Seminario JAVA

Índice

1.- Programación orientada a objetos.

1.1.- Paradigma de la programación.

Cuando utilizamos una computadora, lo que buscamos es una solución a un problema. Pero ellas entienden lenguaje **binario**, y solo son capaces de sumar **bits** y mover **bytes** de un lugar a otro. Como programadores, debemos indicarle a la computadora que es lo que deseamos que ella realice, y para ello nos vemos en la obligación de utilizar un lenguaje de programación. Éste es un lenguaje intermedio, que luego será traducido a binario, siendo una herramienta que nos permite dar órdenes a la computadora sobre como resolver un problema en particular.

La forma en que especifiquemos la solución depende del paradigma de programación que utilicemos. Los paradigmas son un modelo que representa un enfoque específico para el desarrollo de sistemas. No existe paradigma mejor que otro, hay situaciones en que uno resulta más adecuado, todos tienen ventajas y desventajas. El paradigma en que nos enfocaremos en esta ocasión es el de "**Programación Orientada a Objetos**" (POO).

1.1.1.- Propiedades y comportamiento de los objetos.

¿Qué es un objeto? La respuesta es simple, basta con mirar a nuestro alrededor para ver que estamos rodeados de objetos. Si observamos más detalladamente veremos que estos objetos tienen **propiedades** o **atributos**. Por ejemplo, el objeto auto tiene motor, ruedas y asientos. El objeto avión posee estas mismas propiedades, pero se diferencian en que el avión puede volar y el auto no. Vale decir, los objetos también tienen un comportamiento propio.

Entonces, un objeto es una **entidad** compleja que posee **propiedades** (datos, atributos) y **comportamiento** (funcionalidad, métodos). Además, cada objeto expone una **interfaz** a otros objetos que indica cómo éstos pueden interactuar (comunicarse) con él. Esta interfaz provee un conjunto de métodos. La interfaz del automóvil estará dada por los **métodos** "arranca", "frena", "dobla", etc., y es a través de ellos que podemos interactuar con el objeto (**Figura 1**).

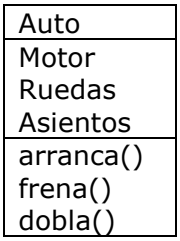


Figura 1. Representación del objeto Auto con sus atributos y métodos.

Cabe aclarar que el comportamiento es exclusivo del objeto. Si bien algunos objetos a simple vista son iguales, internamente pueden ser muy distintos. Por ejemplo, un automóvil cuyo motor utiliza bencina y otro diesel. Si aprendemos a manejar uno, sin problema podemos manejar el otro. Ambos objetos se nos presentan de la misma forma, pero sus motores son muy distintos. Es decir, los objetos presentan la misma interfaz pero ocultan información de su funcionamiento, esto se conoce como **encapsulamiento**, y gracias a él se puede cambiar un objeto por otro que presente la misma interfaz, y todo debería funcionar igual.

Cuando definimos un objeto lo hacemos en función de otros objetos conocidos. Si alguien nos dice "un auto es como una moto, pero tiene cuatro ruedas", nos define al automóvil a través de un objeto de similares características. Sin darnos cuenta hacemos **clasificaciones**. Generalizando, nos damos cuenta de que ambos son medios de transporte. En el paradigma de objetos esto es conocido como **herencia**, y

es útil para no tener que definir **comportamientos** de forma repetitiva.

Por ejemplo, tendremos un objeto transporte, que tendrá propiedades como "cantidad de pasajeros", "numero de puertas", etc., y métodos como "anda", "frena" y "dobla". De esta manera definiríamos un automóvil como un transporte, agregando las particularidades del automóvil que no estén definidas en transporte.

Ahora que sabemos agrupar **clases**, podríamos agregar más objetos a nuestro modelo, como por ejemplo el objeto avión. Es decir, las clases simplemente son conjuntos de objetos que comparten propiedades y comportamientos. El objeto avión definido como un transporte heredará las propiedades y los métodos de éste. Notamos que no es lo mismo hacer andar un avión que un automóvil, de tal forma que necesitamos agregar el método anda para que el avión vuele y el automóvil ruede. Esto se denomina **polimorfismo**, y nos permite tener comportamientos distintos de un método para objetos diferentes.

Revisando el ejemplo del transporte, automóvil y avión notamos que existe una jerarquía, a la cabeza está transporte y de éste cuelgan el avión y el automóvil. En este paradigma esto se denomina **jerarquía de herencia** (**figura 2**).

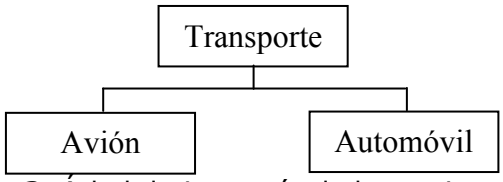


Figura 2. Árbol de jerarquía de herencias. En este caso, los objetos Avión y Automóvil heredan del objeto Transporte.

El automóvil, además de ser un objeto y un transporte, esta **compuesto** de por otros objetos, como un volante, radio, pedales, etc. Aquí nos encontramos frente a una jerarquía de elementos. Esta se conoce como **jerarquía de composición** y sirve para representar que uno o más objetos están dentro de otro (**figura 3**).

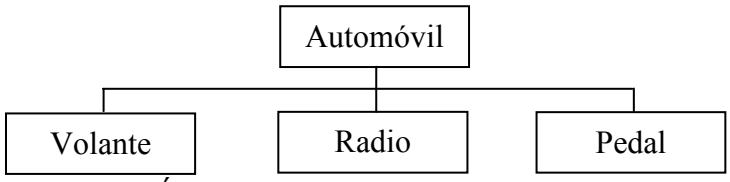


Figura 3. Árbol de jerarquía de composición. En este caso, el Automóvil esta compuesto por los objetos Volante, Radio y Pedal.

Para finalizar, si tenemos un solo objeto automóvil, esto no significa que nuestro programa podrá tener un solo automóvil. Para entender esto debemos conocer el concepto de **instancia**, que nos permite crear la cantidad de automóviles que deseemos. Para lograr esto, cada objeto deberá tener un método que nos permita crear una instancia de éste.

Una clase no está limitada a comportarse sólo como lo define su padre, sino que además puede definir un comportamiento propio, es decir, se pueden agregar nuevos métodos en ella.

1.2.- Propuesta de un modelo de diseño.

Suponiendo el juego Gato, en donde dos participantes tienen fichas y por turno las ubican en un tablero de 3x3. Para resolver el problema lo primero que vamos a hacer es determinar los objetos.

A simple vista tenemos "Jugador", "Tablero" y "Ficha". Cada una de las fichas

será una instancia de la clase ficha. El tablero tiene casilleros, así que agregamos "Casilla" a la lista de objetos. Por otra parte, el jugador puede ser "Humano" o la "Computadora", que también son agregados a la lista de objetos.

Una vez que estamos seguros de haber determinado todos los objetos debemos agregarle atributos y métodos a cada uno. Comenzando por Tablero, el atributo más importante que posee son las casillas. Entre sus métodos tenemos "ponerFicha()", y para verificar si hay ganador esta "buscarLinea()". Para el objeto Casilla tenemos dos atributos, su "coordenada" y el otro es el objeto Ficha. El objeto Jugador, que tiene varias instancias del objeto ficha. Y finalmente Humano tiene el atributo nombre y Computadora tiene el atributo inteligencia, lo que nos permite un juego con varios niveles.

Una vez determinados los atributos y métodos de los objetos, debemos determinar la jerarquía de herencia, en donde los objetos Computadora y Humano heredan del objeto Jugador. Para la jerarquía de composición tenemos que el Tablero esta compuesto de Casillas y el Jugador de Fichas.

Esto que se hizo es un modelo simplificado de representación. Una vez terminado ya estamos listos para escribir el código correspondiente.

Cuando modelamos tendemos a creer que nuestra representación es la más clara que existe, pero puede ser difícil de comprender para otros. Para evitar esto existen herramientas que ayudan al modelado de la programación orientada a objetos a través de estándares. Una de las más utilizadas es **UML** (**Unified Modeling Language**).

Esta herramienta provee de varias representaciones. Algunas son muy comunes, tanto que ya hemos utilizado varias de ellas. UML utiliza como herramientas los diagramas gráficos para representar el sistema. A continuación haremos una explicación del más útil para comprender la orientación a objetos.

1.2.1.- Diagrama de clases.

Los diagramas de clases representan un conjunto de elementos del modelo que son estáticos, como las clases y los tipos, sus contenidos y las relaciones que se establecen entre ellos (**Figura 4**).

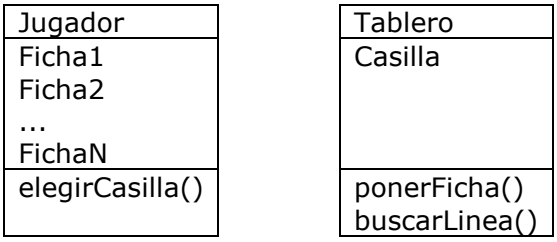


Figura 4. Diagrama de clases del Gato.

2.- Introducción a java.

2.1.- Características de java.

Las principales características de java respecto a otros lenguajes de programación son las siguientes:

- **Simple**: Java ofrece toda la funcionalidad de un lenguaje potente, y a la vez elimina las cosas más confusas y menos usadas de otros lenguajes. Además implementa el **garbage collector** (recolector de basura), gracias al cual el programador no debe asegurar el correcto uso de memoria, como en *C++*.
- **Orientado a objetos**: Fue diseñado como un lenguaje orientado a objetos desde el principio, a diferencia de **C++**, que fue una extensión de *C*. En java, cada objeto se agrupa en estructuras encapsuladas con sus datos y métodos que manipulan esos datos.
- **Distribuido**: Java proporciona una colección de clases para uso en aplicaciones de red, que permite abrir **sockets** y establecer y aceptar conexiones con servidores o clientes remotos. Lo que facilita la creación de aplicaciones distribuidas.
- **Interpretado y compilado a la vez**: Java es compilado ya que su código fuente se transforma en una suerte de código de máquina (**bytecodes**) semejantes a instrucciones en **Assembler**. Es interpretado ya que dichos bytecodes se pueden ejecutar directamente en cualquier máquina en que se haya instalado el intérprete.
- **Robusto**: Java fue diseñado para crear sistemas fiables. Para ellos proporciona muchas comprobaciones en compilación y en tiempo de ejecución.
- **Seguro**: Ya que por naturaleza es distribuido, la seguridad es vital. Implementa barreras de seguridad en el lenguaje y en el sistema de ejecución en tiempo real.
- **Multiplataforma**: Como esta diseñado para soportar aplicaciones distribuidas, y considerando lo heterogéneo de las redes, como Internet, necesitaba ser independiente de la plataforma. Den esta manera una aplicación se puede ejecutar desde un PC a un celular.
- **Multitarea**: Soporta sincronización de múltiples hilos de ejecución (**multithreading**) a nivel de lenguaje.
- **Dinámico**: El lenguaje y su sistema de ejecución en tiempo real son dinámicos en la fase de enlazado. Las clases sólo se enlazan a medida que son necesitadas. Además se pueden enlazar nuevos módulos de código bajo demanda, procedentes incluso desde la Red.

2.2.- Compilación y ejecución de programas.

Antes de comenzar debemos conseguir una distribución de Java de **Sun Microsystems** en su versión para desarrolladores. Podemos descargarla de www.java.sun.com. De acuerdo a nuestras necesidades debemos elegir una distribución particular. Las más populares son:

- **J2SE** (**Java 2 Platform Standard Edition**): La edición estándar provee un entorno para desarrollo de aplicaciones de escritorio y applets. Esta distribución se divide en dos categorías: **Core Java** y **Desktop Java**.
- **J2EE** (**Java 2 Platform Enterprise Edition**): Es la más difundida en el mercado empresarial, ya que define los estándares para desarrollar soportando todas las posibilidades de Java. Además posee herramientas para el desarrollo de servicios Web.
- **J2ME** (**Java 2 Platform Micro Edition**): Sirve para el desarrollo donde el soporte en el cual va a residir la aplicación sea reducido en cuanto a sus capacidades de almacenamiento. Es ideal para desarrollar aplicaciones para teléfonos celulares y PDAs.

Una vez elegida la distribución debemos tener en cuenta que existen dos versiones.

- **SDK** (**Standard Development Kit**): Provee de herramientas de desarrollo estándar.
- **JRE** (**Java Runtime Environment**): Provee el entorno de ejecución de Java.

Lo ideal es tener ambas versiones, **SDK** para programar, compilar y probar, y **JRE** para instalar en el PC donde se ejecutará la aplicación.

En nuestro caso utilizaremos la distribución **J2SE**. Además podemos elegir un **IDE** (**Integrated Development Environment**) que nos facilite el trabajo de compilación y ejecutar desde la línea de comando. Utilizaremos Eclipse, y puede ser bajado desde la página www.eclipse.org.

Una vez instalado el lenguaje Java podemos ver que en la carpeta principal, en el subdirectorio llamado **bin**, encontraremos, entre otros, los siguientes ejecutables de Java:

- **java.exe**: es el lanzador de aplicaciones de Java.
- **javaw.exe**: es el lanzador gráfico de aplicaciones.
- **javac.exe**: es el compilador de Java.

2.2.1.- Compilador de Java.

Una de las características del compilador Java es que junta su código en un archivo objeto de formato independiente sin que sea primordial la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (**run time**) puede ejecutar ese código objeto, sin importar que la máquina en que ha sido generado (**figura 1**). Actualmente existen sistemas **run time** para **Solaris**, **SunOs**, **Windows**, **Linux**, **Iris**, **Aix**, **Mac** y **Apple**.

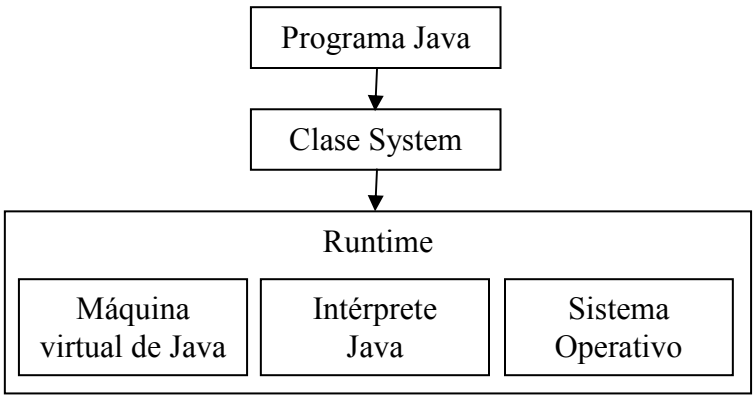


Figura 1. Ambiente de ejecución de Java.

2.3.- Nuestro primer programa Java.

Ahora que sabemos como compilar y ejecutar programas, realizaremos nuestro primer programa:

```
public class HolaMundo
{
   public static void main(String argv[])
   {
      System.out.println("Hola mundo!!");
   }
}
```

La salida del programa será:

```
Hola mundo!!
```

Ahora analicemos la estructura del programa. El nombre del objeto esta determinado por **public class HolaMundo**, mientras que el método principal esta dado por **public static void main(String argv[])**, a través del cual se lanzaran todas las aplicaciones, y cada aplicación ejecutará un solo main por cada ejecución.

Las llaves determinan los bloques de ejecución, y más adelante veremos que también determinarán el alcance de las variables. El punto y coma es el separador de instrucciones, es decir, marcará el fin de una instrucción y el comienzo de la otra. También podemos notar que no todas las líneas comienzas desde el margen izquierdo, esto es conocido como **identación**, y es útil al el programador para leer mas fácilmente el código, ya que determina los bloques de ejecución con mayor claridad.

2.3.1.- Programar comentarios.

Los **comentarios** nos sirven de guía, o para que otro programador vea nuestro código y lo entienda rápidamente. Existen, entre otros, los siguientes tipos:

- **Comentario simple**: se usa para hacer alguna anotación rápida sobe la línea de código siguiente o anterior, El modo de uso es el siguiente:

```
// Esto es un comentario simple
```

Si queremos escribir varias líneas utilizando este tipo de comentario debemos colocar la doble barra al comienzo de cada línea.

```
// Para hacer un comentario simple de varias líneas
// debemos anteponer doble barra a cada una de las líneas
```

- **Comentario de varias líneas**: se usa para escribir aclaraciones de más de una línea. Se utiliza de la siguiente manera:

```
/*
Este es un comentario
de varias líneas
*/
```

De todos modos se puede usar para comentarios de una línea:

```
/* Esto es un comentario de una línea */
```

2.3.2.- Variables y constantes.

Las **variables** son utilizadas para almacenar datos, y como su nombre lo indica, cambian a lo largo de la ejecución del programa. Cuando termina un bloque de ejecución, las variables declaradas en él mueren.

Cada variable posee un **tipo de dato** asociado, que determina el espacio utilizado, los valores que puede almacenar y las operaciones que se pueden realizar con ellos. Las **constantes** tienen el mismo concepto, sólo que no varían durante la ejecución del programa. Los tipos de datos primitivos soportados son:

Tipo de datos	Tamaño	Valores que puede tomar
Boolean	1 byte	true o flase
Char	2 bytes	Cualquier carácter ASCII
Byte	1 byte	Entero entre -128 y 127
Short	2 bytes	Entero entre -32.768 y 32.767
Int	4 bytes	Entero entre -2.147.483.648 y 2.147.483.647
Long	8 bytes	Entero entre -9.223.372.036.854.775.808 y 9.223.372.036.854.775.807
Float	4 bytes	Decimal de -3.402823E38 y -1.401298E-45 y de 1.401298E-45 a 3.402823E38
Duble	8 bytes	Decimal de -1.79769313486232E308 a -4.94065645841247E-324 y de 4.94065645841247E-324 a 1.79769313486232E308

Tabla 1. Tipos de datos primitivos.

Además Java provee un conjunto de datos secundarios, llamados así porque son datos compuestos. Estos datos son:

- **String** (compuesto por una serie de char).
- **Array** (compuesto por una serie de cualquier tipo de dato primitivo).

Antes de utilizar una variable debemos declararla. Veamos un ejemplo:

```
public class Declaracion
{
    public static void main(String argv[])
    {
        // Declaramos i de tipo long
        long i;
        // Declaramos j de tipo entero y la inicializamos en 0
        int j = 0;
    }
}
```

Ahora que ya hemos declarado variables podemos utilizarlas. A continuación veremos un ejemplo, en donde además se informa el resultado por pantalla.

```
public class Prueba
{
    public static void main(String argv[])
    {
        // Declaramos j de tipo entero y la inicializamos en 0
        int j = 0;
        // Sumamos 1 a i
        i = i + 1;
        // Imprimimos en pantalla
        System.out.println("El valor de i es" + i);
    }
}
```

2.3.2.1.- Variables Finales.

No se pueden declarar constantes como **const**, esta es una palabra que fue reservada para uso futuro. Si queremos declarar una constante debemos declarar una variable final de la siguiente forma:

```
final float a = 4.0f;
```

De esta forma la variable no puede ser modificada durante la ejecución del programa. De tener una línea de código que intenta modificar dicha variable se producirá un error en tiempo de compilación.

2.3.2.2.- Nombres de variables.

Las variables pueden tomar cualquier nombre, sin embargo hay una serie de palabras reservadas que no se pueden utilizar como identificadores. El conjunto de palabras reservadas puede clasificarse según el tipo de instrucción.

- **Tipos de datos y retorno**: boolean, byte, short, long, void, char, double, float, int, return, strictfp.
- **Paquetes y clases**: class, super, final, extends, interface, native, static, volatile, import, new, instanceof, abstracts, implements, synchronized, this, transient.
- **Modificadores de acceso**: public, private, protected, package.
- **Ciclos y condiciones**: if, switch, break, for, while, else, case, default, do, continue.

- **Manejo de excepciones**: throws, try, finally, throw, catch.
- **Reservadas para uso futuro**: goto, assert, const.

Por convención, los nombres de las variables empieza con minúscula, y en caso de que éste se encuentre compuesto de varias palabras, las restantes comenzarán con mayúscula. Por ejemplo, un nombre de variable puede ser fechaDeNacimiento. El guión bajo es aceptable en caso de constantes.

2.3.2.3.- Alcance de una variable.

Para entender el alcance de una variable, también llamado **scope**, primero debemos definir lo que son las **variables globales** y las **variables locales**. La primera puede ser usada en todo el programa, mientras que la segunda solo puede utilizarse dentro del bloque en que fue definida. La diferencia entre ambos tipos es precisamente el alcance. Veamos un ejemplo del alcance de las variables:

```
public class AlcanceVariable
{
   static int j = 0;
   public static void main(String argv[])
   {
      // Definimos un bloque de ejecución abriendo llaves
      {
         int a = 0;
         a = a + 1;
      }
      // Se cierra el primer bloque y se crea un segundo bloque
      {
         int c = 2;
         c = c + j;
         c = c + a; // Instrucción inválida, a no existe en este bloque
      }
      // Se cierra el segundo bloque de ejecución
   }
}
```

En este ejemplo la variable j es global a la clase, esto nos indica que en todo el programa se puede acceder a ella. La variable a es local al primer bloque de ejecución, cuando este bloque es ejecutado por completo la variable a deja de existir. En el segundo bloque, se puede sumar c con j, ya que c es local al bloque y j es global a la clase. La tercera instrucción de dicho bloque es invalida, ya que la variable a murió cuando terminó el primer bloque de ejecución.

2.4.- Operadores Java.

Los operadores pueden dividirse en siete grupos, a continuación podemos ver una tabla con los operadores de cada grupo:

- **Operadores básicos**:

Operador	Opera Sobre	Descripción
.	Objetos, datos miembro	Accede a miembros del objeto
(<Tipo>)	Tipos de datos	Convierte a un tipo de dato
instanceof	Objetos	Provee información sobre la instancia

Tabla 2. Tabla de operadores básicos.

- **Operadores aritméticos**:

Operador	Descripción
++/--	Incrementa o decrementa en uno la variable a la que se le ponga como sufijo (ej.: i++) o prefijo (ej.: ++i)
+/-	Invierte el signo de una variable
*	Multiplicación
/	División
%	Resto de la división
+/-	Suma/Resta

Tabla 3. Tabla de operadores aritméticos.

- **Operadores lógicos**:

Operador	Descripción
i	Invierte el valor de un bolean
&	Operador AND (Y lógico) a nivel de bits
^	XOR a nivel de bits
|	Operador OR (O lógico) a nivel de bits
&&	Operador AND condicional
||	Operador OR condicional

Tabla 4. Tabla de operadores lógicos.

- **Operadores de asignación**:

Operador	Descripción	Ejemplo	Equivalencia
=	Asignación simple de derecha a izquierda	a=b+c	
+=	Evalúa a la derecha y suma a la izquierda	a+=b	a=a+b
-=	Evalúa a la derecha y resta a la izquierda	a-=b	a=a-b
*=	Evalúa a la derecha y multiplica a la izquierda	a*=b	a=a*b
/=	Evalúa a la derecha y divide a la izquierda	a/=b	a=a/b

Tabla 5. Tabla de operadores de asignación.

- **Operadores de comparación**:

Operador	Descripción
<	Menor a
>	Mayor a
<=	Menor o igual a
>=	Mayor o igual a
==	Igual a
!=	Distinto a

Tabla 6. Tabla de operadores de comparación.

- **Operadores de nivel de bits**:

Operador	Descripción
~	NOT
<<	Traslado a izquierda
>>	Traslado a derecha
>>>	Traslado a derecha y complementa con ceros a la izquierda
&	AND
!	OR
^	XOR
<<=	Traslado a izquierda con asignación

>>=	Traslado a derecha con asignación
>>>=	Traslado a derecha y complementa con ceros a la izquierda con asignación

Tabla 7. Tabla de operadores de nivel de bits.

- **Operador ternario**:

Operador	Descripción
(expresión 1)?(expresión 2):expresión 3;	Evalúa la expresión 1, si es verdadera ejecuta la expresión 2, de lo contrario ejecuta la expresión 3

Tabla 8. Tabla de operador ternario.

El operador **instanceof** permite saber si un objeto pertenece o no a una clase determinada. Es un operador binario y se utiliza de la siguiente manera:

```
NombreDeLaInstancia instanceof NombreDeLaClase
```

Devuelve **true** si el objeto pertenece a la clase y **false** en caso contrario.

2.4.1.- Precedencia de operadores en Java.

La precedencia determina en qué orden se van a resolver las operaciones. La siguiente tabla nos muestra la precedencia de operadores en Java.

Orden	Operación
1	[] ()
2	++ -- ! - instanceof
3	new (type) expresión
4	
5	* / %
6	+ -
7	<< >> >>>
8	< > <= >=
9	&
10	^
11	!
12	&&
13	!!
14	? :
15	= += -= *= /= %= ^=
16	&= != <<= >>= >>>=

Tabla 9. Precedencia de operadores en Java.

2.5.- Expresiones y sentencias.

Una **expresión** es un conjunto de variables unidas por **operadores**. Una expresión que termina en **punto y coma** es una **sentencia**. En una línea pueden ser incluidas varias sentencias, aunque es recomendable utilizar una línea para cada sentencia, ya que facilita la lectura y la comprensión del programa.

2.6.- Sentencias de control de flujo.

Estas **sentencias** o **instrucciones**, como su nombre lo indica, son utilizadas para controlar el flujo del programa, es decir, qué instrucción deberá ejecutar a continuación nuestro programa. A menudo requerimos que no se ejecuten una detrás de la otra, sino que el programa decida un camino en particular. A continuación vamos a ver las sentencias de control de flujo.

2.6.1.- if-else.

Esta sentencia significa "si – si no", si sucede algo hace una cosa, si no, hace otra. Es utilizada para generar bifurcaciones o saltos. Su sintaxis es:

```
if(condición)
{
   /* Instrucciones que se ejecutarán
   si la condición se cumple*/
}
else
{
   /* Instrucciones que se ejecutarán
   Si la condición no se cumple*/
}
```

Cuando se cumple una condición se dice que es verdadera, y se representa con **true**, si no, es falsa, y es representada por **false**.

En algunas ocasiones necesitamos que el programa haga algo si se cumple la condición, pero si no se cumple, queremos que no haga nada, esto quiere decir que no necesitamos el **else**. Aunque también podemos tener varias bifurcaciones condicionales del tipo **if-else if**. Veamos un ejemplo en donde apreciemos las distintas formas de utilizar la sentencia **if-else**:

```
public class pruebaIfElse
{
   public static void main(String argv[])
   {
      int i = 2;
      if(i == 1)
         System.out.println("La variables es uno");
      else
         System.out.println("La variables es distinta de uno");
      i = i + 2;
      if(i == 3)
         System.out.println("La variables es tres");
      else if (i == 4)
         System.out.println("La variables es cuatro");
      else
         System.out.println("La variables no tomó ningún valor esperado");
      i = i – 3;
      if(i > 0)
         System.out.println("La variables es positiva, y es: " + i);
   }
}
```

Si analizamos el programa, vemos que declaramos la variable i, y es inicializada en 2, a continuación verificamos si es igual a 1, ya que esta condición es false, el programa ejecuta el bloque del else, así se obtiene la siguiente salida:

```
La variable es distinta de uno
```

Luego sumamos 2 a i, quedando en 4, y preguntamos si es igual a 3, como esto es false, continuamos con el siguiente bloque, preguntando si i es igual a 4, ya que esta condición es true se genera la siguiente salida:

```
La variable es cuatro
```

Por ultimo, restamos 3 a la variable i, quedando en 1, luego verificamos si i es mayor que 0, lo que es true, obteniendo la salida:

```
La variable es positiva, y es: 1
```

2.6.2.- switch.

La sentencia **switch** se utiliza cuando una variable puede tomar distintos valores, y dependiendo de cual tome el programa seguirá un camino diferente. Su sintaxis es la siguiente:

```
switch(variable)
{
   case valor1: // instrucciones para valor 1
   case valor2: // instrucciones para valor 2
   ...
   case valorN: // instrucciones para valor N
   default: // instrucciones por defecto
}
```

La palabra reservada **case** define los posibles caminos según el valor de la variable. Al final encontramos la instrucción **default**, y este bloque será ejecutado si la variable no toma ninguno de los valores contemplados. Esta instrucción es opcional, ya que si deseamos que el programa no haga nada en caso de que la variable tome un valor distinto a los contemplados, simplemente la omitimos. La sentencia **switch** puede analizar variables del tipo **char**, **byte**, **short** o **int**, o expresiones que devuelvan resultados de los tipos mencionados.

Además, esta sentencia ejecutará el caso que cumpla con la condición, y todos los que siguen hasta el final. Si deseamos que ejecute de forma excluyente el caso que cumpla con el valor de la variable, debemos agregar al final de cada bloque la instrucción de corte **break**. Para el caso **default** no es necesario agregar **break**, ya que es el último caso, y de ejecutarse, no habrán casos posteriores.

Ahora veremos un ejemplo:

```
public class pruebaSwitch
{
   public static void main(String argv[])
   {
      int i = 2;
      switch(i)
      {
         case 1: System.out.println("primer caso");
                 break;
         case 2: System.out.println("segundo caso");
         case 3: System.out.println("tercer caso");
         default: System.out.println("otro caso");
      }
   }
}
```

La salida que produce el programa será:

```
segundo caso
tercer caso
otro caso
```

En el ejemplo i toma el valor 2, de tal modo que el caso está contemplado en la segunda instrucción del **switch**, es decir, en case 2. Se ejecuta el caso que cumple la condición, pero como este no posee **break**, también ejecuta las instrucciones siguientes, hasta el final. Pero si modificamos el programa para que i tome el valor 1, la salida seria:

```
Primer caso
```

Ya que luego de ejecutarse la instrucción correspondiente a case 1, se ejecuta la instrucción de corte, por lo que no se ejecutan los bloques siguientes.

2.7.- Sentencias de bucle.

En un programa, cada sentencia es ejecutada una sola vez. Sin embrago, algunas veces necesitamos ejecutar varias veces una instrucción o un conjunto de instrucciones. El número de veces que deberá ser ejecutada dicha instrucción está dado por una condición.

Cuando usamos ciclos condicionales debemos asegurarnos que el corte de control se cumpla, de lo contrario el programa se quedará atrapado en el ciclo y no terminará nunca.

2.7.1- while y do-while.

En un ciclo **while** ejecuta un bloque mientras sea verdadera una condición. Su sintaxis es la siguiente:

```
while(condición)
{
   /* bloque que se ejecutará mientras
   la condición sea verdadera*/
}
```

Supongamos que deseamos mostrar por pantalla los números impares menores que 10. El código de dicho programa sería:

```
public class pruebaWhile
{
   public static void main(String argv[])
   {
      int i = 1;
      while(i<10)
      {
         System.out.println(i);
         i = i + 2;
      }
      System.out.println("no hay mas números impares");
   }
}
```

La salida del programa seria:

```
1
3
5
7
9
no hay mas números impares
```

El programa ejecutó el bloque cinco veces. Cada una de las ejecuciones es conocida como iteración. En el comienzo, i toma el valor 1, por lo que la condición del

while es true, entonces ejecuta el bloque, imprime 1, y luego suma 2 a la variable, quedando en 3. Cuando termina el bloque vuelve a revisar la condición del **while**, que nuevamente es verdadera, ejecutando el bloque una vez más. Así sucesivamente. Cuando i toma el valor 11, se vuelve a verificar la condición del **while**, pero esta vez es false, por lo que termina el **while**, saliendo del bloque de iteración, y continúa ejecutando el resto del programa.

En algunos casos necesitamos que el bloque de iteración sea ejecutado por lo menos una vez, para luego verificar la condición, si ésta es true ejecuta nuevamente el bloque, de lo contrario el ciclo es terminado. Para ello existe la sentencia llamada **do-while**, con la siguiente sintaxis:

```
do
{
   /* bloque que se ejecutará mientras
   la condición sea verdadera*/
}
while(condición)
```

2.7.2.- for.

Esta sentencia es útil cuando deseamos realizar un número determinado de iteraciones. Se utiliza de la siguiente forma:

```
for(inicio; condición; iteración)
{
   /* bloque de ejecución*/
}
```

Por ejemplo, si necesitamos imprimir los números entre 0 y 5 debemos realizar 6 iteraciones, desde el 0 al 5, y debemos saltar de uno en uno. El código será éste:

```
public class pruebaFor
{
   public static void main(String argv[])
   {
      int i;
      for(i = 0; i<6; i++)
      {
         System.out.println(i);
      }
   }
}
```

La salida generada es:

```
0
1
2
3
4
5
```

Esta sentencia funciona así. Primero inicializa la variable i en 0, esta parte se ejecuta una sola vez. Luego verifica la condición, como es true ejecuta el bloque de iteración. Una vez terminado dicho bloque, ejecuta la tercera parte del **for**, incrementando i en 1. Ahora vuelve a corroborar la condición, volviendo a ejecutar el bloque, ya que sigue siendo true. Así sucesivamente, hasta que adquiere el valor 6,

dado que 6 no es menor que 6, el ciclo se da por terminado.

La decisión de utilizar **for** o **while** dependerá exclusivamente del programador, ya que en cualquier caso se puede usar ambas sentencias para resolver el mismo problema. Aunque es recomendable utilizar **for**, ya que el código resulta más simple para su lectura. Para el ejemplo anterior, el código del programa utilizando **while** es el siguiente:

```
public class pruebaWhileFor
{
    public static void main(String argv[])
    {
        int i = 0;
        while(i<6)
        {
            System.out.println(i);
            i++;
        }
    }
}
```

2.8.- Clase System.

A lo largo de este texto hemos utilizado la clase **System** para imprimir valores por pantalla, pero esta clase representa varios recursos del sistema, y nos permite acceder a ellos independientemente del sistema operativo que estemos usando. Esta clase no se puede inicializar, y todas sus variables y métodos están declarados como **static**, y en todo momento sólo existirá una instancia de dicha clase en el sistema. Para utilizar sus variables y métodos sólo debemos invocarlos. Por ejemplo, para utilizar la variable out lo hacemos así:

```
System.out
```

De la misma forma procedemos para utilizar **println** de la instancia **out**:

```
System.out.println("Hola mundo!! ");
```

Esto es conocido como **Standard I/O**, por input/output, en español entrada/salida.

- **System.in**: entrada estándar (Standard input). Se utiliza para leer una entrada de datos del usuario.
- **System.out**: salida estándar (Standard output). Se usa para mostrar información al usuario.
- **System.err**: error estándar (Standard error). Se utiliza para mostrar mensajes de error al usuario.

Otro método útil de la clase **System** es **getProperty()**, el cual no entrega información valiosa acerca del sistema. Por ejemplo, con la siguiente línea obtenemos el nombre de usuario del sistema:

```
System.getProperty("user.name");
```

En la siguiente tabla vemos todas las propiedades que podemos entregarle como parámetro a este método, y que obtenemos con ello:

Propiedad	Descripción
"file.separator"	Separador de archivos
"java.class.path"	Classpath de Java
"java.class.version"	Versión del árbol de clases de Java
"java.home"	Directorio de instalación de Java
"java.vendor"	Nombre del distribuidor de Java

"java.vendor.url"	Dirección de Internet del distribuidor de Java
"java.version"	Versión de Java
"line.separator"	Separador de línea
"os.arch"	Arquitectura del sistema operativo
"os.name"	Nombre del sistema operativo
"os.version"	Versión del sistema operativo
"path.separatoe"	Separador de caminos
"user.dir"	Directorio actual de trabajo
"user.home"	Directorio Home del usuario
"user.name"	Nombre del usuario

Tabla 10. Lista de propiedades del método **getProperty()**.

A través de **System**, también podemos forzar la ejecución del recolector de basura. Para realizar este procedimiento sólo hay que ejecutar la siguiente instrucción:

```
System.gc()
```

2.9.- Secuencias de escape.

Los lenguajes poseen caracteres especiales conocidos como **caracteres de escape**. Una serie de estos caracteres es conocida como **secuencia de escape**. Cuando en una línea de texto aparece uno de estos caracteres, el compilador sabe que debe se trata de un **carácter de escape**, y qué por lo tanto debe tratarlo de forma especial. Sabemos que una línea de texto debe estar entre comillas. Por ejemplo, la siguiente sentencia:

```
System.out.println("Esto es una línea de texto");
```

Imprime por pantalla:

```
Esto es una línea de texto
```

Si deseamos que la palabra línea se imprima entre comillas, y lo hacemos de la siguiente manera:

```
System.out.println("Esto es una "línea " de texto ");
```

El compilador arrojará un error, porque interpreta el mensaje como "Esto es una " y el resto no lo entiende. Entonces, la forma correcta de hacerlo es usando caracteres de escape:

```
System.out.println("Esto es una \"línea\" de texto");
```

Ahora, cuando el compilador encuentre el carácter \, sabrá que se trata de un carácter especial y sabrá cómo manejarlo. Esta sentencia nos produce la siguiente salida por pantalla:

```
Esto es una "línea" de texto
```

A continuación se muestran los caracteres de escape posibles y su uso.

Carácter	Secuencia de escape
Contra barra	\\
Backspace	\b
Retorno de carro	\r
Comilla doble	\"
Form feed	\f
Tab horizontal	\t
Nueva línea	\n
Carácter octal	\DDD
Comilla simple	\’
Carácter unicode	\uHHHH

Tabla 11. Secuencias de escape.

3.- Clases.

3.1.- Árbol de clases.

Como lenguaje orientado a objetos puro, en Java todo es una clase o forma parte de una. Lo más tedioso para los programadores que ya tienen alguna experiencia con otros lenguajes es, quizás, conocer todas las clases del árbol de Java, pero una vez entendido cómo está construido dicho árbol, es mucho más fácil incorporar nuevas clases ya implementadas a nuestra aplicación.

Entonces, cualquier aplicación de Java se entiende como una extensión del **árbol de clases**. A diferencia de **C++**, no se pueden declarar métodos o variables globales, se pueden simular a través de la sentencia **static**. La única sentencia que se puede usar fuera de una clase es **import**, pero el compilador reemplaza la sentencia con el contenido del archivo que se indique.

3.1.1.- Tipos de clases.

Si nos fijamos en los ejemplos que hemos desarrollado, notamos que todas las clases fueron definidas como **public**. Sin embargo, existen 4 modificadores para definir el tipo de clase:

- **public**: son las que más libertades poseen, ya que se puede acceder directamente ellas desde otras clases, o por medio de herencia. También son accesibles dentro del mismo paquete en que se han declarado, o desde otros paquetes a través de **import**.
- **abstract**: una clase de este tipo tiene al menos un método abstracto. Este tipo de clase no se **instancia**, sino que se utiliza como clase base para la herencia.
- **final**: es una clase que termina una herencia. Se puede ver como una hoja del árbol de clases. Desde este tipo de clases no se puede heredar.
- **synshronizable**: este modificador especifica que todos los métodos definidos en ella son sincronizados, es decir, que no se puede acceder al mismo tiempo.

3.2.- Declarar, inicializar y usar un objeto.

Cuando creamos una instancia de un objeto, debemos hacerlo a través de un **constructor**. El constructor pertenece al objeto, y lo que hace es tomar todos los recursos necesarios para que el objeto funcione correctamente, y devuelve la instancia del objeto. Entonces, cuando utilizamos la sentencia **new**, lo que estamos haciendo es llamar al constructor del objeto.

Para declarar un objeto se utiliza una sintaxis similar a la usada para declarar variables, es decir:

```
NombreDelObjeto nombreDeLaVariable = new NombreDelObjeto();
```

La sentencia **new** sirve para indicar que queremos una nueva instancia del objeto, y debemos tener en cuenta que este operador nos devuelve una referencia a un objeto, y que esta referencia debe ser asignada a una variable que pueda referenciar dicho objeto.

Si deseamos crear un objeto de la clase **Character,** tenemos que saber que su constructor es parametizado, y el parámetro que recibe es del tipo **char**. Para crear una instancia de **Character** de nombre c, e inicializada con el parámetro a debemos ejecutar la siguiente sentencia:

```
Character c = new Character('a');
```

Utilizando el método "**charValue()** de la clase **Character** podemos ordenarle al objeto que devuelva el valor con que fue instanciado. Para el ejemplo anterior, se envía un mensaje a la instancia c por medio del método **charValue()**, obteniendo el char a. El comando para hacer esto es:

```
c.charValue()
```

3.3.- Referenciar variables de un objeto.

En ocasiones, las variables y métodos que están dentro de un objeto don públicas, es decir, que no están encapsuladas. En estos casos las podemos referenciar, sin la necesidad de crear un objeto, desde cualquier parte de nuestra aplicación. Para ejemplificar esto, veamos las variables "MAX_VALUE" y "MIN_VALUE" de la clase **Integer**.

```
class PruebaInteger
{
   public PruebaInteger()
   {
   }
   public static void main(String argv[])
   {
      // Se muestran las variables públicas de la clase Integer
      System.out.println("Máximo entero: " +Integer.MAX_VALUE);
      System.out.println("Mínimo entero: " +Integer.MIN_VALUE);
   }
}
```

La salida de este programa será:

```
Máximo entero: 2147483647
Mínimo entero: -2147483648
```

No confundamos el objeto **Integer** con el tipo de datos primitivo **int**. En este caso, **Integer** es la clase que representa al tipo de dato **int**, y provee métodos para el manejo de este tipo de dato. En Java todos los tipos de datos primitivos están representados por un objeto. Tenemos el objeto **Long** para **long**, **Character** para **char**, **Double** para **double**, etc.

Dado que "MAX_VALUE" y "MIN_VALUE" son variables públicas de la clase **Integer**, podemos referencialas directamente, sin una instancia. Por otra parte, para respetar un estándar, cuando declaremos variables de este tipo lo haremos siempre en mayúsculas.

En Java existen cuatro niveles de acceso, y todos son válidos para métodos y variables. Los niveles de acceso son los siguientes:

- **public**: se puede acceder a las variables y métodos de instancia publica desde cualquier lugar.
- **protected**: sólo las subclases de la clase pueden acceder a las variables y métodos de instancia protegidos.
- **private**: Las variables y métodos de instancia privados sólo son accesibles desde dentro de la clase. No son accesibles desde las subclases.
- **friendly**: si no se especifica el control de acceso, las variables y métodos de instancia se declaran automáticamente como friendly (amigables). El concepto es el mismo que para **protected**.

3.3.1.- Sentencia static.

En algunos casos vamos a necesitar que una variable no cambie de valor para ninguna instancia de la clase, es decir, que exista una única copia de la variable de instancia. Para esto se utiliza la sentencia **static**. Sirve tanto para variables como métodos. Pero un método declarado como **static** no podrá acceder a ninguna variable de la clase, ésta es una regla de acceso. Por lo tanto, los métodos estáticos sólo pueden modificar variables estáticas o variables locales.

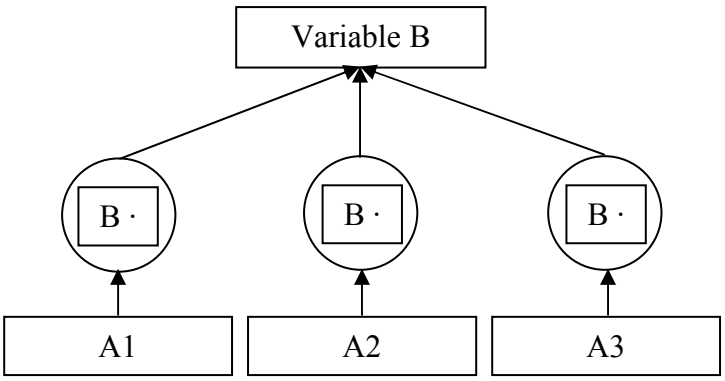


Figura 1. Ejemplo de referencia de varios objetos a una variable static. A1, A2 y A3 son instancias de A.

3.4.- Nuestro primer objeto Java.

A continuación crearemos la clase "empleado", que nos sirve para mantener la información de los empleados de una compañía.

```
public class Empleado
{
   // Los datos miembros son declarados como privados
   private int legajo;
   private String nombre;
   private float sueldo;
   // Variable pública y estática que almacena el sueldo base de los empleados
   public static flota SUELDO_BASE = 150f;
   // En el constructor por defecto vamos a inicializar las variables
   public Empleado()
   {
      this.legajo = 0;
      this.nombre = new String();
      this.sueldo = 0.0f;
   }
   /* En el constructor parametrizado inicializaremos las
   variables con los parámetros correspondientes */
   public Empleado(int legajo, String nombre, float sueldo)
   {
      this.legajo = legajo;
      this.nombre = nombre;
      this.sueldo = sueldo;
   }
   // El método getSueldo nos va a devolver el sueldo del empleado
   public float getSueldo
   {
      return(this.sueldo);
   }
   public static void main(String argv[])
   {
```

```
        /* Se inicializan dos empleados, uno por medio de el constructor
        parametrizado  y el otro utilizando el constructor por defecto */
        Empleado emp1 = new Empleado();
        Empleado emp2 = new Empleado(20, "Pablo", 700f);
        //Mostraremos los sueldos de emp1 y emp2, además el sueldo base
        System.out.println("Sueldo de empleado 1: " +emp1.getSueldo());
        System.out.println("Sueldo de empleado 2: " +emp2.getSueldo());
        System.out.println("Sueldo base: " +emp1.SUELDO_BASE);
    }
}
```

La salida por pantalla que este programa genera es:

```
Sueldo de empleado 1: 0.0
Sueldo de empleado 2: 700.0
Sueldo base: 150.0
```

Podemos notar que a lo largo de este ejemplo se utilizó la palabra reservada **this**. Ésta se utiliza cuando se desea hacer referencia a una variable o a un método que es propio de la clase. Por ejemplo, en el constructor parametrizado, vemos que a "this.nombre", la variable nombre de la clase, le asigna "nombre", el nombre recibido como parámetro.

3.5.- Eliminar objetos Java.

En el ejemplo anterior, cuando termina el bloque de ejecución del **main**, mueren las instancias emp1 y emp2. No debemos preocuparnos de eliminar los objetos, ya que el **Garbage Collector** (recolector de basura) se encargará de eliminar de la memoria los objetos que ya no estén referenciados por nadie.

El **Garbage Collector** llamará al método **finalize()** de cada objeto que encuentre sin referencia. La función de este método es destruir un objeto por orden del **Garbage Colector**. Como en Java todas las clases heredan de **Object**, y ésta tiene el método **finalize()** implementado. En otras palabras, todos los objetos que componen Java tienen un método **finalize()**.

3.6.- Interfaces.

Las interfaces son definiciones de clases abstractas. Éstas pueden contener métodos, pero todos ellos deben ser abstractos. Para los atributos, todos deben ser declarados como **static final**, o sea, constantes. Ahora veamos un ejemplo:

```
public interface Circulo
{
    public void dibujar();
    public double getRadio();
    public double getDiametro();
}
```

Ahora, toda clase que implemente círculo deberá definir los tres métodos obligatorios. Veamos un ejemplo en donde implementemos la interfaz.

```
public class CirculoImplementado implements Circulo
{
    double radio;
    public CirculoImplementado(double r)
    {
        this.radio = r;
    }
```

```
    public void dibujar()
    {
        // Puedo declarar el método y no agregarle código
    }
    public double getRadio()
    {
        return(this.radio);
    }
    public double getDiametro()
    {
        return(2*this.radio);
    }
}
```

Note que aparece la palabra reservada **implements**, y es a través de ella que declaramos que vamos a implementar la interfaz Circulo.

3.6.1.- Clases anónimas.

Una clase anónima es aquella que esta definida e instanciada en el mismo punto. Llevan ese nombre ya que dichas clases no tienen nombre de clase, y por lo tanto no pueden tener constructor y sólo puede existir una instancia de la clase.

Veamos el ejemplo de la interfaz Circulo, ahora vamos a declararla como anónima:

```
public class CirculoAnonimo
{
    public CirculoAnonimo(double r)
    {
        // Declaro la clase anónima
        new Circulo()
        {
            double radio;
            public void dibujar()
            {
            }
            public double getRadio()
            {
                return(this.radio);
            }
            public double getDiametro()
            {
                return(2*this.radio);
            }
        };
    }
}
```

Se recomienda usar las clases anónimas solamente cuanto éstas tengas a lo más un atributo y dos métodos. Siempre que sea posible, debemos evitar el uso de clases anónimas.

4.- Paquetes.

4.1.- La sentencia import.

Como ya hemos visto, la sentencia **import** sirve para importar clases o conjuntos de clases (paquetes) desde el árbol de clases. Debemos recordar que las clases que fueron declaradas como **public** dentro del paquete son las únicas que se pueden importar.

Lo mas difícil de recordar, cuando se aprende Java, es donde se encuentra cada clase, es decir, el árbol de clases. Debemos tener en cuenta que los paquetes, además de contener clases, podrían tener otros paquetes. Se puede hacer una analogía entre los paquetes y los directorios y subdirectorios por un lado, y los archivos y las clases por otro, ya que la organización utilizada para mantener el árbol de clases es la misma.

Por otra parte, debemos tener en cuenta que existe un conjunto de paquetes generales que contienen a otros paquetes y clases agrupados según su función, es decir, que no es necesario conocer todo el árbol de clases para comenzar a programar, sino que iremos incorporando nuevos paquetes y clases a medida que vayamos aprendiendo el lenguaje.

```
import java.util.Vector;
```

Esto nos indica que dentro del paquete **java** se encuentra un paquete llamado **util**, y es dentro de el que está la clase **Vector**. Ahora, si necesitamos la clase **LinkedList**, que también se encuentra dentro de **util**, debemos agregar el siguiente **import**:

```
import java.util.Vector;
import java.util.LinkedList;
```

Ahora, si deseamos importar todas las clases publicas que se encuentran dentro del paquete **util**, debemos hacerlo así:

```
import java.util.*;
```

Al usar **import** debemos tener en cuenta que al final de una sentencia debe haber una clase publica o un asterisco que haga referencia al menos a una clase. Por lo tanto la ésta sentencia es incorrecta:

```
import java.util; // util es un paquete y no una clase
```

4.2.- Archivos JAR.

Luego de desarrollar nuestras aplicaciones, necesitamos distribuirlas de forma práctica y sencilla. Esto lo logramos a través de los archivos **JAR**, que pueden contener un conjunto de paquetes y clases, de tal forma que, agregando el archivo a nuestro proyecto extendemos el árbol de clases. Todas las clases que contenga el archivo **JAR** las importamos por medio de la sentencia **import**. El comando utilizado para crear estos archivos es:

```
jar{ctxu}[vfm0M] [nombre del jar] [archivo mainfest] [-C directorio] archivos
```

Las opciones más comunes son:

Comando	Resultado
C	Crea un nuevo archivo JAR
T	Muestra el contenido de un archivo JAR
X	Extrae las clases de un archivo JAR
U	Actualiza un JAR
V	Genera salida por pantalla
F	Especifica el nombre del archivo
M	Incluye un archivo manifies
0	Sólo almacenamiento, sin compresión

Tabla 1. Comandos para crear archivos **JAR**.

Ahora, como ejemplo, crearemos un archivo **JAR** con las clases del ejemplo de interfaces, las clases eran Circulo y CirculoImplementado. El comando para crear ejemplo.jar seria:

```
jar cvf ejemplo.jar Circulo.class CirculoImplementado.class
```

Ahora, la forma de ejecutar dicho archivo es mediante el comando **java**, pero debemos especificar cual de las clases del **JAR** contiene el programa principal (main). Para ello se usa el **manifes**, o archivo manifiesto. Ahora debemos crear un archivo manifiesto para nuestro **JAR**, que será de texto plano y quedará así:

```
Manifest-Version: 1.0
Main-Class: PruebaHerencia
```

Ahora podemos ejecutar el ejemplo con el comando **java** de la siguiente manera:

```
java -jar ejemplo.jar
```

El formato del archivo **JAR** es similar al de los archivos **ZIP**, por lo que utilizando una herramienta como **WinZip** podemos ver el contenido de estos archivos.

4.3.- Classpath.

El **classpath** es el camino de las clases, y sirve para indicarle a Java dónde están las clases que queremos usar. Podemos definir el classpath de dos formas, una primera es usando la opción **-classpath** del intérprete Java, la segunda es definiendo una variable de entorno en el sistema operativo. Cada una tiene sus ventajas y desventajas. Si lo definimos como variable de entorno no tenemos que escribir el classpath cada vez que compilamos o ejecutamos nuestra aplicación, pero el classpath puede cambiar de acuerdo con la aplicación.

Supongamos la siguiente clase:

```
import CirculoImplementado;
public class PruebaCamino
{
   CirculoImplementado a = new CirculoImplementado();
   public CirculoAnonimo()
   {
   }
}
```

Como vemos, estamos usando la clase CirculoImplementado, que se encuentra dentro del archivo ejemplo.jar, por eso lo importamos con la sentencia **import**. Para poder compilar y ejecutar esta clase debemos agregar ejemplo.jar en el **classpath**, y así extender el árbol de clases de Java, pidiéndole que incorpore las clases Circulo y CirculoImplementado. Suponiendo que el archivo ejemplo.jar se encuentra en c:\misJar, el código para compilar es el siguiente:

```
javac -classpath c:\misJar\ejemplo.jar PruebaCamino.java
```

Y el código para ejecutar es:

```
java -classpath c:\misJar\ejemplo.jar PruebaCamino.java
```

Además, podemos utilizar más de una clase, que estén en distintas partes del disco. Para hacerlo debemos agregar a nuestro **classpath** todos los caminos separados con un punto y coma (;) si estamos en **Windows** o con dos puntos (:) si estamos en **Linux**.

Si deseamos crear la variable de entorno debemos hacerlo utilizando el comando **set**, usando el mismo criterio:

```
set CLASSPATH=c:\misJar\ejemplo.jar
```

Ahora, para compilar y ejecutar se utilizaría, respectivamente:

```
javac PruebaCamino.java
java PruebaCamino.java
```

4.4.- Creación de paquetes.

Ahora que sabemos cómo usar paquetes y crear distribuciones, falta saber cómo crear estos paquetes, es decir, cómo agrupar nuestras clases según un criterio. Para esto existe la sentencia **package**. Esta sentencia se agrega al principio del archivo, y su sintaxis es:

```
package nombreDelPaquete;
```

Con esta línea se declara que la clase que está a continuación pertenece al paquete declarado. Por medio de la sentencia **import**, lo que evitamos es el conflicto entre nombres de clases. Es decir que en Java podemos tener varias clases con el mismo nombre, pero si deseamos usar dos en el mismo programa, estaremos obligados a usarlas con el nombre completo en cada referencia. Esto ocurre por ejemplo con la clase **Date**, que se encuentra en el paquete **java.util** y en **java.sql**. Si deseamos usarlas, no podemos hacer:

```
import java.util.Date;
import java.sql.Date;
public class PruebaImport
{
    Date s;
    Date m;
    public PruebaImport()
    {
    }
}
```

Esto nos arrojará un error de doble **import**, porque no sabrá si queremos declarar las variables como **Date** de **java.util** o de **java.sql**. La forma correcta de hacerlo es:

```
import java.util.Date;
import java.sql.Date;
public class PruebaImport
{
    java.util.Date s;
    java.sql.Date m;
    public PruebaImport()
    {
    }
}
```

De esta forma declaramos a s como **Date** del paquete **java.util** y a m como **Date** del paquete **java.sql**.

4.4.1.- Árbol de paquetes o subpaquetes.

En algunos casos necesitamos crear un paquete dentro de otro, por ejemplo, supongamos que deseamos armar la estructura de la **Figura1**:

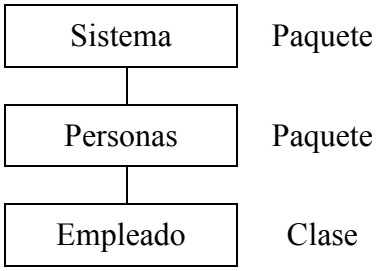


Figura 1. Estructura de paquetes que deseamos lograr.

La declaración para la clase Empleado será:

```
// Declaro que empleado está dentro de personas, y personas dentro de sistema
package sistema.personas;
public class Empleado
{
    public Empleado()
    {
    }
}
```

Ahora, cuando necesitemos usar Empleado, el **import** será:

```
import sistema.personas.*;
```

4.5.- La clase Object.

Como ya hemos visto anteriormente, **Object** es la clase superior de la jerarquía de Java, lo que equivale a decir que todas las clases de Java dependen de **Object**. Esta clase define algunos métodos que por herencia tendrán todas las clases de Java. Los métodos más importantes son:

clone(): crea un objeto a partir de otro de la misma clase. Si no puede hacerlo, lanza una excepción del tipo **CloneNotSupportedException**. Si queremos que nuestras clases tengas esta capacidad, debemos implementar la interfaz **Clonable**.

equals(): sirve para saber si dos objetos son iguales. Devuelve **true** si son referencia de la misma clase, o si son objetos diferentes pero con iguales valores en sus variables miembro, de lo contrario retorna **false**.

toString(): devuelve una cadena que contiene una representación del objeto como cadena de caracteres.

finalize(): es llamado por el **Garbage Collector** para eliminar un objeto.

notify(), **notifyAll()** y **wait()**: como Java fue creado en un momento en que la teoría de hilos estaba totalmente desarrollada, tenemos estos tres métodos que sirven para programar cualquier objeto Java como un **Thread** (hilo).

4.6.- Conversión de tipos.

A veces tenemos determinada clase de datos, pero lo necesitamos representado en otro tipo. Este cambio podemos necesitarlo entre tipos de datos primitivos. La conversión de tipos o **casting** es de una variable de mayor a una de menor precisión, debemos hacerla explícitamente porque es una conversión insegura y puede provocar errores. Para hacer la conversión explícita hay que anteponer el tipo al que deseamos convertir entre paréntesis. Por ejemplo, supongamos que hacemos la división entre dos **double** y queremos almacenar el resultado en un **float**. Entonces haremos:

```
double b=4;
double c=2;
float a = (double) (b/c)
```

Entre clases el casting es igual, sólo que conceptualmente no estamos reduciendo ni aumentando la precisión, sólo estamos cambiando la referencia.

```
Object a;
Integer b = (Integer) a;
```

Es bastante común hacerlo para usar los métodos de la clase a la cual convertimos, por que en la clase actual no tenemos dichos métodos. Sin embargo, para el casting entre clases, Java obliga a que entre ellas haya una relación por herencia. La conversión es implícita de la subclase a la superclase, y explícita si es a la inversa, es decir, de la superclase a la subclase.

4.7.- Algunas clases de Java.

A continuación mencionaremos las clases más usadas de Java, y alguno de sus métodos. Además, entregaremos una serie de aspectos relevante que hay que conocer de una clase antes de usarla. Ellos son:

- **Descripción**: para describir brevemente para qué sirve la clase.
- **Tipo**: para saber cómo está declarada la clase.
- **Ubicación**: para saber en qué rama del árbol de clases se encuentra.
- **Superclase**: cuál es la superclase, es decir, de quién hereda.
- **Subclase**: si tiene subclases, cuáles son.
- **Implementa**: para saber qué interfaz implementa, en caso de que lo haga.
- **Atributos**: cuáles son sus atributos miembro.
- **Constructores**: cómo se crea la clase en caso de ser instanciada.

Finalmente, los métodos más usados los veremos mediante ejemplos.

4.7.1.- Math.

Esta clase nos provee una serie de métodos para evaluar las funciones matemáticas más comunes. A continuación describimos sus principales características:

- **Tipo**: public final class.
- **Ubicación**: **java.lang**.
- **Superclase**: **java.lang.Object**.
- **Subclase**: -
- **Implementa**: -
- **Atributos**: dos constantes **static**: **E** y **PI**.
- **Constructores**: no tiene, recuerde que todos sus atributos y métodos son estáticos.

Ahora veamos un ejemplo:

```
import java.lang.*;
public class UsoClaseMath
{
   public UsoClaseMath()
   {
      System.out.println("El valor de PI es: " +Math.PI);
      System.out.println("La raíz cuadrada de 25 es: " +Math.sqrt(25));
      System.out.println("5 elevado al cuadrado es: " +Math.pow(5,2));
      System.out.println("El coseno de 60 es: " +Math.cos(60));
      System.out.println("El seno de 60 es: " +Math.sen(60));
      System.out.println("La tangente de 60 es: " +Math.tan(60));
      System.out.println("El valor absoluto de -10 es: " +Math.abs(-10));
      System.out.println("El mínimo entre 2 y 8 es: " +Math.min(2,8));
      System.out.println("El máximo entre 2 y 8 es: " +Math.max(2,8));
   }
```

```
    public void main(String argv[])
    {
        new UsoClaseMath();
    }
}
```

La salida del programa es:

```
El valor de PI es: 3.3141592653589793
La raíz cuadrada de 25 es: 5.0
5 elevado al cuadrado es: 25.0
El coseno de 60 es: -0.9524129804151563
El seno de 60 es: -0.3048106211022167
La tangente de 60 es: 0.320040389379563
El valor absoluto de -10 es: 10
El mínimo entre 2 y 8 es: 2
El máximo entre 2 y 8 es: 8
```

4.7.2.- Integer.

La clase **Integer** representa el tipo de dato primitivo **int**. Sus principales características:

- **Ubicación**: **java.lang**.
- **Superclase**: **java.lang.Number**.
- **Subclase**: -
- **Implementa**: **Comparable**, **Serializable**.
- **Atributos**: Tiene tres constantes **static**:
    - **MAX_VALUE**: el valor máximo representable por **int**.
    - **MIN_VALUE**: el valor mínimo representable por **int**.
    - **TYPE**: el tipo de dato que representa.
- **Constructores**:
    - **Integer(int value)**, que crea una instancia **Integer** a partir del tipo primitivo **int.**
    - **Integer(String s)**, que crea una instancia **Integer** a partir del objeto **String**.

Ejemplo de uso:

```
public class UsoClaseInteger
{
    Integer c;
    public UsoClaseInteger()
    {
        // Creamos un Integer desde un int
        Integer a = new Integer(1);
        // Creamos un Integer desde un String
        Integer b = new Integer("1");
        // Si compareTo devuelve 0 quiere decir que los dos Integer son iguales
        if(a.compareTo(b) == 0)
        {
            // intValue() devuelve el int con que está instanciado Integer
            int suma = a.intValue() + b.intValue();
            c = new Integer(suma);
        }
```

```
        // Convertimos el Integer a String usando el método toString()
        System.out.println("El resultado es: " +c.toString());
    }
    public void main(String argv[])
    {
        new UsoClaseInteger();
    }
}
```

Luego, la salida por pantalla es:

```
El resultado es: 2
```

4.7.3.- Random.

Esta clase genera números **pseudorandom**. Sus características son:

- **Ubicación**: **java.util**.
- **Superclase**: **java.util.Object**.
- **Subclase**: **SecureRandom**.
- **Implementa**: **Serializable**.
- **Atributos**: -
- **Constructores**:
    - **Random()**: Crea un nuevo generador de números aleatorios.
    - **Random(long seed)**: Crea un nuevo generador de números aleatorios a partir de una semilla para no repetir secuencias.

Ejemplo de uso:

```
import java.util.*;
public class UsoClaseRandom
{
    public UsoClaseRandom()
    {
        Random r = new Random()
        System.out.println("Flotantes pseudoaleatorios:");
        for(int i = 0; i < 5; i++)
            System.out.print(r.nextFloat() + " ");
        System.out.println("\nEnteros pseudoaleatorios entre 0 y 50:");
        for(int i = 0; i < 5; i++)
            System.out.print(r.nextInt(50) + " ");
    }
    public void main(String argv[])
    {
        new UsoClaseRandom();
    }
}
```

Luego, la salida por pantalla es:

```
Flotantes pseudoaleatorios:
0.22639495 0.8039619 0.51486355 0.37433088 0.26232278
Enteros pseudoaleatorios entre 0 y 50:
30 6 41 45 18
```

4.7.4.- StringTokenizer.

La clase **StringTokenzer** permite dividir un **String** entre **tokens**, o delimitadores, de forma simple. Sus principales características:

- **Ubicación**: **java.util**.
- **Superclase**: **java.util.Object**.
- **Subclase**: -
- **Implementa**: **Enumeration**.
- **Atributos**: -
- **Constructores**:
    - **StringTokenizer(String str)**: crea un string **tokenizer** a partir de un string.
    - **StringTokenizer(String str, String delim)**: crea un string **tokenizer** a partir de un string, permitiendo declarar cuál es el delimitador.
    - **StringTokenizer(String str, String delim, bolean returnDelims)**: crea un string **tokenizer** a partir de un string, permitiendo declarar cuál es el delimitador y si el delimitador es parte de la cadena devuelta.

Ejemplo de uso:

```
import java.util.*;
public class UsoClaseStringTokenizer
{
   String cadena = "Diego|27|011-4486-5863|Amabay 258|1754"
   public UsoClaseStringTokenizer()
   {
      // Creamos el objeto con la cadena y el delimitados
      StringTokenizer stk = new StringTokenizer(cadena,"|",);
      System.out.println("Cantidad de tokens: " +stk.countTokens());
      // Mientras haya más tokend
      while(stk.hasMoreTokens())
      {
         // Tomo el siguiente y lo muestro
         System.out.println(stk.nexElement());
      }
      System.out.println("Cantidad de tokens: " +stk.countTokens());
   }
   public void main(String argv[])
   {
      new UsoClaseStringTokenizer();
   }
}
```

La salida del programa es:

```
Cantidad de tokens: 5
Diego
27
011-4486-5863
Amabay 258
1754
Cantidad de tokens: 0
```

4.7.5.- Runtime.

La clase **Runtime** permite a la aplicación comunicarse con el sistema operativo en el cual se está ejecutando. Sus características son:

- **Ubicación**: **java.util**.
- **Superclase**: **java.util.Object**.
- **Subclase**: -
- **Implementa**:-
- **Atributos**: -
- **Constructores**: -

Veamos un ejemplo:

```
public class UsoClaseRuntime
{
    public UsoClaseStringRuntime()
    {
        Runtime rt = Runtime.getRuntime();
        try{
            // A través del método exec() ejecutamos notepad de Windows
            rt.exec("C:\\WINDOWS\\NOTEPAD:EXE");
        }
        match(Exception ex){}
    }
    public void main(String argv[])
    {
        new UsoClaseRuntime();
    }
}
```

5.- Vectores y matrices.

5.1.- Secuencias de variables.

Muchas veces, cuando programamos, nos encontramos con variables que tienes cosas en común y que guardan elación entre sí. Una de las herramientas que se utilizan para almacenar estas secuencias de variables son los vectores (**array**), en otras palabras, éstos representan una secuencia de cosas.

La representación más común de vectores es la siguiente:

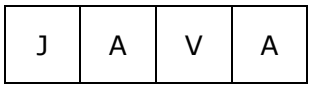


Figura 1. Representación de un arreglo.

Los vectores no son conjuntos, porque en un conjunto no importa el orden y no se pueden repetir elementos. En un vector podemos tener el mismo elemento repetido todas las veces que lo necesitemos.

Para acceder a un elemento del vector lo hacemos a través de su índice. En nuestro ejemplo de la **Figura 1**, dicho índice va de 0 a 3, es decir, que el elemento de la posición 2 es V (**Figura 2**).

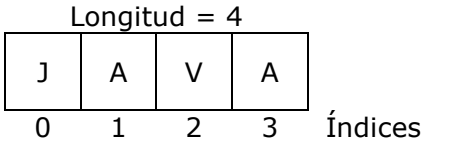


Figura 2. Propiedades de un array.

5.2.- Arreglos unidimensionales.

Los arreglos unidimensionales no son más que vectores de una dimensión. La definición de estos vectores es la siguiente:

```
TipoDeDato nombreDelVector[] = new TipoDeDato[cantidad de elementos];
```

Por ejemplo, para definir un vector de diez números enteros, se hace de la así:

```
int vector[] = new int[10];
```

Para cargar el vector, sencillamente lo hacemos a través del índice. Supongamos que deseamos poner un número en la quinta posición del vector. Recordemos ahora que la quinta posición lleva el índice cuatro, ya que el índice de los vectores comienza desde cero.

```
vector[4] = 8; // En el índice 4 se cargo el número 8
```

Ahora vamos a usar la sentencia **for** para cargar el vector completo:

```
for(int i=9; i<10; i++)
    vector[i] = i;
```

Entonces el vector queda como se observa a continuación:

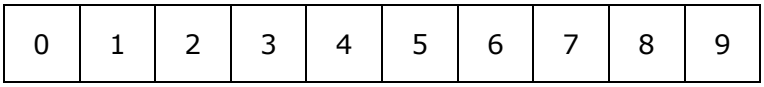


Figura 3. Vector cargado con un **for** de 0 a 9.

5.3.- Algoritmos.

En Java existe la clase **Collections**, la cual implementa una buena cantidad de algoritmos estáticos que operan sobre colecciones.

- **sort()**: implementa un método de ordenamiento mediante **MerdeSort**.
- **reverse()**: invierte el orden de los elementos del arreglo.
- **binarySearch()**: implementa búsqueda binaria de un elemento en particular.
- **copy()**: recibe dos listas y copia todos los elementos de una lista a la otra.
- **fill()**: llena el arreglo con un objeto determinado. Es muy útil para "limpiar".
- **max()**: devuelve el máximo de acuerdo con un comparador.
- **min()**:devuelve el mínimo de acuerdo con un comparador.

6.- Herencia: Subclases y superclases.

6.1.- Herencia simple.

Hay veces que estamos frente a objetos que presentan entre sí ciertas similitudes. Por ejemplo, si tenemos la clase Persona, a partir de ella podemos definir Empleado, Alumno, Cliente, etc., ya que todos son personas. Frente a esto, la programación orientada a objetos provee la herencia, de forma tal que podemos construir objetos a partir de otros, de los cuales heredará sus atributos y métodos.

Java implementa la **herencia simple**, es decir que cada clase puede tener sólo una superclase. Para declarar que una clase hereda de otra se una la palabra reservada **extends** y a continuación el nombre de la superclase. Recordemos que en java "todas" las clases heredan de otra (**Figura 1**), ya que, de no especificar la superclase, se asume que hereda de **java.lang.Object**.

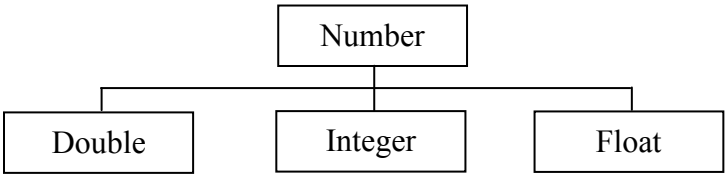


Figura 1. **Number** es superclase o clase padre de **Integer**, **Float** y **Double**. A su vez, éstas son subclases o clases hijas de **Number**.

6.1.1.- ¿Que variable miembro hereda una subclase?.

La herencia depende del modificador que tenga la variable miembro de la clase, es decir, cómo está declarada la clase. Recordemos que los modificadores de los métodos y las variables pueden ser **public**, **protected** o **private**.

La herencia o el acceso a esos métodos y variables depende también de si la clase de la cual heredamos se encuentra en el mismo paquete de clase que estamos programando o en otro.

En Tabla 1 y Tabla 2 vemos la visibilidad de acuerdo con los modificadores.

Modificador	Se hereda	Es accesible
Si modificador	Sí	Sí
Public	Sí	Sí
Protected	Sí	Sí
Private	No	No

Tabla 1. Acceso y herencia de métodos de clases que se encuentran dentro del mismo paquete.

Modificador	Se hereda	Es accesible
Si modificador	No	No
Public	Sí	Sí
Protected	Sí	No
Private	No	No

Tabla 2. Acceso y herencia de métodos de clases que se encuentran en otros paquete.

Ahora vamos a implementar la clase Persona, y a partir de ella vamos implementaremos la clase alumno, para ejemplificar la teoría (**Figura 2**).

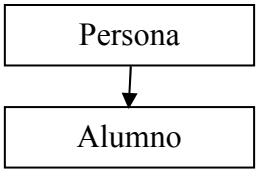


Figura 2. Alumno hereda de Persona.

```
public class Persona
{
    // Note que codigo es declarado private  y nombre protected
    private int codigo;
    protected String nombre;
    // El constructor no hace nada
    public Persona()
    {
    }
    /* Se definen los métodos set y get, para poner y sacar,
    respectivamente, el valor de codigo, ya que éste es privado */
    public void setCodigo(int codigo)
    {
        this.codigo = codigo;
    }
    public int getCodigo()
    {
        return(this.codigo);
    }
    // También agregamos los métodos para acceder al nombre
    public void setNombre(String nombre)
    {
        this.nombre = nombre;
    }
    public String getNombre()
    {
        return(this.nombre);
    }
    // Este método muesta los datos de la persona
    public void mostrar()
    {
        System.out.println("Código: " +this.codigo);
        System.out.println("Nombre: " +this.nombre);
    }
}
```

Ahora veamos la clase alumno:

```
// Usamos extends para declarar que Alumno está heredando a Persona
public class Alumno extends Persona
{
    /* Declaramos los datos de Alumno, nombre y codigo
```

```
    no debemos declararlas porque se heredan */
    float int notaPrimerCertamen;
    float int notaSegundoCertamen;
    public Alumno(int codigo, String nombre, float notaPrimero, float notaSegundo)
    {
        // Con super hacemos referencia a la variable nombre de la clase padre
        super.nombre = nombre;
        // Como codigo es privado se debe acceder a él a través del método setCodigo
        super.setCodigo(codigo);
        // Se inicializan las variables de Alumno
        this.notaPrimerCertamen = notaPrimero;
        this.notaSegundoCertamen = notaSegundo;
    }
    private float getPromedio()
    {
        return((this.notaPrimerCertamen + this.notaSegundoCertamen)/2);
    }
    // Este método muesta los datos del alumno y el promedio
    public void mostrar()
    {
        // Usamos mostrar de la superclase y agregamos el promedio
        super.mostrar();
        System.out.println("Promedio: " +this.getPromedio());
    }
}
```

Finalmente, para probar el ejemplo hacemos la clase Prueba Herencia, que instancia un Alumno y una Persona:

```
public class PruebaHerencia
{
    public PruebaHerencia()
    {
    }
        public static void main(String argv[])
        {
        // Creamos una instancia Alumno y hacemos referencia al método mostrar del alumno
        Alumno a = new Alumno(1,"Diego",80,100);
        System.out.println("-Alumno-");
        a.mostrar();
        /* Creamos una instancia Persona. Luego se hace
        referencia al método mostrar de persona */
        Persona p = new Persona()
        p.setCodigo(10);
        p.setNombre("Fabián");
        System.out.println("-Persona-");
        p.mostrar();
    }
}
```

La salida de este programa es:

```
-Alumno-
Código: 1
Nombre: Diego
Promedio: 90.0
-Persona-
Código: 10
Nombre: Fabián
```

6.2.- Escribir clases, atributos y métodos finales.

Cuando escribimos una clase, podemos usar la sentencia **final**, para asegurarnos de que no habrá subclases de ésta, es decir, las clases declaradas como **final** serán siempre hojas del árbol de clases.

También podemos declarar métodos como **final**, para asegurarnos de que no serán sobrecargados o sobrescritos por una subclase de la clase que los contiene.

En el caso de los atributos o datos miembro de una clase, cuando sean declarados como **final**, tendremos la certeza de que, una vez inicializado, no cambiará de valor durante toda la ejecución del programa. Podemos usar **final** en los atributos de la clase para simular constantes.

6.3.- Clases y métodos abstractos.

Una clase abstracta es una que está incompleta, y sólo ésta puede tener métodos abstractos. Éstos son aquellos que están declarados y pueden ser usados, pero todavía no están implementados. Una clase puede tener métodos abstractos si los hereda de una clase abstracta, o bien, si los tiene declarados explícitamente. Las clases abstractas no pueden ser instanciadas "nunca", si lo hacemos provocaremos un error en tiempo de compilación.

6.4.- Jerarquía de composición.

En el **Capítulo 1** mencionamos que no sólo existía la jerarquía de herencia, sino que también existe la jerarquía de composición, Esta última puede ser vista como "**contiene**" o "**tiene"**.

Aunque éste sea el caso más difícil de percibir, aún sigue siendo el caso más común de común de composición. Esto sucede cuando, en vez de usar los tipos de datos primitivos de Java para definir los atributos de una clase, lo hacemos con objetos que representan dichos tipos de datos.

Analicemos el siguiente ejemplo:

```
public class Empleado
{
    Integer codigo;
    String nombre;
    Double sueldo;
    Direccion direccion;
    public Empleado()
    {
    }
}
```

En este caso tenemos la clase Empleado, que "contiene" tres objetos: **Integer**, **String**, y **Double**. Además el empleado tiene una dirección, representada a través del siguiente Objeto.

```
public class Direccion
{
    String calle;
    Integer numero;
    public Direccion()
    {
    }
}
```

Note que la composición no se da solamente con la clase Direccion, ya que **Integer**, **String** y **Double** también son clases (**Figura 3**).

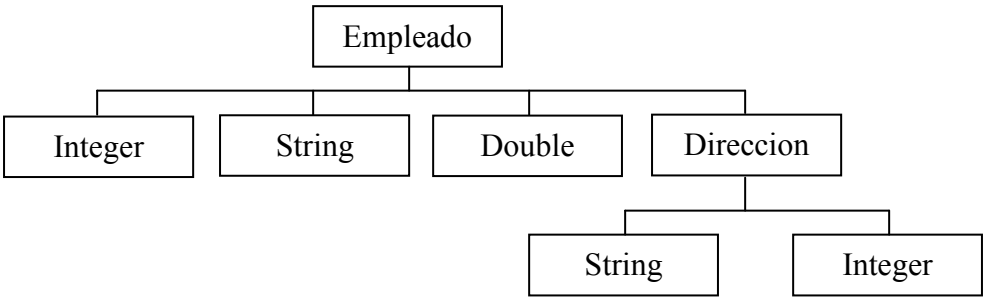


Figura 3. Composición de las clases Empleado y Direccion.

7.- Errores y excepciones.

7.1.- La jerarquía de las excepciones.

De la ejecución de nuestros programas va a surgir una serie de errores que debemos manejar de alguna manera. Para esto debemos determinar qué los provoca y, una vez definido eso, analizar qué camino tomar, evitar el error, mostrar un mensaje comprensible para el usuario, abortar la ejecución del programa, etc.

Java define una excepción como un objeto que es una instancia de la clase **Throwable** o alguna de sus subclases. Es decir que estamos ante una jerarquía de excepciones (**Figura 1**). Para comenzar, de **Throwable** heredan las clases **Error** y **Exception**.

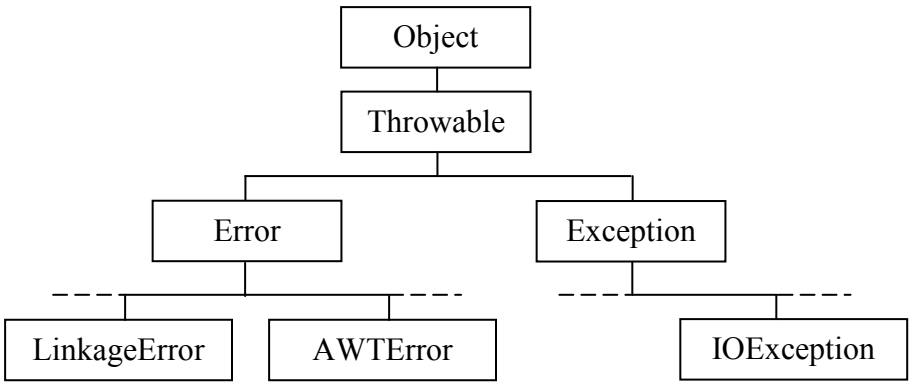


Figura 1. Parte de la jerarquía de errores y excepciones de Java.

Por lo general, un error tiende a provocarse en condiciones anormales, eso significa que no tiene sentido o no es posible capturarlo. Por ejemplo, **OutOfMemoryError** indica que no hay más memoria. Debido a la imposibilidad de capturar los errores, el programa termina.

De la clase **Error** se heredan las siguientes:

- **AWTError**
- **LinkageError**
- **ThreadDeath**
- **VirtualMachineError**

Las excepciones, en cambio, se producen dentro de condiciones normales, es decir que no sólo es posible capturarlas, sino que también el lenguaje nos obliga a capturarlas y manejarlas. Hay algunas excepciones que son las más conocidas, por ejemplo, las que heredan de **Exception**, que son las siguientes:

- **ClassNotFoundException**
- **InterruptedException**
- **IOException**
- **RuntimeException**
- **SQLException**

Hasta ahora hablamos de errores y excepciones, pero sin embargo existe otro conjunto de excepciones que el lenguaje no nos obliga a capturar y manejar, pero si queremos, podemos hacerlo. Éstas son conocidas como **RuntimeException**, excepciones generadas en tiempo de ejecución, que reflejan un error de codificación e imprimen en pantalla un mensaje de error.

Nunca olvidemos que una excepción no es un error, sino un evento que nuestro programa no sabe cómo manejar. Siempre que se provoque una excepción, podemos capturarla y manejarla, o evitarla cambiando la lógica del programa.

7.2.- Uso de try, catch y finally.

Para capturar y manejar las excepciones, Java proporciona tres palabras reservadas, estas son **try**, **catch** y **finally**. Para comenzar veremos **try** y **catch**.

Entre un **try** y un **catch** vamos a escribir el código de programa que pueda provocar una excepción. La estructura general es la siguiente:

```
try
{
   // Acá va el bloque de ejecución que puede provocar una excepción
}
catch(NombreDeLaExcepcion instancia)
{
   // Aca va el bloque de ejecución que maneja la excepción en caso de producirse
}
```

En el **catch** va el nombre de la excepción que deseamos capturar. Si no sabemos qué excepción es, recordemos que hay una jerarquía de excepciones, de tal forma que se puede poner la clase padre de todas las excepciones: **Exception**.

No todos los errores deben ser listados al declarar un método. Las instancias de la clase **Error** y **RunTimeException** no deben ser listadas, pues tienen un tratamiento especial porque pueden ocurrir en cualquier lugar dentro de nuestro programa. Se dan situaciones de las que no somos directamente responsables y que producen estas excepciones, como la situación de **OutOfMemoryError**. Hay solamente seis tipos de excepciones que pueden ser listadas en cláusulas **throws** dentro de **java.lang**.

Veamos un ejemplo que produce una excepción, pero no vamos a capturarla, para analizar un poco la salida de error.

```
// Vamos a crear un paquete de pruebas
package pruebas;
// Importamos java.util.* para usar la clase Vector
import java.util.*;
public class PruebaError
{
   // Declaramos el vector
   Vector v = new Vector();
   public PruebaError()
   {
      c.addElement(new String("Sergio"));
      c.addElement(new String("Francisco"));
      c.addElement(new String("Hismael"));
      c.addElement(new String("Erich"));
      // Note que el for intentará recorrer un vector de diez elementos
      for(int i=0; i<10; i++)
      {
         // Tomamos el elemento de la posición i y lo muestro
         System.out.println(v.get(i));
      }
   }
   public static void main(String argv[])
   {
      /* Creamos el objeto pero no lo referenciamos ya
      que no vamos a usar más que el constructor */
```

```
        new PruebaError();
    }
}
```

Luego de compilar este programa, lo ejecutamos y obtenemos esta salida:

```
Sergio
Francisco
Ismael
Erich
java.lang.ArrayIndexOutOfBoundsException: Array index out of range:
    at java.util.Vector.get(Vector.java:699)
    at pruebas.PruebaError.<init>(PruebaError.java:19)
    at pruebas.PruebaError.main(PruebaError.java:26)
Exception in thread "main"
```

El programa no causa ningún error de compilación, pero sí un error de ejecución. Esto es un **RuntimeException**.

Analizando el error vemos, primero, el nombre de la excepción provocada, en este caso **ArrayIndexOutOfBoundsException**, y a continuación, el mensaje de error. Luego de informar la excepción, muestra la línea exacta en que fue provocada. Es conveniente leer esto de abajo hacia arriba, de manera tal que podamos ver dónde empezó todo e ir siguiéndolo con nuestro código. Vemos que todo empezó en la línea 26, de paquete pruebas, en la clase PruebaError, en el método **main**. Si vamos a esta línea vemos que es donde creamos el objeto PruebaError. Seguimos hacia arriba, y ahora vemos que el error está en la línea 19 del paquete pruebas, en el objeto PruebaError, en el constructor, por eso dice **<init>**. Si vemos en el código, en esta línea es donde sacamos un elemento del vector, es entonces cuando determinamos cuál es la línea de código que provoca el error y por qué.

Ahora debemos elegir una solución. Podemos cambiar el límite del **for** de 10 a 4, para asegurarnos de no querer sacar más elementos, o podemos capturar la excepción y manejarla. Para capturar la excepción debemos poner las líneas que provocaron la excepción entre un **try** y un **catch**. En el ejemplo anterior debemos encerrar al **for**:

```
try
{
    for(int i=0; i<10; i++)
    {
        // Tomamos el elemento de la posición i y lo muestro
        System.out.println(v.get(i));
    }
}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println("Intentó sacar más elementos!!");
}
```

Ahora, la salida es:

```
Sergio
Francisco
Ismael
Erich
Intentó sacar más elementos!!
```

Como vemos, lo único que logramos es mostrar el mismo error, sólo que de manera mas comprensible. En ciertos casos, vamos a usar el bloque **catch** para evitar el error, no sólo para mostrarlo.

El lenguaje no nos obliga a capturar las excepciones que se provocan en tiempo de ejecución, sin embargo, éstas se producen, y muy a menudo. Entonces, empezará ejecutando el **try**, si se produce alguna excepción, corta el **try** y ejecuta el **catch**. Finalmente, lo que va en el bloque **finally**, se ejecutará siempre, haya habido excepciones o no. Algo muy común es usar dicho bloque para limpiar variables, cerrar archivos, restablecer conexiones, es decir, tareas que hayan podido quedar truncadas según se haya ejecutado el **try** completo o no.

La estructura de un bloque con try, catch y finally es la siguiente:

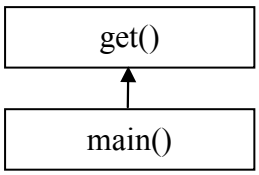
```
try
{
   // Acá va el bloque de ejecución que puede provocar una excepción
}
catch(NombreDeLaExcepcion instancia)
{
   // Aca va el bloque de ejecución que maneja la excepción en caso de producirse
}
finally
{
   //Este bloque se ejecuta siempre, sin importar si ejecutó el bloque del try o del catch
}
```

Java nos obliga a capturar algunas excepciones, y a manejarlas de alguna manera, es decir, que encerremos la línea que produce dicha excepción dentro de un **try** y un **catch**. Pero, si no queremos manejarla dentro del bloque, podemos lanzar la excepción hacia arriba, para que el bloque que llame a éste se encargue de manejar la excepción. Es decir, que la excepción le deberá manejar el padre en la pila de llamadas. Antes de continuar, veamos de qué se trata esto de la pila de llamadas.

7.2.1.- Pila de llamadas.

La pila de llamadas es un arreglo que cera el intérprete de Java en tiempo de ejecución para saber en todo momento quién llamó a quién. Esto sirve al programa para saber dónde tiene que seguir ejecutando cuando termina un bloque de ejecución.

La pila de llamadas para el ejemplo anterior lo podemos observar a continuación:



at java.until.Vector.get(Vector.java:699)

at pruebas.PruebaError.<init>(PruebaError.java:19)
at pruebas.PruebaError.main(PruebaError.java:25)

Figura 2. Pila de llamadas de Java.

Entonces, cuando una excepción no es tratada en el método donde se produce, Java buscar un bloque **try catch** en el método que lo trajo al actual. Si la excepción se lanza hasta el tope de la pila de llamadas sin encontrar un administrador específico para la excepción, entonces la ejecución se detendrá y dará un mensaje. Es decir, podemos suponer que Java nos está proporcionando un bloque **try catch** implícito, que imprime un mensaje de error, indica las últimas entradas en la pila de llamadas y sale. Este mensaje es el **stack trace** o camino de la pila. Para imprimir un **stack**

trace en una excepción capturada, lo hacemos a través del método **printStackTrace()** de **Exception**.

7.3.- Lanzar excepciones.

Entonces, si no queremos manejar la excepción y deseamos que lo haga el padre en la pila de llamadas, usamos la sentencia **throws**.

```
package pruebas;
public class PruebaThrow
{
   public PruebaThrow
   {
   }
   // El método devuelve el resultado de la división
   public flota divide(int a, int b) throws ArithmeticException
   {
      // Esta línea provoca la excepción que el throws lanza hacia arriba en la pila
      return(a/b);
   }
   public static void main(String argv[])
   {
      PruebaThrow p = new PruebaThrow();
      // La llamada provocará un ArithmeticException causado por la división por 0
      try
      {
         flota c = p.divide(15,0);
      }
      catch(ArithmeticException ex)
      {
         // Imprimimos un mensaje de error
         System.out.println("Error de división por 0");
         // Imprimimos el stack trace
         ex.printStackTrace();
      }
   }
}
```

El método divide es el que provoca la excepción, pero a través de **throws**, la lanza hacia arriba para que la maneje su padre en la pila de llamadas. Aquí, el padre es el **main**. Este se encargará de manejar la excepción provocada.

Ahora vamos a agregar un nuevo método a nuestra clase, que también puede provocar una excepción, esta vez, un **ClassCastException**. Este método es:

```
// Convierte de Object a Integer
public Integer convertir(Object a)
{
   return((Integer)a);
}
```

A continuación, el bloque de ejecución puede provocar un **ArithmeticException** o un **ClassCastException**. Para poder manejar ambas excepciones, sólo debemos agregar un nuevo **catch**, es decir que podemos tener

cuantos **catch** como necesitemos. El nuevo **main** y bloque **try** catch quedarían así:

```
public static void main(String argv[])
{
   PruebaThrow p = new PruebaThrow();
   /* La primera sentencia provocará un ArithmeticException causado por la división
   por 0. La segunda un ClassCastException por intentar convertir de Float a Integer */
   try
   {
      flota c = p.divide(15,0);
      p.convertir(new Flota(c));
   }
   catch(ArithmeticException ex)
   {
      System.out.println("Error de división por 0");
   }
   catch(ClassCastException ex)
   {
      System.out.println("Error Convirtiendo Float a Integer");
   }
}
```

7.4.- Capturar excepciones más genéricas.

Como vimos en el ejemplo anterior, podemos tener cuantos **catch** creamos necesarios, pero también vimos que con agregar sólo una línea al bloque de ejecución se provocaba una excepción que el programa no estaba preparado para manejar.

Aprovechando la jerarquía de excepciones, podemos reemplazar todos los **catch** de excepciones particulares por la superclase **Exception**, haciendo una captura genérica. Ahora el **main** del ejemplo anterior quedaría así:

```
public static void main(String argv[])
{
   PruebaThrow p = new PruebaThrow();
   try
   {
      flota c = p.divide(15,0);
      p.convertir(new Flota(c));
   }
   catch(Exception ex)
   {
      System.out.println("Error!!");
      ex.printStackTrace();
   }
}
```

Veamos el mensaje de error. Ahora no sabemos cuál es la excepción que se provocó, porque estamos capturando la excepción genérica. Nos queda por analizar el mensaje, a través del objeto **getMessage()**, o el **stack trace**, a través de **printStackTrace()**.

8.- Streams.

8.1.- Flujo de datos.

La información podrá ser representada por algún tipo de Java, es decir, podrá ser **Object**, **String**, **Charater**, etc. Para poder leer y escribir esta información hay que manejar flujos de información, y para esto Java provee los **streams** (**Figura 1**).

Figura 1. Stream de entrada y stream de salida.

Entonces, los streams representan flujos de datos o de información. Estos flujos podrán ser de entrada o de salida. Generalmente, los algoritmos para manejar los datos son los siguientes:

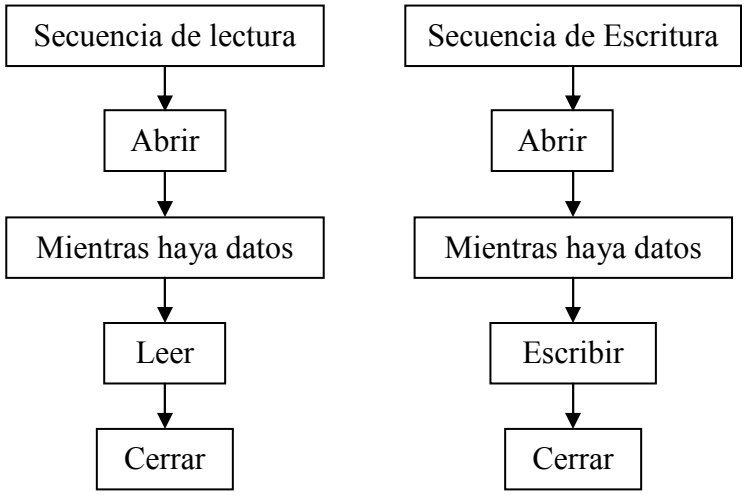


Figura 2. Secuencia de algoritmos de lectura y escritura.

El problema radica ahora en el dispositivo del cual estamos leyendo o escribiendo, es decir, si bien el algoritmo es similar para cualquier dispositivo, no es lo mismo leer o escribir datos del disco duro que de otra PC en una red.

Para que no tengamos que preocuparnos por los tipos de dispositivos, Java ofrece el paquete **java.io** (java input/output). Este paquete contiene un conjunto de clases que nos permitirán manejar cualquier flujo de entrada/salida. Las clases están divididas según dos categorías: **flujos de caracteres** (character stream) y **flujos de bytes** (bytes stream).

- **Bytes stream**: sirve para el manejo de entradas y salidas de bytes, y su uso lógicamente está orientado a la lectura y escritura de datos binarios. Las clases más usadas para el tratamiento de flujos de bytes son **ImputStream** y **OutputStream** (**Figura 3** y **Figura 4**).

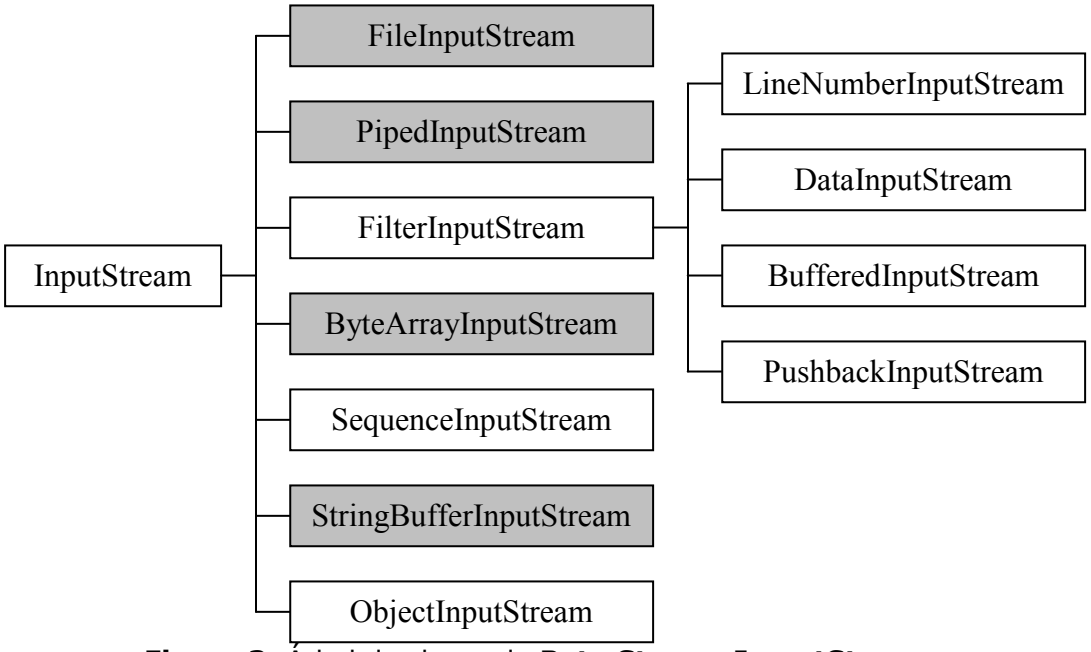


Figura 3. Árbol de clases de **Byte Stream ImputStream**.

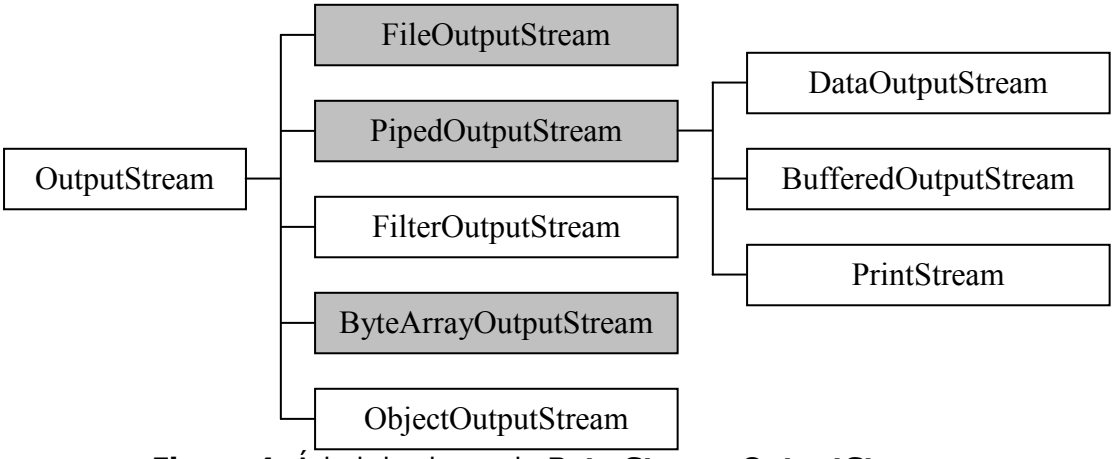


Figura 4. Árbol de clases de **Byte Stream OutputStream**.

- **Character streams**: sirven para el manejo de entradas y salidas de caracteres. Dichos flujos usan codificación Unicote y, por lo tanto, se pueden internacionalizar. Las dos clases para este caso son **Read** y **Write** (**Figura 5** y **Figura 6**).

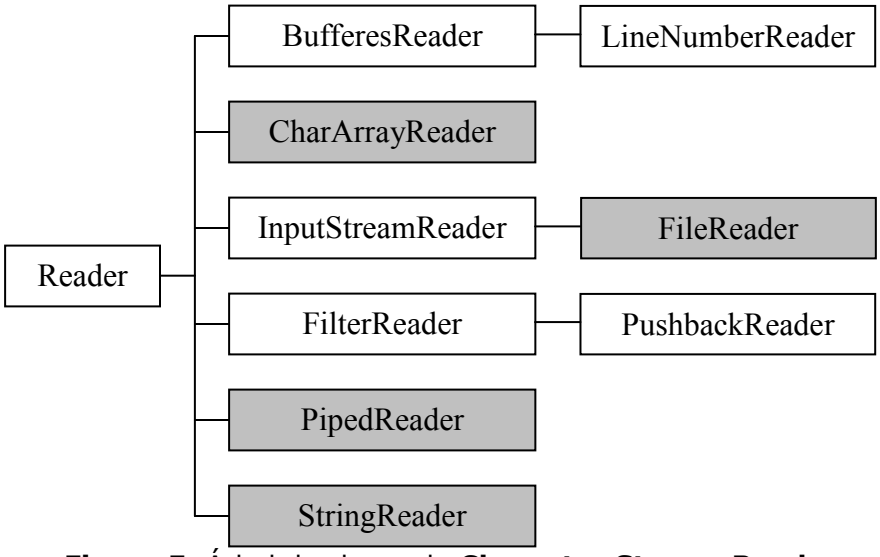


Figura 5. Árbol de clases de **Character Stream Read**.

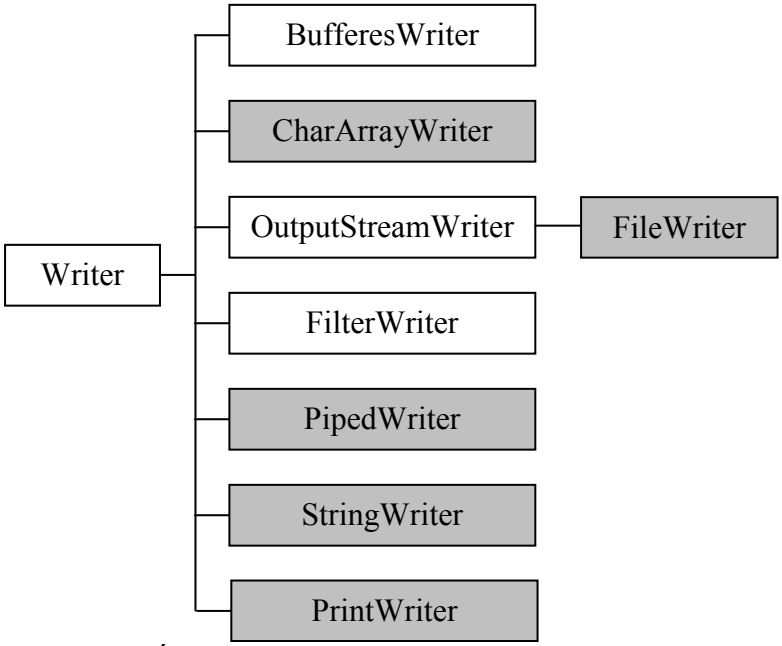


Figura 6. Árbol de clases de **Character Stream Write**.

8.2.- Lecturas y escrituras en archivos usando streams .

El caso más común en que necesitaremos usar streams será el de archivos. Para esto, encontramos las clases **FileOutputStream**, **FileInputStream**, **FileReader** y **FileWriter**. Comenzaremos por **FileOutputStream**, el cual, como su nombre lo indica, crea un flujo de datos de salida de un archivo.

```
package pruebas;
import java.io.*;
public class PruebaArchivoSalida
```

```
{
   public PruebaArchivoSalida()
   {
      // Declaramos la variable miArchivo del tipo FileOutputStream
      FileOutputStream miArchivo;
      // Declaramos el String para gravar en el archivo
      String dato = new String();
      try
      {
         /* Creamos el archivo a través del constructor
         de la clase FileOutputStream(String nombre) */
         miArchivo = new FileOutputStream("C:\\archivo.txt");
         // Grabamos 5 líneas
         for(int i = 0; i<5; i++)
         {
            /* Grabamos el número de línea. Agregamos al final la
            secuencia de escape \n para que no grabe todo seguido*/
            dato = "Línea (" +i +") \n";
            /* Recordemos que el método write espera un array de bytes, así
            que usamos el método getBytes() de String para obtener esto */
            miArchivo.write(dato.getBytes());
         }
         // Cerramos el archivo
         miArchivo.close();
      }
      catch(IOException ex)
      {
         /* Tanto el constructor como los métodos write y close lanzan
         una excepción del tipo IOException que debemos capturar */
         System.out.println("Error: " +ex.getMessage());
      }
   }
   public static void main(String argv[])
   {
      new PruebaArchivoSalida();
   }
}
```

Este programa no genera ninguna salida por pantalla, pero, si todo va bien, genera un archivo en el directorio raíz del disco C. Este archivo se abre con cualquier editor de textos. Aunque debemos tener en cuenta que algunos editores no soportan las secuencias de escape y las representan con algún otro carácter. En el archivo generado vemos:

```
Línea (0)
Línea (1)
Línea (2)
Línea (3)
Línea (4)
```

Ahora, aprovechando que tenemos un archivo generado, usaremos **FileInputStream** para abrirlo, leerlo y mostrarlo por pantalla.

```
package pruebas;
import java.io.*;
public class PruebaArchivoEntrada
{
   public PruebaArchivoEntrada()
   {
      // Declaramos la variable miArchivo del tipo FileInputStream
      FileInputStream miArchivo;
      // Creamos el array de byte para almacenar la información que leemos del archivo
      byte linea[] = new byte[55];
      try
      {
         /* Abrimos el archivo a través del constructor
         de la clase FileInputStream(String nombre) /*
         miArchivo = new FileInputStream("C:\\archivo.txt");
         miArchivo.read(linea);
         /* Debemos convertir el byte[] a String para que la información
         leída del archivo sea entendible para nosotros */
         System.out.println(new String(linea));
         // Cerramos el archivo
         miArchivo.close();
      }
      catch(IOException ex)
      {
         /* Tanto el constructor como los métodos read y close lanzan
         una excepción del tipo IOException que debemos capturar */
         System.out.println("Error: " +ex.getMessage());
      }
   }
   public static void main(String argv[])
   {
      new PruebaArchivoEntrada();
   }
}
```

Para simplificar el ejemplo, creamos el array de 55, ya que sabíamos de antemano que eso era lo que ocupaba el archivo de bytes. De esa forma se pudo leer el archivo completo en una sola lectura. Generalmente no sabemos qué tamaño ocupa el archivo, peor aún, el archivo va a ir cambiando de tamaño. Para manejar esto hay que hacer un programa similar al de escritura, es decir, usando un ciclo.

Por otra parte, el error capturado de esta manera, si lo hubiera, no nos permite especificarle al usuario si se provocó cuando quisimos abrir el archivo, grabarlo o cerrarlo. La forma correcta sería capturar la excepción cada vez que quisiéramos hace algo con el archivo, para poder informar mejor el error o manejarlo de alguna manera.

8.2.1.- FileReader y FileWriter.

Las clases anteriores, como vimos, eran stream de bytes. **FileReader** y **FileWriter**, en cambio, son stream de char, es decir que, tomando como referencia los ejemplos anteriores, cambiando los array de bytes por array de char y convirtiendo cada cadena a array char, vamos a obtener los mismos resultados.

8.3.- Objetos persistentes.

En algunas ocasiones necesitaremos no perder la instancia de un objeto cuando termine el programa, es decir, precisaremos guardar dicha información contenida en el objeto. Para esto se usan los objetos persistentes.

Para programar un objeto persistente debemos implementar la interfaz **Serializable** del paquete **java.io**. Simplemente declaramos la clase como:

```
public class ObjetoPersistente implements Serializable
```

Una vez hecho esto, debemos crear un método que se encargue de grabar el objeto a través del método **writeObject()** de **ObjectOutputStream**. Para ejemplificar, implementaremos el código de la clase ObjetoPersistente.

```
import java.io.*;
public class ObjetoPersistente implements Serializable
{
   String nombre;
   String apellido;
   int edad;
   public ObjetoPersistente(String n, String a, int e)
   {
      this.nombre = n;
      this.apellido = a;
      this.edad = e;
   }
   // Implementamos el método guardarObjeto
   public void guardarObjeto()
   {
      try{
         // Creamos el archivo
         FileOutputStream archivo = new FileOutputStream("C:\\objeto.ser");
         // Creamos el stream de salida del objeto
         ObjectOutputStream out = new ObjectOutputStream(archivo);
         // Grabamos el objeto
         out.writeObject(this);
         out.flush();
         // Cerramos el stream
         out.close;
         // Cerramos el archivo
         archivo.close();
      }
      catch(IOException ex)
      {
         System.out.println("Error al grabar objeto: " +ex.getMessage());
      }
```

```
    }
    public static void main(String argv[])
    {
        ObjetoPersistente op = new ObjetoPersistente("Miguel","Alvarado",22);
        op.guardarObjeto();
    }
}
```

Como resultado obtendremos el archivo objeto.ser con la información del objeto. Note en el ejemplo que el método **writeObject()** recibe **this** como parámetro, se deduce que se puede implementar el método fuera de la clase y enviarle el objeto seriablizable. En nuestro caso, la línea sería:

```
out.writeObject(op);
```

Es decir, en la instancia de ObjetoPersistente, para mostrar esto modificaremos el main para leer el objeto grabado. Antes de modificar el main, haremos la sobrecarga del método **toString()** para mostrar cómo quedó instanciado el objeto al final de la ejecución. Ahora sí, el main será:

```
public static void main(String argv[])
{
    try
    {
        FileInputStream archivo = new FileInputStream("C:\\abjeto.ser");
        ObjectInputStream in = new ObjectInputStream(archivo);
        /* Leemos el objeto, y como readObject devuelve un objero lo
        convertimos a ObjetoPersistente y lo referenciamos con op */
        ObjetoPersistente op = (ObjetoPersistente)in.readObject();
        in.close();
        archivo.close();
    }
    catch(Exception ex)
    {
        ex.printStackTrace();
    }
    System.out.println("Instanciado con: " +opLeido.toString());
}
```

La salida del programa será:

```
Instanciado con: Miguel Alvarado de 22 años de edad
```

9.- AWT.

9.1.- Interfaces gráficas de usuario.

Hoy en día, una de las cosas más importantes de los sistemas es la interfaz de usuario. Al usuario final no le interesan las características técnicas de nuestro software, pero sí, que sea fácil de usar. Para lograr una **GUIs** (Graphical User Interface, en español **interfaces gráficas de usuario**) vamos a usar **AWT** (**Abstract Windows Toolkit**).

Debido a que Java es multiplataforma, provee un conjunto de herramientas para poder construir aplicaciones gráficas multiplataforma. El problema, a veces, consiste en que cuando migramos a otra plataforma, posiblemente no veamos lo mismo en una plataforma que en otra. Por este motivo, hay que tener en cuenta que la versión de **AWT** que **SUN** lanzó al mercado fue desarrollada en sólo dos meses y representa, quizás, el punto más débil del lenguaje. Esta situación hace que construir una ventana simple se vuelva una tarea complicada y muy tediosa.

La biblioteca **AWT** contiene un conjunto completo de clases y métodos para dar soporte al uso de ventanas y a las **GUIs**. La estructura básica del AWT se basa en **Componentes** y **Contenedores**. La versión actual del **AWT** se puede resumir en los puntos que se exponen a continuación:

- Los Contenedores contienen Componentes, que son los controles básicos.
- No se usan posiciones fijas de los Componentes, sino que están situados a través de una disposición controlada (**layouts**).
- El común denominador de más bajo nivel se acerca al teclado, mouse y manejo de eventos.
- Alto nivel de abstracción respecto al entorno de ventanas en que se ejecute la aplicación.
- La arquitectura de la aplicación es dependiente del entorno de ventanas, en vez de tener un tamaño fijo.
- Depende bastante de la plataforma en que se ejecuta la aplicación.
- Carece de un formato de recursos. No se puede separar el código de lo que es propiamente interfaz.

9.2.- Componentes y Contenedores.

Una interfaz gráfica está construida en base a elementos gráficos básicos, los Componentes. Típicos ejemplos de estos Componentes son los botones, barras de desplazamiento, etiquetas, listas, cajas de selección o campos de texto. Los Componentes permiten al usuario interactuar con la aplicación y proporcionar información desde el programa al usuario sobre el estado del programa. En AWT, todos los Componentes de la interfaz de usuario son instancias de la clase **Component** o uno de sus subtipos.

Los Componentes no se encuentran aislados, sino agrupados dentro de Contenedores. Los Contenedores contienen y organizan la situación de los Componentes; además, los Contenedores son en sí mismos Componentes y como tales pueden ser situados dentro de otros Contenedores. También contienen el código necesario para el control de eventos, cambiar la forma del cursor o modificar el icono de la aplicación. En el AWT, todos los Contenedores son instancias de la clase **Container** o uno de sus subtipos.

9.3.- Clase Component.

Component es una clase abstracta que representa todo lo que tiene una posición, un tamaño, que puede ser pintado en pantalla y puede recibir eventos. No tiene constructores públicos, ni puede ser instanciada. Sin embargo, puede ser extendida para proporcionar una nueva característica incorporada a Java, conocida

como componentes **Lightweight**. Los Objetos derivados de dicha clase que se incluyen en **AWT** son los que aparecen a continuación:

- Button
- Canvas
- Checkbox
- Choice
- Container
    - Panel
    - Window
    - Dialog
    - Frame
- Label
- List
- Scrollbar
- TextComponent
    - TextArea
    - TextField

Sobre estos Componentes se podrían hacer más agrupaciones y quizá la más significativa es la que diferencia los Componentes según el tipo de entrada. Así habría Componentes con entrada de tipo no-textual como los botones de pulsación (**Button**), las listas (**List**), botones de marcación (**Checkbox**), botones de selección (**Choice**) y botones de comprobación (**CheckboxGroup**), Componentes de entrada y salida textual como los campos de texto (**TextField**), las áreas de texto (**TextArea**) y las etiquetas (**Label**), y otros Componentes, en donde se encontrarían algunos como las barras de desplazamiento (**Scrollbar**), zonas de dibujo (**Canvas**) e incluso los Contenedores (**Panel**, **Window**, **Dialog** y **Frame**), que también pueden considerarse como Componentes.

9.3.1.- Button.

La clase **Button** es una clase que produce un componente de tipo botón con un título. El constructor más utilizado es el que permite pasarle como parámetro una cadena, que será la que aparezca como título e identificador del botón en el interfaz de usuario. Así:

```
Button nombreBoton = new Button("título");
```

Pone al alcance del programador una serie de métodos entre los que destacan por su utilidad los siguientes:

Método	Descripción
addActionListener()	Añade un receptor de eventos de tipo Action producidos por el botón.
getLabel()	Devuelve la etiqueta o título del botón.
removeActionListener()	Elimina el receptor de eventos para que el botón deje de realizar acción alguna.
setLabel()	Fija el título o etiqueta visual del botón.

Tabla 1. Algunos métodos de la clase **Button**.

ActionListener es uno de los eventos de tipo semántico. Un evento de tipo **Action** se produce cuando el usuario pulsa sobre un objeto **Button**.

A continuación veremos un ejemplo, utilizando sólo un botón con una etiqueta que dice "Oprima aquí". Al oprimir dicho botón haremos que el programa muestre el número de veces que se ha hecho clic sobre el botón. El código es el siguiente:

```
import java.applet.*;
import java.awt.*;
```

```
import java.awt.event.*;
public class PruebaButton extends Applet implements ActionListener
{
    private Button boton;
    private int cuenta = 0;
    public void init()
    {
        boton = new Button("Oprima aquí");
        add(boton);
        boton.addActionListener(this);
    }
    public void actionPerformed(ActionEvent event)
    {
        cuenta++;
        repaint();
    }
    public void paint(Graphics g)
    {
        d.drawString("El número de veces que se a oprimido el botón es: " +cuenta, 10, 50);
    }
}
```

9.3.2.- Choice.

Un cuadro de opción es una lista de opciones similar a un menú. Puede elegirse un elemento de la lista haciendo clic sobre el mismo. Luego, el elemento seleccionado aparece como la única parte visible de la lista.

Para crear un cuadro de opción, veamos un ejemplo:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class PruebaChoice extends Applet implements ItemLisetner
{
    private Choice colorOpcion;
    public void init()
    {
        colorOpcion = new Choice();
        colorOpcion.add("Rojo");
        colorOpcion.add("Blanco");
        colorOpcion.add("Azul");
        add(colorOpcion);
        colorOpcion.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent e)
    {
        if(e.getSource() == colorOpcion)
        {
            String unaOpcion = e.getItem().toString();
```

```
        }
    }
}
```

9.3.3.- Checkbox.

Ésta es una manera en la que el usuario puede especificar una o más opciones al hacer clic en una casilla de verificación. También se pueden agrupar en **CheckboxGrup**, que son un grupo de **Checkbox** que comparten la característica de que sólo una de las casillas puede seleccionarse a la vez. Los grupos de opciones son útiles para algunos programas, por ejemplo para seleccionar color, como vemos en el siguiente código:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class PruebaCheckbox extends Applet implements ItemLisetner
{
    private CheckboxGroup c;
    private Checkbox rojo, blanco, azul;
    public void init()
    {
        c = new CheckboxGroup;
        rojo = new Checkbox("Rojo", c, false);
        add(rojo);
        rojo.addItemListener(this);
        blanco = new Checkbox("Blanco", c, true);
        add(blanco);
        blanco.addItemListener(this);
        azul = new Checkbox("Azul", c, false);
        add(azul);
        azul.addItemListener(this);

    }
    public void itemStateChanged(ItemEvent e)
    {
        if((e.getSource() == rojo) || (e.getSource() == blanco) || (e.getSource() == azul))
        {
            showStatus("Estado de Grupo = " + rojo.getState() + "\t" + blanco.getState()
                + "\t" + azul.getState());
        }
    }
}
```

9.3.4.- List.

Este elemento es una lista de cadenas de texto, de la cual pueden seleccionarse uno o más elementos. Se proporciona una barra de desplazamiento para avanzar hacia arriba o abajo. Su constructor es:

```
List lista = new List(3, false);
```

Aquí, el primer parámetro especifica que sólo habrá tres elementos visibles en

la lista a la vez, mientras que el segundo nos indica que sólo podremos seleccionar un elemento de la lista.

Ahora podemos ver un ejemplo:

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class PruebaList extends Applet implements ItemLisetner
{
    private List lista;;
    public void init()
    {
        lista = new List(3, false);
        lista.add("leche");
        lista.add("azúcar");
        lista.add("té");
        lista.add("café");
        add(lista);
        lista.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource() == lista)
        {
            String listaSeleccion = lista.getSelectedItem();
        }
    }
}
```

Si deseamos una lista de la que se pueda seleccionar más de un elemento, entonces el segundo parámetro del método constructor debe ser true.

9.4.- Clase Container.

La clase **Container** es una clase abstracta derivada de **Component**, que representa a cualquier componente que pueda contener otros componentes. Se trata, en esencia, de añadir a la clase **Component** la funcionalidad de adición, sustracción, recuperación, control y organización de otros Componentes.

Al igual que la clase **Component**, no dispone de constructores públicos y, por lo tanto, no se pueden instanciar objetos de la clase **Container**. Sin embargo, sí se puede extender para implementar los componentes **Lightweight**.

AWT proporciona varias clases de Contenedores:

- Panel
    - Applet
- ScrollPane
    - Window
    - Dialog
        - FileDialog
    - Frame

Aunque los que se pueden considerar como verdaderos contenedores son **Window**, **Frame**, **Dialog** y **Panel**, porque los demás son subtipos con algunas características determinadas y solamente útiles en circunstancias muy concretas.

9.4.1.- Window.

Es una superficie de pantalla de alto nivel, una ventana. Una instancia de la clase **Window** no puede estar enlazada o embebida en otro Contenedor.

Una instancia de esta clase no tiene ni título ni borde, así que es un poco difícil de justificar su uso para la construcción directa de un interfaz gráfico, porque es mucho más sencillo utilizar objetos de tipo **Frame** o **Dialog**. Dispone de varios métodos para alterar el tamaño y título de la ventana, o los cursores y barrar de menús.

9.4.2.- Frame.

Una aplicación que utiliza ventanas necesita crear explícitamente dichas ventanas en la pantalla. Esta área se conoce como **Frame** (marco). La manera más sencilla en que una aplicación puede crear un marco es heredar (o extender) de la clase de biblioteca **Frame**. Para crear un marco se hace así:

```
nombreFrame = new Frame("título");
```

Ahora veamos algunos métodos de esta clase:

```
// Cambiamos su tamaño a 250 píxeles de ancho y 200 de alto
nombreFrame.setSize(250, 200);
// Lo hacemos visible
nombreFrame.setVisible(true);
// Agregamos un botón al Frame
Button b1 = new Button("Botón");
nombreFrame.add(b1);
```

9.4.3.- Dialog.

Es una superficie de pantalla de alto nivel, ventana, con borde y título, que permite entradas al usuario. La clase **Dialog** extiende la clase **Window**, que extiende la clase **Container**, que extiende a la clase **Component**. Y su controlador de posicionamiento por defecto es el **BorderLayout**.

De los constructores proporcionados por esta clase, destaca el que permite que el diálogo sea o no modal. Todos los constructores requieren un parámetro **Frame** y, algunos de ellos, permiten la especificación de un parámetro booleano que indica si la ventana que abre el diálogo será modal o no. Si es modal, todas las entradas del usuario serán recogidas por esta ventana, bloqueando cualquier entrada que se pudiese producir sobre otros objetos presentes en la pantalla. Posteriormente, si no se ha especificado que el diálogo sea modal, se puede hace que adquiera esta característica invocando al método **setModal()**.

9.4.4.- Panel.

Un panel es una forma de agrupar varios componentes de manera conveniente. No tiene un contorno ni alguna otra forma de verse en pantalla, es como un contorno invisible. Por ejemplo, si deseamos que haya cientos de botones en la parte superior de la ventana y otros en la parte inferior, podemos crear paneles separados APRA cada uno de los grupos, y luego especificar que panel debe ir arriba y cual abajo. Su constructor es:

```
Panel p newPanel();
```

Luego podemos agregar componentes a ese panel:

```
Button b1 = new Button("Oprímame");
Button b2 = new Button("No, oprímame a mí");
p.add(b1);
p.add(b2);
```

10.- Applet.

10.1.- ¿Que son los applet?.

Son pequeños programas, comúnmente denominados componentes, ya que se van a embeber en otras aplicaciones, que se enlazan a una página Web. El enlace se logra a través de la etiqueta **APPLET** de **HTML**. Cuando el navegador que está levantando nuestra página encuentra esta etiqueta, entiende que debe ejecutar un applet, para ello levanta primero la maquina virtual y solicita los archivos necesario para la correcta ejecución del applet. Los problemas de esta implementación son los siguientes: la configuración de seguridad del navegador, los applet son pesados y en una ren lenda es posible que no se puedan ejecutar. Por lo tanto, cuando programemos applets deberemos tratar de que sean livianos y que no sean demasiado relevantes para que alguien que no desee ejecutarlos pueda seguir navegando en nuestra página sin problemas.

10.2.- El AppletViewer.

El applet asume que el código se está ejecutando desde dentro de un navegador. Para que no dependamos de un navegador, podemos hacer uso del **AppletViewer**, que tiene el aspecto de un pequeño navegador. Éste espera como argumento el nombre del fichero **HTML** que debe cargar, no se le puede pasar directamente un programa Java. Este fichero **HTML** debe contener una marca que especifica el código que cargará el AppletViewer.

Esta marca, como ya vimos, es la etiqueta de **html APPLET**:

```
<HTML>
<APPLET CODE=PruebaApplet.class WIDTH=300 HEIGHT=100>
</APPLET>
</HTML>
```

De esta manera el AppletViewer genera un espacio de navegación con un área gráfica donde se va a ejecutar nuestro applet PruebaApplet.class.

10.3.- Métodos.

Antes de entrar de lleno en el código, vamos a ver los métodos propios de la clase **Applet** y para qué sirven.

- **init()**: este método es llamado cada vez que la clase se carga por primera vez. El applet puede sobrecargar este método para fijar el tamaño del applet, cargar imágenes y sonidos necesarios para la ejecución del applet, y la asignación de valores a las variables globales a la clase que se utilice.
- **start()**: su función es hacer que el applet comience la ejecución. El método **start()** de la clase **Applet** no hace nada. Cada vez que la zona de visualización del applet queda expuesta, se llama a este método automáticamente, aunque podemos modificarlo para que el applet siga activo aun cuando no está expuesto a la visión.
- **stop()**: detiene la ejecución del applet. Se llama cuando el applet desaparece de la pantalla. Este método también hay que rescribirlo.
- **destroy()**: es llamado cuando ya no se va a usar más el applet, cuando se necesita que sean liberado todos los recursos dispuestos por el applet, por ejemplo, cuando se cierra el navegador. Como el sistema no sabe qué recursos usa el applet, entonces debemos rescribir este método para liberarlos de forma correcta.
- **paint(Graphics g)**: este método es llamado cada vez que el área de dibujo del applet necesita ser actualizada. La clase **Applet** dibuja un rectángulo gris en el área, y la clase derivada debe sobrecargar este método para dibujar lo que necesite. El sistema llama en forma automática a este método cada vez que la zona de visualización del applet cambia. Este cambio puede deberse a que la página se

actualiza, a que el applet queda tapado por otra ventana, queda fuera de la pantalla, etc. El método recibe como parámetro un objeto del tipo **Graphics** que delimita la zona que será pintada.

- **update(Graphics g)**: ésta es la función que realmente se llama cuando se necesita una actualización de la pantalla. La clase **Applet** simplemente limpia e área y llama al método **paint()**.
- **repaint**: llamando a este método se podrá forzar la actualización de un applet.

10.4.- Nuestro Primer applet.

Veamos un ejemplo:

```
package prueba;
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class PruebaApplet extends Applet
{
   GridBagLayout gridBagLayout1 = new GridBagLayout();
   TextField TxNumeroUno = new TextField();
   TextField TxNumeroDos = new TextField();
   Button BotonSumar = new Button();
   TextField TxResultado= new TextField();
   public PruebaApplet()
   {
   }
   // Inicializa el applet
   public void init()
   {
      // Acá ponemos el código para armar la ventana
      this.setLayout(gridBagLayout1);
      TxResultado.setEnabled(false);
      BotonSumar.setLabel("Sumar");
      BotonSumar.addActionListener(new java.awt.event.ActionListener()
      { public void actionPerformed(ActionEvent e)
         { BotonSumar_actionPerformed(e); }});
      this.add(TxNumeroUno, new GridBagConstraints(0, 0, 1, 1, 0.0, 0.0,
         GridBagConstraints.CENTER, GridBagConstraints.NONE,
            new Insets(20, 20, 1, 20), 50, 0));
      this.add(TxNumeroDos, new GridBagConstraints(0, 1, 1, 1, 0.0, 0.0,
         GridBagConstraints. CENTER, GridBagConstraints.NONE,
            new Insets(1, 20, 20, 20), 50, 0));
      this.add(BotonSumar, new GridBagConstraints(0, 2, 1, 1, 0.0, 0.0,
         GridBagConstraints. CENTER, GridBagConstraints.NONE,
            new Insets(0, 0, 0, 0), 0, 0));
      this.add(TxResultado, new GridBagConstraints(0, 3, 1, 1, 0.0, 0.0,
         GridBagConstraints. CENTER, GridBagConstraints.NONE,
            new Insets(20, 20, 20, 20), 50, 0));
   }
   // Acción del botón sumar
```

```
    void BotonSumar_actionPerformed(ActionEvent e)
    {
        int numeroUno = Integer.parseInt(TxNumeroUno.getText());
        int numeroDos = Integer.parseInt(TxNumeroDos.getText());
        int resultado = numeroUno + numeroDos;
        TxResultado.setText(new Integer(resultado).toString());
    }
}
```

Si intentamos ejecutarlo por consola, nos dará un error, ya que ésta ya no es una aplicación de consola, sino un applet. Y para ejecutarla, necesitamos embeberla en una página HTML. Así:

```
<html>
<head>
<title>
Página HTML de prueba
</title>
</head>
<body>
<h1>Ingrese dos números y presione el botón sumar</h1>.<br>
<applet
    codebase = "."
    code = "prueba.PruebaApplet.class"
    name = "PruebaApplet"
    width = "400"
    height = "300"
    hspace = "0"
    vspace = "0"
>
```

10.5.- Sonidos en los applet.

Los applet pueden incluir sonidos a través de la interfaz **AudioClip**. El archivo de sonido puede ser cargado desde una dirección URL a través del método **getAudioClip()** para su posterior ejecución, o a través del método **play()**, también desde una URL, con la diferencia que este método lo carga y lo ejecuta. Otros métodos importantes que provee esta interfaz son **loop()** para repetir el sonido una vez que termine, y **stop()** para detener la ejecución del sonido.