static void main(String [] args) {
        char[] in = new char[50];        // para almacenar el ingreso de datos
        int size = 0;
        try {
            File file = new File(          // es solo un objeto
                "fileWrite2.txt");
            FileWriter fw =
                new FileWriter(file);    // creamos un archivo y un objeto
            FileWriter

            fw.write("howdy\nfolks\n");    // escribimos caracteres en el archivo
            fw.flush();                    // hacemos un flush antes de cerrar el flujo
            fw.close();                    // cerramos el archivo una vez que hemos
                                         // terminado

            FileReader fr =
                new FileReader(file);    // creamos un objeto FileReader

            size = fr.read(in);            // leemos el archivo completo
            System.out.print(size + " ");  // cuántos bytes leímos
            for(char c : in)                // visualizamos el arreglo
                System.out.print(c);
            fr.close();                    // cerramos el flujo
        } catch(IOException e) {}
    }
}
```

La ejecución de este código producirá la siguiente salida:

```
12 howdy
folks
```

Se detallan, a continuación, las primeras líneas de nuestro código:

`FileWriter fw = new FileWriter(file)`, produjo tres cosas:

- Se crea una variable de referencia (fw) a un objeto `FileWriter`.
- Se crea el objeto `FileWriter` el cual es asignado a fw.
- Se crea un archivo vacío en disco (lo cual podríamos verificar).

Cuando usemos archivos, sea para leerlos o escribir sobre ellos, siempre debemos de invocar al método `close()`. La lectura y escritura de archivos implica el uso de recursos costosos y limitados de nuestro sistema operativo, por lo que al invocar al método `close()` conseguimos liberar dichos recursos.

#### Para tomar en cuenta:

En el ejemplo, al leer datos, los estamos almacenando en un arreglo de caracteres. Hemos declarado el tamaño de este arreglo, y podríamos tener problemas si los datos leídos superan el tamaño del arreglo. También, podríamos leer carácter por carácter, pero este procedimiento sería demasiado costoso. Es por ello que se recomienda usar clases tales como `BufferedWriter` o `BufferedReader` en combinación con `FileWriter` o `FileReader`.

### 1.3 Combinación de clases I/O

El uso combinado de clases en java es llamado en algunos casos “wrapping” y en otros, encadenamiento. El paquete `java.io` contiene aproximadamente 50 clases, 10 interfaces, y 15 excepciones. Se muestra a continuación un cuadro resumen de las mismas.

Paquete java.io	Hereda de	Argumentos de sus constructores principales	Métodos clave
File	Object	File, String  String  String, String	<code>createNewFile()</code>  <code>delete()</code>  <code>exists()</code>  <code>isDirectory()</code>  <code>isFile()</code>  <code>list()</code>  <code>mkdir()</code>  <code>renameTo()</code>

FileWriter	Writer	File	close()
		String	flush()
			write()
BufferedWriter	Writer	Writer	close()
			flush()
			newLine()
			write()
PrintWriter	Writer	File (a partir de Java 5)	close()
		String (a partir de Java 5)	flush()
		OutputStream	format()
		Writer	printf()
			print(), println()
			write()
FileReader	Reader	File	read()
		String	
BufferedReader	Reader	Reader	read()
			readLine()

### Escritura de datos en un archivo:

Podemos escribir datos en un archivo y leerlo luego cargándolo en memoria de diferentes maneras. Una de ellas, que podría ser más cómoda, es la que nos proporcionaría el método `newLine()` de la clase `BufferedWriter`. Suena más cómodo que tener que incrustar manualmente un separador de línea (por cada línea). Si continuamos investigando, podremos notar también que la clase `PrintWriter` tiene un método llamado `println()`. Al parecer, esta sería la forma más sencilla de escribir líneas de flujos de caracteres.

Podemos crear un objeto `PrintWriter` con sólo tener un objeto de tipo `File`:

```
File file = new File("fileWrite2.txt"); // creamos un archivo
```

```
PrintWriter pw = new PrintWriter(file); // pasamos el archivo al
//constructor de la clase PrintWriter
```

Debemos tomar en cuenta que es a partir de Java 5 que la clase `PrintWriter` recibe, en su constructor, un `String` o un objeto de tipo `File`. En versiones previas, esto no era posible, por lo que manejar archivos con un objeto `PrintWriter` requería de algunos artificios. Se muestra a continuación un ejemplo para resolver este inconveniente (formal habitual de programar en java 1.4).

```
File file = new File("fileWrite2.txt"); // creamos un objeto File
FileWriter fw = new FileWriter(file); // creamos un FileWriter
                                // que enviará su salida hacia un archivo

PrintWriter pw = new PrintWriter(fw); // creamos un PrintWriter
                                // que enviará su salida a un Writer

pw.println("howdy");           // escribimos datos
pw.println("folks");
```

### Lectura de datos de un archivo:

¿Cuál será la manera más cómoda de leer archivos? A continuación, mostramos un ejemplo que nos muestra una de las maneras más sencillas:

```
File file =
    new File("fileWrite2.txt"); // creamos un objeto File y
                                // lo abrimos "fileWrite2.txt"
FileReader fr =
    new FileReader(file); // creamos un FileReader para obtener
                                // datos del archivo'
BufferedReader br =
    new BufferedReader(fr); // creamos un BufferedReader para
                                // obtener datos del Reader
String data = br.readLine(); // leemos datos
```

## 1.4 Trabajo con archivos y directorios

Analicemos la siguiente sentencia:

```
File file = new File("foo");
```

La sentencia previa siempre crea un objeto `File`, para luego hacer lo siguiente:

Si "foo" no existe, ningún archivo es creado.

Si "foo" existe, el objeto `File` referencia al archivo existente.

Notemos que la sentencia `File file = new File ("foo");` nunca crea automáticamente un archivo, existen dos maneras de hacerlo:

Invocando al método `createNewFile()`. Por ejemplo:

```
File file = new File("foo"); // no existe aún el archivo
file.createNewFile();        // crea el archivo el cual es referenciado por el
                             // objeto file
```

Creación de un Reader, Writer o un Stream. Específicamente, si creamos objetos de tipo `FileReader`, `FileWriter`, `PrintWriter`, `FileInputStream` o `FileOutputStream`, automáticamente crearemos un archivo, a menos que ya exista previamente uno:

```
File file = new File("foo"); // no existe aún un archivo
PrintWriter pw =
    new PrintWriter(file); // creamos un objeto PrintWriter
                           // y creamos un archivo, "foo" el cual es referenciado por
                           // el objeto file y es asignado al objeto pw
```

### Importante:

El tratamiento de un directorio es esivalente al tratamiento de archivos que hemos analizado.

## 2. Serialización

Imaginemos que queremos guardar el estado de uno o más objetos. La serialización en Java permite que grabemos objetos y todas sus variables de instancia, a menos que explícitamente identifiquemos una variable como `transient`, lo que significa que su valor no será incluido como parte del estado de un objeto serializado.

### 2.1 Clases `ObjectInputStream` y `ObjectOutputStream`

Es posible que apliquemos serialización utilizando sólo dos métodos básicos: uno para serializar objetos y escribirlos en un flujo, y otro para leer el flujo y deserializar un objeto.

- `ObjectOutputStream.writeObject()` // serializa y escribe
- `ObjectInputStream.readObject()` // lee y deserializa

Las clases `java.io.ObjectOutputStream` y `java.io.ObjectInputStream` son consideradas clases de alto nivel, lo que significa que serán utilizadas junto con otras de más bajo nivel (haciendo wrapping) tales como `java.io.FileOutputStream` y `java.io.FileInputStream`. A continuación, se muestra un ejemplo básico de serialización:

```
import java.io.*;

class Cat implements Serializable { }

public class SerializeCat {
    public static void main(String[] args) {
        Cat c = new Cat(); /
        try {
            FileOutputStream fs = new FileOutputStream("testSer.ser");
            ObjectOutputStream os = new ObjectOutputStream(fs);
            os.writeObject(c);
            os.close();
        } catch (Exception e) { e.printStackTrace (); }

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

En el ejemplo anterior, la clase Cat implementa la interface serializable. Esta interface sirve como identificador, no es una interface que contenga métodos a implementar.

Creamos un nuevo objeto de tipo Cat (el cual implementa la interface serializable).

Serializamos el objeto c de tipo Cat invocando al método writeObject(). Para hacerlo, hemos creado un bloque try / catch dentro del cual se encuentra el código relacionado con las operaciones de entrada y salida, creamos un objeto de tipo FileOutputStream para escribir el archivo, luego envolvemos el objeto FileOutputStream con un ObjectOutputStream, esta es la clase que cuenta con el método específico que necesitamos. La invocación al método writeObject() ejecuta dos tareas: serializa el objeto y luego escribe el objeto serializado en un archivo.

Luego, deserializamos el objeto Cat invocando al método readObject(). Este método retorna un objeto de tipo Object, por lo que debemos de convertir el objeto deserializado a Cat.

## 2.2 Uso de las clases WriteObject y ReadObject

A continuación, se muestra un interesante ejemplo de sobreescritura de los métodos writeObject y ReadObject:

```
private void writeObject(ObjectOutputStream os) {
    // nuestro código para guardar las variables del Collar
}

private void readObject(ObjectInputStream os) {
    // Nuestro código para leer el estado del Collar, crear un nuevo
    // collar y asignárselo a Dog
}
```

En efecto, escribiremos métodos que tienen el mismo nombre que sus equivalentes en la clase ObjectXXXStream. El resultado final sería el siguiente:

```
class Dog implements Serializable {
    transient private Collar theCollar; // we can't serialize this
    private int dogSize;
    public Dog(Collar collar, int size) {
        theCollar = collar;
        dogSize = size;
    }
    public Collar getCollar() { return theCollar; }

    private void writeObject(ObjectOutputStream os) {
        // throws IOException { // 1
        try {
            os.defaultWriteObject(); // 2
            os.writeInt(theCollar.getCollarSize()); // 3
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void readObject(ObjectInputStream os) {
        // throws IOException { // 1
        try {
            os.defaultReadObject(); // 2
            theCollar = new Collar(os.readInt()); // 3
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```

    } catch (Exception e) { e.printStackTrace(); }
}

private void readObject(ObjectInputStream is) {
    // throws IOException, ClassNotFoundException { // 4
    try {
        is.defaultReadObject();           // 5
        theCollar = new Collar(is.readInt()); // 6
    } catch (Exception e) { e.printStackTrace(); }
    }
}

```

En el escenario planteado en el ejercicio anterior, por alguna razón no queremos serializar el objeto Collar, pero sí queremos serializar el objeto Dog. Para alcanzar nuestro objetivo, debemos implementar explícitamente los métodos `writeObject()` y `readObject()`. Al implementar estos métodos, le indicamos al compilador que si alguien invoca a los métodos `writeObject()` o `readObject()` relacionados con el objeto Dog, usaremos el código de estos métodos como parte de las operaciones normales de lectura y escritura.

Dado que los métodos relacionados con operaciones de entrada y salida pueden lanzar excepciones, debemos de lanzar éstas o, de lo contrario, gestionarlas. En el ejemplo, las excepciones son gestionadas.

Cuando invocamos al método `defaultWriteObject()`, indicamos a la JVM que ejecute el proceso normal de serialización para este objeto. Cuando implementamos el método `writeObject()`, generalmente, ejecutaremos el flujo normal de serialización y agregaremos, probablemente, algún proceso propio de lectura y/o escritura.

En el ejemplo, decidimos escribir un valor entero extra (el tamaño del collar) sobre el flujo que está creando el objeto serializado. Podemos escribir cosas extras antes y/o después de invocar al método `defaultWriteObject()`. Debemos tener cuidado al momento de recuperar la información del objeto, debemos hacerlo en el mismo orden que fue serializado.

#### **Para tener en cuenta:**

a) Es común preguntarnos, ¿Por qué no todas las clases Java son serializables?

¿Por qué la clase Object no es serializable? La respuesta es que existen en Java algunos componentes que no pueden ser serializados por que son específicos al entorno de ejecución (runtime). Componentes tales como flujos, hilos, objetos runtime, etc, incluso algunas clases propias de la interface de usuario. Debemos tener en cuenta que nosotros serializaremos, en general, objetos complejos.



b) Si serializamos un collection o un array, cada elemento debe de ser serializable. Un solo elemento que no sea serializable provocaría que el proceso de serialización falle. Notemos también que mientras las interfaces collection no son serializables, las clases concretas de collection sí lo son.

## 2.4 Serialización y variables estáticas

Debemos notar que la serialización la hemos aplicado solo sobre variables de instancia, no con variables estáticas. ¿Sería correcto guardar el valor de una variable estática como parte del estado de un objeto? ¿Es importante el valor de una variable estática para un objeto serializado en un momento específico? La respuesta es sí y no. Podría ser importante, pero una variable estática nunca es parte del estado de un objeto. Recordemos que las variables estáticas son variables de clase; por lo tanto, no tienen relación alguna con instancias individuales.

Las variables estáticas NUNCA serán grabadas como parte del estado de un objeto, debido a que NO pertenecen al objeto.

### Importante:

Cuando trabajamos con serialización pueden surgir algunos problemas por cuestiones de versionamiento: si grabamos el objeto "Cliente" usando una versión de la clase, y luego intentamos deserializar dicho objeto utilizando, por ejemplo, una versión más reciente de la clase, la deserialización podría fallar.

**Ejercicio de aplicación:**

1. Dada la siguiente clase:

```
import java.io.*;
class Player {
    Player() { System.out.print("p"); }
}
class CardPlayer extends Player implements Serializable {
    CardPlayer() { System.out.print("c"); }
    public static void main(String[] args){
        CardPlayer cl = new CardPlayer();

        try {
            FileOutputStream fos = new FileOutputStream("play.txt");
            ObjectOutputStream os = new ObjectOutputStream(fos);
            os.writeObject(c1);
            os.close() ;
            FileInputStream fis = new FileInputStream("play.txt");
            ObjectInputStream is = new ObjectInputStream(fis);
            CardPlayer c2 = (CardPlayer) is.readObject();
            is.close();
        } catch (Exception x ) { }
    }
}
```

¿Cuál será el resultado de su ejecución?

- a) pc
- b) pcc
- c) pcp
- d) pcpc
- e) Compilation fails.
- f) An exception is thrown at runtime.

2. Sea la variable bw una referencia válida a un objeto de tipo BufferedWriter y dado que contamos con el siguiente código:

```
15. BufferedWriter b1 = new BufferedWriter(new File("f"));
16. BufferedWriter b2 = new BufferedWriter(new FileWriter("f1"));
17. BufferedWriter b3 = new BufferedWriter(new PrintWriter("f2"));
18. BufferedWriter b4 = new BufferedWriter(new BufferedWriter(bw));
```

¿Cuál será el resultado de su ejecución?

- a) Compila exitosamente.
- b) Error de compilación por un error generado solo en la línea 15
- c) Error de compilación por un error generado solo en la línea 16
- d) Error de compilación por un error generado solo en la línea 17
- e) Error de compilación por un error generado solo en la línea 18
- f) Error de compilación por errores en múltiples líneas

## Resumen

- 📖 Las clases dentro del paquete `java.io` que debemos entender como parte del proceso de certificación son las siguientes: `File`, `FileReader`, `BufferedReader`, `FileWriter`, `BufferedWriter`, and `PrintWriter`.
- 📖 Crear un objeto `File` no implica que estemos creando un nuevo archivo en nuestro disco físico.
- 📖 Un objeto `File` puede representar un archivo o un directorio.
- 📖 La clase `File` nos permite gestionar (agregar, renombrar y borrar) archivos y directorios.
- 📖 Los métodos `createNewFile()` y `mkdir()` agregan entradas al File System.
- 📖 Las clases `FileWriter` and `FileReader` son clases de bajo nivel. Podemos usarlas para escribir y leer archivos, aunque usualmente serán envueltas con clases de más alto nivel (wrapping).
- 📖 Las clases en el paquete `java.io` han sido diseñadas para ser “encadenadas” o “envueltas”. (Este es un uso común del patrón de diseño DECORATOR).
- 📖 Una clase debe implementar la interface serializable antes de que sus objetos puedan ser serializados.
- 📖 Si una clase padre implementa la interface `Serializable`, automáticamente, sus clases hijas la implementan.
- 📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

🔗 <http://java.sun.com/j2se/1.5.0/docs/api/java/io/File.html>

Aquí encontrará documentación oficial de Java sobre las principales características y usos de la clase `File`.



[http://www.chuidiang.com/chuwiki/index.php?title=Serializaci%C3%B3n\\_de\\_objetos\\_en\\_java](http://www.chuidiang.com/chuwiki/index.php?title=Serializaci%C3%B3n_de_objetos_en_java)

En esta página, hallará documentación en castellano sobre serialización de objetos en Java.

**UNIDAD DE  
APRENDIZAJE**

**2**

**SEMANA**

**5**

## **GESTIÓN DE STRINGS, FLUJOS DE ENTRADA Y SALIDA, FORMATEO Y PARSEO EN JAVA**

### **LOGRO DE LA UNIDAD DE APRENDIZAJE**

- Al término de la unidad, los alumnos crearán aplicaciones utilizando de manera individual y combinada las clases String, StringBuffer, StringBuilder, clases de manejo de flujos del paquete java.io, clases de formateo y parseo de datos, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo

### **TEMARIO**

- La clase String – Definición y gestión de memoria
- Principales métodos de la clase String
- Las clases StringBuffer y StringBuilder
- Principales métodos de las clases StringBuffer y StringBuilder
- Creación y gestión de las clases:
  - File, FileReader, FileWriter
  - BufferedReader, BufferedWriter
  - PrintWriter
- Serialización de objetos
  - ObjectInputStream, ObjectOutputStream
- Creación y gestión de las clases:
  - Date, Calendar
  - DateFormat, NumberFormat
  - Locale
- Regular Expression en Java, Tokens y delimitadores

### **ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucran el uso de las clases Date, Calendar, DateFormat, NumberFormat y Locale.
- Los alumnos resuelven ejercicios que involucran el uso de Tokens y delimitadores en Java.

## 1. Fechas, números y monedas

La API java nos proporciona un amplio grupo de clases para trabajar con fechas, números y monedas.

A continuación, se muestran las principales:

**java.util.Date** se utiliza como una clase intermedia entre las clases Calendar y DateFormat. Una instancia de la clase Date representa una fecha y hora, expresada en milisegundos.

**java.util.Calendar** proporciona una amplia variedad de métodos que nos permitirán manipular fechas y horas. Por ejemplo, si queremos agregar un mes a una fecha en particular o determinar qué día de la semana será el primer de enero de 2009, la clase Calendar nos permitirá hacerlo.

**java.text.DateFormat** nos permite definir varios formatos y estilos de fecha tales como "01/01/70" o "Enero 1, 1970".

**Java.text.NumberFormat** permite dar formato a números y monedas.

**Java.util.Locale** se usa junto con DateFormat y NumberFormat para formatear fechas, números y monedas sobre la base de idiomas o idiomas-países específicos.

### 1.1 Uso de las clases de tipo fecha y número

Es importante tener presentes algunos conceptos básicos para el uso de fechas y números en Java. Por ejemplo, si deseamos formatear una fecha para un locale específico, necesitamos crear, primero, el objeto Locale antes que el objeto DateFormat, debido a que el locale será un parámetro de los métodos de la clase DateFormat. A continuación, se muestra un resumen con los usos típicos relacionados con fechas y números:

Caso de Uso	Pasos a seguir
Obtener la fecha y hora actual	<ol style="list-style-type: none"> <li>1. Creamos una fecha: <code>Date d = new Date();</code></li> <li>2. Obtenemos su valor: <code>String s = d.toString();</code></li> </ol>
Obtener un objeto que nos permite realizar cálculos utilizando fechas y horas (locale por defecto)	<ol style="list-style-type: none"> <li>1. Creamos un objeto Calendar</li> <li>2. <code>Calendar c = Calendar.getInstance();</code></li> <li>3. Utilizamos <code>c.add(...)</code> y <code>c.roll(...)</code> para manipular la fecha y la hora.</li> </ol>
Obtener un objeto que nos permite realizar cálculos utilizando fechas y horas (locale específico)	<ol style="list-style-type: none"> <li>1. Creamos un objeto Locale:   <code>Locale loc = new Locale (language);</code> o   <code>Locale loc = new Locale (language, country);</code> </li> <li>2. Creamos un objeto Calendar para ese Locale:</li> </ol>

	<ol style="list-style-type: none"> <li>3. <code>Calendar c = Calendar.getInstance(loc)</code></li> <li>4. Utilizamos <code>c.add(...)</code> y <code>c.roll(...)</code> para manipular la fecha y la hora.</li> </ol>
Obtenemos un objeto que nos permita realizar cálculos con fechas y horas, y crear formatos de salida para diferentes Locales con diferentes estilos de fecha.	<ol style="list-style-type: none"> <li>1. Creamos un objeto <code>Calendar</code>:</li> <li>2. <code>Calendar c = Calendar.getInstance();</code></li> <li>3. Creamos un objeto <code>Locale</code> para cada localización:</li> <li>4. <code>Locale loc = new Locale(...);</code></li> <li>5. Convertimos el objeto <code>Calendar</code> en un objeto <code>Date</code>:</li> <li>6. <code>Date d = c.getTime();</code></li> <li>7. Creamos un objeto <code>DateFormat</code> para cada <code>Locale</code>:</li> <li>8. <code>DateFormat df = DateFormat.getDateInstance (style, loc);</code></li> <li>9. Usamos el método <code>format()</code> para crear fechas con formato:</li> <li>10. <code>String s = df.format(d);</code></li> </ol>
Obtenemos un objeto que nos permita dar formato a números y monedas utilizando diferentes Locales.	<ol style="list-style-type: none"> <li>1. Creamos un objeto <code>Locale</code> para cada localización:</li> <li>2. <code>Locale loc = new Locale(...);</code></li> <li>3. Creamos un objeto de tipo <code>NumberFormat</code>:   <code>NumberFormat nf =  NumberFormat.getInstance(loc); - o -  NumberFormat nf =  NumberFormat.getCurrencyInstance(loc);</code> </li> </ol> <p>3. Use el método <code>format()</code> para crear una salida con formato</p> <p><code>String s = nf.format(someNumber);</code></p>

## 1.2 La clase Date

Una instancia de la clase `Date` representa una fecha / hora específica. Internamente, la fecha y hora es almacenada como un tipo primitivo `long`. Específicamente, el valor representa el número de milisegundos entre la fecha que se quiere representar y el 1 de enero de 1970.

En el siguiente ejemplo, determinaremos cuánto tiempo tomaría pasar un trillón de milisegundos a partir del 1 de enero de 1970.

```
import java.util.*;
class TestDates {
    public static void main(String[] args) {
        Date d1 =
            new Date(1000000000000L); // un trillón de milisegundos
        System.out.println("1st date " + d1.toString());
    }
}
```

Obtendremos la siguiente salida:

```
1st date Sat Sep 08 19:46:40 MDT 2001
```

Sabemos ahora que un trillón de milisegundos equivalen a 31 años y 2/3 de año.

Aunque la mayoría de métodos de la clase Date se encuentran deprecados, aún es común encontrar código que utilice los métodos `getTime` y `setTime`:

```
import java.util.*;
class TestDates
public static void main(String[] args) {
    Date d1 = new Date(1000000000000L); // a trillion!

    System.out.println("1st date " + d1.toString());
    d1.setTime(d1.getTime() + 3600000); // 3600000 millis / 1 hora
    System.out.println("new time " + d1.toString());
}
}
```

Obtendremos la siguiente salida:

```
1st date Sat Sep 08 19:46:40 MDT 2001
new time Sat Sep 08 20:46:40 MDT 2001
```

### 1.3 La clase Calendar

La clase Calendar ha sido creada para que podamos manipular fechas de manera sencilla. Es importante notar que la clase Calendar es una clase abstracta. No es posible decir:

```
Calendar c = new Calendar(); // ilegal, Calendar es una clase abstracta
```

Para crear una instancia de la clase Calendar, debemos usar uno de los métodos estáticos sobrecargados `getInstance()`:

```
Calendar cal = Calendar.getInstance();
```

A continuación, mostramos un ejemplo de aplicación:

```
import java.util.*;
class Dates2 {
    public static void main(String[] args) {
        Date d1 = new Date(1000000000000L);
        System.out.println("1st date " + d1.toString());

        Calendar c = Calendar.getInstance();
        c.setTime(d1); // #1

        if(c.SUNDAY == c.getFirstDayOfWeek() // #2
```



```

        System.out.println("Sunday is the first day of the week");
        System.out.println("trillionth milli day of week is "
            + c.get(c.DAY_OF_WEEK)); // #3

        c.add(Calendar.MONTH, 1); // #4
        Date d2 = c.getTime(); // #5
        System.out.println("new date " + d2.toString() );
    }
}

```

Al ejecutar la clase, obtendremos la siguiente salida:

```

1st date Sat Sep 08 19:46:40 MDT 2001
Sunday is the first day of the week
trillionth milli day of week is 7
new date Mon Oct 08 20:46:40 MDT 2001

```

Asignamos el objeto d1 (de tipo Date) a la variable c (referencia a un objeto Calendar). Usamos luego el campo SUNDAY para determinar si para nuestra JVM, SUNDAY es considerado el primer día de la semana. (Para algunos locales, MONDAY es el primer día de la semana). La clase Calendar proporciona campos similares para los días de la semana, meses y días del mes, etc.

El método add. Este método permite agregar o sustraer unidades de tiempo en base a cualquier campo que especifiquemos para la clase Calendar. Por ejemplo:

```

c.add(Calendar.HOUR, -4); // resta 4 horas de c
c.add(Calendar.YEAR, 2); // agrega 2 años a c
c.add(Calendar.DAY_OF_WEEK, -2); // resta dos días a c

```

Otro método importante en la clase Calendar es roll(). Este método trabaja como add(), con la excepción de que cuando una parte de la fecha es incrementada o decrementada, la parte más grande de dicha fecha no será incrementada o decrementada. Por ejemplo:

```

// asumiendo que c representa el 8 de Octubre de 2001
c.roll(Calendar.MONTH, 9); // notemos el año en la salida
Date d4 = c.getTime();
System.out.println("new date " + d4.toString() );

```

La salida será la siguiente:

```

new date Fri Jul 08 19:46:40 MDT 2001

```

Debemos notar que el año no cambió, a pesar de haber agregado 9 meses a la fecha original (Octubre). De manera similar, si invocamos a roll() con el campo HOUR, no cambiará la fecha, el mes o el año.

## 1.4 La clase DateFormat

Dado que ya aprendimos a crear y manipular fechas, es tiempo de aprender a darles un formato específico. A continuación, se muestra un ejemplo en el que una misma fecha es visualizada utilizando diferentes formatos:

```
import java.text.*;
import java.util.*;
class Dates3 {
    public static void main(String[] args) {
        Date d1 = new Date(1000000000000L);

        DateFormat[] dfa = new DateFormat[6];
        dfa[0] = DateFormat.getInstance();
        dfa[1] = DateFormat.getDateInstance();
        dfa[2] = DateFormat.getDateInstance(DateFormat.SHORT);
        dfa[3] = DateFormat.getDateInstance(DateFormat.MEDIUM);
        dfa[4] = DateFormat.getDateInstance(DateFormat.LONG);
        dfa[5] = DateFormat.getDateInstance(DateFormat.FULL);

        for(DateFormat df : dfa)
            System.out.println(df.format(d1));
    }
}
```

Obtendremos la siguiente salida:

```
9/8/01 7:46 PM
Sep 8, 2001
9/8/01
Sep 8, 2001
September 8, 2001
Saturday, September 8, 2001
```

La clase DateFormat es **una clase abstracta**, no es posible utilizar el operador new para crear instancias de la clase DateFormat. En este caso, utilizamos dos métodos de tipo factory: **getInstance()** y **getDateInstance()**. Debemos notar que getDateInstance() está sobrecargado. Este método tiene otra variante para poder ser utilizado con objetos de tipo Locale.

Utilizamos campos estáticos de la clase DateFormat para customizar nuestras instancias de tipo DateFormat. Cada uno de estos campos estáticos representa un estilo de formato. Finalmente, utilizamos el método format() para crear representaciones de tipo String de las diferentes versiones de formato aplicadas a la fecha específica.

El último método con el que deberíamos familiarizarnos es el método parse(). Este método toma un String formateado por DateFormat y lo convierte en un objeto de tipo Date. En realidad, ésta puede ser una operación riesgosa, dado que podría recibir como input un String que se encuentre mal formateado. Debido a ésto, el método parse() podría lanzar la siguiente excepción: ParseException. El código mostrado a continuación, crea una instancia de la clase Date, utiliza el método DateFormat.format() para convertirla a String, y luego usa DateFormat.parse() para convertirla nuevamente a Date:

```
Date d1 = new Date(1000000000000L);
System.out.println("d1 = " + d1.toString());

DateFormat df = DateFormat.getDateInstance(
    DateFormat.SHORT);
String s = df.format(d1);
System.out.println(s);

try {
    Date d2 = df.parse(s);
    System.out.println("parsed = " + d2.toString());
} catch (ParseException pe) {
    System.out.println("parse exc"); }
```

Obtendremos la siguiente salida:

```
d1 = Sat Sep 08 19:46:40 MDT 2001
9/8/01
parsed = Sat Sep 08 00:00:00 MDT 2001
```

### Importante:

Debemos notar que al usar el estilo SHORT, hemos perdido precisión al convertir el objeto Date a un String. Se evidencia esta pérdida de precisión cuando retornamos la cadena a un objeto Date nuevamente, se visualiza como hora: media noche en lugar de 17:46:00.

## 1.5 La clase Locale

Las clases DateFormat y NumberFormat pueden usar instancias de la clase Locale para customizer un formato a una localización específica. Para Java, un Locale es una región geográfica, política o cultural específica. Los constructores básicos de la clase Locale son:

- Locale(String language)
- Locale(String language, String country)

El argumento language representa un código ISO 639. Existen aproximadamente 500 códigos ISO para el lenguaje, incluido el lenguaje Klingon ("tlh").

Podemos ser más específicos y no sólo indicar un lenguaje en particular, sino también en qué país se habla ese lenguaje. En el ejemplo mostrado a continuación, se crea un Locale para italiano y otro para el italiano hablado en Suiza:

```
Locale locPT = new Locale("it"); // Italian
Locale locBR = new Locale("it", "CH"); // Switzerland
```

El uso de estos locales en una fecha producirá la siguiente salida:

sabato 1 ottobre 2005  
sabato, 1. ottobre 2005

A continuación, se muestra un ejemplo más detallado que involucra el uso de la clase Calendar:

```
Calendar c = Calendar.getInstance();
c.set(2010, 11, 14);           // 14 de Diciembre de 2010
                               // (month is 0-based)

Date d2 = c.getTime();

Locale locIT = new Locale("it", "IT"); // Italy
Locale locPT = new Locale("pt");       // Portugal
Locale locBR = new Locale("pt", "BR"); // Brazil
Locale locIN = new Locale("hi", "IN"); // India
Locale locJA = new Locale("ja");       // Japan

DateFormat dfUS = DateFormat.getInstance();
System.out.println("US      " + dfUS.format(d2));

DateFormat dfUSfull = DateFormat.getDateInstance(
    DateFormat.FULL);
System.out.println("US full " + dfUSfull.format(d2));

DateFormat dfIT = DateFormat.getDateInstance(
    DateFormat.FULL, locIT);
System.out.println("Italy   " + dfIT.format(d2));

DateFormat dfPT = DateFormat.getDateInstance(
    DateFormat.FULL, locPT);
System.out.println("Portugal " + dfPT.format(d2));

DateFormat dfBR = DateFormat.getDateInstance(
    DateFormat.FULL, locBR);
System.out.println("Brazil  " + dfBR.format(d2));

DateFormat dfIN = DateFormat.getDateInstance(
    DateFormat.FULL, locIN);
System.out.println("India   " + dfIN.format(d2));

DateFormat dfJA = DateFormat.getDateInstance(
    DateFormat.FULL, locJA);
System.out.println("Japan   " + dfJA.format(d2));
```

Obtendremos la siguiente salida:

```
US      12/14/10 3:32 PM
US full Sunday, December 14, 2010
Italy   domenica 14 dicembre 2010

Portugal Domingo, 14 de Dezembro de 2010
Brazil. Domingo, 14 de Dezembro de 2010
India   ??????, ?? ?????, ????
```

Japan 2010?12?14?

Notemos como el computador en el que se ejecutó el ejercicio previo no está configurado para soportar los locales de India o Japón.

### Importante:

Recordemos que los objetos de tipo `DateFormat` y `NumberFormat` pueden tener Locales asociados sólo en tiempo de instanciación. No es posible cambiar el locale de una instancia ya creada: no existe ningún método que permita realizar dicha operación.

Mostramos, a continuación, un ejemplo de uso de los métodos `getDisplayCountry()` y `getDisplayLanguage()`. Ambos métodos nos permitirán crear cadenas que representen un lenguaje y país específicos correspondientes a un `Locale`:

```
Calendar c = Calendar.getInstance();
c.set(2010, 11, 14);
Date d2 = c.getTime();

Locale locBR = new Locale("pt", "BR"); // Brazil
Locale locDK = new Locale("da", "DK"); // Denmark
Locale locIT = new Locale("it", "IT"); // Italy

System.out.println("def " + locBR.getDisplayCountry());
System.out.println("loc " + locBR.getDisplayCountry(locBR));

System.out.println("def " + locDK.getDisplayLanguage());
System.out.println("loc " + locDK.getDisplayLanguage(locDK));
System.out.println("D>I " + locDK.getDisplayLanguage(locIT));
```

Visualizaremos lo siguiente:

```
def Brazil
loc Brasil
def Danish
loc dansk
D>I danese
```

El locale por defecto para el país Brasil es "Brazil", y el default para el lenguaje danés es "Danish". En Brasil, el país es llamado "Brasil", y en Dinamarca el lenguaje es llamado "dansk". Finalmente, podemos notar que en Italia, el lenguaje danés es llamado "danese".

## 1.6 La clase NumberFormat

Esta clase se usa para formatear fechas y números. Se muestra a continuación un ejemplo básico:

```
float f1 = 123.4567f;
Locale locFR = new Locale("fr");    // francés
NumberFormat[] nfa = new NumberFormat[4];

nfa[0] = NumberFormat.getInstance();
nfa[1] = NumberFormat.getInstance(locFR);
nfa[2] = NumberFormat.getCurrencyInstance();
nfa[3] = NumberFormat.getCurrencyInstance(locFR);

for(NumberFormat nf : nfa)
    System.out.println(nf.format(f1));
```

Se producirá la siguiente salida:

```
123.457
123,457
$123.46
123,46 ?
```

A continuación se muestra otro ejemplo de aplicación de NumberFormat, utilizando el método getMaximumFractionDigits() :

```
float f1 = 123.45678f;
NumberFormat nf = NumberFormat.getInstance();
System.out.print(nf.getMaximumFractionDigits() + " ");
System.out.print(nf.format(f1) + " ");

nf.setMaximumFractionDigits(5);
System.out.println(nf.format(f1) + " ");

try {
    System.out.println(nf.parse("1234.567"));
    nf.setParseIntegerOnly(true);
    System.out.println(nf.parse("1234.567"));
} catch (ParseException pe) {
    System.out.println("parse exc");
}
```

Se visualizará lo siguiente:

```
3 123.457 123.45678
1234.567
1234
```

Debemos notar que, en este caso, el número por defecto de dígitos fraccionarios es 3. El método `format()` redondea los valores decimales, no los trunca.

## 2. Búsquedas

### 1.1 Búsquedas simples

Para nuestro primer ejemplo, nos gustaría buscar en la siguiente cadena:

**abaaaba**

todas las ocurrencias de :

**ab**

Asumimos que el índice de nuestras fuentes de datos siempre empieza en cero:

fuente: **abaaaba**  
Indice: 0123456

Podemos notar que existen dos ocurrencias de la expresión **ab**: una comenzando en la posición 0 y la segunda a partir de la posición 4. Si enviamos la cadena original y la “**expresión de búsqueda**” a un **motor de búsqueda** (regular expresión Engine), éste nos daría el mismo resultado: ocurrencias encontradas en las posiciones 0 y 4:

```
import java.util.regex.*;
class RegexSmall {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("ab");    // la expresión
        Matcher m = p.matcher("abaaaba");    // la fuente
        boolean b = false;
        while(b = m.find()) {
            System.out.print(m.start() + " ");
        }
    }
}
```

Esto produce:

0 4

Tenemos, ahora, un ejemplo de mayor complejidad:

fuente: Abababa  
Indice: 0123456  
expresión: ABA

¿Cuántas ocurrencias encontramos para el ejemplo previo? Se hace evidente que existe una ocurrencia a partir de la posición 0 y otra a partir de la posición 4, pero ¿existe otra ocurrencia en la posición 2? La cadena ABA que comienza en la posición 2 no se considerará válida.

### **Importante:**

Las búsquedas se hacen de izquierda a derecha. Una vez que un carácter ha sido utilizado en una búsqueda, éste no puede ser reutilizado.

En el ejemplo anterior, en la primera verificación se utilizaron las posiciones 0, 1, y 2 para que coincida con la expresión. (Se consumieron los tres primeros caracteres). Debido a que el carácter en la posición 2 se consume en la primera verificación, no puede ser utilizado de nuevo. Sin embargo, esta regla tendrá sus excepciones.

## **1.2 Búsquedas usando metacaracteres**

El uso de motores de búsqueda en Java, involucra un mecanismo muy poderoso para encontrar expresiones: el uso de metacaracteres. Por ejemplo, si queremos buscar dígitos numéricos, podemos usar la siguiente expresión:

`\d`

Si aplicamos la expresión `\d` a la siguiente cadena de origen:

fuente: a12c3e456f  
índice: 0123456789

Obtendremos como resultado que se encontraron dígitos en las posiciones 1, 2, 4, 6, 7 y 8. (Para hacer pruebas en el programa, deberemos de ingresar "`\d`")

Podemos utilizar un rico conjunto de metacaracteres que se encuentra descrito de manera detallada en la documentación de la API para `java.util.regex.Pattern`.

- `\d` Un dígito
- `\s` Un blanco
- `\w` Una palabra de caracteres (letras, dígitos, o "`_`" (subrayado))

Así, por ejemplo, si tenemos:

fuente: "a 1 56 \_Z"  
índice: 012345678  
patrón: `\w`



Se retornará 0, 2, 4, 5, 7 y 8. Los únicos caracteres de la fuente que no coinciden con el patrón, son los espacios en blanco.

También se pueden especificar conjuntos de caracteres que se buscan mediante el uso de corchetes y rangos de caracteres separados por un guión:

[abc] Búsqueda de sólo las letras a, b o c

[a-f] Búsqueda de los caracteres a, b, c, d, e, y f.

Se pueden buscar también varios rangos a la vez:

[a-fA-F]

Busca las seis primeras letras del alfabeto, tanto en su versión mayúscula como minúscula.

Sea:

fuentes: "cafeBABE"  
índice: 01234567  
patrón: [a-cA-C]

Obtendríamos como respuesta las posiciones 0, 1, 4, 5, 6.

### 1.3 Búsquedas usando cuantificadores

Supongamos que queremos crear un patrón de búsqueda para expresiones literales hexadecimal. Como primer ejemplo, buscaremos un dígito hexadecimal de los números:

0 [xX] [0-9a-fA-F]

La expresión anterior se podría leer: "Encontrar un conjunto de caracteres en la que el primer carácter es un "0", el segundo carácter una "x" o una "X", y el tercer carácter es un dígito entre "0" y "9", una letra minúscula entre "a" y "f" o mayúscula entre "A" y "F".

Sean:

fuentes: "12 0x 0x12 0xf 0xg"  
índice: 012345678901234567

Obtendremos 6 y 11. (0x y 0xg no son números hexadecimales válidos).

Si queremos especificar una o más repeticiones de una expresión podemos utilizar **cuantificadores**. Por ejemplo, "+" significa "uno o más".

En el ejemplo mostrado a continuación, encontraremos la posición inicial y luego el grupo identificado para dicha posición inicial:

fuelle: "1 A12 234b"  
patrón de búsqueda: \d+

Podemos leer el ejemplo anterior como "Encontrar uno o más dígitos en una fila." La salida sería la siguiente:

0 1  
3 12  
6 234

La siguiente expresión añade los paréntesis para limitar el cuantificador "+" sólo a los dígitos hexadecimal:

0 [xx] ([0-9a-FA-F]) +

Los otros dos cuantificadores que podemos utilizar son:

\* **Cero o más ocurrencias**  
? **Cero o una ocurrencia**

Dada la siguiente cadena fuente:

... "proj3.txt, proj1sched.pdf, proj1, proj2, proj1.java" ...

Podríamos armar una expresión con el símbolo ^ (negación) y el operador \* (cero o más):

**proj1 ([^,])\***

Si aplicamos esta expresión obtendríamos:

10 proj1sched.pdf  
25 proj1  
37 proj1.java

## 1.4 El punto predefinido (.)

Además de los metacaracteres \s, \d, y \w, también existe el metacaracter "." (punto). Cuando aparece el punto en una expresión, debemos interpretarlo de la siguiente manera: "cualquier caracter es válido". Por ejemplo, Sea:

fuelle: "abc un ac c"  
patrón: a.c

Obtendremos la siguiente salida:

3 abc  
7 a c

El "." reemplazó tanto la letra "b" como " " en la fuente de datos.

## 2. Ubicando datos a través de patrones de Equivalencia

Dado que ya sabemos usar expresiones de búsqueda dentro de una fuente de datos, el uso de las clases `java.util.regex.Pattern` y `java.util.regex.Matcher` será bastante sencillo. La clase `Pattern` se utiliza para crear una expresión que pueda luego ser utilizada y reutilizada por instancias de la clase `Matcher`. La clase `Matcher` invoca al motor de búsqueda de Java para ejecutar operaciones de búsqueda. A continuación, se muestra un ejemplo de aplicación de ambas clases:

```
import java.util.regex.*;
class Regex {
    public static void main(String [] args) {
        Pattern p = Pattern.compile(args[0]);
        Matcher m = p.matcher(args[1] );
        boolean b = false;
        System.out.println("Pattern is " + m.pattern());
        while(b = m.find()) {
            System.out.println(m.start() + " " + m.group());
        }
    }
}
```

Este programa utiliza el primer argumento de línea de comandos (`args [0]`) para representar a la expresión que se desea utilizar. Utiliza el segundo argumento (`args [1]`) para representar a la fuente de datos en donde se realizará la búsqueda.

He aquí una prueba:

```
% java  Regex "\\d\\w" "ab4 56_7ab"
```

Obtendremos la siguiente salida:

```
Pattern is \\d\\w
4 56
7 7a
```

(Recordemos que si queremos que una expresión sea válida dentro de una cadena, debemos usar `\\d\\w`). Dado que normalmente tendremos caracteres en blanco o especiales como parte de nuestros argumentos, es conveniente nos acostumbremos a encerrar los argumentos entre comillas.

Debemos notar que no usamos el operador `new` para crear un patrón. Usamos el método estático `compile()` para crear una instancia de la clase `Pattern`. Usamos, luego, el método `Pattern.matcher()` para crear una instancia de la clase `Matcher`.

El método más importante en este programa es `find()`. El método `find()` retorna verdadero si encuentra alguna equivalencia, y recuerda la posición inicial de dicha correspondencia. Si `find()` retorna verdadero, podremos invocar al método `start()` para obtener la posición inicial de la equivalencia y, luego, invocar

al método `group()` para obtener la cadena que representa la equivalencia encontrada.

Importante:

La clase `Matcher` nos proporciona varios métodos para ejecutar operaciones de búsqueda y reemplazo de caracteres. Los métodos más usados son:

- `appendReplacement()`,
- `appendTail()`,
- y `replaceAll()`

### Búsquedas utilizando la clase `java.util.Scanner`

La clase `Scanner` nos permite determinar cuantas instancias de una expresión dada existen en una fuente de datos. El programa mostrado a continuación, usa el primer argumento de la línea de comando como una expresión, luego pregunta por los caracteres ingresados a través del teclado (usando `System.in`). Se generará un mensaje cada vez que una correspondencia es encontrada:

```
import java.util.*;
class ScanIn
{
    public static void main(String[] args) {
        System.out.print("input: ");
        System.out.flush();
        try {
            Scanner s = new Scanner(System.in);
            String token;
            do {
                token = s.findInLine(args[0]);
                System.out.println("found " + token);
            } while (token != null);
        } catch (Exception e) { System.out.println("scan exc"); }
    }
}
```

La invocación a la clase y los datos ingresados son:

```
java ScanIn "\d\d"
input: 1b2c335f456
```

Obtendremos la siguiente salida:

```
found 33
found 45
found null
```

### 3. Gestión de Tokens

Tokenizing es el proceso de adopción de grandes piezas de la fuente de datos, rompiendo en pedacitos, y el almacenamiento de las pequeñas piezas en las variables. Probablemente, la más común tokenizing situación está leyendo un archivo delimitado con el fin de obtener el contenido del archivo se trasladó a útiles lugares como objetos, colecciones o arrays. Pronto nos ocuparemos de dos clases en la API que proporcionan capacidades tokenizing: String (utilizando el split () el método) y escáner, que tiene muchos métodos que son útiles para tokenizing

#### 3.1 Tokens y Delimitadores

Hablar de “tokenizing” en Java, implica hablar de dos cosas básicas: tokens y delimitadores. Los tokens son los bloques de datos y los delimitadores son las expresiones que separan los tokens uno de otro.

Cuando hablamos de delimitadores es común pensar en caracteres simples tales como espacios en blanco, comas o backslashes (\); sin embargo, los delimitadores pueden ser mucho más dinámicos. Un delimitador puede ser cualquier expresión. A continuación, se muestra un ejemplo:

fuelle: "ab,cd5b,6x,z4"

Si nuestro delimitador es **una coma**, obtendríamos los siguientes tokens:

```
ab
cd5b
6x
z4
```

Si usamos la misma fuente y definimos \d como delimitador, obtendríamos los siguientes tokens:

```
ab,cd
b,
X,Z
```

### 3.2 El método String.split()

El método `split()` de la clase `String` toma como argumento una expresión y retorna un arreglo de `Strings` con los tokens producidos por el proceso de tokenizing. El programa mostrado a continuación, utiliza `args[0]` para definir la fuente sobre la que se realizará la búsqueda y `arg[1]` para identificar la expresión que representa al delimitador:

```
import java.util.*;
class SplitTest {
    public static void main(String[] args) {
        String[] tokens = args[0].split(args[1]);
        System.out.println("count " + tokens.length);
        for(String s : tokens)
            System.out.println(">" + s + "<");
    }
}
```

Dada la siguiente invocación:

```
% java SplitTest "ab5 ccc 45 @" "\d"
```

Obtendremos la siguiente salida:

```
count 4
>ab<
>ccc<
><
>@<
```

#### Importante:

Recordemos que para representar el carácter `"\"` en una cadena requerimos utilizar la secuencia de escape `"\\"`.

Se han colocados los tokens dentro de los caracteres `"> <"` para poder visualizar los espacios en blanco. Notemos como cada dígito fue utilizado como delimitador.

Una limitación del método `splits()` es que no soporta terminar un proceso de búsqueda una vez que hemos encontrado un token en particular. Por ejemplo, si buscamos un número telefónico en un archivo, sería ideal culminar el proceso de búsqueda ni bien encontremos la primera ocurrencia de dicho número. La clase `Scanner` nos proporciona la posibilidad de implementar esta funcionalidad o tras operaciones muy interesantes con tokens.

### 3.3 La clase Scanner

La clase `java.util.Scanner` es la clase más completa para realizar operaciones con Tokens. Esta clase nos proporciona las siguientes características:

Scanners puede ser construída usando archivos, flujos, o cadenas como fuentes para realizar nuestras búsquedas. El proceso de búsqueda de tokens es ejecutado dentro de una iteración de la cual podremos salir en cualquier momento. Por otro lado, los tokens pueden ser convertidos a sus tipos primitivos específicos de manera automática.

A continuación, se muestra un ejemplo en el que usaremos los principales métodos de la clase Scanner:

```
import java.util.Scanner;
class ScanNext {
    public static void main(String [] args) {
        boolean b2, b;
        int i;
        String s, hits = " ";
        Scanner s1 = new Scanner(args[0]);
        Scanner s2 = new Scanner(args[0]);
        while(b = s1.hasNext()) {
            s = s1.next(); hits += "s";
        }
        while(b = s2.hasNext()) {
            if (s2.hasNextInt()) {
                i = s2.nextInt(); hits += "i";
            } else if (s2.hasNextBoolean()) {
                b2 = s2.nextBoolean(); hits += "b";
            } else {
                s2.next(); hits += "s2";
            }
        }
        System.out.println("hits " + hits);
    }
}
```

Si invocamos este programa de la siguiente manera:

```
java ScanNext "1 true 34 hi"
```

Obtendremos el siguiente resultado:

```
hits ssssisibis2
```

Una característica clave de los métodos identificados con el nombre `hasNextXxx()` es que prueban el valor del siguiente token, pero no lo recuperan en ese momento, no nos movemos hacia el siguiente token. Todos los métodos `nextXxx()` ejecutan dos funciones: obtienen el siguiente token y nos movemos hacia el siguiente token.

### Ejercicios de aplicación:

1. Dada la siguiente clase:

```
import java.util.regex.*;
class Regex2 {
    public static void main(String[] args) {
        Pattern p = Pattern.compile(args[0]);
        Matcher m = p.matcher(args[1]);
        boolean b = false;
        while(b = m.find()) {
            System.out.print(m.start() + m.group());
        }
    }
}
```

Y la siguiente línea de comando:

```
java Regex2 "\\d*" ab34ef
```

¿Cuál será el resultado de su ejecución?

- a) 234
- b) 334
- c) 2334
- d) 0123456
- e) 01234456
- f) 12334567
- g) Compilation fails

2. Dada la siguiente clase:

```
1. import java.util.*;
2. class Brain {
3.     public static void main(String[] args) {

4.         // insert code block here

5.     }
6. }
```



¿Cuáles de las siguientes opciones insertadas de manera independiente en la línea 4 compilará y producirá la siguiente salida: "123 82"?

a) Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L");  
while(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");

b) Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L").  
useDelimiter(" ");  
while(sc.hasNextInt()) System.out.print(sc.nextInt() + " ");

c) Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L");  
useDelimiter(" ");  
while(sc.hasNext()) {  
if (sc.hasNextInt()) System.out.print(sc.nextInt() + " " );  
else sc.next(); }

d) Scanner sc = new Scanner("123 A 3b c,45, x5x,76 82 L").  
useDelimiter(" ");  
while(sc.hasNext()) }  
if (sc.hasNextInt()) System.out.print(sc.nextInt() + " " );  
else sc.next(); }

e) Scanner sc = new Scanner ("123 A 3b c,45, x5x, 76 82 L");  
do {  
if (sc.hasNextInt()) System.out.print (sc.nextInt() + "");  
} while (sc.hasNext());

f) Scanner sc = new Scanner("12,3 A 3b c,45, x5x, 76 82 L").  
useDelimiter(" ");  
do {  
if(sc.hasNextInt()) System.out.print(sc.nextInt() + " " );  
} while ( sc.hasNext() );

## Resumen

- 📖 La mayoría de métodos de la clase `Date` están actualmente obsoletos.
- 📖 Un objeto de tipo `Date` se almacena como un tipo `long` expresado en milisegundos, el número de milisegundos transcurridos desde el 1 de enero de 1970.
- 📖 Actualmente, la clase `Date` es utilizada como clase intermediaria entre los objetos de tipo `Calendar` y `Locale`.
- 📖 La clase `Calendar` nos proporciona un poderoso conjunto de métodos para manipular fechas y realizar operaciones, tales como obtener los días de la semana, o la adición de meses, años (u otro tipo de incrementos) a una fecha.
- 📖 Podemos crear instancias de la clase `DateFormat` utilizando los métodos estáticos `getInstance()` y `getDateTimeInstance()`.
- 📖 Las clases `Pattern` y `Matcher` nos proporcionan las mayores capacidades para procesar expresiones de búsqueda en grandes fuentes de datos.
- 📖 Podemos utilizar la clase `java.util.Scanner` para hacer búsquedas simples aunque su principal utilidad es la gestión de “tokenizing” en java.
- 📖 Tokenizing es el proceso de división de datos separados en trozos pequeños por delimitadores.
- 📖 Los delimitadores son caracteres, tales como comas, espacios en blanco o expresiones más complejas.
- 📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

🔗 <http://java.sun.com/j2se/1.5.0/docs/api/java/io/File.html>

Aquí encontrará documentación oficial de Java sobre las principales características y usos de la clase `File`.



[http://www.chuidiang.com/chuwiki/index.php?title=Serializaci%C3%B3n\\_de\\_objetos\\_en\\_java](http://www.chuidiang.com/chuwiki/index.php?title=Serializaci%C3%B3n_de_objetos_en_java)

En esta página, hallará documentación en castellano sobre serialización de objetos en Java.

**UNIDAD DE  
APRENDIZAJE**

**3**

**SEMANA**

**6**

## GENERICOS Y COLLECTIONS EN JAVA

### LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos crearán aplicaciones que utilicen eficientemente las clases e interfaces del Framework Collections tales como Set, List y Map, manipulando arreglos para ordenarlos o implementar búsquedas binarias basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

### TEMARIO

- Framework Collections – Principales Clases e Interfaces
- Clase ArrayList – principales métodos
- Ordenamiento de Collections y arreglos
- Tipos genéricos
  - Polimorfismo y tipos genéricos
  - Métodos genéricos
  - Declaraciones genéricas

### ACTIVIDADES PROPUESTAS

- Los alumnos resuelven ejercicios que involucren el uso de las principales clases e interfaces del Framework collections.
- Los alumnos resuelven ejercicios que involucren el uso de las principales estrategias de ordenamiento de collections y arreglos.

## 1. Collections

Las colecciones en Java, se empiezan a utilizar a partir de la versión 1.2, se amplía luego su funcionalidad en la versión 1.4 mejorándose aún más en la versión 5.

¿Qué podemos hacer con una colección?

Existen diversas operaciones básicas que podremos realizar con las colecciones. A continuación, se mencionan las principales:

- Añadir objetos a la colección
- Eliminar objetos de la colección
- Determinar si un objeto (o grupo de objetos) se encuentra en la colección
- Recuperar un objeto de la colección (sin eliminarlo)

### 1.1 Principales clases e interfaces del Framework Collections

La API collections de Java involucra un grupo de interfaces y también un amplio número de clases concretas. Las principales interfaces de este Framework son las siguientes:

Collection	Set	SortedSet
List	Map	SortedMap
Queue		

Las principales clases del Framework son las siguientes:

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector		Arrays
TreeMap	TreeSet	LinkedList		
LinkedHashMap				

El término colección, dentro del contexto de Java, posee tres significados, los cuales detallamos a continuación:

**colección** (con “c” minúscula) representa cualquiera de las estructuras de datos en la que se almacenan los objetos y pueden ser iterados.

**Colección** (con “C” mayúscula), es en realidad la interfaz java.util.Collection a partir de la cual heredan los componentes Set, List y Queue.

**Colecciones** (con “C” mayúscula y terminando con “s”). Es la clase `java.util.Collections`, la cual nos proporciona métodos estáticos de utilidad para la gestión de colecciones.

### Tipos básicos de una colección:

Las colecciones pueden ser de cuatro tipos básicos:

**Lists:** Listas de elementos (clases que implementan la interface `List`).

**Sets:** Elementos únicos (clases que implementan la interface `Set`).

**Maps:** Elementos con un único ID (Clases que implementan la interface `Map`).

**Queues:** Elementos ordenados en base al “orden” en que fueron procesados.

Dentro de estos cuatro tipos básicos existen los siguientes sub tipos:

Sorted   Unsorted   Ordered   Unordered

Una clase que implemente una interface de la familia `Collection`, puede ser desordenada y no estar clasificada (`unsorted` y `unordered`), clasificada pero no ordenada, o ambas, clasificada y ordenada.

Una implementación nunca podrá ser ordenada, pero no clasificada, porque el ordenamiento es un tipo específico de clasificación. Por ejemplo, un `HashSet` es un conjunto no clasificado y desordenado, mientras que un `LinkedHashSet` es un conjunto clasificado (pero no ordenado) que mantiene el orden en el cual los objetos fueron insertados.

**Colecciones clasificadas.** Una colección clasificada significa que podemos iterar a través del `collection` en un orden específico (`not-random`). Un `Hashtable` no está clasificado. Aunque el `Hashtable` en sí mismo tiene una lógica interna para determinar el orden (basado en `hashcodes` y en la implementación del `collection` en sí misma), no encontraremos ningún orden cuando iteremos sobre un `Hashtable`. Un `ArrayList`, por otro lado, mantiene el orden establecido por la posición del índice de cada elemento. Un objeto `LinkedHashSet` mantiene el orden establecido por la inserción.

**Colecciones ordenadas.** Una colección ordenada significa que el orden en el `collection` está determinado por alguna regla o reglas conocidas como “`sort order`”. Un `sort order` no tiene nada que hacer con el hecho de agregar un objeto a un `collection`, cuándo fue la última vez que fue accedido o en qué posición fue agregado. El ordenamiento se hace sobre la base de las propiedades de los objetos en sí mismos. Cuando colocamos objetos en un `collection`, el `collection` asumirá el orden en que lo colocamos, basado en el `sort order`. Una colección que mantiene un orden (Como cualquier `List`, los

cuales usan el orden de inserción) no es considerada realmente ordenada, a menos que el collection ordene sus elementos utilizando algún tipo de sort order. El sort order más comúnmente utilizado es uno llamado **natural order**.

¿Qué significa esto?

Para un collection de objetos de tipo String, el orden natural es alfabético. Para una colección de enteros, el orden natural es por valor numérico, 1 antes que 2 y así sucesivamente. Para objetos creados por el usuario, no existe un orden natural, a menos que o hasta que le proporcionemos uno a través de una interface (Comparable) que define como las instancias de una clase pueden ser comparadas unas con otras. Si el desarrollador decide que los objetos deben ser comparados usando el valor de alguna variable de instancia, entonces una colección ordenada ordenará los objetos sobre la base de las reglas en dicha clase definidas para hacer ordenamiento sobre la base de una variable específica. Evidentemente, este objeto podría también heredar un orden natural de una super clase.

### **Importante:**

Tenga en cuenta que un sort order (incluyendo el orden natural), no es lo mismo que el orden dado por la inserción, acceso o un índice.

## **1.2 La interface List**

Se orienta al manejo de índices. Cuenta con un conjunto de métodos relacionados con la gestión de los índices. Así tenemos, por ejemplo, los métodos:

```
get(int index),  
indexOf(Object o),  
add(int index, Object obj), etc.
```

Las tres implementaciones de la interface List están ordenadas por la posición del índice, una posición que determinamos ya sea cargando un objeto en un índice específico o agregándolo sin especificar posición. En este caso, el objeto es agregado al final. A continuación, se describen las implementaciones de esta interface:

**ArrayList** es una lista dinámica que nos proporciona una iteración rápida y un rápido acceso aleatorio. Es una colección clasificada (por índice) pero no ordenada. Implementa la interface RandomAccess, la cual permite que la lista soporte un rápido acceso aleatorio. Es ideal y superior a LinkedList para iteraciones rápidas, mas no para realizar muchas operaciones de insert o delete.

**Vector** es básicamente igual que un ArrayList, pero sus métodos se encuentran sincronizados para para soportar thread safety. Vector es la única clase junto con ArrayList que cuentan con acceso aleatorio.

**LinkedList** Este objeto está clasificado por posición del índice como un ArrayList, con la excepción de que sus elementos se encuentran doblemente enlazados uno con otro. Este esquema permite que contemos con nuevos

métodos para agregar y eliminar desde el inicio o desde el final, lo que lo hace una opción interesante para implementar pilas o colas. Puede iterar más lentamente que un `ArrayList`, pero es la mayor opción para hacer inserts o deletes rápidamente. A partir de la versión 5, implementa la interface `java.util.Queue`, por lo que soporta los métodos relacionados con colas: `peek()`, `poll()`, and `offer()`.

### 1.3 La interface Set

La interface `Set` no permite objetos duplicados. El método `equals()` determina si dos objetos son idénticos (en cuyo caso sólo uno puede ser parte del conjunto).

**HashSet.** Es un conjunto sin clasificar y sin ordenar. Se utiliza el valor `hashCode` del objeto que se inserta como mecanismo de acceso. Esta clase es ideal cuando requerimos un `Collection` que no tenga objetos duplicados y no nos importe como iteraremos a través de ellos.

**LinkedHashSet.** Versión mejorada de `HashSet` que mantiene una lista doble través de todos sus elementos. Es conveniente utilizar esta clase en vez de `HashSet` cuando importa el orden en la iteración: iteraremos a través de los elementos en el orden en que fueron insertados.

#### Importante:

Cuando utilicemos objetos `HashSet`, o `LinkedHashSet`, los objetos que agreguemos a estos `collections` deben sobrescribir el método `hashCode()`. Si no lo hacemos, el método `hashCode()` que existe por defecto permitirá agregar múltiples objetos que podríamos considerar iguales a nuestra conjunto de objetos “únicos”.

**TreeSet.** Un `TreeSet` es uno de los dos `collections` ordenados con que se cuentan dentro del framework (el otro es `TreeMap`). Esta estructura garantiza que los elementos se encontrarán en orden ascendente de acuerdo con el orden natural. Opcionalmente, podremos construir un `TreeSet` utilizando un constructor que le indique una regla de ordenamiento específica.

### 1.4 La interface Map

Un objeto `Map` permite solo identificadores únicos. Debemos asociar una `key` única con un valor específico, donde ambos, la `key` y el valor, son objetos. Las implementaciones de la interface `Map` nos permitirán hacer operaciones tales como buscar valores basados en una `key`, preguntar por sólo los valores de un `collection`, o sólo por sus `keys`. De la misma manera que los conjuntos, un `Map` utiliza el método `equals()` para determinar si dos `keys` son las mismas o diferentes.

## 1.5 La interface Queue

Un objeto Queue está diseñado para contener objetos que serán procesados de la misma manera. De manera típica, presentan un orden FIFO (first-in, first-out).

**PriorityQueue.** Esta clase es nueva en Java 5. Desde que la clase LinkedList fue mejorada al implementar la interface QUEUE, colas básicas pueden ser gestionadas con este objeto. El propósito de PriorityQueue es crear una cola con prioridades: "priority-in, priority out", en contraposición a una típica cola FIFO. Los elementos de este tipo de objeto están ordenados ya sea por orden natural o sobre la base de un Comparator.

A continuación, se muestra una tabla con las principales implementaciones del Framework Collection:

Class	Map	Set	List	Ordered	Sorted
HashMap	x			No	No
HashTable	x			No	No
TreeMap	x			Sorted	Por orden natural o reglas de comparación customizadas por el usuario.
LinkedHashMap	x			By insertion order or last access order	No
HashSet		x		No	No
TreeSet		x		Sorted	Por orden natural o reglas de comparación customizadas por el usuario.
LinkedHashSet		x		By insertion order	No
ArrayList			x	By index	No
Vector			x	By index	No
LinkedList			x	By index	No
PriorityQueue				Sorted	Por orden en el que fueron procesados.

## 2. ArrayList

La clase `java.util.ArrayList` es una de las más utilizadas dentro del framework Collections. A continuación se describen alguna de sus principales ventajas:

Puede crecer dinámicamente.

Proporciona un mecanismo de inserción y búsqueda más eficiente que los arreglos normales.

Un objetivo de diseño clave del framework collection fue proporcionar una funcionalidad muy rica a nivel de las interfaces principales: List, Set, y Map. En la práctica, podremos instanciar polimórficamente un ArrayList de la siguiente manera:



```
List myList = new ArrayList();
```

o a partir de la versión 5 de Java:

```
List<String> myList = new ArrayList<String>();
```

Este tipo de declaración sigue el principio de programación orientada a objetos "coding to an interface". Este principio implica el uso de generics.

De muchas maneras, `ArrayList<String>` es similar a un `String[]`, contenedor que almacenará solo Strings, pero con mayor flexibilidad y ventajas que un `String[]`. Veamos el siguiente ejemplo:

```
import java.util.*;
public class TestArrayList {
    public static void main(String[] args) {
        List<String> test = new ArrayList<String>();
        String s = "hi";
        test.add("string");
        test.add(s);
        test.add(s+s);
        System.out.println(test.size());
        System.out.println(test.contains(42));

        System.out.println(test.contains("hihi"));
        test.remove("hi");
        System.out.println(test.size());
    }
}
```

obtendremos la siguiente salida:

```
3
false
true
2
```

### 3. Autoboxing con Collections

En general, un collection puede contener objetos, pero no tipo primitivos. Antes de la versión 5 de java, un uso típico de las clases wrapper era proporcionar una mecanismo para que un tipo primitivo pueda ser parte de un collection. A partir de la versión 5 de java, los tipos primitivos todavía deben de ser transformados, sin embargo, el proceso de transformación (autoboxing) es automático.

```
List myInts = new ArrayList(); // pre Java 5 declaration myInts.add(new
Integer(42)); // had to wrap an int
```

As of Java 5 we can say

```
myInts.add(42); // autoboxing handles it!
```

En el último ejemplo, continuamos agregando un objeto de tipo Integer a la lista myInts (no un tipo primitivo); sin embargo, actúa de manera automática el autoboxing.

## 4. Ordenando Collections y Arreglos

### 4.1 Ordenando Collections

A continuación se muestra un ejemplo de aplicación:

```
import java.util.*;
class TestSortl {
    public static void main(String[] args) {
        ArrayList<String> stuff = new ArrayList<String>(); // #1
        stuff.add("Denver");
        stuff.add("Boulder");
        stuff.add("Vail");
        stuff.add("Aspen");
        stuff.add("Telluride");
        System.out.println("unsorted " + stuff);
        Collections.sort(stuff); // #2
        System.out.println("sorted " + stuff);
    }
}
```

Obtendremos la siguiente salida:

Salida desordenada: [Denver, Boulder, Vail, Aspen, Telluride]

Salida ordenada: [Aspen, Boulder, Denver, Telluride, Vail]

En la línea 1 declaramos un ArrayList de Strings, y en la línea 2 ordenamos el ArrayList alfabéticamente.

### 4.2 La interface Comparable

La interface Comparable es usada por los métodos Collections.sort() y java.util.Arrays.sort() para ordenar listas y arreglos de objetos respectivamente. Para implementar Comparable, una clase debe implementar el método compareTo():

```
int x = thisObject.compareTo(anotherObject);
```

El método compareTo() retorna un valor entero con las siguientes características:

#### **Negative**

Si este objeto < otro Object

#### **zero**

Si este objeto == otro Object

### positive

If este objeto > otro Objeto

El método `sort()` usa `compareTo()` para determinar cómo la lista o arreglo de objetos debe ser ordenada. Dado que podemos implementar el método `compareTo()` para nuestras propias clases, podremos usar cualquier criterio de ordenamiento para nuestras clases:

```
class DVDInfo implements Comparable<DVDInfo> { // #1
    // existing code
    public int compareTo(DVDInfo d) {
        return title.compareTo(d.getTitle()); // #2
    }
}
```

En la línea 1, declaramos que la clase `DVDInfo` implementa `Comparable` de tal manera que los objetos de tipo `DVDInfo` pueden ser comparados con otros objetos de su mismo tipo. En la línea 2, implementamos `compareTo()` para comparar los títulos de los objetos `DVDInfo`. Dado que sabemos que los títulos son de tipo `Strings` y los `String` implementan `Comparable`, esta será una manera sencilla de ordenar objetos de tipo `DVDInfo` por título. Antes de la versión 5 de Java, debíamos implementar `Comparable` con un código similar al siguiente:

```
class DVDInfo implements Comparable {
    // existing code
    public int compareTo(Object o) { // takes an Object rather
        // than a specific type
        DVDInfo d --- (DVDInfo)o;
        return title.compareTo(d.getTitle());
    }
}
```

Este código es válido, pero puede ser riesgoso dado que debemos de hacer un casteo y debemos de asegurarnos de que este casteo no fallará antes de que lo utilicemos.

### Importante:

Cuando sobreescribimos el método `equals()`, debemos recibir un argumento de tipo `Object`, pero cuando sobreescribimos `compareTo()` tomaremos un argumento del tipo que estemos ordenando.

## 4.3 Ordenando con Comparator

La interface `Comparator` nos proporciona la capacidad de ordenar un collection de muchas formas diferentes. Podemos, también, usar esta interface para ordenar instancias de cualquier clase, incluso clases que no podamos modificar, a diferencia de la interface `Comparable`, la cual nos obliga a modificar la clase de cuyas instancias queremos ordenar. La

interface `Comparator` es también muy fácil de implementar, sólo tiene un método: `compare()`. A continuación, se muestra un ejemplo de aplicación:

```
import java.util.*;
class GenreSort implements Comparator<DVDInfo> {
    public int compare(DVDInfo one, DVDInfo two) {
        return one.getGenre().compareTo(two.getGenre());
    }
}
```

El método `Comparator.compare()` retorna un valor entero cuyo significado es el mismo que el valor retornado por `Comparable.compareTo()`. A continuación, se muestra un ejemplo de aplicación de ambos métodos:

```
import java.util.*;
import java.io.*; // populateList() needs this
public class TestDVD {
    ArrayList<DVDInfo> dvdlist = new ArrayList<DVDInfo>();
    public static void main(String[] args) {
        new TestDVD().go();
    }
    public void go() {
        populateList();
        System.out.println(dvdlist); // output as read from file
        Collections.sort(dvdlist);
        System.out.println(dvdlist); // output sorted by title

        GenreSort gs = new GenreSort();
        Collections.sort(dvdlist, gs);
        System.out.println(dvdlist); // output sorted by genre
    }

    public void populateList() {
        // read the file, create DVDInfo instances, and
        // populate the ArrayList dvdlist with these instances
    }
}
```

Se muestra, a continuación, una tabla con la comparación de ambas interfaces:

<b>java.lang.Comparable</b>	<b>java.util.Comparator</b>
<code>int objOne.compareTo(objTwo)</code>	<code>int compare(objOne, objTwo)</code>
Retorna: Negativo si <code>obj One &lt; objTwo</code> Cero si <code>objOne == objTwo</code> Positivo si <code>objOne &gt; objTwo</code>	El mismo que <code>Comparable</code>
Debemos modificar la clase cuyas instancias queremos ordenar.	Debemos construir una clase independiente de la clase que queremos ordenar.
Solo podemos crear una secuencia de ordenamiento.	Podemos crear muchas secuencias de ordenamiento
Implementada frecuentemente por la API java: <code>String</code> , <code>Wrapper</code> classes, <code>Date</code> , <code>Calendar</code> ...	Creada para ser implementada por instancias de Terceros.

## 4.4 Ordenando con la clase Arrays

Ordenar arreglos de objetos es igual que ordenar collections de objetos. El método `Arrays.sort()` es sobreescrito de la misma manera que el método `Collections.sort()`.

**`Arrays.sort(arrayToSort)`**

**`Arrays.sort(arrayToSort, Comparator)`**

Recordemos que los métodos `sort()` de las clases `Collections` y `Arrays` son métodos estáticos, y ambos trabajan directamente sobre los objetos que están ordenando, en vez de retornar un objeto diferente ordenado.

## 4.5 Convirtiendo arreglos a listas y listas a arreglos

Existen dos métodos que permiten convertir arreglos en listas y listas en arreglos. Las clases `List` y `Set` tienen los métodos `toArray()`, y las clases `Arrays`, tienen un método llamado `asList()`.

El método `Arrays.asList()` copia un arreglo en un objeto tipo `List`:

```
String [] sa = {"one", "two", "three", "four"};
List sList = Arrays.asList(sa);           // make a List
System.out.println("size " + sList.size());
System.out.println("idx2 " + sList.get(2));

sList.set(3,"six");                       // change List
sa[1] = "five";                           // change array
for(String s : sa)
    System.out.print(s + " ");
System.out.println("\nsl[1] " + sList.get(1));
```

Obtendremos la siguiente salida:

```
size 4
idx2 three
one five three six
sl[1] five
```

A continuación veamos el uso del método `toArray()`. Existen dos versiones de este método. La primera, retorna un nuevo objeto de tipo `Array`. La segunda, usa como destino un arreglo pre existente.

```
List<Integer> iL = new ArrayList<Integer>();
for(int x=0; x<3; x++)
    iL.add(x);
Object[] oa = iL.toArray();               // create an Object array
Integer[] ia2 = new Integer[3];
ia2 = iL.toArray(ia2);                   // create an Integer array
```

## 4.6 Usando Listas

Las listas son usadas para almacenar elementos en algún tipo de orden. Podemos usar un `LinkedList` para crear una cola FIFO. Podemos usar un `ArrayList` para mantener el rastro de que ubicaciones fueron visitadas, y en qué orden. En ambos casos, es totalmente razonable pensar que podemos tener elementos duplicados. Antes de la versión 5 de Java, la manera más común de examinar los elementos de una lista era utilizando un objeto `Iterator`. Un `iterator` es un objeto asociado con un `collection` en particular. Los métodos que debemos de conocer en un `iterator` son los siguientes:

**boolean hasNext():** Retorna verdadero si al menos existe un elemento en el `collection` que está siendo procesado, `hasNext()` no nos desplaza al siguiente elemento del `Collection`.

**Object next():** Este método retorna el siguiente elemento del `collection`, y nos desplaza hacia ese elemento después de que éste fue retornado.

A continuación, se muestra un ejemplo que utiliza `Iterator`:

```
import java.util.*;
class Dog {
    public String name;
    Dog(String n) { name = n; }
}
class ItTest {
    public static void main(String[] args) {
        List<Dog> d = new ArrayList<Dog>();
        Dog dog = new Dog("aiko");
        d.add(dog);
        d.add(new Dog("clover"));
        d.add(new Dog("magnolia"));
        Iterator<Dog> i3 = d.iterator(); // make an iterator
        while (i3.hasNext()) {
            Dog d2 = i3.next();           // cast not required
            System.out.println(d2.name);
        }
        System.out.println("size " + d.size());
        System.out.println("get1 " + d.get(1).name);
        System.out.println("aiko " + d.indexOf(dog));

        d.remove(2);
        Object[] oa = d.toArray();
        for(Object o : oa) {
            Dog d2 = (Dog)o;
            System.out.println("oa " + d2.name);
        }
    }
}
```

Obtendremos la siguiente salida:

```
aiko
clover
```

```
magnolia
size 3
getl clover
aiko 0
oa aiko
oa clover
```

#### 4.7 Uso de Conjuntos (Sets)

Recordemos que los conjuntos se usan cuando no queremos ningún elemento duplicado dentro de un collection. Si queremos agregar un elemento a un set que ya contiene dicho elemento, el elemento duplicado no será agregado, y el método add() retornará el valor false. Los HashSets tienden a ser muy rápidos, porque utilizan hashcodes.

Podemos crear, también, un TreeSet, el cual es un Set cuyos elementos se encuentran ordenados.

```
import java.util.*;
class SetTest {
    public static void main(String [] args) {
        boolean[] ba = new boolean[5];
        // insert code here

        ba[0] = s.add("a");
        ba[1] = s.add(new Integer(42));
        ba[2] = s.add("b") ;
        ba[3] = s.add("a") ;
        ba[4] = s.add(new Object());
        for(int x=0; x<ba.length; x++)
            System.out.print(ba[x] + " ");
        System.out.println("\n");
        for(Object o : s)
            System.out.print(o + " ");
    }
}
```

Si insertamos la siguiente línea de código:

```
Set s = new HashSet(); // insert this code
```

Obtendríamos la salida mostrada a continuación:

```
true true true false true
a java.lang.Object@e09713 42 b
```

Es importante destacar que el orden de los objetos en el segundo for no es predecible: HashSets y LinkedHashSets no garantizan orden alguno. Notemos también que la cuarta invocación del método add() falló, esto debido a que intentar insertar una entrada duplicada.

Si insertamos la siguiente línea de código:

```
Set s = new TreeSet(); // insert this code
```

Obtendríamos:

```
Exception in thread "main" java.lang.ClassCastException: java.
lang.String
    at java.lang.Integer.compareTo(Integer.java:35)
    at Java.util.TreeMap.compare(TreeMap.java:1093)

    at java.util.TreeMap.put(TreeMap.java:465)
    at java.util.TreeSet.add(TreeSet.java:210)
```

Si queremos que un collection esté ordenado, sus elementos deben ser mutuamente comparables. Objetos de diferentes tipos no son comparables.

## 4.8 Uso de objetos tipo Maps

Recordemos que, cuando usamos una clase que implementa Map, cualquier clase que usemos como parte de la key debe sobrescribir los métodos hashCode() e equals().

**Importante:** de manera estricta, solo necesitaremos sobrescribir estos métodos si queremos recuperar objetos de un Map.

```
import java.util.*;
class Dog {
    public Dog(String n) { name = n; }
    public String name;
    public boolean equals(Object o) {
        if( (o instanceof Dog) &&
            (((Dog)o).name == name)) {
            return true;
        } else {
            return false;
        }
    }
    public int hashCode() {return name.length(); }
}

class Cat { }

enum Pets {DOG, CAT, HORSE }

class MapTest {
    public static void main(String[] args) {
        Map<Object, Object> m = new HashMap<Object, Object>();

        m.put("k1", new Dog("aiko")); // add some key/value pairs
        m.put("k2", Pets.DOG);
        m.put(Pets.CAT, "CAT key");
        Dog d1 = new Dog("clover"); // let's keep this reference

        m.put(d1, "Dog key");
        m.put(new Cat(), "Cat key");
    }
}
```



```

        System.out.println(m.get("k1"));           // #1
        String k2 = "k2";
        System.out.println(m.get(k2));             // #2
        Pets p = Pets.CAT;
        System.out.println(m.get(p));              // #3
        System.out.println(m.get(dl));             // #4
        System.out.println(m.get(new Cat()));      // #5
        System.out.println(m.size());              // #6
    }
}

```

Obtendremos la siguiente salida:

```

Dog@1c
DOG
CAT key
Dog key
null
5

```

El primer valor recuperado es un objeto Dog. El segundo valor recuperado es un valor de tipo enum (DOG). El tercer valor recuperado es un String; notemos que la key fue un valor enum. Enum sobrescribe los métodos equals() y hashCode().

El cuarto valor recuperado es un String. La parte destacable en relación a esta salida es que la key utilizada para recuperar el String fue construida en base a un objeto de tipo Dog. La quinta salida es null. El punto importante aquí es que el método get() falló en su objetivo de encontrar el objeto Cat insertado inicialmente. ¿Por qué no se pudo encontrar la key String de Cat?

La respuesta es muy sencilla, a diferencia de Dog, Cat no sobrescribió los métodos equals() y hashCode().

## 4.9 Uso de la clase PriorityQueue

La clase PriorityQueue ordena sus elementos usando una prioridad definida por el usuario. La prioridad puede ser tan simple como un orden natural (en la que, por ejemplo, una entrada con el valor 1 será de mayor prioridad que una entrada con el valor 2). Adicionalmente, la clase PriorityQueue puede ser ordenada utilizando un Comparator, componente que nos permitirá definir el orden que estimemos conveniente.

```

import java.util.*;
class PQ {

    static class PQsort
        implements Comparator<Integer> { // inverse sort
        public int compare(Integer one, Integer two) {
            return two - one;              // unboxing
        }
    }
}

```

```

public static void main(String[] args) {
    int[] ia= {1,5,3,7,6,9,8};           // unordered data
    PriorityQueue<Integer> pq1 =
        new PriorityQueue<Integer>();      // use natural order

    for(int x : ia)                       // load queue
        pq1.offer(x);
    for(int x : ia)                       // review queue

        System.out.print(pq1.poll() + " ");
    System.out.println("");

    PQsort pqs = new PQsort();           // get a Comparator
    PriorityQueue<Integer> pq2 =
        new PriorityQueue<Integer>(10,pqs); // use Comparator

    for(int x : ia) // load queue
        pq2.offer(x);
    System.out.println("size " + pq2.size());
    System.out.println("peek " + pq2.peek());
    System.out.println("size " + pq2.size());
    System.out.println("poll " + pq2.poll());
    System.out.println("size " + pq2.size());
    for(int x : ia)                       // review queue
        System.out.print(pq1.poll() + " ");
    }
}

```

Este código produce el siguiente resultado:

1 3 5 6 7 8 9

size 7  
 peek 9  
 size 7  
 poll 9  
 size 6  
 8 7 6 5 3 1 null

El primer for itera sobre el arreglo ia, y utiliza el método offer() para agregar elementos a pq1, el segundo for itera sobre pq1 usando el método poll(), el cual retorna la entrada de más alta prioridad en pq1 y elimina la entrada de la cola. Notemos que los elementos son retornados en orden de prioridad (en este caso, orden natural). Luego creamos un comparator, en este caso, el Comparator ordena los elementos en orden opuesto al natural. Usamos luego el Comparator para construir un PriorityQueue, llamado pq2, y cargamos los datos con el arreglo utilizado inicialmente. Finalmente, el tamaño de pq2 antes y después de llamar a los métodos peek() y poll(). Esto confirma que peek() retorna el elemento de más alta prioridad en la cola sin removerlo de la misma.

### Ejercicios de aplicación:

1. Dada la siguiente clase:

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        // insert code here
        x.add("one");
        x.add("two");
        x.add("TWO");
        System.out.println(x.poll());
    }
}
```

¿Cuáles de las siguientes líneas de código compilarán?

- a) `List<String> x = new LinkedList<String>();`
- b) `TreeSet<String> x = new TreeSet<String>();`
- c) `HashSet<String> x = new HashSet<String>();`
- d) `Queue<String> x = new PriorityQueue<String>();`
- e) `ArrayList<String> x = new ArrayList<String>();`
- f) `LinkedList<String> x = new LinkedList<String>();`

2. Dada la siguiente clase:

```
import java.util.*;
class Flubber {
    public static void main(String[] args) {
        List<String> x = new ArrayList<String>();
        x.add(" x"); x.add("xx"); x.add("Xx");

        // insert code here
        for(String s: x) System.out.println(s);
    }
}
```

Y la salida:

```
xx
Xx
x
```

¿Cuáles de las siguientes líneas de código producirán la salida mostrada?

- a) `Collections.sort(x);`
- b) `Comparable c = Collections.reverse(); Collections.sort(x,c);`
- c) `Comparator c = Collections.reverse(); Collections.sort(x,c);`
- d) `Comparable c = Collections.reverseOrder(); Collections.sort(x,c);`
- e) `Comparator c = Collections.reverseOrder(); Collections.sort(x,c);`

# Resumen

📖 Las actividades más comunes dentro del framework collections incluyen agregar objetos, eliminarlos, verificar la inclusion de objetos, recuperarlos, iterarlos, etc.

📖 Tenemos tres significados para "collection" :

- Collection representa la estructura de datos en la cual los objetos son almacenados.
- Es la interface java.util.Collection de la cual heredan Set y List
- Collections es una clase que contiene varios métodos utilitarios estáticos.

📖 Los collections se clasifican en objetos de tipo Lists, Sets, Maps y Queue.

📖 Existen cuatro subclasificaciones para los collections: Sorted, Unsorted, Ordered, Unordered.

📖 Las operaciones de ordenamiento pueden ser alfabéticas, numéricas o definidas programáticamente.

📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

🔗 <http://www.reloco.com.ar/prog/java/collections.html>

Aquí hallará un artículo muy interesante en castellano sobre el uso de Collections en Java.

🔗 <http://java.sun.com/docs/books/tutorial/collections/interfaces/order.html> En esta página, hallará un torial oficial sobre ordenamiento de objetos en java.



**UNIDAD DE  
APRENDIZAJE**

**3**

**SEMANA**

**9**

## GENERICOS Y COLLECTIONS EN JAVA

### LOGRO DE LA UNIDAD DE APRENDIZAJE

- Al término de la unidad, los alumnos crearán aplicaciones que utilicen eficientemente las clases e interfaces del Framework Collections tales como Set, List y Map, manipulando arreglos para ordenarlos o implementar búsquedas binarias basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

### TEMARIO

- Framework Collections – Principales clases e Interfaces
- Clase ArrayList – principales métodos
- Ordenamiento de Collections y arreglos
- Tipos genéricos
  - Polimorfismo y tipos genéricos
  - Métodos genéricos
  - Declaraciones genéricas

### ACTIVIDADES PROPUESTAS

- Los alumnos resuelven ejercicios que involucran el uso de polimorfismo y tipos genéricos de Java.

## 1. Tipos genéricos

### 1.1 Polimorfismo y tipos genéricos

Las colecciones genéricas nos proporcionan los mismos beneficios de seguridad en los tipos que los arreglos, aunque existen algunas diferencias fundamentales que están relacionadas con el uso de polimorfismo en Java.

El Polimorfismo se aplica al tipo base del collection:

```
List<Integer> myList = new ArrayList<Integer>();
```

En otras palabras, es válido asignar un `ArrayList` a un objeto `List` porque `List` es clase padre de `ArrayList`.

Veamos ahora el siguiente ejemplo:

```
class Parent { }  
class Child extends Parent { }  
List<Parent> myList = new ArrayList<Child>();
```

¿Cuál será el resultado?

La respuesta es No funcionará. Existe una regla básica para el ejemplo anterior: El tipo de una declaración de variable debe corresponder al tipo que que pasamos en el objeto actual. Si declaramos `List<Foo> foo`, entonces lo que asignemos a la variable `foo` debe ser un tipo genérico `<Foo>`. No un subtipo de `<Foo>`. Tampoco un supertipo de `<Foo>`. Sólo `<Foo>`.

**Esto sería un error:**

```
List<object> myList = new ArrayList<JButton>(); // NO!  
List<Number> numbers = new ArrayList<Integer>(); // NO!  
// remember that Integer is a subtype of Number
```

**Esto sería correcto:**

```
List<JButton> myList = new ArrayList<JButton>(); // yes  
List<Object> myList = new ArrayList<Object>(); // yes  
List<Integer> myList = new ArrayList<Integer>(); // yes
```



## 1.2 Métodos genéricos

Uno de los mayores beneficios del polimorfismo es que podemos declarar el parámetro de un método como de un tipo en particular y, en tiempo de ejecución, hacer que este parámetro referencie a cualquiera de sus subtipos.

Por ejemplo, imaginemos un ejemplo clásico de polimorfismo: La clase `AnimalDoctor` con el método `checkup()`. Tenemos, también, tres subtipos de la clase `Animal`: `Dog`, `Cat`, y `Bird`. A continuación, se muestran las implementaciones del método abstracto `checkup()` de la clase `Animal`:

```
abstract class Animal {
    public abstract void checkup();
}
class Dog extends Animal {
    public void checkup() { // implement Dog-specific code
        System.out.println("Dog checkup");
    }
}
class Cat extends Animal {
    public void checkup() { // implement Cat-specific code
        System.out.println("Cat checkup");
    }
}
class Bird extends Animal {
    public void checkup() { // implement Bird-specific code
        System.out.println("Bird checkup");
    }
}
```

Si requerimos que la clase `AnimalDoctor` invoque al método `checkup()` adecuado correspondiente al animal específico que se está atendiendo, podríamos tener el siguiente código:

```
public void checkAnimal(Animal a) {
    a.checkup(); // no importará el subtipo de animal que se reciba
               // siempre invocaremos al método checkup() adecuado.
}
```

Si ahora requerimos atender a un conjunto variado de animals diversos, los cuales son recibidos como un arreglo de objetos de la clase `Animal[]`, podríamos tener un código similar al siguiente:

```
public void checkAnimals(Animal[] animals) {
    for(Animal a : animals) {
        a.checkup();
    }
}
```

A continuación, se muestra el ejemplo completo en el que probamos el uso de polimorfismo a través de arreglos de subtipos de la clase `Animal`:

```

import java.util.*;
abstract class Animal {
    public abstract void checkup();
}
class Dog extends Animal {
    public void checkup() { // implement Dog-specific code
        System.out.println("Dog checkup");
    }
}
class Cat extends Animal {
    public void checkup() { // implement Cat-specific code
        System.out.println("Cat checkup");
    }
}

class Bird extends Animal {
    public void checkup() { // implement Bird-specific code
        System.out.println("Bird checkup");
    }
}

public class AnimalDoctor {
    // method takes an array of any animal subtype
    public void checkAnimals(Animal[] animals) {
        for(Animal a : animals) {
            a.checkup();
        }
    }

    public static void main(String[] args) {
        // test it
        Dog[] dogs = (new Dog(), new Dog());
        Cat[] cats = (new Cat(), new Cat(), new Cat());
        Bird[] birds = (new Bird());

        AnimalDoctor doc = new AnimalDoctor();
        doc.checkAnimals(dogs); // pass the Dog[]
        doc.checkAnimals(cats); // pass the Cat[]
        doc.checkAnimals(birds); // pass the Bird[]
    }
}

```

El ejemplo anterior trabaja correctamente; sin embargo, debemos tener presente que este enfoque no es válido si trabajamos con collections bajo el esquema de manera segura (type safe collections).

Si un método, por ejemplo, recibe un collection de tipo `ArrayList<Animal>`, no podrá aceptar una colección constituida por cualquier subtipo de la clase `Animal`. Eso significa que, por ejemplo, no podremos recibir un `Collection` de tipo `ArrayList<Dog>` en un método que soporta argumentos de tipo `ArrayList<Animal>`.

Como podemos observar, el ejemplo anterior, evidencia las diferencias entre el uso de arreglos de un tipo específico versus collections de un tipo específico.

Aunque sabemos que no se ejecutará correctamente, haremos los siguientes cambios a la clase `AnimalDoctor` para que utilice objetos genéricos en vez de arreglos:

```
public class AnimalDoctorGeneric {

    // change the argument from Animal[] to ArrayList<Animal>
    public void checkAnimals(ArrayList<Animal> animals) {
        for(Animal a : animals) {
            a.checkup();
        }
    }

    public static void main(String[] args) {
        // make ArrayLists instead of arrays for Dog, Cat, Bird
        List<Dog> dogs = new ArrayList<Dog>();
        dogs.add(new Dog());
        dogs.add(new Dog());
        List<Cat> cats = new ArrayList<Cat>();
        cats.add(new Cat());
        cats.add(new Cat());
        List<Bird> birds = new ArrayList<Bird>();
        birds.add(new Bird());

        // this code is the same as the Array version
        AnimalDoctorGeneric doc = new AnimalDoctorGeneric();
        // this worked when we used arrays instead of ArrayLists
        doc.checkAnimals(dogs); // send a List<Dog>
        doc.checkAnimals(cats); // send a List<Cat>
        doc.checkAnimals(birds); // send a List<Bird>
    }
}
```

Si ejecutamos la siguiente línea:

```
javac AnimalDoctorGeneric.Java
```

**Obtendremos:**

```
AnimalDoctorGeneric.Java:51: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Dog>)
    doc.checkAnimals(dogs);
        ^
AnimalDoctorGeneric.Java: 52: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Cat>)
    doc.checkAnimals(cats);
        ^
AnimalDoctorGeneric.Java:53: checkAnimals(java.util.
ArrayList<Animal>) in AnimalDoctorGeneric cannot be applied to
(java.util.List<Bird>)
    doc.checkAnimals(birds);
        ^
3 errors
```

El compilador generó tres errores debido a que no podemos asignar ArrayLists de subtipos de Animal (<Dog>, <Cat>, or <Bird>) a un ArrayList del supertipo <Animal>.

Ante esta situación ¿Cuál podría ser una salida válida utilizando objetos genéricos?

**Podríamos hacer lo siguiente:**

```
List<Animal> animals = new ArrayList<Animal>();  
animals.add(new Cat()); // OK  
animals.add(new Dog()); // OK
```

El código anterior es válido tanto para arreglos como para collections genéricos, podemos agregar una instancia de un subtipo en un arreglo o colección declarados con un supertipo.

Por lo tanto, y bajo ciertas circunstancias, el código mostrado a continuación sería válido:

```
public void addAnimal(ArrayList<Animal> animals) {  
    animals.add(new Dog()); // sometimes allowed...  
}
```

El código mostrado en líneas anteriores, compilará satisfactoriamente, siempre y cuando el parámetro que le pasemos al método sea de tipo ArrayList<Animal>.

**importante:**

El único objeto que podremos pasar a un método con un argumento de tipo ArrayList<Animal> es un objeto de tipo ArrayList<Animal>!

Existe en Java, también, un mecanismo para indicar al compilador que aceptaremos cualquier subtipo genérico del argumento declarado en un método. Es importante estar seguros que no agregaremos cualquier elemento (no válido) a el collection, el mecanismo es el uso de <?>.

La firma del método cambiará de:

```
public void addAnimal(List<Animal> animals)
```

por:

```
public void addAnimal(List<? extends Animal> animals)
```

El decir **<? extends Animal>**, nos permite indicar al compilador que podremos asignar un collection que sea un subtipo de List y de tipo <Animal> o cualquier otro tipo que herede de Animal.

Sin embargo, el método `addAnimal()` no funcionará debido a que el método agrega un elemento al `collection`.

```
public void addAnimal(List<? extends Animal> animals) {
    animals.add(new Dog()); // NO! Can't add if we
                           // use <? extends Animal>
}
```

El error que obtendríamos sería el siguiente:

```
javac AnimalDoctorGeneric.java
AnimalDoctorGeneric.java:38: cannot find symbol
symbol : method add(Dog)
location: interface java.util.List<capture of ? extends Animal>
    animals.add(new Dog());
                ^
1 error
```

Si cambiamos el método para que no agregue un elemento, funcionará correctamente el código anterior.

### 1.3 Declaraciones genéricas

Veamos el siguiente ejemplo:

```
public interface List<E>
```

El identificador `<E>` representa, de manera “general”, el tipo que pasaremos a la lista. La interface `List` se comporta como una plantilla genérica, de modo que cuando creamos nuestro código, podamos hacer declaraciones del tipo: `List<Dog>` or `List<Integer>`, and so on.

El uso de “**E**”, es solo una convención. Cualquier identificador válido en Java funcionará adecuadamente. Se escogió **E** porque se puede interpretar como “Elemento”. La otra convención básica es el uso de **T** (que significa **Type**), utilizada para tipos que no son `collections`.

## Ejercicios de aplicación:

1. Dada la siguiente clase:

```
import java.util.*;
class Test {
    public static void main(String[] args) {
        // insert code here
        x.add("one");
        x.add("two");
        x.add("TWO");
        System.out.println(x.poll());
    }
}
```

¿Cuáles de las siguientes líneas de código, al ser insertadas en lugar de:

// insert code here, compilarán?

- a) `List<String> x = new LinkedList<String>();`
- b) `TreeSet<String> x = new TreeSet<String>();`
- c) `HashSet<String> x = new HashSet<String>();`
- d) `Queue<String> x = new PriorityQueue<String>();`
- e) `ArrayList<String> x = new ArrayList<String>();`
- f) `LinkedList<String> x = new LinkedList<String>();`

2. Dadas las siguientes líneas de código:

```
10. public static void main(String[] args) {
11.     Queue<String> q = new LinkedList<String>();
12.     q.add("Veronica");
13.     q.add("Wallace");
14.     q.add("Duncan");
15.     showAll(q);
16. }
17.
18. public static void showAll(Queue q) {
19.     q.add(new Integer(42));
20.     while (!q.isEmpty ( ) )
21.         System.out.print(q.remove( ) + " ");
22. }
```

¿Cuál será el resultado obtenido?

- a) Veronica Wallace Duncan
- b) Veronica Wallace Duncan 42
- c) Duncan Wallace Veronica
- d) 42 Duncan Wallace Veronica
- e) Compilation fails.
- f) An exception occurs at runtime.

3. Dadas las siguientes clases:

```
import java.util.*;
class MapEQ {
    public static void main(String[] args) {
        Map<ToDos, String> m = new HashMap<ToDos, String>();
        ToDos t1 = new ToDos("Monday");
        ToDos t2 = new ToDos("Monday");
        ToDos t3 = new ToDos("Tuesday");
        m.put(t1, "doLaundry");
        m.put(t2, "payBills");
        m.put(t3, "cleanAttic");
        System.out.println(m.size());
    }
}
class ToDos{
    String day;
    ToDos(String d) { day = d; }
    public boolean equals(Object o) {
        return ((ToDos)o).day == this.day;
    }
    // public int hashCode() { return 9; }
}
```

¿Cuáles de las afirmaciones mostradas a continuación son válidas?

- a) As the code stands it will not compile.
- b) As the code stands the output will be 2.
- c) As the code stands the output will be 3.
- d) If the hashCode() method is uncommented the output will be 2.
- e) If the hashCode() method is uncommented the output will be 3.
- f) If the hashCode() method is uncommented the code will not compile

## Resumen

- 📖 Dentro del contexto de las colecciones, las asignaciones que usan polimorfismo son válidas para el tipo base, no para el tipo específico del Collection. Será válido decir:

```
List<Animal> aList = new ArrayList<Animal>(); // yes
```

Será inválido:

```
List<Animal> aList = new ArrayList<Dog>(); // no
```

- 📖 El uso de caracteres comodín dentro de un método genérico, nos permite aceptar subtipos (o supertipos) del tipo declarado en el argumento del método:

```
void addD(List<Dog> d) {} // can take only <Dog>
void addD(List<? extends Dog>) {} // take a <Dog> or <Beagle>
```

- 📖 La palabra clave extends es usada para significar "extends" o "implements". Por lo tanto, en el ejemplo <? extends Dog>, Dog puede ser una clase o una interface.

- 📖 Cuando usamos una definición del tipo: List<? extends Dog>, la colección puede ser accesada pero no modificada por la aplicación.

- 📖 Las convenciones para la declaración de tipos genéricos sugieren utilizar T para tipos y E para elementos:

```
public interface List<E> // API declaration for List
    boolean add(E o)      // List.add() declaration
```

- 📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

- 🔗 <http://javabasico.osmosislatina.com/curso/polimorfismo.htm>

Aquí hallará un interesante artículo sobre el uso de polimorfismo y arreglos en java.

- 🔗 <http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html>

En esta página, hallará información oficial y detallada sobre el uso de genéricos en java.



**UNIDAD DE  
APRENDIZAJE**

**4**

**SEMANA**

**10**

## **CLASES INTERNAS Y GESTIÓN DE HILOS EN JAVA**

### **LOGRO DE LA UNIDAD DE APRENDIZAJE**

- Al término de la unidad, los alumnos crearán aplicaciones que utilicen clases internas de los tipos Inner Classes, Method-Local Inner Classes, Anonymous Inner Classes y Static Nested Classes así como hilos de ejecución utilizando la clase Thread y la interface Runnable, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

### **TEMARIO**

- Clases internas - Definición
- Codificación de una clase interna estándar
- Referencias desde una clase interna
- Clases internas locales a un método – Definición y características
- Clases internas anónimas – Definición y tipos
- Clases internas anónimas como argumentos de un método
- Clases estáticas anidadas – Definición y gestión
- Hilos – Definición
- Estados y ciclo de vida de un hilo
- Priorización de hilos
- Sincronización y bloqueo de código

### **ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucran el uso de clases internas estándares y clases internas locales a un método.

## 1. Clases internas estándares

Se utiliza el término estándar para representar clases internas que no son de:

- Estáticas
- Definidas como parte de un método local.
- Anónimas

Una clase interna es aquella que se define dentro de los límites de una clase “externa”, dentro de los símbolos { } de la clase externa:

```
class MyOuter {  
    class MyInner { }  
}
```

El ejemplo mostrado es bastante simple, y si ejecutamos la siguiente línea de código:

**javac MyOuter.java**

Obtendremos dos archivos .class:

**MyOuter.class**

**MyOuter\$MyInner.class**

La clase interna, por definición, es una clase independiente de la clase externa, y está representada por su propio archivo físico. Sin embargo, una clase interna no puede ser accedida de la manera tradicional, por ejemplo, no sería válido:

```
%java MyOuter$MyInner
```

Tampoco es posible invocar al método main de una clase interna estándar, dado que una clase de este tipo no puede tener declaraciones estáticas de ningún tipo. La única manera de acceder una clase interna es a través de una instancia de la clase externa.

```
class MyOuter {  
    private int x = 7;  
  
    // inner class definition  
    class MyInner {  
        public void seeOuter() {  
            System.out.println("Outer x is " + x);  
        }  
    } // close inner class definition  
  
} // close outer class
```

El código mostrado en líneas previas es totalmente válido. Debemos notar que la clase interna accede a un atributo privado de la clase externa. Eso es correcto, debido a que la clase interna es también un miembro de la clase externa. Como cualquier miembro de la clase externa (por ejemplo, un método de instancia), puede acceder a cualquier otro miembro de la clase externa, privado o no privado.

## 1.1 Instanciando una clase interna

Para crear una instancia de una clase interna, debemos tener una instancia de la clase externa, no existe ninguna excepción a esta regla.

Generalmente, la clase externa usa la clase interna como una clase “helper”. A continuación, se muestra un ejemplo en el que se crea una instancia de la clase interna dentro de la clase externa:

```
class MyOuter {
    private int x = 7;
    public void makeInner() {
        MyInner in = new MyInner(); // make an inner instance
        in.seeOuter();
    }

    class MyInner {
        public void seeOuter() {
            System.out.println("Outer x is " + x);
        }
    }
}
```

El código mostrado anteriormente funciona correctamente: instanciamos la clase interna dentro de la clase externa; sin embargo, ¿existirá alguna manera de instanciar la clase interna desde algún punto externo a la clase externa?

A continuación, se muestra el código que nos permitirá crear una instancia de la clase interna desde cualquier lugar externo:

```
public static void main(String[] args) {
    MyOuter mo = new MyOuter(); // gotta get an instance!
    MyOuter.MyInner inner = mo.new MyInner();
    inner.seeOuter();
}
```

Es válido también que hagamos lo siguiente:

```
public static void main(String[] args) {
    MyOuter.MyInner inner = new MyOuter().new MyInner();
    inner.seeOuter();
}
```

La instanciación de una clase interna es el único escenario en el que invocaremos al operador `new` tal como se visualiza en el ejemplo anterior.

### Importante:

Dentro del código de una instancia de la clase externa, use el nombre de la clase externa de manera normal:

```
MyInner mi = new MyInner();
```

Desde fuera de una instancia de la clase externa, el nombre de la clase interna, debe incluir ahora el nombre de la clase externa:

### **MyOuter.MyInner**

Para crear una instancia debemos usar una referencia a la clase externa:

```
new MyOuter().new MyInner.(); o outerObjRef.new MyInner();
```

## 2. Referenciando instancias desde una clase interna

La referencia **this** es una referencia al objeto que actualmente se está ejecutando; en otras palabras, el objeto cuya referencia fue usada para invocar al método que actualmente se esté ejecutando.

La referencia **this** es la manera en que un objeto puede pasar una referencia de si mismo a otro código como el argumento de un método:

```
public void myMethod() {
    MyClass mc = new MyClass();
    mc.doStuff(this); // pass a ref to object running myMethod
}
```

Dentro del código de una clase interna, la referencia `this`, se refiere a la instancia de la clase interna, dado que `this` siempre se refiere al objeto que se está ejecutando actualmente. Pero ¿Qué sucede si el código de la clase interna requiere tener una referencia explícita a la instancia de la clase externa que la contiene? En otras palabras, ¿cómo referenciamos al `this` externo? A continuación, se muestra un ejemplo que podrá aclarar estas dudas:

```
class MyInner {
    public void seeOuter() {
        System.out.println("Outer x is " + x);
        System.out.println("Inner class ref is " + this);
        System.out.println("Outer class ref is " + MyOuter.this);
    }
}
```

Si ejecutamos el siguiente código:

```
class MyOuter {
    private int x = 7;
    public void makeInner() {
        MyInner in = new MyInner();
        in.seeOuter();
    }
    class MyInner
        public void seeOuter() {
            System.out.println("Outer x is " + x);
            System.out.println("Inner class ref is " + this);
            System.out.println("Outer class ref is " + MyOuter.this);
        }
}
public static void main (String[] args) {
    MyOuter.MyInner inner = new MyOuter().new MyInner();
    inner.seeOuter();
}
```

Obtendremos la siguiente salida:

```
Outer x is 7
Inner class ref is MyOuter$MyInner@113708
Outer class ref is MyOuter@33f1d7
```

Las reglas para que una clase interna se referencie a sí misma o a la instancia de la clase externa que la contiene son las siguientes:

Para referenciar la instancia de la clase interna se referencie a sí misma, dentro del código de la clase interna, usaremos **this**.

Para referenciar el “this externo” (la instancia de la clase externa) desde dentro del código de la clase interna, usemos el nombre de la clase externa seguida del operador this; Por ejemplo, **MyOuter.this**

Los siguientes modificadores pueden ser aplicados a una clase interna:

- final
- abstract
- public
- private
- protected
- static—aunque usar este modificador implicaría tener una clase interna estática, no sólo una clase interna.
- strictfp

### 3. Clases internas locales a un método

Una clase interna local a un método puede ser instanciada sólo dentro del método en que la clase interna es definida. En otras palabras, ningún otro código, ejecutándose en cualquier otro método, dentro o fuera de la clase externa, puede ser instanciado por la clase interna local a un método. Como cualquier clase interna estándar, la clase interna local a un método comparte una relación especial con la clase externa que la contiene y puede acceder a sus miembros privados o identificados con otro modificador. Sin embargo, la clase no puede acceder a las variables locales del método que la contiene.

Dado que no existe garantía alguna de que las variables locales se encuentren disponibles siempre, mientras siga activa la clase interna, no es posible acceder a ellas, a menos que estas variables, sean identificadas como de tipo **final**.

```
class MyOuter2 {
    private String x = "Outer2";
    void doStuff() {
        String z = "local variable";
        class MyInner {
            public void seeOuter() {
                System.out.println("Outer x is " + x);
                System.out.println("Local variable z is " + z); //Won't Compile!
            } // close inner class method
        } // close inner class definition
    } // close outer class method doStuff()
} // close outer class
```

Al compilar el código previo, obtendríamos la siguiente salida:

```
MyOuter2.java:8: local variable z is accessed from within inner class;
needs to be declared final
```

```
    System.out.println("Local variable z is " + z);
                        ^
```

Si hacemos la variable local z, de tipo final, corregiremos el problema:

```
final String z = "local variable"; // Now inner object can use it
```

Recordemos también qué modificadores podemos utilizar dentro de un método: las mismas reglas aplican para una clase interna dentro de un método que para la declaración de variables locales. Por ejemplo, **no es posible** identificar una clase interna dentro de un método como **public**, **private**, **protected**, **static** o **transient**. Los únicos modificadores que podemos utilizar con una clase interna dentro de un método son **abstract** y **final**, pero no ambos a la vez.

#### Importante:

Una clase local declarada dentro de un método estático, tiene acceso solo a los miembros estáticos de la clase que la contiene. Si estamos dentro de un método estático, no existirá el operador **this**; en otras palabras, no será posible acceder a variables de instancia.

### Ejercicios de aplicación:

1. Dada la siguiente clase:

```
1. class Foo {
2.   class Bar{ }
3. }
4. class Test {
5.   public static void main(String[] args) {
6.     Foo f = new Foo();
7.     // Insert code here
8.   }
9. }
```

¿Cuáles de las siguientes líneas de código, al ser insertadas en la línea 7, creará una instancia de la clase interna Foo?

- a) Foo.Bar b = new Foo.Bar();
- b) Foo.Bar b = f.new Bar ();
- c) Bar b = new f.Bar();
- d) Bar b = f.new Bar();
- e) Foo.Bar b = new f.Bar();

2. Dada la siguiente clase:

```
public class Foo {
    Foo() {System.out.print("foo");}
    class Bar{
        Bar() {System.out.print("bar");}
        public void go() {System.out.print("hi");}
    }
    public static void main(String[] args) {
        Foo f = new Foo ();
        f.makeBar();
    }
    void makeBar() {
        (new Bar() {}).go();
    }
}
```

¿Cuál será el resultado de su ejecución?

- a) Compilation fails.
- b) An error occurs at runtime.
- c) foobarhi
- d) barhi
- e) hi
- f) foohi



# Resumen

- 📖 Una clase interna estándar o "regular" es declarada dentro de los símbolos { } de otra clase, pero fuera de cualquier método u otro bloque de código.
- 📖 Una instancia de una clase interna comparte una relación especial con la instancia de la clase que la contiene. Esta relación proporciona a la clase interna acceso a **todos** los miembros de la clase externa, incluyendo aquellos identificados como privados.
- 📖 Para instanciar una clase interna, debemos tener una referencia a la clase externa que la contiene.
- 📖 Una clase interna local a un método es definida dentro de un método de la clase externa que la contiene.
- 📖 Para poder usar la clase interna, debemos instanciarla. La instanciación debe de suceder dentro del mismo método pero después de la definición del código de la clase interna.
- 📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

🖱 <http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html>

Aquí hallará un interesante ejemplo y documentación oficial de java sobre el uso de clases internas.

🖱 <http://www.arrakis.es/~abelp/ApuntesJava/ClasesEmbebidas.htm>

En esta página, hallará un artículo y ejemplos introductorios al uso de clases internas



**UNIDAD DE  
APRENDIZAJE**

**4**

**SEMANA**

**11**

## **CLASES INTERNAS Y GESTIÓN DE HILOS EN JAVA**

### **LOGRO DE LA UNIDAD DE APRENDIZAJE**

- Al término de la unidad, los alumnos crearán aplicaciones que utilicen clases internas de los tipos Inner Classes, Method-Local Inner Classes, Anonymous Inner Classes y Static Nested Classes así como hilos de ejecución utilizando la clase Thread y la interface Runnable, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

### **TEMARIO**

- Clases Internas - Definición
- Codificación de una clase interna estándar
- Referencias desde una clase interna
- Clases internas locales a un método – Definición y características
- Clases internas anónimas – Definición y tipos
- Clases internas anónimas como argumentos de un método
- Clases estáticas anidadas – Definición y gestión
- Hilos – Definición
- Estados y ciclo de vida de un hilo
- Priorización de hilos
- Sincronización y bloqueo de código

### **ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucren el uso de clases internas anónimas, clases internas anónimas como argumento y clases estáticas anidadas.

## 1. Clases internas anónimas – Tipo 1

Dadas las siguientes clases:

```
class Popcorn {
    public void pop() {
        System.out.println("popcorn");
    }
}

class Food {
    Popcorn p = new Popcorn() {

        public void pop() {
            System.out.println("anonymous popcorn");
        }

    };
}
```

Que tal si procedemos a analizarlas y entendemos este nuevo estilo de creación de clases internas:

En primer lugar, estamos definiendo dos clases, Popcorn and Food.

Popcorn tiene un método, pop().

Food tiene una variable de instancia, declarada de tipo Popcorn.

Food no tiene métodos.

Es en este punto donde debemos de prestar mayor atención:

La variable p de Food no referencia a una instancia de Popcorn, sino mas bien a una instancia anónima (sin nombre) de una subclase de Popcorn.

Si analizamos solo el código de la clase anónima:

```
2. Popcorn p = new Popcorn() {
3.   public void pop() {
4.       System.out.println("anonymous popcorn");
5.   }
6. };
```

Podríamos empezar la línea 2 con la creación de una variable de instancia de la clase Popcorn, el código se vería tal como se muestra a continuación:

Popcorn p = new Popcorn(); // notemos el punto y coma al final de la sentencia

Sin embargo, iniciamos llaves ( { ) donde normalmente iría un punto y coma:

Popcorn p = new Popcorn() { // una llave de inicio, no un punto y coma

Podemos leer la línea 2 de la siguiente manera:

Declaramos una variable de referencia, p, de tipo Popcorn. Luego, declaramos una clase nueva que no tiene nombre, pero que es una subclase de Popcorn. Es en este punto donde abrimos llaves “ { “ para iniciar la definición de la clase.

En la línea 3 tenemos la primera sentencia dentro de la definición de la clase. En ella, sobrescribimos el método pop() de la superclase Popcorn.

### Importante:

El punto y coma que termina la definición de la clase interna es a veces difícil de detectar. Podemos notar este detalle en el código mostrado a continuación:

```

2. Popcorn p = new Popcorn() {
3.   public void pop() {
4.     System.out.println("anonymous popcorn");
5.   }
6. }           // Es importante colocar el punto y coma (;) para poder
               // terminar la sentencia iniciada en la línea 2!

7. Foo f = new Foo();
    
```

Se requiere que seamos bastante cuidadosos en términos de sintaxis cuando una clase interna está involucrada. Por ejemplo, el código en la línea 6 se ve perfectamente natural, sin embargo, no es correcto.

Los conceptos básicos de polimorfismo se siguen aplicando cuando trabajamos con clases internas. En el ejemplo anterior, utilizamos una variable de referencia de una super clase para referenciar a una subclase. Esto implica que solo podremos invocar métodos dentro de una clase interna, que estén definidos en el tipo de la variable de referencia. Por ejemplo:

```

class Horse extends Animal{
    void buck() { }
}

class Animal {
    void eat() { }
}

class Test {
    public static void main (String[] args) {
        Animal h = new Horse();
        h.eat(); // Legal, la clase Animal tiene el método eat()
        h.buck(); // No legal!, la clase Animal no tiene el método buck()
    }
}
    
```

Debemos tener presente también que una clase interna anónima, en vez de sobrescribir un método de la clase padre, puede también definir su propio método. Dentro de este contexto, el tipo de la variable de referencia (superclase) no conoce el nuevo método (definido en la subclase anónima), por lo tanto, el compilador generará un error, si tratamos de invocar cualquier método en una clase interna anónima, que no esté definido en la superclase. Se muestra, a continuación, el siguiente ejemplo:

```
class Popcorn {
    public void pop() {
        System.out.println("popcorn");
    }
}

class Food {
    Popcorn p = new Popcorn() {
        public void sizzle () {
            System.out.println("anonymous sizzling popcorn");
        }
        public void pop() {
            System.out.println("anonymous popcorn");
        }
    };

    public void popIt() {
        p.pop(); // OK, Popcorn tiene un método pop()
        p.sizzle(); // Not Legal! Popcorn no tiene el método sizzle()
    }
}
```

Al compilar el ejercicio previo obtendríamos el siguiente mensaje de error:

```
Anon.Java:19: cannot resolve symbol
symbol : method sizzle ()

location: class Popcorn
    p.sizzle();
    ^
```

Podemos interpretar el mensaje del compilador de la siguiente manera: "No puedo encontrar el método sizzle() en la clase Popcorn"

## 2. Clases internas anónimas – Tipo 2

La única diferencia entre las clases internas anónimas de tipo 1 y las clases internas anónimas de tipo 2 es que las primeras crean una subclase anónima de un tipo de clase especificado. Las segundas crean un implementador anónimo del tipo de interface especificado. En los ejemplos que hemos revisado hasta el momento, hemos definido subclases anónimas del tipo PopCorn, tal como se muestra a continuación:

```
Popcorn p = new Popcorn() {
```

Pero si Popcorn fuera una interface, entonces la nueva clase clase anónima sería una implementadora de dicha interface:

```
interface Cookable {
    public void cook();
}

class Food {
    Cookable c = new Cookable() {
        public void cook() {
            System.out.println("anonymous cookable implementer");
        }
    };
}
```

En el ejemplo anterior creamos una clase interna anónima, la cual es una clase implementadora de la interface Cookable. Notemos que este es el único caso en que veremos sintaxis tal como se muestra a continuación:

**new Cookable()**

Cookable es una interface y no una clase no abstracta. Sabemos que **no es posible** instanciar una interface, sin embargo, eso es lo que aparentemente estamos haciendo en el ejercicio previo. En realidad, no estamos creando un objeto Cookable, sino, una instancia anónima, implementadora de la interface Cookable.

Las clases anónimas implementadoras de interfaces pueden implementar **solo una interface**. No existe mecanismo alguno para indicar que nuestra clase anónima implementará más de una interface. Una clase interna anónima tampoco puede implementar una interface y heredar de una clase a la vez. Si una clase interna anónima es subclase de una clase en particular, la clase anónima será automáticamente implementadora de cualquier interface que implemente la superclase.

### Importante:

El siguiente código no es legal:

```
Runnable r = new Runnable (); // no podemos instanciar una interface
```

El siguiente código es legal:

```
Runnable r = new Runnable() { // abrimos llaves, no usamos punto y coma:
    // “;”
    public void run() { }
};
```

### 3. Clases internas anónimas definidas como argumento

Imaginemos el siguiente escenario:

```
1. class MyWonderfulClass {
2.   void go() {
3.     Bar b = new Bar();
4.     b.doStuff(new Foo() {
5.       public void foof() {
6.         System.out.println("foofy");
7.       } // end foof method
8.     }); // end inner class def, arg, and b.doStuff stmt.
9.   } // end go()
10. } // end class
11.
12. interface Foo {
13.   void foof();
14. }
15. class Bar {
16.   void doStuff(Foo f) {}
17. }
```

Haremos el análisis a partir de la línea 4. Estamos invocando al método `doStuff` de un objeto de tipo `Bar`. Pero el método recibe una instancia de tipo **IS-A** `Foo`, donde `Foo` es una interface. Lo que hacemos a continuación es crear una clase implementadora de `Foo`, dentro de la llamada al método `doStuff()`.

Decimos:

```
new Foo() {
```

para iniciar la definición de la nueva clase anónima que implementará la interface `Foo`. `Foo` tiene un solo método, `foof()`, en las líneas 5, 6, y 7 implementamos el método `foof()`. En la línea 8 terminamos la definición de la clase implementadora con la siguiente sintaxis:

```
});
```

Mientras que, si trabajamos con una clase interna de los tipos 1 o 2, la terminación es la siguiente:

```
};
```



## 4. Instanciando y utilizando clases estáticas anidadas.

Debemos usar la sintaxis estandar de Java para acceder a una clase anidada estática desde la clase externa que la contiene. La sintaxis para instanciar una clase anidada estática desde una clase distinta a la que la contiene es ligeramente diferente a la utilizada con una clase interna normal:

```
class BigOuter {
    static class Nest {void go() { System.out.println("hi"); } }
}
class Broom {
    static class B2 {void goB2() { System.out.println("hi 2"); } }
    public static void main(String[] args) {
        BigOuter.Nest n = new BigOuter.Nest(); // both class names
        n.go();
        B2 b2 = new B2(); // access the enclosed class
        b2.goB2();
    }
}
```

Obtendremos la siguiente salida:

```
hi
hi 2
```

### Importante:

De la misma manera en que un método estático no tiene acceso a las variables de instancia y a los métodos no estáticos de una clase, una clase anidada estática, no tiene acceso a las variables de instancia y a los métodos no estáticos de la clase externa que la contiene.

**Ejercicios de aplicación:**

1. Dada la siguiente clase:

```
public abstract class AbstractTest {
    public int getNum() {
        return 45;
    }
    public abstract class Bar {
        public int getNum() {
            return 38;
        }
    }
    public static void main(String[] args) {
        AbstractTest t = new AbstractTest() {
            public int getNum() {
                return 22;
            }
        };
        AbstractTest.Bar f = t.new Bar() {
            public int getNum() {
                return 57;
            }
        };
        System.out.println(f.getNum() + " " + t.getNum());
    }
}
```







¿Cuál será el resultado obtenido?


- a) 57 22
- b) 45 38
- c) 45 57
- d) An exception occurs at runtime.
- e) Compilation fails.

2. ¿Cuáles de las siguientes afirmaciones son verdaderas en relación a las clases internas locales a un método específico? (Seleccione todas las opciones válidas.)

- a) Debe ser marcada como final.
- b) Puede ser marcada como abstract.
- c) Puede ser marcada como public.
- d) Puede ser marcada como static.
- e) Puede acceder a los miembros privados de la clase que la contiene.

## Resumen

-  Las clases internas anónimas no tienen nombre y son subclases de un tipo en particular o implementadores de una interfaz específica.
-  Una clase interna anónima es siempre creada como parte de una sentencia. No debemos olvidar cerrar la sentencia después de la definición de la clase con una llave y un punto y coma. Este es un caso especial de Java en el que una llave es seguida de un punto y coma.
-  Las clases anidadas estáticas son clases internas identificadas con el modificador static.
-  Una clase anidada estática no comparte relación alguna con una instancia de la clase externa. Por lo tanto, no necesitamos una instancia de la clase externa para instanciar una clase interna estática.
-  Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.
  -  <http://www.developer.com/java/other/article.php/3300881>

Aquí hallará un interesante artículo acerca del uso de clases anónimas en Java.
  -  <http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html>

En esta página, hallará documentación oficial acerca del uso de clases anidadas en Java.

**UNIDAD DE  
APRENDIZAJE**

**4**

**SEMANA**

**12**

## **CLASES INTERNAS, GESTIÓN DE HILOS EN JAVA Y DESARROLLO DE APLICACIONES JAVA**

### **LOGRO DE LA UNIDAD DE APRENDIZAJE**

- Al término de la unidad, los alumnos crearán aplicaciones que utilicen clases internas de los tipos Inner Classes, Method-Local Inner Classes, Anonymous Inner Classes y Static Nested Classes así como hilos de ejecución utilizando la clase Thread y la interface Runnable, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

### **TEMARIO**

- Clases Internas - Definición
- Codificación de una clase interna estándar
- Referencias desde una clase interna
- Clases internas locales a un método – Definición y características
- Clases internas anónimas – Definición y tipos
- Clases internas anónimas como argumentos de un método
- Clases estáticas anidadas – Definición y gestión
- Hilos – Definición
- Estados y ciclo de vida de un hilo
- Priorización de hilos
- Sincronización y bloqueo de código

### **ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucran el uso de threads utilizando la interface Runnable y la clase Thread.

## 1. Definiendo un Thread

Para definir un hilo (thread), necesitamos contar con un lugar para colocar nuestro método run(). Ésto lo podemos hacer de dos formas básicas:

- Heredando de la clase Thread o
- Implementando la interface Runnable.

### 1.1 Heredando de la clase java.lang.Thread

La manera más simple de definir código que se ejecute en un thread de manera independiente es heredando de la clase the java.lang.Thread.

Debemos sobrecribir el método run().

Un ejemplo podría ser el siguiente:

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Important job running in MyThread");  
    }  
}
```

La limitación de este enfoque es que al heredar de la clase Thread no podremos heredar de ninguna otra clase. Esta limitación es importante, dado que si queremos utilizar threads de manera práctica, nuestra clase debe heredar de alguna diferente de Thread.

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Important job running in MyThread");  
    }  
    public void run(String s) {  
        System.out.println("String in run is " + s);  
    }  
}
```

La clase thread espera contemos con un método run() sin argumentos, el cual será ejecutado después de que el Thread haya sido iniciado.

### 1.2 Implementando la interface Runnable

Al implementar la interface Runnable tenemos la posibilidad de heredar de cualquier clase y, a la vez, definir un comportamiento que será ejecutado por un thread de manera independiente:

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Important job running in MyRunnable");  
    }  
}
```

## 2. Instanciando un Thread

Cada thread de ejecución inicia con una instancia de la clase Thread. Sin importar si nuestro método run() está dentro de una subclase de la clase Thread o en una clase implementadora de la interface Runnable, necesitamos contar con un objeto de la clase Thread para empezar a trabajar:

Si heredamos de la clase Thread, debemos hacer lo siguiente:

```
MyThread t = new MyThread( )
```

Si implementamos la interface Runnable:

Creamos una instancia de la clase implementadora de Runnable:

```
MyRunnable r = new MyRunnable();
```

Luego, creamos una instancia de Thread y le pasamos el objeto Runnable como argumento de su constructor:

```
Thread t = new Thread(r);
```

Si creamos un thread usando un constructor sin argumentos, el thread invocará a su propio método run(). Ese es el comportamiento que queremos tener cuando heredamos de la clase Thread, pero cuando usamos la interface Runnable, necesitamos indicar al nuevo thread que use nuestro método run() y no el suyo. Es por ello que pasamos el objeto Runnable al constructor del nuevo Thread.

Podemos pasar el mismo objeto Runnable a muchos objetos de tipo Thread:

```
public class TestThreads {
    public static void main (String [] args) {
        MyRunnable r = new MyRunnable();
        Thread foo = new Thread(r);
        Thread bar = new Thread(r);
        Thread bat = new Thread(r);
    }
}
```

Al proporcionar el mismo objeto Runnable a multiples Threads, logramos que varios threads de ejection ejecuten el mismo trabajo (y que el mismo trabajo se ejecute multiples veces).

### Importante:

La clase Thread en sí misma implementa la interface Runnable. Esto significa que podemos pasar al constructor de un thread una instancia de la clase Thread:

```
Thread t = new Thread(new MyThread());
```

Esto no es muy eficiente, pero es válido. En el ejemplo, requerimos, realmente, un objeto Runnable; crear un Thread completo para pasarlo como argumento no es óptimo.

A continuación se muestran los principales constructores de la clase Thread:

- Thread()
- Thread(Runnable target)
- Thread(Runnable target, String name)
- Thread(String name)

Cuando un thread ha sido instanciado pero no iniciado (en otras palabras, no hemos invocado al método start() ), se dice que el thread se encuentra en el estado **new** . En este estado, aún no se considera que el thread se encuentre vivo. Una vez que invocamos al método start() el thread es considerado vivo. Un thread es considerado muerto, una vez que el método run() ha completado su ejecución. El uso del método isAlive() es la mejor manera de determinar si un thread ha sido iniciado pero aún no ha completado la ejecución de su método run().

### 3. Instanciando un Thread

Para poder iniciar la ejecución de un thread una vez instanciado, debemos de invocar a su método start():

```
t.start();
```

Previamente a la invocación del método start() en una instancia de la clase Thread, el thread se encuentra en el estado **new**. El estado new significa que tenemos un objeto de la clase Thread, pero aún no un thread de ejecución realmente.

- ¿Qué sucede después de invocar al método start()?
- Un Nuevo thread de ejecución es iniciado.
- El thread se mueve del estado new al estado **runnable**.
- Cuando el thread tenga la oportunidad de ejecutar, su método run() destina a ejecutar dicho método.

Debemos tener presente siempre lo siguiente: Nosotros iniciamos la ejecución de un Thread, no un Runnable. Podemos invocar al método start() en una instancia



de la clase Thread, no en una instancia que implemente la interface Runnable. El ejemplo mostrado a continuación involucra todos los conceptos que hemos aprendido hasta este momento sobre threads:

```
class FooRunnable implements Runnable {
    public void run() {
        for(int x =1; x < 6; x++) {
            System.out.println("Runnable running");
        }
    }
}

public class TestThreads {
    public static void main (String [] args) {
        FooRunnable r = new FooRunnable();
        Thread t = new Thread(r);
        t.start();
    }
}
```

Si ejecutamos el código anterior, obtendremos la siguiente salida:

```
% java TestThreads
Runnable running
Runnable running
Runnable running
Runnable running
Runnable running
```

### Importante:

Si tenemos código que invoca al método run() dentro de un objeto Runnable o en una instancia de Thread, es totalmente válido. Pero no significa que el método run() se esté ejecutando en un thread separado; es decir, no se inicia ningún nuevo thread, sino que se ejecutará como cualquier método del thread actual que se esté ejecutando.

```
Runnable r = new Runnable();
r.run(); // Legal, pero no se inicia ningún thread realmente.
```

¿Qué sucede si iniciamos múltiples threads? A continuación, se muestra un ejemplo de aplicación que aclarará esta pregunta:

```
class NameRunnable implements Runnable {
    public void run() {
        System.out.println("NameRunnabl.e running");
        System.out.println("Run by "
            + Thread.currentThread().getName());
    }
}
```

```

public class NameThread {
    public static void main (String [] args) {
        NameRunnable nr = new NameRunnable();
        Thread t = new Thread(nr);
        t.setName("Fred");
        t.start();
    }
}

```

La ejecución del código anterior generará la siguiente salida adicional:

```

% java NameThread
NameRunnable running
Run by Fred

```

Para obtener el nombre de un thread debemos de invocar al método **getName()** de la instancia del thread. Pero la instancia de la interface Runnable no tiene una referencia a la instancia del Thread. Es por ello que invocamos al método estático **Thread.currentThread()**, el cual nos retorna una referencia al thread que actualmente se está ejecutando y, luego, invocamos al método **getName()**.

Incluso, si no asignamos explícitamente un nombre a un thread, éste tendrá uno. A continuación, se muestra un ejemplo de aplicación:

```

public class NameThread {
    public static void main (String [] args) {
        NameRunnable nr = new NameRunnable();
        Thread t = new Thread(nr);
        // t.setName("Fred");
        t.start();
    }
}

```

Si ejecutamos el código anterior obtendremos:

```

% java NameThread
NameRunnable running
Run by Thread-0

```

Dado que obtenemos el nombre del thread actual usando el método estático **Thread.currentThread()**, podremos obtener el nombre del thread ejecutando nuestro código principal:

```

public class NameThreadTwo {
    public static void main (String [] args) {
        System.out.println("thread is "
            + Thread.currentThread().getName());
    }
}

```

### 3.1 Inicio y ejecución múltiples hilos

Ejecutar múltiples hilos quiere decir, generalmente, ejecutar más de dos simultáneamente. Esto porque, normalmente, ya tenemos dos threads ejecutándose a la vez: el método main() se ejecuta en su propio thread de ejecución y cuando decimos dentro de este método, por ejemplo t.start(), un segundo thread es ejecutado. El código mostrado a continuación crea una instancia de la clase Runnable y tres instancias de la clase Thread. Las tres instancias de Thread utilizan la instancia de Runnable y cada thread tendrá un nombre distinto.

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x <= 3; x++) {
            System.out.println("Run by "
                + Thread.currentThread().getName()
                + ", x is " + x);
        }
    }
}

public class ManyNames {
    public static void main(String [] args) {
        // Make one Runnable
        NameRunnable nr = new NameRunnable();
        Thread one = new Thread(nr);
        Thread two = new Thread(nr);
        Thread three = new Thread(nr);

        one.setName("Fred");
        two.setName("Lucy");
        three.setName("Ricky");
        one.start();
        two.start();
        three.start();
    }
}
```

La ejecución de este código generará la siguiente salida:

```
% java ManyNames
Run by Fred, x is 1
Run by Fred, x is 2
Run by Fred, x is 3

Run by Lucy, x is 1
Run by Lucy, x is 2
Run by Lucy, x is 3
Run by Ricky, x is 1
Run by Ricky, x is 2
Run by Ricky, x is 3
```

Sin embargo, el comportamiento obtenido en líneas previas “no está garantizado”. Es importante tener presente que no existe en la especificación java nada que indique a los threads que inicien su ejecución en el orden que fueron iniciados (started). Tampoco hay ninguna garantía de que, una vez un thread es iniciado, éste se mantendrá ejecutándose hasta que termine su lógica o que una iteración se completará siempre antes que otro thread empiece su ejecución. La única garantía que existe es la siguiente:

**Cada thread se iniciará, y cada thread se ejecutará hasta completar su lógica.**

Dentro de cada thread, las cosas pasarán en un orden predecible; sin embargo, las acciones de diferentes threads pueden mezclarse de maneras impredecibles. Si ejecutamos un programa múltiples veces o en múltiples máquinas, observaremos diferentes salidas. En el siguiente ejemplo, iteraremos 400 veces en vez de 3 (iteración del ejercicio original):

```
public void run() {  
    for (int x = 1; x <= 400; x++) {  
        System.out.println("Run by "  
            + Thread.currentThread().getName()  
            + ", x is " + x);  
    }  
}
```

La ejecución del ejemplo anterior nos permite comprobar que ésta no es lineal:

```
Run by Fred, x is 345  
  
Run by Lucy, x is 337  
  
Run by Ricky, x is 310  
  
Run by Lucy, x is 338  
  
Run by Ricky, x is 311  
  
Run by Lucy, x is 339  
  
Run by Ricky, x is 312  
  
Run by Lucy, x is 340  
  
Run by Ricky, x is 313  
  
Run by Lucy, x is 341  
  
Run by Ricky, x is 314  
  
Run by Lucy, x is 342  
  
Run by Ricky, x is 315
```

...

Podemos notar que no existe ningún patrón claro en la ejecución del ejemplo anterior y es que, para cualquier grupo de threads iniciados, no existe ningún orden garantizado de ejecución por el scheduler. Sin embargo, existe una manera de iniciar un thread y decirle que no se ejecute hasta que algún otro thread haya terminado. Podemos hacer esto utilizando el método `join()`.

Un thread termina su ejecución cuando se completa la ejecución de su método `run()`.

Cuando un thread completa su método `run()`, el thread deja de ser un thread de ejecución (hilo de ejecución) y es considerado muerto (`dead`). No muerto he ido (`"Not dead and gone"`), pero si muerto. Esto significa que es aún un objeto de la clase `Thread`, a pesar de no ser ya un thread de ejecución: podemos aún invocar métodos en el objeto thread, pero lo que no podremos hacer es invocar nuevamente al método `start()`.

**Una vez que un thread ha sido iniciado, nunca podrá ser iniciado nuevamente.**

Si tenemos una referencia a un objeto `Thread` e invocamos al método `start()`, iniciamos el thread. Si invocamos al método `start()` por segunda vez, se generará una excepción (`IllegalThreadStateException`, la cual es un tipo de `RuntimeException`). Esto sucede se haya completado o no la ejecución del método `run()` del thread. Un thread en plena ejecución (estado `runnable`) o un thread muerto (`dead`) **no** pueden ser nunca reiniciados.

#### **Importante:**

Hasta el momento ya hemos vistos los siguientes estados de un thread: `new`, `runnable`, y `dead`.

### **3.2 El Thread Scheduler**

El thread scheduler es la parte de la JVM que decide que thread debe ejecutarse en un momento dado y, también, puede sacar a un thread del estado **run**. Si asumimos que tenemos solo un procesador, solo un thread podrá ejecutarse a la vez. Es el thread scheduler el que decide qué thread, de todos los elegibles, se ejecutará. Un thread elegible es aquel que se encuentra en el estado **runnable**.

Cualquier thread en el estado `runnable` puede ser seleccionado por el scheduler para ser el único en ejecución.

#### **Importante:**

El orden en el cual los threads en estado `runnable` son seleccionados para ejecutarse no está garantizado de ninguna manera.

Aunque el comportamiento basado en colas es típico, este comportamiento no está garantizado. Comportamiento basado en colas significa que, cuando un

thread ha terminado con su turno, éste se mueve al final de la línea de todos los objetos runnable (pool de objetos runnable) y espera hasta que “eventualmente” sea el primero de la línea de modo que pueda ser seleccionado nuevamente por el scheduler

Aunque no controlamos el thread scheduler, si podemos influenciarlo. Los métodos mostrados a continuación nos proporcionan algunas herramientas para influenciar el scheduler (Aunque jamás para controlarlo).

### **Importante:**

Debemos tener muy claro, dentro del contexto de los threads, cuándo un comportamiento está garantizado y cuándo no. Debemos ser capaces de revisar la codificación de cualquier thread y determinar si se ejecutará de una manera en particular o indeterminada, si podemos garantizar su salida o no.

### **Métodos de la clase `java.lang.Thread`**

Algunos de los métodos que nos pueden ayudar a trabajar con el thread scheduler son:

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

Notemos que los métodos `sleep()` y `join()` tienen versiones sobrecargadas que son detalladas en el presente documento.

### **Métodos de la clase `java.lang.Object`**

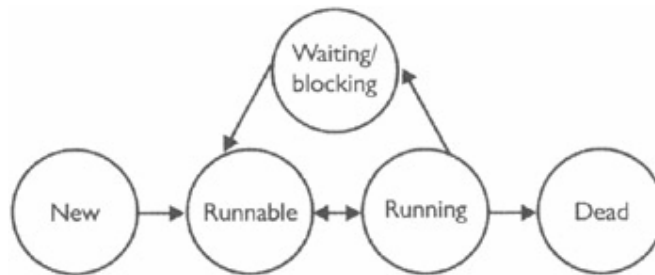
Cada clase en java hereda los siguientes tres métodos relacionados con los threads:

- `public final void wait() throws InterruptedException`
- `public final void notify()`
- `public final void notifyAll()`

El método `wait()` cuenta con tres versiones sobrecargadas incluyendo la versión mostrada en líneas previas.

## **4. Estados de un Thread**

Los threads pueden asumir sólo uno de los siguientes estados:



### New

Este es el estado en que un thread se encuentra después de que una instancia de la clase Thread es creada, pero aún no se ha invocado a su método start(). Es un objeto de la clase, pero no un thread de ejecución.

### Runnable

Un thread se encuentra en este estado cuando es elegible para ser ejecutado, pero el scheduler aún no lo ha seleccionado para ser un thread en ejecución. Un thread ingresa al estado runnable cuando el método start() es invocado, pero un thread puede regresar al estado runnable después haberse estado ejecutando o desde los estados blocked, waiting, o sleeping. Cuando un thread se encuentra en el estado runnable, es considerado un **thread vivo**.

### Running

Este es el estado en el que el thread scheduler selecciona a un thread (desde el pool de threads en estado runnable) para ser el proceso que se esté ejecutando actualmente. Un thread puede salir del estado running por varias razones. Podemos observar en el gráfico previo que existen varias maneras de llegar al estado runnable, pero sólo una [ha](#) de llegar al estado running: el scheduler escoge un thread del pool de threads en estado runnable.

### Waiting/blocked/sleeping

Este es el estado en que un thread se encuentra cuando es elegible para ser runnable. En realidad, debemos de entender que son tres estados combinados en uno donde todos estos posibles estados tienen algo en común: el thread está todavía vivo, pero no es elegible para ser ejecutado. En otras palabras, no se encuentra en el estado runnable, pero podría retornar a dicho estado posteriormente, si un evento en particular ocurre. Un thread puede estar bloqueado, esperando por un recurso. En este caso, el evento que enviará al thread nuevamente al estado runnable será la disponibilidad de dicho recurso. Un thread puede encontrarse en estado sleeping porque el código de ejecución del thread puede indicarle que “descanse” por un período de tiempo específico, en cuyo caso el evento que retornará al thread al estado runnable es que éste “despierte” cuando su tiempo de descanso haya expirado. O el thread puede estar en estado waiting, porque el código de ejecución del thread puede provocar que el

thread pase a un estado de espera, en cuyo caso el evento que enviará al thread nuevamente al estado runnable será que otro thread envíe una notificación indicando que ya no es necesario que continúe la espera.

### Importante:

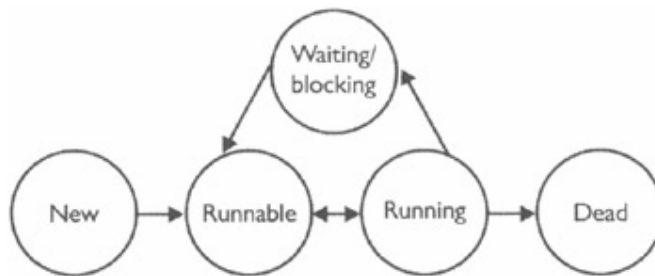
Un thread no indica a otro que debe de bloquearse, puede parecer que algunos métodos indican a otro thread que deben de bloquearse, sin embargo, no lo hacen. Si tenemos una referencia a otro thread, podríamos escribir algo como lo siguiente:

```
t.sleep(); o t.yield()
```

Pero estos métodos son métodos estáticos de la clase Thread, éstos no afectan a la instancia t, en vez de ello, siempre afectarán al thread que actualmente se encuentre ejecutando. (Este es un buen ejemplo de por qué es una mala idea usar una variable de instancia para acceder a un método estático).

### Dead

Un thread es considerado muerto cuando su método run() ha sido completado, puede ser aún un objeto de la clase Thread, pero ya no es un thread de ejecución. Una vez que un thread está muerto, este nunca podrá retornar a la vida. Si invocamos al método start() de un thread muerto, obtendremos un error en tiempo de ejecución (no de compilación).



Un thread que ha sido detenido usualmente es interpretado como un thread que ha sido movido al estado dead. Sin embargo, debemos ser capaces de reconocer cuando un thread ha sido sacado del estado running, pero no ha sido enviado al estado runnable o dead.

## 5. El método sleep

El método sleep() es un método estático de la clase Thread. Usamos este método en nuestro código para hacer más lenta la ejecución de nuestro thread, forzándolo a ingresar al estado sleep antes de que retorne al estado runnable.



¿Por qué necesitaríamos hacer que un thread pase al estado `sleep`?

A continuación, planteamos un ejemplo como respuesta: Imaginemos que un thread se está ejecutando dentro de un loop, descargando los últimos precios de un producto y analizándolos. La descarga de precios, uno tras otro, puede ser una pérdida de tiempo y más aún, de ancho de banda. La manera más simple de resolver este inconveniente es provocando por ejemplo la pausa de un thread (`sleep`) por cinco minutos, después de cada descarga.

```
try {
    Thread.sleep(5*60*1000); // Sleep for 5 minutes
} catch (InterruptedException ex) { }
```

Notemos que el método `sleep()` puede lanzar una excepción de tipo `InterruptedException` (esto es posible, si es que otro thread explícitamente interrumpe su ejecución).

Modifiquemos ahora nuestro ejercicio original para hacer uso del método `sleep()`:

```
class NameRunnable implements Runnable {
    public void run() {
        for (int x = 1; x < 4; x++) {
            System.out.println("Run by "
                + Thread.currentThread().getName());

            try {
                Thread.sleep(1000);
            } catch (InterruptedException ex) { }
        }
    }
}

public class ManyNames {
    public static void main (String [] args) {

        // Make one Runnable
        NameRunnable nr = new NameRunnable();

        Thread one = new Thread(nr);
        one.setName("Fred");
        Thread two = new Thread(nr);
        two.setName("Lucy");
        Thread three = new Thread(nr);
        three.setName("Ricky");

        one.start();
        two.start();
        three.start();
    }
}
```

La ejecución del código previo muestra como Fred, Lucy, y Ricky son visualizados de manera alternativa:

```
% java ManyNames
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Lucy
Run by Ricky
Run by Fred
Run by Lucy
Run by Ricky
```

Debemos tener muy presente que el comportamiento visualizado en líneas anteriores **no es garantizado**. El uso del método `sleep()` es la mejor manera de ayudar a todos los threads a obtener una oportunidad para ejecutarse. Cuando un thread encuentra una invocación al método `sleep()`, éste debe descansar, al menos, el número de milisegundos especificado en el método (a menos que sea interrumpido antes de la culminación de dicho tiempo, lanzándose inmediatamente una `InterruptedException`).

### Importante:

Que el tiempo especificado en el método `sleep()` expire, y por lo tanto, el thread despierte, no significa que el thread retornará al estado `running` inmediatamente. Recordemos que, cuando un thread despierta, este simplemente regresa al estado `runnable`. Por lo tanto, el tiempo especificado en el método `sleep()` es el tiempo mínimo que el thread no se ejecutará. No es la duración exacta del tiempo que no se ejecutará.

El tiempo expresado en el método `sleep()` **no es una garantía** de que el thread empiece a ejecutarse nuevamente tan pronto el tiempo expire y el thread despierte.

Recordemos que el método `sleep()` es un método estático, un thread jamás podrá poner “a dormir” a otro.

A continuación, crearemos un thread con un contador. Este thread contará hasta 100 generando una pausa de un segundo entre cada número. En conformidad con el conteo de números, visualizará un `String` cada diez números.

Una posible solución podría ser la siguiente:

```
class TheCount extends Thread {
    public void run() {
        for(int i = 1; i <= 100; ++i) {
            System.out.print(i + " ");
        }
    }
}
```

```

        if(i % 10 == 0) System.out.println("Hahaha");
        try { Thread.sleep(1000); }
        catch(InterruptedException e) {}
    }
}
public static void main(String [] args) {
    new TheCount().start();
}
}

```

## 6. Prioridades de un Thread y el método yield()

Para entender el método yield(), debemos entender el concepto de prioridades de un thread. Los threads siempre se ejecutan con la misma prioridad, usualmente representada por un número entre 1 y 10 (aunque en algunos casos el rango es menor de 10).

En la mayoría de JVMs, el scheduler usa las prioridades de un thread de una manera muy importante: si un thread ingresa al estado runnable, y tiene una prioridad más alta que cualquiera de los otros threads en el pool y una prioridad más alta que el thread que se está ejecutando actualmente, el thread que actualmente se está ejecutando (y que es de menor prioridad) será enviado al estado runnable y el thread de mayor prioridad será seleccionado para ser ejecutado.

Dado que el comportamiento de las prioridades dentro del scheduler **no está garantizado**, solo debemos usar las prioridades para mejorar la eficiencia de nuestro programa.

**Tampoco existe garantía alguna del comportamiento** cuando los threads en el pool son de la misma prioridad o cuando el thread que actualmente se está ejecutando tiene la misma prioridad que los threads en el pool. Si todas las prioridades son iguales, entonces el scheduler es libre de hacer lo que estime conveniente; por ejemplo, podría:

- Seleccionar un thread para que se ejecute, y ejecutarlo hasta que se bloquee o termine.
- División de tiempo entre los threads del pool, para proporcionarle a todos ellos la oportunidad de ejecutarse.

### Definiendo la prioridad de un Thread:

Un thread obtiene una prioridad por defecto, la cual es la prioridad del thread de ejecución que lo creó. Por ejemplo:

```

public class TestThreads {
    public static void main (String [] args) {
        MyThread t = new MyThread();
    }
}

```

El thread referenciado por `t` tendrá la misma prioridad que el thread main, debido a que el thread main está ejecutando el código que crea la instancia de la clase `MyThread`.

Podemos, también, definir la prioridad de un thread directamente invocando al método `setpriority()`:

```
FooRunnable r = new FooRunnable();
Thread t = new Thread(r);
t.setPriority(8);
t.start();
```

Las prioridades son definidas usando un valor entero positivo entre 1 y 10 y la JVM nunca cambiará las prioridades de un Thread; sin embargo, los valores entre 1 y 10 **no están garantizados**. Algunas JVMs podrían no reconocer 10 valores distintos y, quizás, podrían manejar valores equivalentes, por ejemplo para un rango entre 1 y 5.

Aunque la prioridad por defecto es 5, la clase Thread tiene las siguientes constantes (variables de tipo static final) que definen el rango de las prioridades de un thread:

```
Thread.MIN_PRIORITY (1)
Thread.NORM_PRIORITY (5)
Thread.MAX_PRIORITY (10)
```

### ¿Cuál es la utilidad del método `yield()` ?

En la práctica, no es mucha su utilidad. Se supone que `yield()` hace que el thread que actualmente se está ejecutando regrese al estado runnable para permitir que otros threads con su misma prioridad puedan obtener su turno. En realidad, **no existe garantía alguna** de que `yield()` pueda hacer lo que en teoría debería lograr. Incluso, si logra que un thread pase de estado running a runnable, tampoco existe garantía alguna de que dicho thread sea seleccionado nuevamente (inmediatamente) por sobre todos los otros threads.

El método `yield()` no provocará jamás que un thread cambie al estado waiting/sleeping/ blocking. En la mayoría de los casos, provocará que un thread pase de estado running a runnable, aunque ello podría no tener ningún efecto en lo absoluto.

## 6.1 El método `Join()`

El método no estático `join()` de la clase Thread permite que un thread **se una al final** de otro thread. Si por ejemplo, tenemos un thread B que no puede hacer su trabajo hasta que otro thread (el thread A) haya completado el suyo, entonces lo que queremos es que el thread B **se una** al thread A. Esto significa que el thread B no estará en estado runnable hasta que A haya terminado e ingresado al estado dead.

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

El código anterior toma el thread que se está ejecutando actualmente (si esto estuviera en el método `main()`), entonces tomaríamos el thread principal ) y lo une al final del thread referenciado por `t`. Esto bloquea al thread actual para que pueda ser runnable hasta que el thread referenciado por `t` deje de estar vivo. En otras palabras, el código `t.join()` significa:

"Únete a mi (thread actual) al final de la ejecución de `t`, dado que `t` debe terminar antes que yo (el thread actual) “.

Podemos bloquear de tres maneras un thread en ejecución de modo que salga de su estado running:

**Una llamada a `sleep()`.** Se garantiza que el thread actual detenga su ejecución por al menos el tiempo la duración especificada de descanso (aunque podría ser interrumpido antes del tiempo especificado).

**Una llamada a `yield()`.** No existe la garantía de hacer mucho, aunque típicamente la llamada a este método provocará que el thread que actualmente se esté ejecutando retorne al estado runnable de modo que un thread con la misma prioridad tenga, también, la oportunidad de ejecutarse.

**Una llamada a `join()`.** Se garantiza que el thread actual detenga su ejecución hasta que el thread al cual se una haya terminado su ejecución o deje de estar vivo.

Adicionalmente a estas tres llamadas, existen algunos escenarios en los que un thread podría dejar de estar ejecutándose:

El método `run` del thread termina su ejecución.

Una llamada al método `wait()` de un objeto (Nosotros no invocamos al método `wait()` de un thread).

El thread scheduler puede decidir mover el thread actual de su estado running a runnable para proporcionar a otro thread la oportunidad de ejecutarse. Ninguna razón específica es necesaria, el thread scheduler puede cambiar entre los estados mencionados a un thread cuando desee hacerlo.

## Ejercicios de aplicación:

1. Dada la siguiente clase:

```
public class Messenger implements Runnable {  
    public static void main(String[] args) {  
        new Thread(new Messenger("Wallace")).start() ;  
        new Thread(new Messenger("Gromit")).start();  
    }  
    private String name;  
    public Messenger(String name) { this.name = name; }  
    public void run() {  
        message(1);  
        message(2);  
    }  
    private synchronized void message(int n) {  
        System.out.print(name + "-" + n + " ");  
    }  
}
```

¿Cuáles son las posibles respuestas válidas?

- a) Wallace-1 Wallace-2 Gromit-1
- b) Wallace-1 Gromit-2 Wallace-2 Gromit-1
- c) Wallace-1 Gromit-1 Gromit-2 Wallace-2
- d) Gromit-1 Gromit-2
- e) Gromit-2 Wallace-1 Gromit-1 Wallace-2
- f) The code does not compile.
- g) An error occurs at run time.

2. Dada la siguiente clase:

```
public class Letters extends Thread {
    private String name;
    public Letters(String name) {
        this.name = name;
    }

    public void write () {
        System.out.print(name);
        System.out.print(name);
    }
    public static void main(String[] args) {
        new Letters("X").start();
        new Letters("Y").start();
    }
}
```

Deseamos garantizar que la salida pueda ser, ya sea XXYX o YYXX, pero nunca XYYX o cualquier otra combinación. ¿Cuáles de las siguientes definiciones de métodos pueden ser agregadas a la clase Letters para garantizar la salida solicitada?

- a) public void run() { write(); }
- b) public synchronized void run() { write(); }
- c) public static synchronized void run() { write(); }
- d) public void run() { synchronized(this) { write(); } }
- e) public void run() { synchronized(Letters.class) { write(); } }
- f) public void run () { synchronized (System.out) { write (); } }
- g) public void run() { synchronized(System.out.class) { write(); } }

## Resumen

- 📖 Los threads pueden ser creados heredando de la clase Thread y sobrescribiendo su método run().
- 📖 Los objetos de tipo Thread también pueden ser creados invocando al constructor de la clase Thread que recibe como argumento un objeto Runnable. Se dice que el objeto runnable será el destino (target) del thread.
- 📖 Una vez que un nuevo thread es iniciado (con el método start()), éste pasa siempre al estado runnable.
- 📖 El thread scheduler puede mover un thread entre los estados runnable y running.
- 📖 El sleeping es usado para retardar la ejecución de un thread por un periodo de tiempo. Ningún bloqueo es liberado cuando un thread cambia al estado sleep.
- 📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.
  - 🔗 <http://java.sun.com/docs/books/tutorial/essential/concurrency/>  
Aquí hallará información oficial sobre la gestión de concurrencia y threads en Java.
  - 🔗 <http://www.arrakis.es/~abelp/ApuntesJava/Threads.htm>  
En esta página, hallará un interesante artículo en castellano sobre el uso de Threads en Java.



**UNIDAD DE  
APRENDIZAJE**

**4**

**SEMANA**

**13**

## **CLASES INTERNAS, GESTIÓN DE HILOS EN JAVA Y DESARROLLO DE APLICACIONES JAVA**

### **LOGRO DE LA UNIDAD DE APRENDIZAJE**

- Al término de la unidad, los alumnos crearán aplicaciones que utilicen clases internas de los tipos Inner Classes, Method-Local Inner Classes, Anonymous Inner Classes y Static Nested Classes así como Hilos de ejecución utilizando la clase Thread y la interface Runnable, basados en un caso propuesto y utilizando Eclipse como herramienta de desarrollo.

### **TEMARIO**

- Clases Internas - Definición
- Codificación de una clase interna estándar
- Referencias desde una clase interna
- Clases internas locales a un método – Definición y características
- Clases internas anónimas – Definición y Tipos
- Clases internas anónimas como argumentos de un método
- Clases estáticas anidadas – Definición y gestión
- Hilos – Definición
- Estados y ciclo de vida de un hilo
- Priorización de hilos
- Sincronización y bloqueo de código

### **ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucran el uso de bloqueo y sincronización de métodos utilizando la palabra reservada synchronized.

## 1. Sincronización y bloqueo de hilos

¿Cómo trabaja la sincronización? Trabaja con bloqueos. Cada objeto en java maneja un bloqueo implícito que sólo actúa cuando el objeto tiene código en un método sincronizado. Cuando ingresamos a un método sincronizado no estático, automáticamente adquirimos el bloqueo asociado con la instancia actual de la clase cuyo código estamos ejecutando.

El hecho de adquirir un bloqueo para un objeto es conocido también como obtener el bloqueo, bloquear el objeto, o sincronizando en el objeto. Podemos usar también el término “monitor” para referirnos al objeto cuyo bloqueo estamos adquiriendo. Técnicamente, el bloqueo y el monitor son dos cosas diferentes, pero la mayoría de las personas, utilizan ambos términos de manera intercambiable.

Dado que existe solo un bloqueo por objeto, si un thread ha tomado el bloqueo, ningún otro thread podrá tomarlo hasta que el primer thread libere “o retorne” el bloqueo. Esto, en la práctica, significa que ningún otro thread podrá ingresar al código sincronizado; es decir, no podrá ingresar a ningún método sincronizado de dicho objeto hasta que el bloqueo haya sido liberado. Típicamente, liberar el bloqueo significa que el thread que mantiene el bloqueo (en otras palabras, el thread que se encuentra actualmente en el método sincronizado), salga del método sincronizado. En este punto, el bloqueo es libre hasta que algún otro thread acceda a un método sincronizado en dicho objeto. A continuación, se enumeran algunos puntos claves que debemos recordar siempre en relación a bloqueos y sincronización:

- Solo métodos o bloques de código pueden ser sincronizados, **no variables o clases**.
- Cada objeto tiene solo un bloqueo.
- No todos los métodos en una clase necesitan ser sincronizados. Una clase puede tener tanto métodos sincronizados como no sincronizados.

Si dos threads desean ejecutar un método sincronizado en una clase, y ambos threads están usando la misma instancia de la clase para invocar al método, sólo un thread a la vez podrá ejecutar el método. El otro thread necesitará esperar hasta que el primero termine su invocación al método. En otras palabras, una vez que un thread adquiere el bloqueo sobre un objeto, ningún otro thread puede [invocar](#) a cualquiera de los métodos sincronizados de dicha clase para ese objeto.

Si una clase tiene métodos sincronizados y no sincronizados, múltiples threads podrán, todavía, acceder a los métodos no sincronizados de la clase. Si tenemos métodos que no acceden a la data que estamos tratando de proteger, entonces no necesitamos sincronizar dichos métodos. El uso de sincronización puede afectar la performance de nuestra aplicación, incluso, generar situaciones de tipo deadlock si es usada incorrectamente, por lo que debemos de ser cuidadosos de no excedernos en su uso.

Si un thread cambia al estado `sleep`, mantendrá cualquier bloqueo que tenga, **no los libera**.

Un thread puede adquirir más de un bloqueo. Por ejemplo, un thread puede ingresar a un método sincronizado, por lo tanto, adquirir el bloqueo, luego, invocar inmediatamente a un método sincronizado en un objeto diferente, por lo tanto adquiriría dicho bloqueo también.

Si un thread adquiere un bloqueo y luego intenta invocar a un método sincronizado del mismo objeto, no existirá ningún problema. La JVM sabe que este thread ya tiene el bloqueo para este objeto, por lo tanto, el thread es libre de invocar a otro método sincronizado dentro del mismo objeto, usando el bloqueo que el thread ya tiene.

Podemos sincronizar un bloque de código en vez de un método completo:

Dado que la sincronización no afecta la concurrencia, es probable no deseemos sincronizar más código del que realmente debemos de proteger. Por lo tanto, si el ámbito de un método es mayor al que necesitamos, podemos reducir el ámbito de la parte sincronizada a **solo un bloque** de dicho método:

```
class SyncTest {
    public void doStuff() {
        System.out.println("not synchronized");
        synchronized(this) {
            System.out.println("synchronized");
        }
    }
}
```

Cuando un thread está ejecutando código dentro de un bloque sincronizado, se dice que el código se está ejecutando en un contexto sincronizado, pero, ¿qué significa sincronizado, sincronizado sobre qué o sobre qué bloqueo?

Cuando sincronizamos un método, el objeto usado para invocar al método, es el objeto cuyo bloqueo debe ser adquirido. Pero cuando sincronizamos un bloque de código, podemos especificar que el bloqueo de qué objeto queremos usar como bloqueo; por lo que podremos usar, por ejemplo, un objeto independiente (third-party object) como bloqueo para este bloque de código. Esto nos proporciona la capacidad de tener más de un bloqueo para sincronizar código dentro de un único objeto.

Podemos sincronizar sobre la instancia actual (this) como lo hemos hecho en el código anterior.

```
public synchronized void doStuff () {
    System.out.println("synchronized");
}
```

Es equivalente a:

```
public void doStuff() {
    synchronized(this) {
        System.out.println("synchronized");
    }
}
```

Ambos métodos, en términos prácticos, tienen el mismo efecto. Los bytecodes compilados pueden no ser exactamente los mismos para ambos métodos, pero su comportamiento sí. La primera forma es más corta y más familiar para la mayoría de personas; sin embargo, la segunda forma es más flexible.

## 1.1 Sincronización de métodos estáticos

Los métodos estáticos pueden ser sincronizados. Existe solo una copia de la data estática que estamos tratando de proteger; por lo tanto, sólo necesitamos un bloque por clase para sincronizar métodos estáticos, un bloqueo para toda la clase. Cada clase cargada en Java tiene una instancia equivalente de `java.lang.Class` representando a la clase. La instancia de la clase `java.lang.Class` utiliza su propio bloqueo para proteger los métodos estáticos de la clase (si es que ellos están sincronizados). En realidad, no existe nada especial que debamos hacer para sincronizar un método estático:

```
public static synchronized int getCount() {
    return count;
}
```

Nuevamente, esto podría ser reemplazado con el código que usa un bloque sincronizado. Si el método está definido en una clase llamada `MyClass`, el código equivalente sería el siguiente:

```
public static int getCount() {
    synchronized(MyClass.class) {
        return count;
    }
}
```

¿Cómo se entiende el uso de `MyClass.class`? Esto se conoce como un **class literal**. Una característica especial en el lenguaje Java es que le indica al compilador (el cual le dirá a la JVM): anda y busca la instancia de la clase que representa la clase llamada `MyClass`. También podemos hacer esto con el siguiente código:

```
public static void classMethod() {
    Class c1 = Class.forName("MyClass");
    synchronized (c1) {
        // do stuff
    }
}
```

## Sincronización de un bloque de código

Mostramos, a continuación, un ejercicio en el que sincronizamos un bloque de código. Dentro del bloque de código obtendremos el bloqueo sobre un objeto, de modo que otros threads no puedan modificarlo mientras el bloque de código se esté ejecutando. Crearemos tres threads que intentarán manipular el mismo objeto. Cada thread generará una salida 100 veces, e incrementará dicha salida en un carácter cada vez. El objeto utilizado será un `StringBuffer`.

A continuación, el detalle del ejercicio:

Creemos una clase que hereda de la clase `Thread`.

Sobreescribimos el método `run()` de la clase `Thread`. Es aquí donde ubicaremos el bloque de código sincronizado.

Para los tres objetos de la clase `Thread` que compartirán el mismo objeto, necesitaremos crear un constructor que acepte un objeto de tipo `StringBuffer` como argumento.

El bloque de código sincronizado obtendrá un bloqueo sobre el objeto `StringBuffer`.

Dentro del bloque, se visualiza el `StringBuffer` 100 veces y se incrementa un carácter en el `StringBuffer`.

Finalmente, en el método `main()`, se crea un objeto `StringBuffer` usando la letra A; luego, creamos tres instancias de nuestra clase e iniciamos los threads.

La solución sería la siguiente:

```
class InSync extends Thread {
    StringBuffer letter;
    public InSync(StringBuffer letter) { this.letter = letter; }
    public void run() {
        synchronized(letter) { // #1
            for(int i = 1; i <= 100; ++i) System.out.print(letter);
            System.out.println();
            char temp = letter.charAt(0);
            ++temp; // Increment the letter in StringBuffer:
            letter.setCharAt(0, temp);
        } // #2
    }
}

public static void main(String [] args) {
    StringBuffer sb = new StringBuffer("A");
    new InSync(sb).start(); new InSync(sb).start();
    new InSync(sb).start();
}
}
```

Sólo para “divertirnos”, eliminemos las líneas identificadas como #1 y #2 y ejecutemos nuevamente el programa. No estará sincronizado, analicemos la salida obtenida.

## 1.2 Cuándo necesitamos sincronizar

Cuando usamos threads, usualmente necesitaremos usar algún tipo de sincronización para asegurar que nuestros métodos no interrumpan en el momento equivocado pudiendo ocasionar pérdidas de datos. Generalmente, en cualquier momento, más de un thread está accediendo data mutable (modificable). Nosotros sincronizamos para proteger esta data, para asegurarnos que, por ejemplo, dos threads no están cambiando la misma información al mismo tiempo. Con respecto a las variables locales, no es necesario que nos preocupemos por ellas, dado que cada thread obtiene su propia copia de variables locales. Dos threads ejecutando el mismo método y simultáneamente, usan diferentes copias de variables locales. Por otro lado, sí debemos preocuparnos por los campos estáticos y no estáticos si es que contienen data que puede ser modificada.

Para data modificable en un campo no estático, usualmente, empleamos un método no estático para acceder a él. Sincronizando dicho método nos aseguramos de que cualquier thread que trate de ejecutar dicho método usando la misma instancia no pueda tener un acceso simultáneo junto con otro thread. Pero un thread trabajando con una instancia diferente, no será afectado, dado que adquiere el bloqueo sobre otra instancia.

Para data modificable en un campo estático el comportamiento es similar.

Sin embargo, ¿qué sucede si un método no estático accede a un campo estático o si un método estático accede a un campo no estático?

En estos casos, debemos prepararnos para comportamientos impredecibles o errados. Si, por ejemplo, un método estático accede a un campo no estático y sincronizamos dicho método, adquiriremos el lock sobre el objeto de tipo Class. Pero qué sucede si otro método también accede al campo no estático y en esta oportunidad el método es no estático, probablemente se sincronize sobre la instancia actual del thread. Recordemos que un método estático sincronizado y un método no estático sincronizado no se bloquean el uno al otro, ambos pueden ejecutarse simultáneamente.

Para mantener las cosas más simples, de modo que podamos hacer una clase “**thread-safe**”, los métodos que accedan a campos modificables, deben de estar sincronizados.

El acceso a campos estáticos, debe de hacerse desde métodos estáticos sincronizados. El acceso a campos no estáticos debe de hacerse desde métodos no estáticos sincronizados. A continuación, se muestra un ejemplo de aplicación:

```
public class Thing {  
    private static int staticField;  
    private int nonstaticField;  
    public static synchronized int getStaticField() {  
        return staticField;  
    }
```

```

    }
    public static synchronized void setStaticField(
        int staticField) {

        Thing.staticField = staticField;
    }
    public synchronized int getNonstaticField() {
        return nonstaticField;
    }
    public synchronized void setNonstaticField(
        int nonstaticField) {
        this.nonstaticField = nonstaticField;
    }
}

```

### 1.3 Clases Thread Safe

Cuando una clase ha sido cuidadosamente sincronizada para proteger sus datos, nosotros decimos que la clase es "**thread-safe**." Muchas clases en la API Java ya usan sincronización internamente para hacer la clase "thread-safe." Por ejemplo, `StringBuffer` y `StringBuilder` son clases casi idénticas, con la excepción de que todos los métodos en la clase `StringBuffer` estarán sincronizados cuando sea necesario, mientras que los de la clase `StringBuilder` no. Generalmente, esto hace a la clase `StringBuffer` segura para usarla dentro de un entorno multithreaded, mientras que `StringBuilde` no (En contraposición, `StringBuilder` es un poco más rápida que `StringBuffer` porque no debe preocuparse de aspecto alguno de sincronización).

### Ejercicios de aplicación:

1. Dada la siguiente clase:

```
public class ThreadDemo {
    synchronized void a() { actBusy(); }
    static synchronized void b() { actBusy(); }
    static void actBusy() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
    public static void main(String[] args) {
        final ThreadDemo x = new ThreadDemo();

        final ThreadDemo y = new ThreadDemo();
        Runnable runnable = new Runnable(){
            public void run() {
                int option = (int) (Math.random() * 4);
                switch (option) {
                    case 0: x.a(); break;
                    case 1: x.b(); break;
                    case 2: y.a(); break;
                    case 3: y.b(); break;
                }
            }
        };
        Thread thread1 = new Thread(runnable);
        Thread thread2 = new Thread(runnable);
        thread1.start();
        thread2.start();
    }
}
```

¿ Cuáles de las siguientes invocaciones no deben nunca estar ejecutándose a la vez?

- a) x.a() in thread1, and x.a() in thread2
- b) x.a() in thread1, and x.b() in thread2
- c) x.a() in thread1, and y.a() in thread2
- d) x.a() in thread1, and y.b() in thread2
- e) x.b() in thread1, and x.a() in thread2
- f) x.b() in thread1, and x.b() in thread2
- g) x.b() in thread1, and y.a() in thread2
- h) x.b() in thread1, and y.b() in thread2



2. Dada la siguiente clase:

```
public class TwoThreads {
    static Thread laurel, hardy;
    public static void main(String[] args) {
        laurel = new Thread() {
            public void run() {
                System.out.println("A");
                try {
                    hardy.sleep (1000);
                } catch (Exception e) {
                    System.out.println("B");
                }
                System.out.println("C");
            }
        };
        hardy = new Thread() {
            public void run() {
                System.out.println("D");
                try {
                    laurel.wait();
                } catch (Exception e) {
                    System.out.println("E");
                }
                System.out.println("F");
            }
        };
        laurel.start();
        hardy.start();
    }
}
```

¿Qué letras eventualmente aparecerán en algún lugar de la salida?

- a) A
- b) B
- c) C
- d) D
- e) E
- f) F
- g) The answer cannot be reliably determined.
- h) The code does not compile.

## Resumen

📖 Los métodos sincronizados previenen que más de un thread acceda de manera simultánea a código crítico dentro de un método.

📖 Podemos usar la palabra clave **synchronized** como modificador de un método o para iniciar un bloque de código sincronizado.

📖 Para sincronizar un bloque de código (en otras palabras, un ámbito más pequeño que el de todo el método), debemos especificar un argumento que es objeto cuyo bloqueo queremos sincronizar.

📖 Mientras solo un thread puede acceder a código sincronizado de una instancia en particular, múltiples threads pueden, todavía, acceder al código no sincronizado del mismo objeto.

📖 Si desea saber más acerca de estos temas, puede consultar las siguientes páginas.

🖱 <http://java.sun.com/docs/books/tutorial/essential/concurrency/syncmeth.html>

Aquí hallará documentación oficial sobre el uso de métodos sincronizados con Java.

🖱 <http://www.janeg.ca/scjp/threads/synchronization.html>

En esta página, hallará interesante documentación sobre sincronización de threads en Java.

**UNIDAD DE  
APRENDIZAJE**

**5**

**SEMANA**

**14**

## **DESARROLLO DE APLICACIONES JAVA**

### **LOGRO DE LA UNIDAD DE APRENDIZAJE**

- Al término de la unidad, los alumnos crearán y desplegarán aplicaciones Java JSE creando estructuras de directorios que permitan su ejecución, basados en un caso propuesto, utilizando Eclipse como herramienta de desarrollo.

### **TEMARIO**

- Compilación de clases java utilizando el comando javac
- Ejecución de aplicaciones java
- Uso de sentencias import estáticas.

### **ACTIVIDADES PROPUESTAS**

- Los alumnos resuelven ejercicios que involucran el uso de los comandos javac, java y sentencias import estáticas.

## 1. Compilación con el comando `javac`

El comando `javac` es usado para invocar al compilador de java. Existen muchas opciones que podríamos utilizar cuando utilizamos `javac`, sin embargo, las más relevantes son las opciones `-classpath` y `-d`. A continuación, se muestra la estructura base del comando `javac`:

```
javac [options] [source files]
```

A continuación se muestran dos sentencias válidas del uso del comando `javac`:

```
javac -help  
javac -classpath com:. -g Foo.java Bar.java
```

La primera invocación no compila archivo alguno, pero imprime un resumen de las opciones válidas del comando. La segunda invocación le pasa al compilador dos opciones (`-classpath`, la cual en sí misma tiene el argumento **com:.** y **-g**), y pasa al compilador dos archivos `.java` para compilar (`Foo.java` and `Bar.java`). Cuando especificamos múltiples opciones y/o archivos, éstos deben estar separados por espacios.

### 1.1 Compilación con el parámetro `-d`

Por defecto, el compilador coloca un archivo `.class` en el mismo directorio que el archivo fuente. Esto es correcto para proyectos pequeños, sin embargo, para proyectos de mayor envergadura y en realidad, debería ser para cualquier proyecto, es recomendable mantener nuestros archivos `.java` separados de nuestros archivos `.class` (Este enfoque nos ayuda con el control de versiones, las pruebas, deployment, etc). La opción `-d` nos permite indicar al compilador en qué directorio colocar los archivos `.class` generados (**d** es la abreviación de “destination”). Dada la siguiente estructura de directorio:

```
myProject
|
|--source
|   |
|   |-- MyClass.java
|
|--classes
|   |
|   |--
```

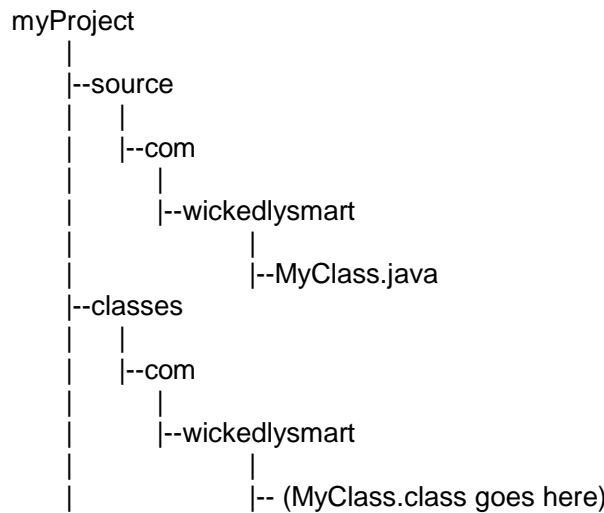
El siguiente comando, iniciado desde el directorio `myProject`, compilará `MyClass.java` y colocará la clase resultante `MyClass.class` en el directorio `classes`. (Este ejemplo assume que `MyClass` no tiene una sentencia **package**)

```
cd myProject
javac -d classes source/MyClass.java
```

Este commando también demuestra cómo podemos seleccionar un archivo .java de un subdirectorio del directorio desde el cual el comando fue ejecutado.

Supongamos ahora que tenemos un archivo .java en la siguiente estructura de directorio:

```
package com.wickedlysmart;
public class MyClass { }
```



Si nos encontramos en el directorio fuente, compilaremos MyClass.java y colocaremos el archivo resultante MyClass.class dentro del directorio **classes/com/wickedlysmart directory** al invocar al siguiente comando:

```
javac -d ../classes com/wickedlysmart/MyClass.java
```

Podemos leer el commando de la siguiente manera: "Para definir el directorio de destino, retrocedemos al directorio myProject; luego, ingresamos al directorio classes, el cual será nuestro destino. Compilamos el archivo llamado MyClass.java. Finalmente, colocamos la clase resultante MyClass.class dentro de la estructura de directorio que corresponde a su paquete, en este caso classes/com/wickedlysmart."

Dado que la clase Myclass.java está en un paquete, el compilador sabe que debe colocar el archivo resultante .class dentro del directorio classes/com/wickedlysmart.

Adicionalmente, el commando javac puede, algunas veces, ayudarnos con la construcción de la estructura de directorios que nuestra clase requiera. Supongamos que tenemos:

```
package com.wickedlysmart;
public class MyClass { }
```

```
myProject
|
|--source
|   |
|   |--com
|       |
|       |--wickedlysmart
|           |
|           |--MyClass.java
|
|--classes
|   |
```

Y el siguiente comando:

```
javac -d ../classes com/wickedlysmart/MyClass.java
```

En este caso, el compilador construye dos directorios llamados **com** y **com/wickedlysmart** para colocar el archive resultante `MyClass.class` en la estructura de directorio (`com/wickedlysmart/`) la cual es construída dentro del directorio existente **../classes**.

Si el directorio destino no existe, obtendremos un error de compilación. Si en el ejemplo previo el directorio `classes` no existe, el compilador visualizará lo siguiente:

```
java:5: error while writing MyClass: classes/MyClass.class (No
such file or directory)
```

## 2. Ejecución de aplicaciones en Java

El comando `java` es usado para invocar a la Java virtual machine. Dentro de este comando, es importante entender las opciones `-classpath` (and su gemela `-cp`) y la opción `-D`. A continuación se muestra la estructura del comando:

```
java [options] class [args]
```

Las partes `[options]` y `[args]` del comando `java` son opcionales y ambas partes pueden tener multiples valores. Debemos especificar de manera precisa un archivo `.class` a ejecutar. El comando `java` asume que estamos hablando de un archivo `.class`, por lo tanto, no es necesario **que** especiquemos la extension `.class`.

```
java -DmyProp=myValue MyClass x 1
```

Este comando puede ser leído como: "Crea una propiedad del sistema llamada **myProp** y carga su valor con **myValue**. Luego, lanza el archivo llamado `yClass.class` y envíale dos argumentos de tipo `String` cuyos valores son `x` y `1`."

## 2.1 Uso de propiedades del Sistema

A partir de la version 5 de Java contamos con una clase llamada `java.util.Properties` que puede ser usada para acceder a información persistente del sistema tal como la versión actual del sistema operativo, el compilador Java y la JVM. Adicionalmente, podemos agregar y recuperar también nuestras propias propiedades:

```
import java.util.*;
public class TestProps {
    public static void main(String[] args) {
        Properties p = System.getProperties();
        p.setProperty("myProp", "myValue");
        p.list(System.out);
    }
}
```

Si este archive es compilado y lo invocamos tal como se muestra a continuación:

```
java -DcmdProp=cmdVal TestProps
```

Obtendremos:

```
...
os.name=Mac OS X
myProp=myValue
...
java.specification.vendor=Sun Microsystems Inc.
user.language=en
java.version=1.5.0_02
...
cmdProp=cmdVal
...
```

Donde ... representa muchas otras parejas de nombres y valores. Dos nombres y valores fueron agregados a las propiedades del sistema:

`myProp=myValue` fue agregado via el método `setProperty` y `cmdProp=cmdVal` fue agregado via la opción `-D` en la línea de comando.

Cuando usamos la opción `-D`, si el valor contiene espacios en blanco, el valor entero debe ser definido entre comillas tal como se observa:

```
java -DcmdProp="cmdVal take 2" TestProps
```

El método `getProperty()` es usado para recuperar una propiedad simple. Puede ser invocado con un único argumento (un `String` que representa el nombre (o key)) o puede ser invocado con dos argumentos, (un `String` que

representa el nombre (o key) y un String por defecto, que será usado como propiedad, si la propiedad solicitada ya no existe). En ambos casos `getProperty()` retorna la propiedad como un String.

### 3. Realización de imports estáticos

A partir de la version 5 de Java es posible realizar imports estáticos. Los imports estáticos pueden ser usados cuando queremos usar miembros estáticos de una clase. A continuación, se muestra un ejemplo:

Antes de los imports estáticos:

```
public class TestStatic {
    public static void main(String[] args) {
        System.out.println(Integer.MAX_VALUE);
        System.out.println(Integer.toHexString(42));
    }
}
```

Después de los imports estáticos:

```
import static java.lang.System.out;      // 1
import static java.lang.Integer.*;      // 2
public class TestStaticImport {
    public static void main(String[] args) {
        out.println(MAX_VALUE);          // 3
        out.println(toHexString(42));    // 4
    }
}
```

Ambas clases producen la misma salida:

```
2147483647
2a
```

Veamos a continuación que sucede con la clase que utilice la característica de imports estáticos:

La sintaxis debe iniciar con las palabras reservadas **import static** seguidas del nombre completo del miembro estático que queremos importar o de caracteres comodín. En el ejemplo, hacemos un import estático del objeto `out` de la clase `System`.

Ahora, podemos observar los beneficios del uso de la característica de import estáticos. No tenemos que digitar `System` en `System.out.println`. Tampoco tenemos que escribir `Integer` en `Integer.MAX_VALUE`.



A continuación, se muestran un par de reglas para el uso de imports estáticos:

Debemos decir siempre **import static**; **no** es posible decir **static import**.

Debemos tener cuidado con los nombres ambiguos de miembros estáticos. Por ejemplo, si hacemos un import estático tanto para la clase Integer como para la clase Long, cuando referenciamos a la constante MAX\_VALUE, generaremos un error de compilación, dado que tanto Integer como Long poseen una constante llamada MAX\_VALUE y no sabremos a cual realmente nos referimos.

### Ejercicios de aplicación:

1. Dadas las siguientes clases en archivos diferentes:

```
package xcom;
public class Useful {
    int increment(int x) { return ++x; }
}

import xcom.*;                // line 1
class Needy3 {
    public static void main(String[] args) {
        xcom.Useful u = new xcom.Useful();    // line 2
        System.out.println(u.increment(5));
    }
}
```

¿Qué sentencias son verdaderas?

- a) The output is 0.
- b) The output is 5.
- c) The output is 6.
- d) Compilation fails.
- e) The code compiles if line 1 is removed.
- f) The code compiles if line 2 is changed to read:

Useful u = new Useful( );

2. Dada la siguiente estructura de directorios y archivo fuente:

```
org
|-- Robot.class
|
|-- ex
|   |-- Pet.class
|   |
|   |-- why
|       |-- Dog.class
```

```
class MyClass {
    Robot r;
    Pet p;
    Dog d;
}
```

¿Qué sentencia(s) deben ser agregadas al archivo fuente para que éste compile?

- a) package org;
- b) import org.\*;
- c) package org.\*;
- d) package org.ex;
- e) import org.ex.\*;
- f) package org.ex.why;
- g) package org.ex.why.Dog;

## Resumen

- 📖 Usemos **-d** para cambiar el destino de un archive .class cuando es generado por primera vez por el commando **javac**.
- 📖 Tanto java como javac usan el mismo algoritmo para buscar clases.
- 📖 Las búsquedas inician en la ubicación que contiene las clases que vienen estándar con JSE.
- 📖 Cuando una clase es parte de un paquete, debemos utilizar siempre su “nombre completo” el cual implica nombre de paquete más nombre de clase.
- 📖 Debemos iniciar una sentencia de import estática tal como se muestra a continuación: **import static**
- 📖 Podemos usar imports estáticos para crear atajos hacia miembros estáticos (variables estáticas, constantes y métodos) de cualquier clase.

🔗 <http://www.geocities.com/CollegePark/Quad/8901/intro.htm> Aquí hallará una interesante introducción a la compilación de aplicaciones Java.

🔗 <http://java.sun.com/j2se/1.4.2/docs/tooldocs/solaris/javac.html> En esta página, hallará importante información oficial sobre el comando javac de Java.