1 Introducción al lenguaje de programación Java

Como cualquier idioma humano, Java provee una manera de expresar conceptos. En caso de tener éxito, este medio de expresión será significativamente fácil y más flexible que las demás alternativas cuando los problemas se vuelven más grandes y complejos.

Java no puede ser considerado solamente como una colección de características (algunas de ellas no tienen mucho sentido) por lo que es importante que sólo se considere todo el conjunto en el aspecto del diseño y no simplemente como código de programación.

Breve historia de Java

Para apreciar el significado e importancia de Java, es muy importante conocer su lugar de origen y cuales fueron sus propósitos:

En Diciembre de 1990, Patrick Naughton, un empleado de la empresa Sun, reclutó a sus colegas James Gosling y Mike Sheridan para trabajar sobre un nuevo tema conocido como "El proyecto verde". Este a su vez estaba auspiciado por la compañía "Sun founder Bill Joy" y tenía como objetivo principal crear un lenguaje de programación accesible, fácil de aprender y de usar, que fuera universal, y que estuviera basado en un ambiente C++ ya que había mucha frustración por la complejidad y las limitaciones de los lenguajes de programación existentes.

En abril de 1991, el equipo decidió introducir sistemas de software con aplicaciones para consumidores smart como plataforma de lanzamiento para su proyecto. James Gosling escribió el compilador original y lo denominó "Oak", y con la ayuda de los otros miembros del equipo desarrollaron un decodificador que mas tarde se convertiría en lenguaje Java.

Para 1992, el equipo ya había desarrollado un sistema prototipo conocido como "*7", que era una especie de cruce entre un asistente digital personalizado y un mecanismo inteligente de control remoto.

Por su parte el presidente de la compañía Sun, Scott McNealy, se dio cuenta en forma muy oportuna y estableció el Proyecto Verde como una subsidiaria de Sun. De 1993 a 1994, el equipo de Naughton se lanzó en busca de nuevas oportunidades en el mercado, mismas que se fueron dando mediante el sistema operativo base. La incipiente subsidiaria fracasó en sus intentos de ganar una oferta con Time-Warner, sin embargo el equipo concluyó que el mercado para consumidores electrónicos smart y las cajas Set-Up en particular, no eran del todo eficaces. La subsidiaria Proyecto Verde fue amortizada por la compañía Sun a mediados del 94’.

Afortunadamente, el cese del Proyecto Verde coincidió con el nacimiento del fenómeno mundial Web. Al examinar las dinámicas de Internet, lo realizado por el ex equipo verde se adecuaba a este nuevo ambiente ya que cumplía con los mismos requerimientos de las set-top box OS que estaban diseñadas con un código de plataforma independiente pero sin dejar de ser pequeñas y confiables.

Patrick Naugthon procedió a la construcción del lenguaje de programación Java que se accionaba con un browser prototipo, más tarde se le fueron incorporando algunas mejoras y el browser Hot Java fue dado a conocer al mundo en 1995.

Con el paso del tiempo el Hot Java se convirtió en un concepto práctico dentro del leguaje Java y demostró que podría proporcionar una forma segura multiplataforma para que el código pueda ser bajado y corrido del Host del World Wide Web y que de otra forma no son seguros.

Una de las características más atractivas del Hot Java fue su soporte para los "applets", que son las partes del código Java que pueden ser cargadas mediante una red de trabajo para después ejecutarlo localmente y así lograr o alcanzar soluciones dinámicas en computación acordes al rápido crecimiento del ambiente Web.

Para dedicarse al desarrollo de productos basados en la tecnología Java, Sun formó la empresa Java Soft en enero de 1996, de esta forma de se dio continuidad al fortalecimiento del programa del lenguaje Java y así trabajar con terceras partes para crear aplicaciones, herramientas, sistemas de plataforma y servicios para aumentar las capacidades del lenguaje.

Durante ese mismo mes, Java Soft dio a conocer el Java Developmet Kit (JDK) 1.0, una rudimentaria colección de componentes básicos para ayudar a los usuarios de software a construir aplicaciones de Java. Dicha colección incluía el compilador Java, un visualizador de applets, un debugger prototipo y una máquina virtual Java(JVM), necesaria para correr programas basados en Java, también incluía paquetería básica de gráficos, sonido, animación y trabajo en red.

Asimismo el Netscape Comunications Inc, mostró las ventajas de Java y rápidamente se asoció con Java Soft para explotar su nueva tecnología. No pasó mucho tiempo antes de que Netscape Communications decidiera apoyar a los Java applets en Netscape Navigator 2.0. Este fue el factor clave que lanzó a Java a ser reconocido y famoso, y que a su vez forzó a otros vendedores para apoyar el soporte de applets en Java.

Como parte de su estrategia de crecimiento mundial y para favorecer la promoción de su nueva tecnología, Java Soft otorgó permisos a otras compañías para que pudieran tener acceso al código fuente de Java y al mismo tiempo mejorar sus navegadores , dicha licencia también les permitía crear herramientas de desarrollo para programación Java y los facultaba para acondicionar Máquinas Virtuales Java (JVM), a varios sistemas operativos.

Muy pronto las licencias o permisos contemplaban a prestigiadas firmas como IBM, Microsoft, Symantec, Silicon Graphics, Oracle, Toshiba y por supuesto Novell.

Desde su aparición, Java se ha ganado una impresionante cantidad de apoyo. Virtualmente cada vendedor importante de software ha obtenido autorización de Java y ahora ha sido incorporado en los principales sistemas operativos base de PC’s de escritorio hasta estaciones de trabajo UNIX.

Los applets Java (basados en JDK 1.02) son apoyados por los dos más populares navegadores web (Netscape Navigator 3.0 y Microsoft Internet Explorer 3.0). I.B.M./Lotus, Computer Asociates, Symantec, Informix, Oracle, Sybase y otras poderosas empresas de software están construyendo Software 100% puro JAVA, por ejemplo el Corel Office que actualmente está en versión Beta.

Un gran número de nuevas empresas ya están preparadas para recibir a la Ola Java o para ingresar a los Mercados de software basados en Java, en algunos casos como "Marimba´s Castanet" se han concebido desde un principio con bases de tecnología Java para Internet y han sido autorizados bajo licencia de Netscape para poner al corriente "netcast", un producto informativo para PC’s de escritorio.

Los nuevos proyectos de Java son co-patrocinados por cientos de millones de dólares en capital disponible de recursos tales como la Fundación Java, un fondo común de capital formado el verano pasado por 11 compañías, incluyendo Cisco Systems, IBM, Netscape y Oracle.

Los Colegios y Universidades alrededor del mundo están adoptando Java como un lenguaje universal y de enseñanza indispensable, hoy en día existen más de 150 libros en Java que se están imprimiendo en este momento.

En un reciente estudio se encontró que el 60% de los empresarios están usando Java y el 40% expresaron que Java representa la solución estratégica que estaban buscando para sus negocios..

Para darse una idea de la rápida aceptación que tiene Java en el mundo, tenemos el ejemplo de las conferencias "Java Soft Java One" en San Francisco, el primer Java One fue celebrado en abril de 1996 y atrajo a 5000 usuarios, un año después, en la segunda conferencia Java One albergó a 10,000 usuarios, asistentes. Java Soft estima que el número actual de usuarios Java llega a 400 mil y sigue creciendo. Java también está ganando aceptación en el área empresarial, se ha estimado que actualmente las compañías de hoy que cuentan con más de 5000 empleados, una tercera parte están usando Java.

2 Programación Orientada a Objetos

¿Qué es orientado a objetos?

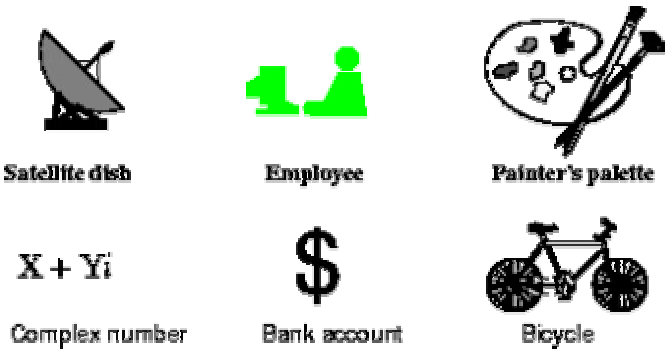
El término Orientado a Objetos significa que el software se organiza como una colección de objetos discretos que contienen tanto estructuras de datos como un comportamiento. Esto se opone a la programación convencional, en la cual las estructuras de datos y el comportamiento estan relacionadas de forma débil.

En el enfoque orientado a objetos existen aspectos que lo caracterizan:

- Identidad
- Clasificación
- Polimorfismo
- Herencia

Identidad

Quiere decir que los datos están cuantificados en entidades discretas y distinguibles llamadas objetos. Por ejemplo:



Los objetos pueden ser concretos o conceptuales.

Clasificación

Significa que los objetos con los mismos atributos (estructuras de datos), se aglutinan para formar una clase, como ejemplo de clase tenemos:

Clase Polígono

Atributos:

- *Vértice*
- *Color del borde*
- *Color del interior*

Métodos o Comportamientos:

- *Dibujar*
- *Borrar*
- *Mover*

Clase Automóvil

Atributos:

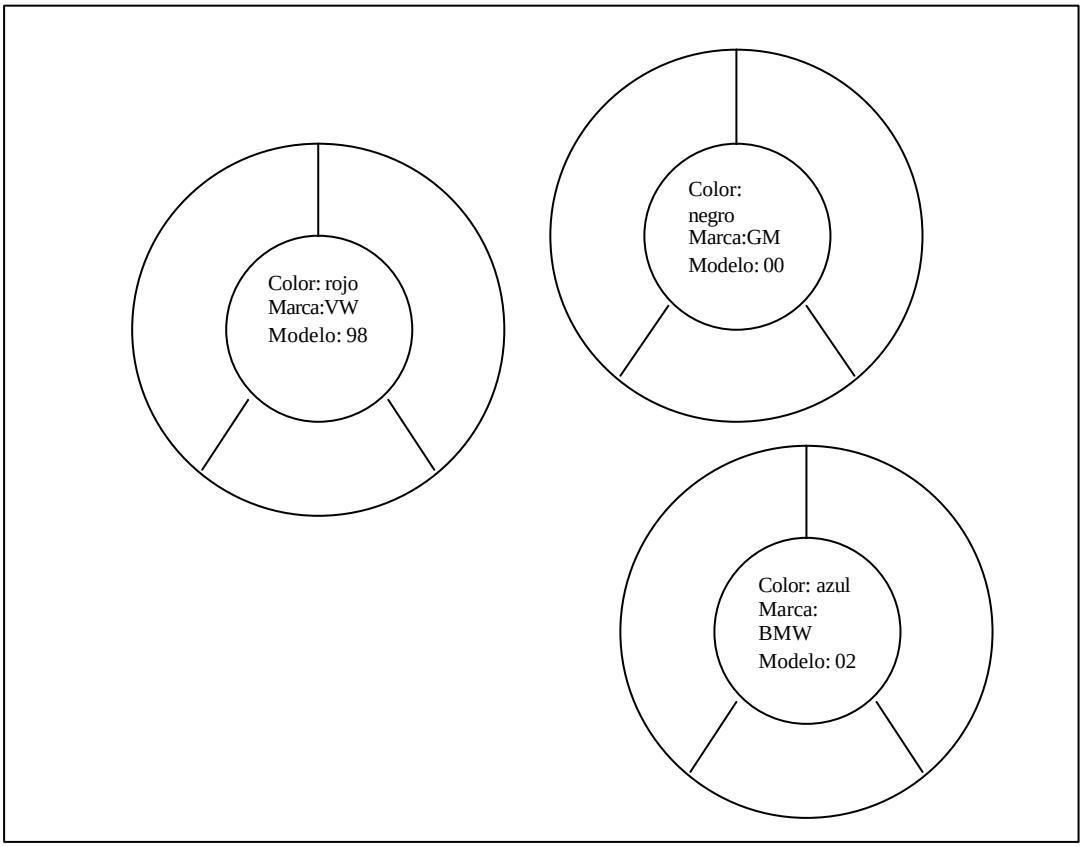
- *Marca*
- *Color*
- *Modelo, etc.*

Métodos o Comportamientos:

- *Arrancar*
- *Acelerar*
- *Frenar, etc.*

Toda clase describe un conjunto posiblemente infinito de objetos individuales. Se dice que cada objeto es una instancia de su clase. Todo objeto de la clase posee su propio valor para cada uno de los atributos, pero comparte el nombre de dichos atributos y los métodos o las operaciones de las demás instancias de la clase.

Clase Automóvil



Polimorfismo

Significa que una misma operación puede comportarse de modos distintos en distintas clases. La operación mover puede comportarse de modo distinto en las clases *Ventana* y *PiezaDeAjedrez*. Una operación es una acción o una transformación que se lleva a cabo o que se aplica a un objeto. *Justificar–a-la-derecha*, *Visualizar* y *Mover*, son ejemplos de operaciones.

Herencia

Es compartir atributos y operaciones entre clases tomando coma base una relación jerárquica. Todas las subclases heredan cada una de las propiedades de su superclase y añaden además sus propiedades exclusivas.

El Anaisis Orientado a Objetos, tiene yna metodología que consiste en construir un modelo de un dominio de aplicación. Esta aproximación se denomina tecnica del modelado de objetos (OMT), dicha metodología consta de 4 pasos:

1. Análisis
2. Diseño del Sistema
3. Diseño de Objetos
4. Implementación

Análisis

El modelo de análisis es una abstracción resumida y precisa de lo que debe hacer el sistema deseado y no de la forma en que se hará.

Diseño del Sistema

El diseñador de sistemas deberá decidir qué características de rendimiento hay que optimar. Seleccionando una estrategia para atacar el problema y efectuando unas reservas de recursos tentativas.

Diseño de objetos

El diseñador añade detalles al modelo de acuerdo con la estrategia establecida durante el diseño del sistema.

Implementación

Las clases de objetos y las relaciones desarrolladas durante su diseño se traducen finalmente a un lenguaje de programación concreto, a una base de datos o a una implementación de hardware.

Durante la implementación es importante respetar las buenas ideas de la ingeniería del software, de tal manera que el seguimiento hasta el diseño sea sencillo y de tal forma que el sistema implementado siga siendo flexible y extensible.

Algunas clases no forman parte del análisis sino que se presentan como parte del diseño o de la implementación. Por ejemplo, las estructuras de datos tales como: *árboles, tablas de dispersión* y *listas enlazadas* no suelen estar presentes en el mundo real.

Ejercicio:
Cree las siguientes clases con sus respectivos atributos y métodos, además cree dos instancias de dicha clase.

1. Edificio
2. Deporte
3. Mueble

3 Primer programa en Java

1.-Tipos de Datos

Existen tipos de datos primitivos. Se crean a partir de una variable al igual que en otros lenguajes de programación. Esta variable guarda el valor asignado a ella en la pila. Java determina automáticamente el valor de cada tipo de dato primitivo. Este tamaño no cambia independientemente de la plataforma donde se use el programa. Esta es una de las razones por las cuales los programas de Java son bastante portables.

Dato Primitivo	*Tamaño*	*Mínimo*	*Máximo*	*Tipo Wrapper*
boolean	-	-	-	Boolean
char	16 bit	Unicode 0	Unicode 2 -1	Character
byte	8 bit	-128	+127	Byte
short	16 bit	-2	+2 -1	Short
int	32 bit	-2	+2 -1	Integer
long	64 bit	-2	+2 -1	Long
float	32 bit	IEEE754	IEEE754	Float
double	64 bit	IEEE754	IEEE754	Double
void	-	-	-	Void

El tamaño del tipo **boolean** no está definido explícitamente, es el único donde se permite definirlo como literales ya sea **true** o **false**.

Los tipos primitivos también tienen “Wrapper” (fundas) clases que se forman con ellas. Eso significa que si se quiere hacer un objeto no primitivo para representar a ese tipo de dato primitivo, se utiliza el wrapper asociado.

Pe:

```
char c = 'x';
Character c =  new Character( c );
```

O también:

```
Character C = new Character( 'x' );
```

La utilidad de hacer esto se verá más adelante.

Números de alta precisión

Java cuenta con dos clases para el manejo de números de alta precisión aritmética: **BigInteger** y **BigDecimal**. Ambas clases cuentan con métodos que proveen operaciones que se realizan con datos primitivos.

Valores por default de datos primitivos

A continuación se presentan los valores por default de los datos primitivos.

Tipo Primitivo	*Default*
boolean	false
char	'\uoooo'(null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

2.-Métodos, argumentos y valores regresados.

Hasta ahora, el término "función" se utilizaba para describir una subrutina dentro de un programa. El término apropiado en Java es "método" ya que se utilizan como una forma de "hacer algo".

Los métodos determinan los mensajes que puede recibir un objeto. Las partes fundamentales de un método son: nombre, argumentos, tipo de valor que regresa y cuerpo o estructura. Esto es:

```
TipoQueRegresa NombreDelMetodo( /* argumentos */ ) {
/*
  cuerpo o estructura
*/
}
```

Los argumentos representan la información que se desea procesar en el método. El nombre sirve para identificar al método de otros.

Los métodos en Java pueden ser creados como parte de una clase. Deben de ser llamados a partir de algún objeto y ese objeto debe ser capaz de realizar el método al que llame. La sintaxis para llamar algún método desde un objeto es:

NombreObjeto.NombreDelMetodo(arg1,arg2,arg3)

Pe: Supóngase que se tiene un método f() que no tiene argumentos y regresa un entero. Además se cuenta con un objeto el cual puede llamar a dicho método.

```
int x = a.f();
```

Al acto de llamar un método se le conoce como "mandar un mensaje a un objeto".

Lista de Argumentos

La lista de argumentos especifica qué tipo de información se le pasa al método. Como se puede suponer, la información toma forma de objetos. Entonces lo que se debe de especificar en la lista de argumentos es el tipo de objeto y el nombre de cada uno.

Pe:

Considérese un método que toma un **String** como argumento. Esta sería su definición:

```
int storage (String  s ){
 return s.length()*2;
}
```

Este método te dice cuántos bytes se requieren para algún **String** en particular. El argumento es de tipo **String** y su nombre es s. Una vez dentro del método, s puede ser manipulado como cualquier objeto. Aquí, el método **lenght**() es llamado y regresa el número de caracteres de un **String**. En este caso, return es utilizado para indicar que el método terminó y para regresar un valor entero.

Se pueden regresar valores de cualquier tipo, pero si se desea que no se regrese nada se utilizará void simplemente.

Pe:

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nada() { return ; }
void nada2() { }
```

Hasta ahora, se ha visto que un programa es un conjunto de objetos con métodos que toman esos objetos como argumentos y mandan mensajes a esos argumentos.

Usando otros componentes

Cuando se desea utilizar una clase predefinida en algún programa, el compilador necesita saber dónde localizarla. Para este propósito se utiliza la palabra clave **import**. Import le indica al compilador que “traiga” todos los elementos de un paquete (“package”).

La mayor parte de las clases utilizadas serán las existentes en Java que vienen en el compilador. Para estos casos, se haría lo siguiente, por ejemplo:

```
import java.util.ArrayList;
```

Con esto se le indica el compilador que se desean utilizar los métodos de la clase **ArrayList**. Sin embargo, si se deseara utilizar las clases definidas en el subdirectorio **util**, el cual contiene muchas clases bastantes utilizadas, se haría lo siguiente:

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

Es más común utilizar la segunda forma ya que se evita el importar clases individualmente.

La palabra clave "STATIC"

Regularmente, cuando se crea una clase se describe el comportamiento de los objetos de esta clase y qué métodos podrá realizar cada uno de ellos. Pero si por ejemplo se desea tener algún método al cual se le pueda mandar llamar aunque no haya objetos creados, se utilizará la palabra "**static**". Entonces, se puede llamar e algún método "**static**" a partir de algún dato "**static**".

Pe:

Para hacer que algún tipo de dato sea **static** se haría lo siguiente:

```
Class StaticTest{
        static int i = 47;
}
```

Ahora, si se crearan dos objetos StaticTest, ambos objetos compartirían la misma variable i . Esto es:

```
StaticTest st1 =  new StaticTest();
StaticTest st2 =  new StaticTest();
```

Eso significa que **st1.i** y **st2.i** tendrían el mismo valor de 47 en memoria. Existe otra forma para referirse a una variable **static**. Puede hacerse como se hizo en el ejemplo anterior o simplemente:

```
StaticTest.i++;
```

En este caso el operador ++ incrementó la variable **i**.

Una lógica similar se aplica a métodos static. Se puede hacer referencia a ellos a través de objetos o con la sintaxis **NombreClase.metodo()**. Se define un método static de la siguiente manera:

```
class StaticFun{
        static void incr(){ StaticTest.i++ }
}
```

En este caso, para llamar al método **incr()** se puede hacer lo siguiente:

```
StaticFun sf = new StaticFun();
sf.incr();
```

o simplemente:

```
StaticFun.incr();
```

2.-Primer Programa en JAVA

A continuación se presenta un ejemplo que utiliza la clase **Date** de la librería estándar de Java. Nótese que para los comentarios simples se usará:

```
//HolaFecha.java

import java.util.*;

public class HolaFecha
{
  public static void main( String[ ] args)
  {
    System.out.println( "Hola, hoy es : ");
    System.out.println( new Date() );
   }//fin main
 }//fin clase
```

Al principio de cada programa, se debe de poner import para traer clases extras predefinidas en compilador. La clase **Date** se encuentra precisamente en la librería **util**. El método **println** está definido en **java.lang**, sin embargo no se especifica. Sirve para imprimir una línea nueva en la salida estándar (**System.out**).

Otra cosa a resaltar es que el nombre dela clase debe de ser el mismo que el del archivo, cuando se crea un programa stand-alone como este. Este tipo de clases deben de tener el método **main()** con la siguiente "firma":

```
public static void main( String[ ] args ){
```

La palabra **public** indica que el método es visible para todo el "mundo". El argumento de **main** es una serie de objetos de tipo **String**. En este caso no se utilizan argumentos para este método.

La línea que imprime la fecha es:

```
System.out.println( new Date() );
```

En este caso un objeto de tipo **Date** se crea sólo para que sea mostrado en pantalla por el método **println()**.

Compilando y ejecutando

Para compilar y ejecutar cualquier ejemplo o aplicación de Java primero se debe de contar con el **JDK**(Java Development Kit) de Sun que se puede conseguir de forma gratuita. Se puede Conseguir directamente en java.sun.com.

Una vez instalado el **JDK**, se tiene que configurar el path de la computadora para indicar donde encontrar los comandos **javac** y **java**. Para esto se puede usar el sysedit de Windows y agregar las siguientes líneas:

```
set PATH=C:\java\bin;%PATH%
set CLASSPATH=C:\java\jre\lib;c:\objetos;
```

donde el directorio **c:\java** sería el nombre del directorio del jdk y el CLASSPATH sirve para indicar en qué lugar se pueden localizar las clases definidas por el sistema (en este caso c:\java\jre\lib) o por el usuario (c:\ objetos).

Una vez hechas las modificaciones necesarias, lo siguiente sería compilar el programa. Para esto se ejecutará la siguiente línea desde la el prompt de MS-DOS o el de UNÍS:

```
javac HolaFecha.java
```

Si no genera errores el compilador, entonces se habrá creado exitosamente el archivo HolaFecha.class . Una vez creado éste, se podrá ejecutar el programa tecleando el siguiente comando:

```
java HolaFecha
```

Este es el proceso para correr y ejecutar cualquier programa .

3.- Comentarios y documentación

Existen dos tipos de comentarios en Java. El primero es el conocido por la mayoría que han programado en C.

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

También existe el comentario de C++ de una sola línea:

```
//este comentario es de una línea.
```

Documentación

Una característica importante en el lenguaje Java es que fue diseñado no sólo para escribir código de una manera sencilla, sino también para documentar cada programa de manera eficaz. Una de las cosas más molestas de documentar un programa es separar los comentarios o documentación del código en sí de un programa. Para estos casos Java cuenta con una herramienta llamada **javadoc**.

Esta herramienta genera código HTML a partir de etiquetas ("tags") especiales que se colocan en el código del programa fuente.

Sintaxis de Javadoc

Todos los comandos **Javadoc** se generan cuando se comenta entre /** y terminan con */ . Las etiquetas de **javadoc** comienzan con el símbolo **@** al principio de cada línea de comentario.

Existen tres tipos de comentarios: de clase, de variables y de métodos. Entonces, para comentar una clase, un método y una variable se haría lo siguiente:

```
/** Comentario de clase */
public class docTest {
        /** Comentario de variable */
        public int i;
        /** Comentario de método */
        public void f() { }
}
```

Es decir, el comentario correspondiente a cada una se coloca una línea antes de éstos. Una cosa importante es que **Javadoc** no generará código mas que para miembros **public** y **protected** (Sin embargo si se puede generar documentación para miembros **private** con la opción –**private**).

Etiquetas especiales de Javadoc

@version

Genera información de la versión.

@autor

Genera información del autor del código

@param

Describe los parámetros que puede recibir una clase o un método.

@return

Describe los valores que regresa algún método

@throws

Muestra las excepciones que puede generar algún método.

@deprecated

Se utiliza para indicar cuando algún método ya está obsoleto.

Pe:

Aquí está el ejemplo anterior con documentación

```
//HolaFecha.java

import java.util.*;

/** Primer Programa de ejemplo de este manual
  *Genera la fecha actual.
  *@author Sergio Jardón
  *@version  1.1
*/
public class HolaFecha
{
 /** Aplicación sin valores de entrada
   *@param args arreglo de String
   *@return No regresa valores
   *@exception No hay excepciones
 */
  public static void main( String[ ] args)
  {
    System.out.println( "Hola, hoy es : ");
    System.out.println( new Date() );
   }//fin main
 }//fin clase
```

Codificación

Para finalizar este capítulo existen varias cosas que se tienen que aclarar acerca de la notación existente en Java acerca de las clases y los métodos. Para la notación de clases se utilizará la siguiente sintaxis:

```
class TodosLosColoresDelArcoiris{ //...
```

es decir que cada palabra que conforme a la clase se iniciará con mayúscula sin dejar espacios en blanco. Para métodos y variables se utilizará una sintaxis parecida sólo que la primera letra será minúscula:

```
class TodosLosColoresDelArcoiris{ //...
        int unEnteroRepresentandoColors;
        void cambioDeColor( int nuevoColor ){ //...
         //...
        }
}
```

Ejercicios del capítulo.

1.- Crear una clase llamada HolaMundo siguiendo el ejemplo de la clase HolaFecha. Sólo se necesita el método main. Compila el programa con javac y ejecútelo con java.

2.- Escriba un programa que imprima tres argumentos tomados de la línea de entrada (del prompt) . Para esto necesitará el uso del método main.

3.- Observe lo que genera el uso del comando javadoc de la línea de comandos del ejemplo HolaFecha.

4.- Convierta el ejemplo de docTest en clase y ejecute el comando javadoc . Observe qué es lo que genera.

5.- Comente el primer ejercicio de este capítulo y genere documentación con javadoc como en el ejemplo de HolaFecha.

4 Control de Flujo

En Java los objetos y los datos se manipulan utilizando operadores. La mayoría de ellos deben de ser conocidos ya que son los que se utilizaron en C y C++. Sin embargo Java cuenta con alguno s propios permitiendo un mejor uso de éstos.

I.- Operadores

Un operador toma uno o más argumentos y produce un nuevo valor. Los operadores se manejan de la misma forma que en otros lenguajes de programación. Los operadores de Suma **(+)** , Resta **(-)** , Multiplicación **(*)** , División **(/)** y de asignación **(=)** funcionan de la manera tradicionalmente usada en otros lenguajes como C o C++.

Casi todos los operadores funcionan con datos primitivos. Las excepciones son '=' , **'=='** y **'!='** que también pueden ser utilizados por objeto. Además, la clase **String** puede utilizar los operadores **'+'** y **'+='**.

Precedencia

La precedencia indica cómo se evalúa una expresión cuando muchos operadores son utilizados. Java tiene sus reglas al respecto. La más fácil de recordar es que la multiplicación y la división se evalúan antes que la suma y la resta. Las demás reglas de precedencia son más difíciles de recordar, por lo que se aconseja utilizar paréntesis para evitar malas interpretaciones de las operaciones y hacer el orden de evaluación explícito.

Pe:

A = X + Y – 2 / 2 + Z;

Tiene un significado diferente que:

A = X + (Y –2) / (2 + Z);

Asignación

La asignación ocurre cuando ocupamos el operador **'='** . El compilador lo interpreta de la siguiente manera: " toma el valor de la derecha y asígnalo al de la izquierda " . El valor de la derecha (conocido como "**rvalue** ") , puede ser una constante, variable o expresión que produzca un valor; mientras que el valor dela izquierda ("**lvalue** "), debe de ser una variable.

Esto resulta bastante evidente para el manejo de datos primitivos (si decimos A = B, simplemente se copia el valor de B en A), sin embargo con objetos no ocurre lo mismo . Cuando se dice " C = D" significa que C y D apuntan hacia el mismo objeto.
Pe:

```
//Asignación.java
//Asignar con objetos es distinto.

class Numero
{
  int i;
}

public class Asignacion
{
  public static void main( String[] args )
  {
    Numero n1 = new Numero();
    Numero n2 = new Numero();
    n1.i = 9;
    n2.i = 47;

    System.out.println( "1: n1.i : " + n1.i  + " ,n2.i : " + n2.i );
    n1 = n2 ;
     System.out.println( "2: n1.i : " + n1.i  + " ,n2.i : " + n2.i );
    n1.i = 27;
     System.out.println( "3: n1.i : " + n1.i  + " ,n2.i : " + n2.i );
   }//fin main
  }//fin clase
```

La clase Número es simple, tiene dos instancias **n1** y **n2** creadas en **main** . El valor "**i**" es diferente para cada uno al principio . Sin embargo ,cuando se asigna **n1 = n2** , entonces ambos objetos apuntan al mismo valor (47 al principio y después a 27). La salida del programa sería :

```
1: n1.i : 9 , n2.i: 47
2: n1.i : 47 , n2.i: 47
3: n1.i : 27 , n2.i: 27
```

Este fenómeno es conocido como ***aliasing*** en Java . Pero si se quisiera mantener independientes a cada una de las variables se podría hacer los siguiente:

```
n1.i = n2.i ;
```

Pe:

```
//PasandoObjeto.java

class Letra
{
 char c;
}

public class PasandoObjeto
{
 static void f( Letra y )
  {
    y.c = 'z';
  }
 public static void main (String [ ] args )
 {
  Letra x = new Letra ();
  x.c = 'a';
  System.out.println( "1: x.c: " + x.c );
  f(x);
  System.out.println( "2: x.c: " + x.c );
 }//fin main
}//fin clase
```

En este caso el método **f()** cambia al objeto utilizado como argumento. La salida sería:

```
1: x.c : a
2: x.c: z
```

Como se ve el aliasing no es tan sencillo por lo que se le dedicará un Apéndice para su manejo y mejor comprensión.

Operadores Matemáticos

Los operadores matemáticos son en esencia iguales en la mayoría delos lenguajes de programación: suma **(+)**, resta **(-)** , multiplicación **(*)** , división **(/)** y módulo **(%)** . La división entera se trunca en lugar de redondear el resultado.

Java también utiliza la notación "corta" de asignación – operación que existe en C o C++. Esta se denota por el operador seguido del signo de igualdad .

Pe: Para sumar cuatro a la variable X se haría:

```
X + = 4;
```

El siguiente ejemplo muestra el manejo de operadores matemáticos:

```
//MathOps.java

import java.util.*;

public class MathOps {
 //esto evita la repetición de escritura
   static void prt( String s ) {
      System.out.println( s );
    }
 //esto imprime cadenas y enteros
    static void pInt( String s , int i ) {
        prt( s + " = " + f );
    }
 //esto imprime una cadena y un flotante
    static void pFlt( String s, float f ) {
         prt( s +  "  = " + f );
    }

 public static void main( String[ ] args ) {
   //genera números aleatoriamente a partir de la fecha
   Random rand = new Random();
   int i, j, k;
   // '%' limita el máximo valor a 99
   j = rand.nextInt() % 100;
   k= rand.nextInt() % 100;
   pInt( "j" , j ); pInt ("k", k);
   i = j + k ; pInt ("j + k", i);
   i = j – k; pInt ("j – k ",i);
   i = k / j; pInt ("k / j ",i);
    i = k * j; pInt ("k * j ",i);
   i = k % j; pInt ("k % j ",i);
    j % = k ; pInt( "j % = k ",j );
   //estos son de punto flotante
   float u, v, w;
   v = rand.nextFloat();
   w = rand.nextFloat();
   pFlt( "v", v); pFlt( "w", w);
   u = v + w; pFlt(" v + w ", u );
   u = v - w; pFlt(" v - w ", u );
    u = v * w; pFlt(" v * w ", u );
   u = v / w; pFlt(" v / w ", u );
   //Los siguientes funcionan también con char, byte, short, int, long y double
   u + = v ; pFlt( "u + = v ",u );
  u - = v ; pFlt( "u  - = v ",u );
   u / = v ; pFlt( "u / = v ",u );
 }//fin main
```

```
}//fin clase
```

La primera cosa a recalcar es que se utilizaron métodos para imprimir ya sea cadenas con enteros o cadenas con flotantes. En éstos se utilizó el método **System.out.println().** Para generar números al azar se creó un objeto **Random** y se llamó a alguno de sus métodos(ya sea **nextInt()** que genera enteros, **nextLong(), nextFloat()** o **nextDouble()**).

Operadores (+) y (-)

El operador **(-)** se puede utilizar para indicar cambio de signo:

```
X = - a;
```

El compilador también es capaz de interpretar la siguiente expresión:

```
x = a * - b;
```

pero para cuestiones prácticas es mejor la siguiente notación para evitar confusiones:

```
x = a * (- b );
```

Incremento y Decremento

Los operadores de incremento y decremento permiten la codificación de algún programa de manera más corta y legible. Existen dos versiones de cada uno de estos operadores: cuando se postincrementa o se preincrementa (también funciona con el decremento).

Cuando se preincrementa la variable se incrementa antes de ser utilizada, es decir, se opera y luego se asigna. Cuando se postincrementa ocurre lo contrario, primero se asigna y luego se opera.

Pe:

```
//Autoinc.java

public class AutoInc {
        public static void main( String [ ] args ) {
                int i = 1;
                prt( "i :  " + i );
                prt( "++i :  " + ++i ); //Preincremento
                prt( "i++ :  " + i++ ); //Postincremento

                prt( "i :  " + i );
                prt( "--i :  " + --i ); //Predecremento
                prt( "i-- :  " + i-- ); //Postincremento
                prt( "i :  " + i );
                }

        static void prt( String s ){
```

```
            System.out.println( s );
            }
}// fin clase
```

La salida del programa es:

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

Operadores Relacionales

Un operador relacional genera resultados Boléanos . Evalúan el resultado de alguna operación relacional y producen ya sea falso o verdadero dependiendo de qué es lo que se está evaluando. La lista de operadores relacionales es : mayor **(>)**, menor **(<),** mayor o igual **(>=),** menor o igual **(<=),** igual a **(==)** y diferente de **(!=).**

Los operadores == y **!=** funcionan con todos los tipos de datos y objetos pero a veces resultan confusos en su uso.

Pe:

```
//Equivalente.java

public class Equivalente {
        public static void main( String [ ] args ) {
                Integer n1 = new Integer( 47 );
                Integer n2 = new Integer( 47 );
                System.out.println( n1==n2 );
                System.out.println( n1!=n2 );
                }
}//fin clase
```

La expresión **System.out.println(n1==n2)** imprimirá el resultado de esta comparación, en este caso false porque los objetos no son iguales aunque tengan una instancia que sí. Pero ¿ que pasaría si se deseara comparar los contenidos de objetos ?. Aquí se muestra cómo:

Pe:

```
//IgualandoMetodo.java

public class IgualandoMetodo {
        public static void main( String [ ] args ) {
                Integer n1 = new Integer( 47 );
```

```
                Integer n2 = new Integer( 47 );
                System.out.println( n1.equals(n2) );
        }
}// fin clase
```

El resultado sería true como es de esperarse (se están comparando instancias).

Pe:

```
//IgualandoMetodo2.java
class Value {
        int i;
}
public class IgualandoMetodo2 {
        public static void main( String [ ] args ) {
                Value v1 = new Value();
                Value v2 = new Value();
                v1.i=v2.i=100;
                System.out.println( v1.equals(v2) );
        }
}//fin clase
```

En esta caso volvería a imprmir false porque se comparan el dato "**i**". El método **equals()** se puede "sobrecargar" pero se verá más adelante.

Operadores Lógicos

Los operadores lógicos **AND (&&), OR (||) y NOT (!)** producen un valor Boleano, ya sea **true** o **false** basado en el resultado lógico de sus operandos.

Pe:

```
//Bool.java

import java.util.*;

public class Bool {
        public static void main( String [ ] args ) {
                Random rand = new Random();
                int i = rand.nextInt() % 100;
                int  j = rand.nextInt() % 100;
                prt( " i = " + i ) ;
                prt( " j = " + j ) ;
                prt( " i  > j es = " + ( i > j ) );
                prt( " i  < j es = " + ( i < j ) );
                prt( " i  >= j es = " + ( i >= j ) );
                prt( " i  <= j es = " + ( i <= j ) );
                prt( " i  == j es = " + ( i == j ) );
                prt( " i  != j es = " + ( i != j ) );
```

```
        //Tratar a un entero como booleano es un error
        // ! prt("i && j es " + (i &&j));
        // ! prt("i ||  j es " + (i || j));
        // ! prt("!i  es " + (!i));
                prt( "( i < 10 ) && ( j < 10 ) es " + ( ( i< 10 ) && ( j < 10 )) );
                prt( "( i < 10 ) || ( j < 10 ) es " + ( ( i< 10 ) || ( j < 10 )) );
        }
static void prt( String s ) {
        System.out.println( s );
        }
}//fin clase
```

Estos operadores sólo se pueden aplicar a valores booleanos solamente. Sin embargo se puede utilizar una solución como la del ejemplo anterior. La salida sería la siguiente:

```
i = 85
j = 4
i > j es true
i < j es false
i >= j es true
i <= j es false
i == j es false
i != j es true
( i < 10) && ( j < 10 ) es false
( i <10 ) || ( j < 10 ) es true
```

"Short circuiting"

Cuando se trabaja cn comparaciones lógicas ocurre el fenómeno de "***short circuiting***". Esto significa que la expresión sólo se evalúa hasta que la parte falsa o verdadera de toda la expresión pueda ser determinada . El siguiente ejemplo ilustra lo que esto significa:

Pe:

```
        //ShortCircuit.java

        public class ShortCircuit {
                static boolean prueba1( int val ) {
                        System.out.println( "prueba1 ( " + val + " ) " ) ;
                        System.out.println( "resultado : " +  ( val <  1)  ) ;
                        return val < 1;
                }

                static boolean prueba2( int val ) {
                        System.out.println( "prueba2 ( " + val + " ) " ) ;
                        System.out.println( "resultado : " +  ( val < 2 )) ;
```

```
            return val < 2;
        }

        static boolean prueba3( int val ) {
            System.out.println( "prueba3 ( " + val + " ) " ) ;
            System.out.println( "resultado : " +  ( val < 3 )) ;
            return val < 3;
        }

        public static void main( String [ ] args ) {
            if(prueba1(0) && prueba2(2) && prueba3(2))
                System.out.println( "expresión true " );
            else
                System.out.println( "expresión False" );
        }
    }//fin clase
```

Aquí se esperaría que las tres expresiones del **if** de **main** se evaluaran, sin embargo la salida es:

```
prueba1(0)
resultado: true
prueba2(2)
resultado: false
expresión False
```

Como se ve no se evaluó la tercera expresión debido a que la segunda fue falsa.

Operaciones con Bits

Las operaciones con bits permiten manipular bit por bit un tipo de dato primitivo. Los operadores de bit efectúan álgebra de Boole.

Los operadores son: **AND &, OR | , XOR ^ y NOT** . Estos operadores se pueden combinar con el de asignación : &= , etc. (excepto en NOT).

Operadores SHIFT

Estos operadores sirven también para el manejo de bits. Sólo pueden ser utilizados con tipos de datos primitivos. El operador "**left-shift" (<<)** produce que el operando de la izquierda se recorra hacia la izquierda un "n" número de bits (insertando ceros de ser necesario). El operador " **right-shift" (>>)** tiene un efeco similar.

Si se utiliza este operador con datos **char**, **byte** o **short**, primeron serán promovidos a enteros.Java cuenta con el operador **r-shift unsigned (>>>).** El uso de este operador se muestra a continuación:

Pe:

```
//URShift.java

public class URShift {
        public static void main( String [ ] args ) {
                int i = -1;
                i >>>= 10;
                System.out.println( i );
                long l = -1 ;
                l >>>=10;
                System.out.println(l);
                short s = -1;
                s >>>= 10;
                System.out.println(s);
                byte b= -1;
                b >>>=10;
                System.out.println(b);
                b = -1;
                System.out.println(b>>>10);
        }
}//fin clase
```

El siguiente ejemplo muestra el uso de todos los operadores que manejanbits.

```
//BitManipulation.java

import java.util.*;

public class BitManipulation {
        public static void main( String [ ] args ) {
                Random rand= new Random();
                int i = rand.nextInt();
                int j = rand.nextInt();
                pBinInt("-1",-1);
                pBinInt("+1",+1);
                int maxpos = 2147483647;
                pBinInt("maxpos",maxpos);
                int maxneg= -2147483648;
                pBinInt("maxneg",maxneg);
                pBinInt("i",i);
                pBinInt("i ~ ", ~ i );
                pBinInt("-i",-i);
                pBinInt("j",j);
```

```
        pBinInt("i & j ", i & j);
        pBinInt("i | j ", i | j);
        pBinInt("i ^ j ", i ^ j);
        pBinInt("i << 5 ", i << 5);
        pBinInt("i >> 5 ", i >> 5);
        pBinInt(" (~ i ) >> 5 ", (~ i ) >> 5);
        pBInt( " i  >>> 5 ", i  >>> 5 );
        pBinInt(" (~ i ) >>> 5 ", (~ i ) >>> 5);

        long l = rand.nextLong();
        long m = rand.nextLong();
        pBinLong( " –1L ", -1L );
        pBinLong( " +1L ", +1L );
        long ll = 9223372036854775807L;
        pBinLong( "maxpos",ll);
        long lln = -9223372036854775808L;
        pBinLong( "minpos",lln);
        pBinLong("l",l);
        pBinLong(" ~ l  ", ~ l );
        pBinLong("-l",-l);
        pBinLong("m",m);
        pBinLong("l & m ", l & m);
        pBinLong("l | m ", l | m);
        pBinLong("l ^ m ", l ^ m);
        pBinLong("l << 5 ", l << 5);
        pBinLong("l >> 5 ", l >> 5);
        pBinLong(" (~ l ) >> 5 ", (~ l ) >> 5);
        pBinLong( " l  >>> 5 ", l  >>> 5 );
        pBinLong(" (~ l ) >>> 5 ", (~ l ) >>> 5);
}

static void pBinInt( String s, int i ) {
        System.out.println(
                S + ", int : " + I + " , binario : " );
        System.out.print( "    ");
        for( int j=31; j>=0 ; j- )
                if((( 1 << j ) & i ) != 0 )
                        System.out.print(" 1 " );
                else
                        System.out.print(" 0 ");
        System.out.println();
}

static void pBinLong( String s, long l){
        System.out.println(
                S + ", long : " + l  + " , binario : " );
        System.out.print( "   " );
```

```
                for( int i=63; i>=0; i--)
                        if(((1L << i) & 1) != 0)
                                System.out.print("1");
                        else
                                System.out.print("0");
                System.out.println();
        }
}//fin clase
```

Los métodos **pBinInt()** y **pBinInt** toman un entero y un long como argumetos respectivamente e imprimen la salida en formato binario. También se utilizó el método **System.out.print()** en lugar del utilizado anteriormente para que no realice el salto de línea.

Operador Ternario if else

El uso de este operador es menos frecuente porque tiene tres operandos. Es un operador porque produce un resultado. La expresión es de la siguiente forma:

Boolean-exp ? valor0 : valor 1

Si la expresión Boolean-exp resulta true, valor0 es evaluada y su resultado se transforma en el valor producido por el operador. Si es falsa valor1 es evaluada y será el valor a asignar.

Pe:

```
static int ternario(int i) {
        return i<10?i*100:i*10;
        }
```

Este código podría ser fácilmente escrito con na estructura if-else normal pero en este caso resulta más corto, sin embargo es más comprensible con la estructura if-else.

El operador coma

Este operador sólo se utiliza dentr de ciclos **for** que serán descritos más adelante.

Operador + de cadenas

Este operador tiene un uso especial en Java. Sirve para concatenar cadenas. Sin embargo debe de tenerse cuidado con su uso en métodos que impriman resultados:

```
int x = 0, y=1, z=2;
String  String="x,y,z";
System.out.println(sString+x+y+z);
```

En este caso el compilador tranformará x,y y z a sus respectivas representaciones **String** en lugar de sumarlas porque la expresión comienza con un dato **String**.Además todos el texto que se encuentre entre " " será tranformado en Strings.

Errores comunes en el uso de operadores

Uno de los errores más comunes dentro de la programación de cualquier lenguaje es querer evaluar una expresión**(==)** y hacer una asignación en lugar de eso**(=).** Si se hace esto en Java el compilador marcará un error debido a que espera un boolean en la expresión y no un entero. En otros lenguajes como C o C++ podría obtenerse un ciclo infinito sin dar ninguna información sobre posibles errores.

Operadores "cast".

La palabra "***cast***" se utliza en el sentido de moldear. Java cambia automáticamente algún tipo de dato al apropiado cuando sea necesario. Por ejemplo, si se asigna algún valor entero a uno de tipo punto flotante el compilador convertirá el entero a flotante. El " ***cast***" hace que este tipo de conversión sea explícita.

Para realizar algún cast, se pone el tipo al que se desea convertir entre paréntesis. Aquí hay un ejemplo:

```
void casts() {
        int i=200;
        long l=(long)i;
        longl2=(long)200;
        }
```

Como se puede observar el cast se puede realizar tanto en valores numéricos como variables. En este caso, la operación "**cast**" resulta realmente trivial porque el compilador la haría automáticamente, sin embargo para cuestiones de comprensión del código si resulta más explícito.

En C y C++ el casting podía resultar un verdadero problema. En Java casting es más flexible con la excepción de que si se desea convertir de algún tipo de dato que puede almacenar más información a otro que almacena menos, se corre el riesgo de perder información .En estos casos el compilador lanza mensajes de advertencia.

Java permite el uso del cast en cualquier tipo de dato primitivo, excepto para los bolean. Para las clases existen métodos especiales que se tratarán posteriormente.

Literales

Cuando se inserta un valor literal a un programa el compilador sabe regularmente que hacer con él. Sin embargo, algunas veces resulta ambiguo el tipo. Cuando esto pasa se debe indicar al compilador alguna información extra.

Pe:

```
//Literales.java

class Literals {
        char c=0xffff; // maximo carácter hexadecimal
        byte b=0x7f;   // maximo byte hexadecimal
        short s=0x7fff; // maximo short hexadecimal
        int i1=0x2f;  //Hexadecimal(minúsculas)
        int i2=0X2F; //Hexadecimal(mayúsculas)
        int i3=0177;   //Octal
        //Hex y Oct también trabajan con long
        long n1=200L ; //sufijo long
        long n2=200l;    //sufijo long
        long n3=200;
        float f1=1;
        float f2=1F; //sufijo float
        float f3=1f;  //sufijo flota
        float f4=1e-45f; //potencia 10
        float f5=1e+9f;  //sufijo float
        double d1=1d;  //sufijo double
        double d2=1D; //sufijo double
        double d3=47e47d; //potencia 10
        }//fin clase
```

Los hexadecimales (base 16) se denotan por 0x ó 0X seguidos del número en hexadecimal escrito en mayúculas o minúsculas. En este código se muestra los mayores valores en hexadecimal para **char**, **byte** y **short**, si se sobrepasan esos valores, el compilador marcará un mensaje de error.

La base octal se denota por un cero al principio y seguido del número octal.

Las letras F ó f, D ó d y L ó l denotan datos tipo float, double y long respectivamente. La notación exponencial se realiza con la letra “e” seguido del número.

Promoción

Cuando se realizan operaciones matemáticas o binarias en datos primitivos más chicos que **int** (**char**, **byte** o **short**), estos valores serán promovidos a **int** antes de efectuar la operación y el resultado será tipo **int**. Por lo que si se desea que el resultado sea del tipo original, se deberá utilizar un ***cast*** (con la advertencia de que podría perderse información ya que son tipos más chicos). En general, cuando se realizan operaciones, el tamaño del dato más grande determina de qué tipo será el resultado. (por ejemplo, un **float** por un **double** resultará un **double**).

Java no tiene “sizeof()”

Java no tiene este método debido a que el tamaño de los tipos de datos es el mismo sin importar en qué máquina o S.O se está ejecutando el programa. El programador no necesita preocuparse por la portabilidad a este nivel.

Revisión de Precedencia.

En la tabla se presentan los operadores de mayor precedencia a menor:

Tipo de Operador	Operadores
Unuario	+ - ++ –
Aritméticos	* / % + - << >>
Relacionales	> < >= <= == !=
Lógicos	&& ¦ & ^
Condicional	A > B ? X : Y
Asignación	=

Compendio de Operadores

A continuación se presenta un ejemplo con el uso de todos los operadores.

//AllOps.java

```
class AllOps {
void f(boolean b) {}
void boolTest(boolean x, boolean y) {
        f(x==y);
        f(x!=y);
        f(!y);
        x=x&&y;
        x=x||y;
        x=x&y;
        x=x|y;
        x=x^y;
        x&=y;
        x^=y;
        x|=y;
        }
void charTest(char x,char y) {
        x=(char)(x*y);
        x=(char)(x/y);
        x=(char)(x+y);
        x=(char)(x-y);
        x++;
        x–;
        x=(char)+y;
```

```
        x=(char)-y;
        f(x>y);
        f(x>=y);
        f(x<y);
        f(x<=y);
        f(x==y);
        f(x!=y);
        //x=(char)~y;
        x=(char)(x&y);
        x=(char)(x|y);
        x=(char)(x^y);
        x=(char)(x<<1);
        x=(char)(x>>1);
        x=(char)(x>>>1);
        x+=y;
        x-=y;
        x*=y;
        x/=y;
        x%=y;
        x<<=1;
        x>>=1;
        x>>>=1;
        x&=y;
        x^=y;
        x|=y;
        byte B=(byte)x;
        short s=(short)x;
        int i=(int)x;
        long l=(long)x;
        float f=(float)x;
        double d=(double)x;
        }
void byteTest(byte x,byte y){
        x=(byte)(x*y);
        x=(byte)(x/y);
        x=(byte)(x%y);
        x=(byte)(x+y);
        x=(byte)(x-y);
        x++;
        x--;
        x=(byte)+y;
        x=(byte)-y;
        f(x>y);
        f(x>=y);
        f(x<y);
        f(x<=y);
        f(x==y);
```

```
        f(x!=y);
        //x=(byte)~y;
        x=(byte)(x&y);
        x=(byte)(x|y);
        x=(byte)(x^y);
        x=(byte)(x<<1);
        x=(byte)(x>>1);
        x=(byte)(x>>>1);
        x+=y;
        x-=y;
        x*=y;
        x/=y;
        x%=y;
        x<<=1;
        x>>=1;
        x>>>=1;
        x&=y;
        x^=y;
        x|=y;
        char c=(char)x;
        short s=(short)x;
        int i=(int)x;
        long l=(long)x;
        float f=(float)x;
        double d=(double)x;
        }
void shortTest(short x,short y){
        x=(short)(x*y);
        x=(short)(x/y);
        x=(short)(x%y);
        x=(short)(x+y);
        x=(short)(x-y);
        x++;
        x--;
        x=(short)+y;
        x=(short)-y;
        f(x>y);
        f(x>=y);
        f(x<y);
        f(x<=y);
        f(x==y);
        f(x!=y);
        //x=(short)~y;
        x=(short)(x&y);
        x=(short)(x|y);
        x=(short)(x^y);
        x=(short)(x<<1);
```

```
        x=(short)(x>>1);
        x=(short)(x>>>1);
        x+=y;
        x-=y;
        x*=y;
        x/=y;
        x%=y;
        x<<=1;
        x>>=1;
        x>>>=1;
        x&=y;
        x^=y;
        x|=y;
        char c=(char)x;
        byte B=(byte)x;
        int i=(int)x;
        long l=(long)x;
        float f=(float)x;
        double d=(double)x;
        }

void intTest(int x,int y){
        x=(int)(x*y);
        x=(int)(x/y);
        x=(int)(x%y);
        x=(int)(x+y);
        x=(int)(x-y);
        x++;
        x–;
        x=(int)+y;
        x=(int)-y;
        f(x>y);
        f(x>=y);
        f(x<y);
        f(x<=y);
        f(x==y);
        f(x!=y);
        //x=(int)~y;
        x=(int)(x&y);
        x=(int)(x|y);
        x=(int)(x^y);
        x=(int)(x<<1);
        x=(int)(x>>1);
        x=(int)(x>>>1);
        x+=y;
        x-=y;
        x*=y;
```

```
        x/=y;
        x%=y;
        x<<=1;
        x>>=1;
        x>>>=1;
        x&=y;
        x^=y;
        x|=y;
        char c=(char)x;
        short s=(short)x;
        byte B=(byte)x;
        long l=(long)x;
        float f=(float)x;
        double d=(double)x;
        }
void longTest(long x,long y){
        x=(long)(x*y);
        x=(long)(x/y);
        x=(long)(x%y);
        x=(long)(x+y);
        x=(long)(x-y);
        x++;
        x–;
        x=(long)+y;
        x=(long)-y;
        f(x>y);
        f(x>=y);
        f(x<y);
        f(x<=y);
        f(x==y);
        f(x!=y);
        //x=(long)˜y;
        x=(long)(x&y);
        x=(long)(x|y);
        x=(long)(x^y);
        x=(long)(x<<1);
        x=(long)(x>>1);
        x=(long)(x>>>1);
        x+=y;
        x-=y;
        x*=y;
        x/=y;
        x%=y;
        x<<=1;
        x>>=1;
        x>>>=1;
        x&=y;
```

```
        x^=y;
        x|=y;
        char c=(char)x;
        short s=(short)x;
        int i=(int)x;
        byte B=(byte)x;
        float f=(float)x;
        double d=(double)x;
        }
void floatTest(float x,float y){
        x=(float)(x*y);
        x=(float)(x/y);
        x=(float)(x%y);
        x=(float)(x+y);
        x=(float)(x-y);
        x++;
        x--;
        x=(float)+y;
        x=(float)-y;
        f(x>y);
        f(x>=y);
        f(x<y);
        f(x<=y);
        f(x==y);
        f(x!=y);
        //x=(float)~y;
        //x=(float)(x&y);
        //x=(float)(x|y);
        //x=(float)(x^y);
        //x=(float)(x<<1);
        //x=(float)(x>>1);
        //x=(float)(x>>>1);
        x+=y;
        x-=y;
        x*=y;
        x/=y;
        x%=y;
        //x<<=1;
        //x>>=1;
        //x>>>=1;
        //x&=y;
        //x^=y;
        //x|=y;
        char c=(char)x;
        short s=(short)x;
        int i=(int)x;
        long l=(long)x;
```

```
        byte B=(byte)x;
        double d=(double)x;
        }
void doubleTest(double x,double y){
        x=(double)(x*y);
        x=(double)(x/y);
        x=(double)(x%y);
        x=(double)(x+y);
        x=(double)(x-y);
        x++;
        x–;
        x=(double)+y;
        x=(double)-y;
        f(x>y);
        f(x>=y);
        f(x<y);
        f(x<=y);
        f(x==y);
        f(x!=y);
        //x=(double)~y;
        //x=(x&y);
        //x=(x|y);
        //x=(x^y);
        //x=(x<<1);
        //x=(x>>1);
        //x=(x>>>1);
        x+=y;
        x-=y;
        x*=y;
        x/=y;
        x%=y;
        //x<<=1;
        //x>>=1;
        //x>>>=1;
        //x&=y;
        //x^=y;
        //x|=y;
        char c=(char)x;
        short s=(short)x;
        int i=(int)x;
        long l=(long)x;
        float f=(float)x;
        byte B=(byte)x;
        }
}
```

El siguiente ejemplo muestra un error de desbordamiento:

```
//Desborde.java

public class Desborde {
        public static void main(String [ ] args) {
                int big=0x7fffffff; //máximo valor entero
                prt("big= "+ big);
                int bigger=big*4;
                prt("bigger= "+bigger);
                }
        static void prt(String s){
                System.out.println(s);
                }
        }//fin clase
```

La salida es:

```
Big=2147483647
Bigger= -4
```

El compilador no enviará mensajes de error ni excepciones.

II .-Control de flujo

Java utiliza las estructuras de control de C o C++ por lo que si ha programado en estos lenguajes el uso de estas sentencias no deberá resultar complicado. Sin embargo Java no cuenta con el famoso "**goto**".

La sentencia If-Else

La sentencia **if- else** es probablemente la más básica forma de control de flujo. El **else** es opcional, entonces la sintaxis sería:

```
if(expresión booleana)
        sentencias
```

ó

```
if(expresión booleana)
        sentencias
else
        sentencias
```

La expresión booleana debe producir un resultado bolean. Las sentencias pueden ser simples o compuestas. En caso de ser compuestas deben encontarse entre { }. A continuación se presenta un ejemplo se una sentencia if-else.

```
//IfElse.java

public class IfElse {
        static int test(int testval,int target) {
                int resultado=0;
                if(testval>target)
                        resultado=+1;
                else if(testval<target)
                        resultado=-1;
                else
                        resultado=0;
                return resultado;
                }
        public static void main(String [ ] args) {
                System.out.println(test(10,5));
                System.out.println(test(5,10));
                System.out.println(test(5,5));
                }
        }//fin clase
```

Return

La palabra clave **return** tiene dos propósitos: especifica qué valor regresa algún método y provoca que ese valor sea regresado inmediatamente. El código anterior puede ser escrito de la siguiente forma:

```
//IfElse2.java

public class IfElse2 {
        static int test(int testval,int target) {
                int resultado=0;
                if(testval>target)
                        return +1;
                else if(testval<target)
                        return -1;
                else
                        return 0;
                }
        public static void main(String [ ] args) {
                System.out.println(test(10,5));
                System.out.println(test(5,10));
                System.out.println(test(5,5));
```

```
        }
}//fin clase
```

Iteración

Las sentencias **while**, **do-while** y **for** sirven para controlar iteraciones y se les clasifica regularmente como "***sentencias de iteración***". Las iteraciones se repetirán hasta cumplir con ciertas condiciones (generalmente hasta que alguna condición se vuelva falsa).

La forma del **while** es:

```
while(expresión booleana)
        Sentencias
```

La expresión booleana se evalua antes de comenzar el primer ciclo y así cada vez que comienza uno nuevo.

Pe:

```
//WhileTest.java

public class WhileTest {
        public static void main(String[ ] args) {
                double r=0;
                while(r<0.99d) {
                        r=Math.random();
                        System.out.println( r);
                }
        }
        }//fin clase
```

Esta clase utilize el método **random()** de la librería **Math**, que genera un valor **double** entre 0 y 1.

La forma de **do-while** es:

```
do
        sentencias
while(expresión booleana);
```

La única diferencia entre **while** y **do-while** es que **do-while** se ejecuta cuando menos una vez y hasta el final se verifica la condición.

La forma del ciclo **for** es:

```
for(valor inicial;Expresión booleana;"paso")
```

```
        sentencias
```

La inicialización, expresión o paso pueden estar vacíos. La expresión se evalua para cada iteración y tan pronto como sea falsa, se saldrá del ciclo **for** y continuará con la ejecución del código.

Pe:

```
//ListaCaracteres.java

public class ListaCaracteres {
        public static void main(String[ ] args) {
                for(char c=0;c<128;c++)
                        if(c!=26) //ANSI clear screen
                                System.out.println("valor:      "+(int)c+"caracter:
"+c);
        }
}//fin clase
```

Nótese que la variable "c" no se define hasta el punto donde es utilizada, es decir, dentro del ciclo "**for**". Esto significa que el alcance de esta variable sólo es dentro de este ciclo.

En otros lenguajes como C las variables debían ser declaradas al principio de cada bolque. En Java o C++ se pueden declarar variables donde se necesiten haciendo el código más compresensible.

En un ciclo **for** pueden ser definidas muchas variables, pero deben de ser del mismo tipo:

```
for(int i=0,j=1;i<10&&j!=11;i++,j++)
        sentencias
```

Sólo en el ciclo **for** se pueden definer muchas variables para utilizarlas en él.

Operador Coma

Ya se había mencinado anteriormente en este capítulo que el operador "," sólo tenía una utilidad: el control en una sentencia **for**. El código anterior mostró su utilidad: utilizar distintas variables dentro del control del ciclo **for**. A continuación se presenta otro ejemplo:

```
//OperadorComa.java

public class OperadorComa {
        public static void main(String [ ] args) {
                for(int i=1,j=i+10;i<5;i++,j=i*2) {
                        System.out.println("i= "+i+"j= "+j);
                        }
                }
```

```
}//fin clase
```

La salida es la siguiente:

```
i=1  j=11
i=2  j=4
i=3  j=6
i=4  j=8
```

Break y Continue

Dentro de cualquier sentencia de control de flujo se pueden utilizar ya sea el **break** o **continue** para ciertas siuaciones específicas. **Break** sale del ciclo sin ejecutar el resto de las sentencias del ciclo. **Continue** detiene la ejecución de la iteración actual y regresa al principio del ciclo con la siguiente iteración.

Pe:

```
//BreakAndContinue.java

public class BreakAndContinue {
        public static void main(String[ ] args) {
                for(int i=0;i<100;i++) {
                        if(i==74)
                                break; //sale del ciclo
                        if(i%9!=0)
                                continue;//siguiente iteración
                        System.out.println(i);
                }//fin for
                int i=0;
                //ciclo infinito
                while(true) {
                        i++;
                        int j=i*27;
                        if(j==1269)
                                break;//sale del ciclo
                        if(i%10!=0)
                                continue;
                        System.out.println(i);
                }//fin while
        }//fin main
}//fin clase
```

El famoso "goto"

La palabra clave "**goto**" ha estado presente en casi todo lenguaje de programación desde el principio. De hecho, el "**goto**" fue el génesis del control de los programas en lenguaje

compilador. Sin embargo, también ha sido el centro de muchos debates acerca de su verdadera utilidad dentro de los lenguajes ya que con su presencia se pierde la continuidad del flujo de un programa.

El verdadero problema no está en la utilización del "**goto**" sino en sobreutilizarlo, sin embargo, su uso puede ser la mejor forma de resolver algún problema en particular.

Aunque **goto** es una palabra reservada en Java, no se utiliza en el lenguaje porque no existe. Sin embargo, cuenta con "***etiquetas***" que pueden ser utilizadas en conjunto con "**break**" y "**continue** ". No es precisamente un salto dentro del programa , más bien es una forma de romper con una iteración. Una etiqueta se usa de la siguiente forma:

Etiqueta1:

El único lugar donde es útil una etiqueta es justo antes de comenzar una iteración:

```
Etiqueta1:
Iteración externa {
        Iteración interna {
                //...
                break; //1
                //...
                continue; //2
                //...
                continue  etiqueta1; //3
                //...
                break etiqueta1; //4
                }
        }
```

En el caso 1, el **break** rompe con la iteración interna y sigue con la externa. En el caso 2, el **continue** regresa al principio de la iteración interna. En el caso 3, **continue** rompe con la iteració n interna y sigue con la externa. En el caso 4, **break** rompe con ambas iteraciones.

Pe:

```
//LabeledFor.java

public class LabeledFor {
        public static void main(String[ ] args) {
                int i=0;
                outer:
                for(;true;) { //ciclo infinito
                        inner:
                        for(;i<10;i++) {
                                prt("i= "+i);
                                if(i==2) {
```

```
                        prt("continuar");
                        continue;
                        }
                if(i==3) {
                        prt("break");
                        i++;//se incrementa sino nunca sería
                            //incrementada
                        break;
                        }
                if(i==7) {
                        prt("continuar outer");
                        i++;//lo mismo
                        continue outer;
                        }
                if(i==8) {
                        prt("break outer");
                        break outer;
                        }
                for(int k=0;k<5;k++) {
                        if(k==3) {
                                prt("continuar inner");
                                continue inner;
                                }
                }//fin for de k
            }//fin for
        }//fin  for infinito
    }//fin main
    static void prt(String s) {
        System.out.println(s);
        }
}//fin clase
```

La salida es la siguiente:

```
        i=0
continuar inner
        i=1
continuar inner
        i=2
continue
        i=3
break
        i=4
continuar inner
        i=5
continuar inner
        i=6
```

```
continuar inner
        i=7
continuar inner
        i=8
break outer
```

Si no existiera el **break *outer*** no habría manera de salir del ciclo infinito ya que rompe con ambos. Otro ejemplo es el siguiente:

```
//LabeledWhile.java

public class LabeledWhile {
        public static void main(String[ ] args) {
                int i=0;
                outer:
                while(true) {
                        i++;
                        prt("i= "+i);
                        if(i==1) {
                                prt("continue");
                                continue;
                                }
                        if(i==3) {
                                prt("continue outer");
                                continue outer;
                                }
                        if(i==5) {
                                prt("break");
                                break;
                                }
                        if(i==7) {
                                prt("break outer");
                                break outer;
                                }
                        }
                }
        static void prt(String s) {
                System.out.println(s);
                }
}//fin clase
```

Las reglas generales para el **break** y **continue** serán las siguientes:

1.- Un continue regresa al principio del ciclo más interno y continúa con él.
2.- Un continue "etiquetado" va a la etiqueta y vuelve a entrar al ciclo que se encuentra después de la eqitueta.
3.- Un break "va hasta el final" de un ciclo.

4.- Un break "etiquetado" va hacia donde está la etiqueta y no entra el ciclo de nuevo.

Switch

Switch es normalmente clasificado como una estructura de selección. Su forma es:

```
switch(entero-selección) {
        case entero-valor1 : sentecias ; break;
        case entero-valor2 : sentencias; break;
        .
        .
        .
        case entero-valorn: sentencias;break;
        default: sentencias;
```

El entero de selección es una expresión que produce un valor entero. **Switch** compara el resultado con cada entero-valor. Si lo encuentra, ejecuta las correspondientes sentencias(simples o compuestas). Si no, ejecuta el valor default.

Se notará que en cada case existe un **break** correspondiente, lo que ocasiona que salga del cuerpo del **switch**. Esta es la forma "tradicional" de sintaxis de una estructura **switch**, ya que en realidad el **break** es opcional. Si no se encuentra en el código fuente, el programa se ejecutará hasta que encuentre algún **break**.

La estructura **switch** es ideal para implementaciones multiselección, pero el inconveniente es que requiere que el entero-selección sea un valor **int** o **char**.

Pe:

```
//VocalesAndConsonantes.java

public class VocalesAndConsonantes {
        public static void main(String[ ] args) {
                for(int i=0;i<100;i++) {
                        char c=(char)(Math.random()*26+'a');
                        System.out.print(c + ": ");
                        switch( c ) {
                                case 'a':
                                case 'e':
                                case 'i':
                                case 'o':
                                case 'u':
                                        System.out.println("vocal");
                                        break;
                                case 'y':
                                case 'w':
```

```
                                        System.out.println("A veces vocal");
                                        break;
                                default:
                                        System.out.println("consonante");
                                }
                        }
                }
        }//fin clase
```

Ya que **Math.random()** genera un valor entre 0 y 1, se necesita multiplicar por 26 para el rango de letras del alfabeto. Además produce un valor **double**,lo que significa que el valor **double** producido por este método necesita ser convertido a "**char**". Pero qué valor se toma cuando por ejemplo se tiene un resultado **double** 29.7, ¿se toma 30 ó 29?. La respuesta se puede ver en el siguiente ejemplo.

Pe:

```
//CastingNumbers.java

public class CastingNumbers {
        public static void main(String[ ] args) {
                double sobre=0.7,bajo=0.4;
                System.out.println("sobre: "+sobre);
                System.out.println("bajo: "+bajo);
                System.out.println("(int)sobre :"+(int)sobre);
                System.out.println("(int)bajo :"+(int)bajo);
                System.out.println("(char)('a'+sobre) :"+(char)('a'+sobre));
                System.out.println("(char)('a'+bajo) :"+(char)('a'+bajo));

                }
        }//fin clase
```

La respuesta es que siempre lo trunca, es decir, no redondea el resultado, así que en este ejemplo sería cero. El siguiente ejemplo nos muestra si **Math.random()** produce valores entre 1 y 0 cerrados([0,1]) o abiertos( ]0,1[ )

Pe:

```
//LimitesRandom.java

public class LimitesRandom {
        static void uso() {
        System.out.println("Uso: \n\t"+"LimitesRandom bajo\n\t"+
                                "LimitesRandom alto");
        System.exit(1);
        }

public static void main(String[ ] args) {
        if(args.length!=1) uso();
```

```
        if(args[0].equals("bajo")) {
                while(Math.random()!=0.0)
                        ;//sigue intentando
                System.out.println("Produjo 0.0!");
                }
        else if(args[0].equals("alto")) {
                while(Math.random()!=1.0)
                        ;//sigue intentando
                System.out.println("Produjo 1.0!");
                }
        else
                uso();
        }
}//fin clase
```

Para corer este programa se teclea en la línea de comandos:

java LimitesRandom bajo
ó
java LimitesRandom alto

El resultado es que **Math.random()** si produce un cero pero no el uno. ([0,1)).

Ejercicios

1.- Escriba un programa que imprima los números enteros del 1 al 100.

2.-Modifique el programa anterior para que se detenga en el valor 47 (break).

3.- Escriba una función que tome dos String como argumentos y los compare e imprima los resultados, utilice = = ,!= y el método equals. En el método main llame a sus funciones con varios String como ejemplos.

4.-Escriba un programa que genere 25 valores enteros al azar, Para cada uno, use sentencias if-then-else para clasificarlos en orden ascendente

5. Inicialización y limpieza

Muchos bugs de C ocurren cuando el programador olvida inicializar una variable. Esto ocurre especialmente cuando se trabaja con librerías que no se manejan frecuentemente y no se sabe como inicializar un componente o siquiera si se debe hacer. La limpieza es un problema especial porque es fácil olvidar liberar memoria que ya no se va a utilizar.

C++ introdujo el concepto de constructor; un método especial que se llama cada vez que un objeto es creado. Java adoptó el método del constructor, además, cuenta con un recolector de basura que libera memoria automáticamente cuando un objeto no se utiliza. Este capítulo está dedicado a la inicialización y liberación de memoria en Java.

I.-Inicialización garantizada con el constructor

Como se mencionó anteriormente, Java cuenta con un método especial para crear un objeto cualquiera llamado "constructor". Si una clase tiene un constructor, Java llama automáticamente al constructor cuando un objeto es creado. Entonces realmente la inicialización está garantizada.

Una cosa importante, es que el constructor tiene el mismo nombre que la clase. Así tiene más sentido una inicialización automática.

Pe:

```
//SimpleConstructor.java

class Rock {
        Rock() { //Este es el constructor
                System.out.println("Creando un constructor");
                }
        }
public class SimpleConstructor {
        public static void main(String[ ] args) {
                for(int i=0;i<10;i++)
                        new Rock();
        }
}//fin clase
```

Nótese que el nombre del constructor es exactamente el mismo que el de la clase(mayúsculas y minúsculas).

Como cualquier método , el constructor puede tener argumentos para especificar cómo va a ser creado un objeto.

Pe:

```
//SimpleConstructor2.java

class Rock2 {
        Rock2(int i) {
                System.out.println("Creando Rock numero: "+i);
                }
        }//fin clase Rock2

public class SimpleConstructor2 {
        public static void main(String[ ] args) {
                for(int i=0;i<10;i++)
                        new Rock2(i);
                }
        }//fin clase SimpleConstructor2
```

Los argumentos del constructor son una vía para proveer parámetros para la inicialización de un objeto. Por ejemplo, la clase Arbol tiene un constructor tiene un constructor que toma un entero como argumento para denotar la altura de ese árbol, para crear el objeto Arbol se haría lo siguiente:

```
Arbol a = new Arbol(12) ;//12 m de altura
```

Si el único constructor existente es Arbol(int), el compilador no dejará crear un objeto Arbol de otra forma.

Los constructores son métodos especiales porque no regresan valores, lo cual es totalmente distinto a algún método que es de tipo void

II.-Sobrecarga de Métodos

Varios lenguajes de programación (C en particular) requieren que cada función tenga un identificador único(nombres distintos). Por lo que no puedes tener una función llamada print() para imprimir enteros y la misma función print() para imprimir float.

En Java el constructor es un método bastante versátil ya que permite crear objetos de distintas formas, sin forzar a una sola. Por ejemplo, si necesitamos un constructor que no tenga argumentos(el default) y otro que tome un String como argumento. Ambos son constructores, entonces deben tener el nombre de la clase. Pero , en este caso, se sobrecarga el constructor, ya que es esencial tener el mismo nombre para estos dos métodos aunque tengan distintos argumentos. Además Java permite la sobrecarga de métodos como se muestra en el siguiente ejemplo.

Pe:

```
//Sobrecarga.java

import java.util.*;
class Arbol {
        int altura;
        Arbol() {
                prt("Plantando una semilla");
                altura=0;
                }
        Arbol(int i) {
                prt("Creando un arbol que es "+i+"m de altura");
                altura=i;
                }
        void info() {
                prt("El arbol es "+altura+"m de alto");
                }
        void info(String s) {
                prt(s+":Arbol es "+altura+"m de alto");
                }
        static void prt(String s) {
                System.out.println(s);
        }
}

public class Sobrecarga {
        public static void main(String[ ] args) {
                for(int i=0;i<5;i++) {
                        Arbol a=new Arbol(i);
                        a.info();
                        a.info("Sobrecargando metodo");
                        }
        //Sobrecarga de constructor
        new Arbol();
        }
}//fin clase
```

En este caso, Arbl puede ser sembrado con o sin argumento. Para permitir esto, existen dos constructores, el constructor default y otro creado para que tome un valor entero como argumento.

Además, el método info() es llamado de dos formas.

Distinguiendo Métodos sobrecargados

Si los métodos tienen el mismo nombre,¿Cómo distingue el compilador cuál es el método que se desea utilizar?. La respuesta es simple: cada método debe tener una lista de argumentos única.

Además esto tiene mucho sentido para el programador ya que es la única formaen que puede distinguir entre ellos.

Pe:

```
//SobrecargandoOrden.java

public class SobrecargandoOrden {
        static void print(String s,int i) {
                System.out.println("String: "+s+",int: "+i);
                }
        static void print(int I,String s) {
                System.out.println("int: "+i+",String: "+s);
                }
        public static void main(String[ ] args) {
                print("String first",11);
                print(99,"Int first");
                }
        }//fin clase
```

Los dos métodos print() tienen argumentos idénticos, pero el orden de éstos es distinto, lo cual hace la diferencia.

Sobrecargando con Primitivas

Un dato primitivo puede ser automáticamente promovido de un tipo más pequeño a uno más grande y esto puede resultar confuso con la sobrecarga de métodos:

```
//SobrecargaPrimitiva.java

public class SobrecargaPrimitiva {
        //booleano no puede ser convertido automáticamente
        static void print(String s) {
                System.out.println(s);
                }
        void f1(char x) { prt("f1(char)"); }
        void f1(byte x) { prt("f1(byte)"); }
        void f1(short x) { prt("f1(short)"); }
        void f1(int x) { prt("f1(int)"); }
        void f1(long x) { prt("f1(long)"); }
        void f1(float x) { prt("f1(float)"); }
```

```
void f1(double x) { prt("f1(double)"); }

void f2(byte x) { prt("f2(byte)"); }
void f2(short x) { prt("f2(short)"); }
void f2(int x) { prt("f2(int)"); }
void f2(long x) { prt("f2(long)"); }
void f2(float x) { prt("f2(float)"); }
void f2(double x) { prt("f2(double)"); }

void f3(short x) { prt("f3(short)"); }
void f3(int x) { prt("f3(int)"); }
void f3(long x) { prt("f3(long)"); }
void f3(float x) { prt("f3(float)"); }
void f3(double x) { prt("f3(double)"); }

void f4(int x) { prt("f4(int)"); }
void f4(long x) { prt("f4(long)"); }
void f4(float x) { prt("f4(float)"); }
void f4(double x) { prt("f4(double)"); }

void f5(long x) { prt("f5(long)"); }
void f5(float x) { prt("f5(float)"); }
void f5(double x) { prt("f5(double)"); }

void f6(float x) { prt("f6(float)"); }
void f6(double x) { prt("f6(double)"); }

void f7(double x) { prt("f7(double)"); }

void testConstVal() {
        prt(" Probando con 5 " );
        f1(5) ;f2(5) ;f3(5) ;f4(5) ;f5(5) ;f6(5);f7(5) ;
        }

void testChar() {
        char x='x' ;
        prt("Argumento char: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        }

void testByte() {
        byte x=0;
        prt("argumento byte: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        }

void testShort() {
```

```
        short x=0;
        prt("argumento short: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        }
void testInt() {
        int x=0;
        prt("argumento int: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        }

void testLong() {
        long x=0;
        prt("argumento long: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        }

void testFloat() {
        float x=0;
        prt("argumento float: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        }

void testDouble() {
        double x=0;
        prt("argumento double: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x);
        }

pulic static void main(String[ ] args) {
        SobrecargaPrimitiva p=new SobrecargaPrimitiva();
        p.testConstVal();
        p.testChar();
        p.testByte();
        p.testShort();
        p.testInt();
        p.testLomg();
        p.testFloat();
        p.testDouble();
        }
}//fin clase
```

III.- Constructores default

Como se mencionó anteriormente, un constructor por default es aquel que no tiene argumentos. Si se utiliza una clase que no tiene constructores, el compilador creará automáticamente uno para tal clase. Por ejemplo:

```
//ConstructorDeafault.java

class Bird {
        int i;
        }

public class ConstructorDefault {

        public static void main( String[] args ) {

                        Bird nc= new Bird(); //default !!
                        }
        }
}//Fin clase
```

La línea

```
new Bird();
```

crea un nuevo objeto y llama al constructor por default, aunque no haya sido declarado uno previamente. Sin embargo, si se definen nuevos constructores, el compilador no los reconocerá automáticamente:

```
class Bush {
        Bush (int i ) {}
        Bush ( double d ) {}
        }
```

y si se dice:

```
new Bush() ;
```

el compilador no encontrará el constructor que corresponda al nuevo objeto. Es como si el compilador dijera: " Si no tienes un constructor, yo hago uno para ti. Si ya hiciste un constructor, entonces te dejo el trabajo a ti ".

La palabra clave THIS.

Supóngase que se tienen dos objetos del mismo tipo llamados " a " y " b ", entonces resulta interesante cómo se puede llamar a un método f() para ambos objetos:

```
class Banana { void f(int i) { /*...*/ } }
Banana a = new Banana() , b= new Banana();
a.f(1);
b.f(2);
```

Si solo hay un método llamado f() , cómo saber si el método ha sido llamado para el objeto "a" o "b".?

En estos casos, el compilador manda un argumento al método de manera secreta, y es precisamente la referencia al objeto. Ahora si se necesita hacer referencia dentro del método al objeto actual se necesita utilizar la palabra reservada this. La palabra reservada this – que debe ser utilizada exclusivamente dentro de algún método – produce una referencia al objeto que llamó al método. Por ejemplo:

```
//Leaf.java
//ejemplo de la palabra reservada  this

    public class Leaf {
        int i=0;
        Leaf incremento() {
            i++;
            return this;
            }
        void print() {
            System.out.println("i= " +i );
            }
        public static void main( String[] args ) {
            Leaf x = new Leaf();
            x.incremento().incremento().incremento().print();
            }
    }//fin clase
```

Ya que incremento() regresa la referencia al objeto actual con la ayuda de this, se pueden efectuar múltiples operaciones al mismo objeto.

Llamando constructores desde constructores

Cuando se escriben varios constructores para una clase, hay veces que se necesita llamar a un determinado constructor en vez de otro para evitar la repetición de código. Para esto se puede utilizar this.

Normalmente, cuando se utiliza this, se utiliza en el sentido del "objeto actual" o "referencia al objeto actual". Sin embargo, en un constructor, la palabra this toma un sentido diferente: hace que el constructor corresponda al que tiene la lista correcta de argumentos. Por ejemplo:

```
//Flores.java
//Llamando constructores con this.

        public class Flores {
                int cuentaPetalos=0;
                String s= new String("null");

                Flores(int petalos) {
                        cuenraPetalos=petalos;
                        System.out.println("Constructor con argumento int
,petalos= "+cuentaPetalos);
                        }

                Flores(String ss) {
                        System.out.println("Constructor con argumento String
,s= "+ss);
                        s=ss;
                        }

                Flores(String s, int petalos) {
                        this(petalos);
                        // ! this(s); //No se pueden llamar dos!
                        this.s=s; //otro uso de this
                        System.out.println("String & int args");
                        }

                Flores() {
                        this("Hola",47);
                        System.out.println("Constructor por default sin args");
                        }

                void print() {
                        System.out.println("cuentaPetalos= "+cuentaPetalos+"s =
" +s);

                public static void main(String[] args) {
                        Flores x= new Flores();
                        x.print();
                        }
        }//fin clase
```

El constructor Flores(String s, int petalos) muestra que, aunque se puede llamar un constructor usando this, no se puede llamar dos. Además, el constructor debe aparecer primero en el código , sino el compilador mandará un mensaje de error.

Este ejemplo también muestra otro uso de la palabra reservada this. El nombre del argumento “s” y el nombre del dato “s” son el mismo, existe ambiguidad. Se puede resolver utilizando this.s para hacer referencia al miembro dato. Esta forma de utilizar this es muy común.

El significado de static.

Un método static funciona de manera similar a una función global de C. Con un método static no existen mensajes al objeto, ya que no se necesitan. Esto ha generado una controversia de su presencia en un lenguaje orientado a objetos. Sin embargo, su correcta utilización puede llegar a solucionar problemas dentro de un programa de manera que se respete el principio de la OPP.

IV.-Limpieza: finalización y colecta de basura.

Los programadores en general conocen la importancia de la inicialización, pero muchas veces olvidan la importancia de limpiar. Java cuenta con su propio recolector de basura para recolectar objetos que no se utilizan.

Para casos especiales en los que no se utiliza la palabra reservada “new” , sin la cual el recolector de basura de Java sería incapaz de saber que cierta memoria está siendo utilizada, el lenguaje cuenta con un método llamado finalize(). Funciona de la siguiente manera: si se pone en el código de una clase el método finalize() , al momento de que el compilador realiza la limpieza, se liberará memoria que el programador ya no va a utilizar.

Hay que señalar que no es lo mismo que un destructor para C++, ya que el destructor es una función utilizada para destruir objetos. Java no cuenta con algo similar, así que si se necesita para algún programa en específico (como por ejemplo limpiar una imagen de la pantalla) , se debe de programar por cuenta propia.

Para que sirve finalize()?

Hasta ahora sólo se ha mencionado que el método finalize() tiene como objetivo general de limpieza. ¿ Qué tantas ventajas tiene esto ? La principal ventaja es que libera memoria que ya no va a ser utilizada a lo largo del programa.

Esto no significa que finalize() libere memoria relacionada con otros objetos (por ejemplo si el objeto contiene otros objetos).

Una de las cosas para las que puede resultar interesante finalize() es para observar el proceso de limpieza:

Pe:

```
//Garbage.java
//Demuestra el proceso de liberación de memoria

class Chair {
        static bolean gcrun=false;
        static bolean f=false;
        static int created =0;
        static int finalized=0;
        int i;
        Chair() {
                i=++created;
                if(created==47)
                        System.out.println("Creado 47");
                }

        public void finalize() {
                if(¡gcrun) {
                        //la primera vez que es llamado finalize()
                        gcrun=trae;
                        System.out.println("Comenzando la limpieza despues de "+created+"sillas creadas");
                }
                if(i==47) {
                        System.out.println("Limpiando silla 47, poniendo la bandera para parar la creacion de siilas");
                        f=true;
                        }
                finalizad++;
                if(finalized >= created)
                        System.out.println("Todos"+finalized+ " limpiados");
                }
        }//fin clase

public class Garbage {
        public static void main(String[] args) {
                        //mientras la bandera no sea cierta
                        //crea sillas y cadenas
                        while(!Chair.f) {
                                new Chair();
                                new String("Tomando espacio de memoria");
                                }
                        System.out.println("Despues de que todas las sillas fueron creadas:\n"+"total creadas= " + Chair.created + ", total limpiadas= " +Chair.finalized);
```

```
                    if(args.length>0) {
                        if(args[0].equals("gc") || args[0].equals("all")) {
                            System.out.println("gc():");
                            System.gc();
                            }
                        if(args[0].equals("finalize") || args[0].equals("all"))
{
                            System.out.println("runFinalization():");
                            System.runFinalization();
                            }
                        }
                    System.out.println("Adios!");
                    }//fin main
            }//fin clase
```

Este programa crea varios objetos Chair y en algún punto de él comienza la limpieza o liberación de memoria. Para saber cuando comienza este proceso creamos una bandera llamado gcrun. Una segunda bandera "f" es nuestra forma de avisar al método main que pare de crear objetos. Finalmente , cada objeto Chair tiene su propio int i para rastrear que número es. En el loop del while, puede parecer que no hay nada que cambie el valor de la condición Chair.f, sin embargo, de esto se encarga finalize().

Cuando se corre el programa, se le provee un argumento a trave´s de la línea de comandos "gc", "finalize", o "all". El primer argumento llama a System.gc() (el cual forza la ejecución del colector de basura). El segundo llama a un método llamado System.runFinalization() que , en teoría, provocará que todos los objetos sean finalizados. El tercero provoca que ambos sean llamados.

Cabe señalar que todo este asunto de colleción de basura y finalización ha sido modificado con cada versión del JDK. De hecho, la ejecución de este programa puede variar dependiendo de la versión de JDK en el que sea utilizada.

La condición Death

En general no se puede confiar en el método fnalize(), por lo que se deben crear métodos propios para relizar dicha tarea. Sin embargo, existe un uso bastante interesante del método finalize(). Este es la verificación de la death condition de un objeto.

Hay veces en que ya no interesa el uso de algún objeto, por lo cual dicho objeto debería estar listo para que su memoria sea liberada. Por ejemplo, si algún objeto representa un archivo abierto, ese archivo debe ser cerrado antes de que el objeto sea puesto en el recolector de basra. Si algunas porciones del objeto no son propiamente limpiadas, causará un bug muy difícil de encontrar en el programa. En este caso, el método finalize() resulta muy valioso:

Pe:

```
//DeathCondition.java
//Usando finalize() para detectar un objeto que no ha
//sido propiamente limpiado

class Book {
        bolean checkedOut=false;
        Book(booleancheckOut) {
                checkedOut=checkOut;
                }

        void checkIn() {
                checkedOut=false;
                }

        public void finalize() {
                if(checkedOut)
                        System.out.println("Error: checked out");
                }
        }//fin clase

public class DeathCondition {
        public static void main(String[] args) {
                Book novel=new Book(true);
                //limpieza aduecada
                novel.checkIn();
                new Book(true);
                //Forza la limpieza y coleccion
                System.gc();
                }
        }//fin clase
```

La death condition consiste en que todos los objetos Book deben ser verificados antes de ser puestos en el recolector de basura.

V.- Inicialización

Java garantiza la inicialización de variables para su uso adecuado. En el caso de las variables que son definidas localmente:

```
void f() {
        int i;
        i++;
        }
```

el compilador enviará un mensaje de error diciendo que la variable “i” pudiera no haber sido inicializada. El compilador podría haber dado un valor a la variable, pero en este caso se interpreta más como un error de programación, forzando al programador a darlo poscuenta propia.

Sin embargo, cuando un tipo de dato primitivo es miembro de una clase, no funciona de la misma manera. Ya que cada método puede inicializar sus propios datos, no es forzoso que el programador ponga para estos datos valores iniciales. Pero no es recomendable que cada dato tenga basura como valor inicial, por lo que cada tipo de dato primitivo tiene garantizado un valor inicial.

Pe:

```
//ValoresIniciales.java
//Muestra los valores iniciales por default

class Medida {
        bolean t;
        char c;
        byte b;
        short s;
        int i,
        long l;
        float f;
        double d;
        void print() {
                System.out.println(
                “Tipo de dato        Valor inicial\n” +
                “bolean              “ + t + “\n” +
                “char                [“ + c + “] ” + (int)c + “\n” +
                “byte                “ + b + ”\n” +
                “short               “ + s + “\n” +
                “int                 “ + i + “\n” +
                “long                “ + l + “\n” +
                “float               “ + f + “\n” +
                “double              “ + d);
                }
        }

public class ValoresIniciales {
        public static void main(String[] args) {
                Medida d = new Medida();
                d.print();
                /* en este cao tambien sirve new Medida().print();*/
                }
        }//fin clase
```

Al ejecutar el programa la salida muestra los valores iniciales de cada tipo: cero y false para el tipo bolean. Más adelante se verá que cuando se inicia un objeto se le dará un valor especial de null.

Inicialización especificada

Para darle un valor inicial a alguna variable simplemente se le asigna el valor:

```
class Medida {
        bolean b= true;
        char c= 'x';
        byte B= 47;
        short s=0xff;
        int i=999;
        long l=1;
        flota f=3.14f;
        double d=3.14159;
//...
```

También se pueden utilizar métodos para dar estos valores iniciales:

```
class   CInit {
        int i=f();
//...
```

Estos métodos pueden tener argumentos pero deben ser previamente inicializados:

```
class Cinit {
        int i=f();
        int j=g(i);
        }
```

Pero no se puede hacer esto:

```
class Cinit {
        int j=g(i);
        int i=f();
        }
```

Ya que la variable "i" no ha sido inicializada.

Inicialización por medio del Constructor.

El constructor puede ser utilizado para inicializar variables, lo cual da una gran flexibilidad a los programas porque al mismo tiempo que creamos objetos podemos inicializar sus propiedades. Por ejemplo:

```
class Contador {
        int i;
        Counter() { i=7 ; }
...
```

Orden de la inicialización.

Dentro de una clase, el orden de inicialización lo determina el orden en que son definidas las variables. Un aspecto importante a considerar es que nuestras variables pueden aparecer dispersas a través de todo el programa, sin embargo , son inicializadas antes de siquiera ser utilizadas. Por ejemplo:

```
//OrdenInicializacion.java
//Demuestra el orden de inicialización.

class Tag{
        Tag(int marca) {
                System.out.println("Tag("+marca+")");
                }
        }

class Card {
        Tag t1=new Tag(1); //antes del constructor
        Card() {
                //Estamos en el constructor!
                System.out.println("Card()");
                t3=new Tag(33); //Reinicializa t3
                }
        Tag t2=new Tag(2); //Después del constructor
        void f() {
                System.out.println("f()");
                }
        Tag t3=new Tag(3); //Al final
        }

public class OrdenInicializacion {
        public static void main(String[] args) {
                Card t=new Card();
                t.f(); //Muestra que el constructor ya acabó
                }
}
```

Las definiciones de los objetos Tag están dispersas por todo el programa de manera intencional. La salida del programa es:

```
Tag(1)
```

```
Tag(2)
Tag(3)
Card()
Tag(33)
f()
```

El objeto t3 es inicializado dos veces, una antes del constructor y una durante la construcción. Esto puede no parecer muy útil o eficiente, sin embargo garantiza que nuestros objetos son inicializados propiamente.

Inicialización de objetos Static.

Cuando se inicializa un dato de tipo static ocurren cosas sismilares a lo visto anteriormente; en caso de que sea un objeto primitivo sin inicializar, el valor inicial será aquel que tenga el tipo primitivo. Si es alguna referencia a un objeto, su valor será null.

Para inicializar un objeto de tipo primitivo, se procede de la misma manera que con otros objetos. Por ejemplo:

```
//StaticInicializacion.java
//Se especifican valores iniciales en la definición de la clase.

class Bowl{
        Bowl(int marca) {
                System.out.println("Bowl("+marca+")");
                }
        }

class Table{
        static Bowl b1=new Bowl(1);
        Table(){
                System.out.println("Table()");
                b2.f(1);
                }
        void f2(int marca){
                System.out.println("f2("+marca+")");
                }
        static Bowl b2=new Bowl(2);
        }

class Cupboard{
        Bowl b3=new Bowl(3);
        static Bowl b4=new Bowl(4);
        Cupboard(){
                System.out.println("Cupboard()");
                b4.f(2);
                }
```

```
        void f3(int marca){
                System.out.println("f3("+marca+")");
                }
        static Bowl b5=new Bowl(5);
        }

public class StaticInicializacion{
        public static void main(String[] args){
                System.out.println("Creando nueva Cupboard() en main");
                new Cupboard();
                System.out.println("Creando nueva Cupboard() en main");
                new Cupboard();
                t2.f2(1);
                t3.f3(1);
        }
        static Table t2=new Table();
        static Cupboard t3=new Cupboard();
}
```

La salida del programa es:

```
Bowl(1)
Bowl(2)
Table()
f(1)
Bowl(4)
Bowl(5)
Bowl(3)
Cupboard()
f(2)
Creando nueva Cupboard() en main
Bowl(3)
Cupboard()
f(2)
Creando nueva Cupboard() en main
Bowl(3)
Cupboard()
f(2)
f2(1)
f3(1)
```

La inicialización static ocurre sólo cuendo es necesario. Si nunca se hace referencia a Table.b1 o Table.b2, los objetos static Bowl b1 y b2 nunca serán creados.

Java permite también la inicialización static a través de "una estructura static especial", conocida también como bloque static. Su sintaxis es :

```
class loquesea{
        static int i;
```

```
static{
        i=47;
        ...
}
```

Nótese que puede confundirse con un método. Este código es ejecutado sólo una vez, esto es cuando un objeto de esta clase se crea o la primera vez que se accede a un miembro de esta estructura. . Por ejemplo:

```
//StaticExplicito.java
//Inicialización explicita de static.

class Cup{
        Cup(int marca){
                System.out.println("Cup("+marca+")");
                }
        void f(int marca){
                System.out.println("f("+marca+")");
                }
        }

class Cups{
        static Cup c1;
        static Cup c2;
        static {
                c1= new Cup(1);
                c2=new Cup(2);
                }
        Cups(){
                System.out.println("Cups()");
                }
        }

public class StaticExplicito{
        public static void main(String[] args){
                System.out.println("Dentro de main()");
                Cups.c1.f(99); // (1)
                }
        //static Cups x=new Cups();// (2)
        //static Cups y=new Cups();//(2)
}
```

Inicialización de objetos No Static.

La sintaxis para la inicialización de objetos no static es muy similar. Por ejemplo:

```
//Mugs.java
```

```
//Inicialización no rmal.


class Mug{
        Mug(int marca){
                System.out.println("Mug("+marca+")");
                }
        void f(int marca){
                System.out.println("f("+marca+")");
                }
public class Mugs{
        Mug c1;
        Mug c2;
        {
                c1=new Mug(1);
                c2=new Mug(2);
                System.out.println("c1 & c2 inicializados");
        }
        Mugs() {
                System.out.println("Mugs()");
                }
        public static void main(String[] args){
                System.out.println("Dentro de main()");
                Mugs x= new Mugs();
        }
}
```

Inicialización de objetos Array.

El manejo de arreglos resulta una labor muy tediosa en otros lenguajes de programación como C o C++. Un arreglo es una secuencia de objetos o datos primitivos del mismo tipo almacenados bajo un mismo identificador.
Para definir un arreglo, simplemente :

```
Tipodedato[] nombredelarreglo;
```

o de la misma forma:

```
Tipodedato nombredelarreglo[];
```

Por ejemplo:

```
int[] a1;
String[] args;
double[] d1;
int a1[];
```

```
String args[];
...
```

Existen diversas formas de inicializar un arreglo, la inicialización puede aparecer en cualquier parte del código de la clase, sin embargo pueden ser inicializados en el momento mismo que es creado:

```
int[] a1={1,2,3,4,5};
```

También es posible asignar un arreglo a otro arreglo:

```
a2=a1;
```

Lo que en realidad ocurre es que se copia la referencia al arreglo, veamos el siguiente ejemplo:

```
//Arrays.java;
//Arreglos de primitivas.

public class Arrays{
        public static void main(String[] args){
                int[] a1={1,2,3,4,5};
                int[] a2;
                a2=a1;
                for(int i=0;i<a2.length;i++)
                        a2[i]++;
                for(int i=0;i<a1.length;i++)
                        System.out.println("a1["+i+"]="+a1[i]);
                }
        }
```

En este caso, el arreglo a1 es inicializado. Luego a2 modifica los valores del arreglo a1, dado que sólo se copia la referencia al arreglo. La salida del programa es:

```
a[0]=2
a[1]=3
a[2]=4
a[3]=5
a[4]=6
```

También en este ejemplo podemos observar la aparición de un elemento intrínseco en un arreglo: length. Todos los arreglos comienzan por el índice cero, uno, etc. El máximo elemento de un arreglo es siempre length-1, dado que length es una propiedad de un arreglo que nos dice la longitud del arreglo.

Cuando utilicemos un elemento del arreglo que no este asignado (que ese encuentre fuera de sus límites), Java nos enviará un mensaje de error de compilación.

Sin embargo, si no conocemos el máximo número de elementos del arreglo, ¿ cómo podemos crear el arreglo ? Simple, utilizamos new para crear los elementos del arreglo dinámico. Por ejemplo:

```
//ArregloDinamico.java
//Creando arreglos con new.

import java.util.*;
public class ArregloDinamico{
        static Random rand=new Random();
        static int pRand(int mod){
                return Math.abs(rand.nextInt())%mod+1;
                }
        public static void main(String[] args){
                int[] a;
                a=new int[pRand(20)];
                System.out.println("longitude de de a="+a.length);
                for(int i=0;i<a.length;i++)
                        System.out.println("a["+i+"]="+a[i]);
                }
        }
```

Ahora veamos un arreglo de objetos de tipo Integer:

```
//ArregloObjeto.java
//Creando arreglos de objetos.

import java.util.*;
public class ArregloObjeto{
        static Random rand=new Random();
        static int pRand(int mod){
                return Math.abs(rand.nextInt())%mod+1;
                }
        public static void main(String[] args){
                Integer[] a=new Integer[pRand(20)];
                System.out.println("longitud de a="+a.length);
                for(int i=0;i<a.length;i++)
                        System.out.println("a["+i+"]="+a[i]);
                }
        }
```

La diferencia entre estos dos últimos dos programas es que en el primero se trata de datos primitivos, mientras que en el Segundo se trata de objetos. Por lo tanto la diferencia entre ambos es la salida, en el primero los elementos del arreglo son inicializados en cero, mientras que en el segundo al ser objetos, su valor inicial es null.

También es posible inicializar un arreglo de la siguiente manera:

```
public class ArregloInt{
        public static void main(String[] args){
                Integer[] a= {
                        new Integer(1);
                        new Integer(2);
                        new Integer(3);
                        };
                Integer[] b=new Integer[]{
                        new Integer(1);
                        new Integer(2);
                        new Integer(3);
                        };
        }
}
```

En este caso el arreglo “a” está formado por:
a[0]=1,a[1]=2,a[3]=3

Esta forma de inicialización es bastante limitada porque la longitud del arreglo está predeterminada.

La segunda forma de inicializar un arreglo es a través de métodos. En este ejemplo crearemos un método que tome un argumento tipo Object:

```
//VarArgs.java
//Usando métodos para arreglos.

class A{ int i; }

public class VarArgs{
        static void f(Object[] x){
                for(int i=0;i<x.length;i++)
                        System.out.println(x[i]);
        }
        public static void main(String[] args){
                f(new Object[] {
                                        new Integer(47),new VarArgs(),
                                        new Float(3.14),new Double(11.11)
                                });
                f(new Object[] { “uno”,”dos”,”tres” });
                f(new Object[] {new A(),new A(), new A()});
        }
}
```

Arrays Multidimensionales.

Java permite también la creación de arreglos multidimensionales:

```
//MultiDimArray.java
//Creando arreglos multidimensionales.

import java.util.*;

public class MultiDimArray{
        static Random rand=new Random();
        static int pRand(int mod){
                return Math.abs(rand.nextInt()) % mod +1;
                }
        static void prt(String s){
                System.out.println(s);
                }
        public static void main(String[] args){
                int[][] a1={
                                {1,2,3},
                                {4,5,6}
                           };
        for(int i=0;i<a1.length;i++)
                for(int j=0;j<a1[i].length;j++)
                        prt("a1["+i+"]["+j+"]="+a1[i][j]);
        //arreglo de 3D con longitud fija
        int[][][] a2=new int[2][2][4];
        for(int i=0;i<a2.length;i++)
                for(int j=0;j<a2[i].length;j++)
                        for(int k=0;k<a2[i][j].length;k++)
                                prt("a2["+i+"]["+j+"]["+k+"]="+a2[i][j][k]);
        //arreglo 3D con longitud variada

        int[][][] a3=new int[pRand(7)][][];
        for(int i=0;i<a3.length;i++){
                a3[i]=new int[pRand(5)][];
                for(int j=0;j<a3[i].length;j++)
                        a3[i][j]=new int[pRand(5)];
                        }
        for(int i=0;i<a3.length;i++)
                for(int j=0;j<a3[i].length;j++)
                        for(int k=0;k<a3[i][j].length;k++)
                                prt("a3["+i+"]["+j+"]["+k+"]="+a3[i][j][k]);

        //Arreglo de objetos
        Integer[][] a4={
                                {new Integer(1),new Integer(2)},
```

```
                        {new Integer(3),new Integer(4)},
                        {new Integer(5),new Integer(6)},};
        for(int i=0;i<a4.length;i++)
                for(int j=0;j<a4[i].length;j++)
                        prt("a4["+i+"]["+j+"]="+a4[i][j]);

        Integer[][] a5;
        a5=new Integer[3][];
        for(int i=0;i<a5.length;i++){
                a5[i]=new Integer[3];
                for(int j=0;j<a5.length;j++)
                        a5[i][j]=new Integer(i*j);
                }
        for(int i=0;i<a5.length;i++)
                for(int j=0;j<a5[i].length;j++)
                        prt("a5["+i+"]["+j+"]="+a5[i][j]);
        }
}
```

6 Especificadores de Acceso

Uno de los aspectos más importantes de los lenguajes orientados a Objetos es el diseño de los programas. Dentro de dicho diseño, es importante saber separar “las cosas que cambian de las cosas que nunca lo hacen”.

Un ejemplo claro de esta filosofía son las librerías. El usuario de dichas librerías confía en que no es necesario rescribir el código de sus clases si una nueva versión de la librería está disponible. Por otro lado, el creador de dicha librería debe tener la libertad de hacer modificaciones pero con la seguridad de que los códigos del usuario seguirán funcionando.

Java provee de los llamados “Especificadotes de Acceso” para dicho tipo de problemas. Estos niveles de acceso van desde “más restringido” hasta “menos restringido”. Los especificadotes son **public**, **protected**, “**friendly**” (que no tiene palabra reservada) y **private**. Del ejemplo anterior se puede pensar que como creador de librerías, un programador debe mantener la mayor parte de su código como “**private**” y mostrar sólo los métodos que deseamos que el cliente utilice. Esto es exactamente la funcionalidad de dichos especificadotes de acceso, nosotros controlamos el acceso de nuestras clases.

Sin embargo el concepto de librería no se encuentra completo con dichos especificadores. La característica de una librería es que tienen una funcionalidad común. Para dichos casos Java provee la palabra “**package**”. Para comenzar este capítulo veremos como crear nuestras propias librerías a través de los paquetes o **package**.

Creación de paquetes con “Package”.

El uso de paquetes lo hemos estado trabajando en varias ocasiones en ejemplos anteriores, cuando tenemos una sentencia de este tipo:

import java.util,*;

estamos importando un paquete, en este caso ,el asterisco indica que queremos todos la librería que se encuentran en java.util.

En caso que queramos utilizar una librería en particular utilizamos por ejemplo:

import java.util.ArrayList;

En este caso sólo la librería ArrayList de java.util será utilizada.

Una librería es un conjunto de archivos class. Cada archivo tiene una sola clase **public**. Para que un conjunto de clases pertenezcan a la misma librería es necesario incluir la siguiente línea al principio del archivo:

package mipaquete;

Por ejemplo, supongamos que el nombre del archivo es MiClase.java. Esto implica que dicho archivo sólo puede tener una clase public cuyo nombre debe ser el mismo del archivo *.java:

```
package mipaquete;
public class MiClase {
        ...
```

Nótese que el nombre de la clase **public** es ***IDENTICO*** al del archivo **.java.**

Ahora, si alguien quisiera utilizar por alguna razón alguna clase **public** de nuestro paquete, tendría que incluir la siguiente línea de código en su archivo:

```
import mipaquete.*;
class OtraClase{
        MiClase m = new MiClase();
```

O bien especificando toda la ruta, lo cual resulta algo molesto:

```
mipaquete.MiClase m = new mipaquete.MiClase();
```

Creando nuestros propios paquetes.

Como hemos visto , un paquete puede estar compuesto por varios archivos con extensión ***.class** . Para evitar confusiones, los archivos class de un paquete en particular deben estar contenidos dentro de un mismo directorio. Hacer esto nos resuelve dos problemas: crear paquetes con nombre único y encontrar rápidamente las clases correspondientes a dichos paquetes en el directorio adecuado.

Cuando hacemos directorios y paquetes Java trabaja de la siguiente manera: primero busca la variable de ambiente de nombre CLASSPATH, la cual contiene uno o más directorios donde se encuentran los archivos .class. Comenzando desde la raíz, el intérprete toma nuestro paquete y reemplaza cada punto que encuentra por un flash para generar el directorio correcto. (el paquete **package foo.bar.baz** se convierte en **foo/bar/baz** ó **foo\bar\baz**).

Veamos un ejemplo:

```
//com:serge:simple:Vector.java
//Creando un paquete!

package com.serge.simple;

public class Vector{
        public Vector(){
                System.out.println("Vector de com.serge.simple");
                }
        }
```

Como vemos la primera línea corresponde a la creación o especificación de nuestro paquete. Veamos otro ejemplo sencillo:

```
//com:serge:simple:List.java
//Creando paquetes!

package com.serge.simple;

public class Lista{
        public  Lista(){
                System.out.println("com.serge.simple.Lista");
                }
        }
```

Ambos archivos se encuentran dentro del subdirectorio de mi sistema:

C:\DOC\Java\com\serge\simple

y el la variable de ambiente CLASSPATH está configurada de tal manera:

CLASSPATH = ; D:\JAVA\LIB; C:\ DOC\ Java

Aquí podemos ver también que el CLASSPATH puede contener diversos directorios alternativos.

Una vez configurado correctamente esta variable, podemos poner el siguiente archivo en cualquier directorio:

```
//Utilizando nuestra librería

import com.serge.simple.*;

public class LibTest{
        public static void main(String[] args){
                Vector v=new Vector();
                Lista l=new Lista();
                }
        }
```

Colisiones

Las colisiones ocurren cuando importamos dos librerías que contengan una clase con el mismo nombre, por ejemplo, tomando la librería que acabamos de crear:

```
import java.util.*;
import com.serge.simple.*;
```

Como **java.util** contiene una clase llamada **Vector** y nuestro paquete también, cuando creemos un objeto con la sentencia:

Vector v=new Vector();

ocurrirá una colisión.

Para distinguir correctamente entre ambos vectores debemos utilizar la notación larga vista al principio del capítulo:

java.util.Vector v=new java.util.Vector();

pero caemos en una contradicción muy absurda, ¿ para qué importamos el paquete **java.util** si vamos a especificar con notación larga ? La respuesta es que estemos utilizando otra clase distinta a Vector en este caso , si no, no tiene caso importar dicho paquete.

Más librerías.

Una vez comprendido dichos conceptos de paquetes, podemos crear paquetes y librerías propias. Por ejemplo, creemos una librería que sea un alias de System.out.println.

```
//com:serge:tools:P.java
//Alias de System.out.println! y System.out.print!

package com.serge.tools;

public class P{
        public static void rint(String s){
                System.out.print(s);
                }
        public static void rintln(String s){
                System.out.println(s);
                }
        }
```

La localización de este archivo es : **C:\Doc\Java\com\serge\tools**
Ahora veamos una aplicación con un ejemplo:

```
//ToolTest.java
//probando nuestras tools!

import com.serge.tools.*;

class ToolTest{
        public static void main(String[] args){
```

```
                P.rintln("Imprimiendo desde ahora!");
                P.println(""+100); //Se hace String
                P.println(""+100L);//Se hace String
                P.println(""+3.14159);//Se hace String
                }
        }
```

Como vemos, podemos crear cuantas aplicaciones necesitemos, ahora veamos un ejemplo de debugger:

```
//com:serge:tools:debug:Debugger.java
//Herramienta de debugger.

package com.serge.tools.debug;

public class Debugger{
        private static void perr(String msg){
                System.err.println(msg);
                }
        public final static void is_true(boolean exp){
                if(!exp) perr("Prueba fallida");
                }
        public final static void is_false(boolean exp){
                if(exp) perr("Prueba fallida");
                }
        public final static void is_true(boolean exp,String msg){
                if(!exp) perr("Prueba fallida:"+ msg);
                }
        public final static void is_false(boolean exp,String msg){
                if(exp) perr("Prueba fallida: "+msg);
                }
        }
```

Esta clase nos sirve para pruebas booleanas, las cuales imprimen mensajes de error si fallan. Veamos un segundo ejemplo de debugger:

```
//com:serge:tools:Debugger.java
//Otro ejemplo de Debugger

package com.serge.tools;

public class Debugger{
        public final static void is_true(boolean exp){}
        public final static void is_false(boolean exp){}
        public final static void is_true(boolean exp,String msg){}
        public final static void is_false(boolean exp,String msg){}
        }
```

Veamos un ejemplo de aplicación:

```
//TestDebugger.java
//Probando nuestra herramienta de Debugger

import com.serge.tools.debug.*;
//import com.serge.tools.*;

class TestDebugger{
        public static void main(String[] args){
                Debugger.is_true((2+2)==5);
                Debugger.is_false((1+1)==2);
                Debugger.is_true((2+2)==5,"2+2==5");
                Debugger.is_false((1+1)==2,"1+1!=2");
                }
        }
```

Al cambiar el paquete importado veremos el cambio en el comportamiento del programa.

Especificadores de Acceso de Java.

Como se mencionó anteriormente, los especificadores de acceso de Java son : **public, protected y private**. Cada uno especifica el control de acceso a los miembros de una clase o incluso a una clase misma.

"Friendly"

Si una clase no tiene ningún especificador en particular, como muchos de los ejemplos manejados hasta este momento, ¿ qué tipo de acceso tiene ? Aunque no tenga una palabra reservada, a dicho tipo de acceso se le conoce como "**friendly**".

Este acceso permite que todas las clases dentro del mismo paquete tengan acceso a ella, por lo que las clases ajenas a dicho paquete no lo harán.

Public: Acceso libre.

El especificador **public** indica que todos tienen acceso a dicho miembro o clase, sin importar el paquete. Veamos un ejemplo:

```
//desayuno:Galleta.java
//Creando una librería
package esp.menu;

public class Galleta{
        public Galleta(){
```

```
            System.out.rpintln("Constructor de Galletas!");
            }
        void mordida(){
            System.out.println("Mordida");
            }
        }
```

Recuerda que Galleta.java debe residir en un directorio llamado menú en esp.
Ahora creamos un programa que use Galleta:

```
//Cena.java
//Usando librería
import esp.menu.*;

public class Cena{
        public Cena(){
            System.out.println("Construyendo Cena!");
            }
        public static void main(String[] args){
            Galleta x=new Galleta();
            //x.mordida() ;// Sin acceso!
            }
        }
```

Podemos observar lo siguiente: **Cena** y **Galleta** no pertenecen al mismo paquete, sin embargo, podemos construir un objeto **Galleta** ya que su constructor es **public**. También observamos que el método mordida no tiene especificador explícito, por lo que su especificador es "**friendly**", por lo tanto las clases ajenas a su paquete no tendrán acceso a dicho método.

private: prohibido tocar!

Este especificador indica que sólo la clase actual tiene acceso a dichos miembros. Otros miembros del mismo paquete también son ajenos a los miembros **private**. Su uso principal es cuando nuestro programas se vuelven multihilo. Veamos un ejemplo:

```
//esp:Helado.java
//Demostrando el uso de private.

class Sundae{
        private Sundae(){}
        static Sundae makeSundae(){
            return new Sundae();
            }
        }
```

```
public class Helado{
        public static void main(String[] args){
                //! Sundae x=new Sundae();
                Sundae x=Sundae.makeSundae();
                }
        }
```

El uso de **private** puede resultar útil cuando queremos que algun objeto sea creado de alguna forma en particular. En este caso, no permitimos el uso del constructor default.

*El uso de **private** en el constructor permite también evitar que una clase sea heredada, siempre y cuando se especifique sólo el constructor default en la clase y sea **private**.

protected: una especie de "friendly".

El uso de **protected** tiene que ver directamente con el concepto de herencia. El concepto de herencia es tratado a parte en este manual por lo que sólo se dará una idea de su uso. Para heredar de una clase la sintaxis es:

```
class hija extendí padre {
        ...
```

Veamos un ejemplo:

```
//esp:ChocolateChip.java
//No podemos acceder a miembros de la familia!
//de otra clase

import esp.menu.*;

public class ChocolateChip extendí Galleta{
        public ChocolateChip(){
                System.out.println("Construyendo Chocolatechip");
                }
        public static void main(String[] args){
                ChocolateChip x=new ChocolateChip();
                //!x.mordida();//No tenemos acceso
                }
        }
```

Lo interesante de este ejemplo, es que al heredar de la clase **Galleta**, todos sus métodos son heredados por clases hijas, en este caso **ChocolateChip**. Sin embargo, el

método **mordida**() no es accesible ya que **Galleta** está dentro del paquete **menú**, por lo que es "**amiga**" de **ChocolateChip**. Si cambiamos el código de **Galleta** de la siguiente manera, entonces las clases hijas tendrán accesible el método **mordida**():

```
Public class Galleta{
        Public Galleta(){
                System.out.println("Construyendo Galleta!");
                }
        protected void bite() {
                System.out.println("Mordida");
                }
        }
```

ya que **protected** permite acceso a la clase actual así como a los que heredan de ella.

Acceso de Clases

Para controlar el acceso a una clase, el especificador debe aparecer antes de la palabra **class**:

```
public class Widget{
```

Sin embargo, existen ciertas restricciones:

1. **Sólo puede existir una clase pública por archivo .java. Si existe más de una, el compilador enviará un mensaje de error.**
2. **El nombre de la clase pública debe ser el mismo del archivo .java.**
3. **También es posible que un archivo no contenga ninguna clase pública.**

Una clase no puede ser **private** o **protected**. Por lo que en realidad una clase sólo puede ser "**friendly**" o **public**. Si queremos que nadie tenga acceso a los métodos de alguna clase, simplemente declaramos los métodos como **private**. Veamos un ejemplo:

```
//esp:Lunch.java
//Uso de especificadotes de clase.
//Hacemos una clase privada al volver
//todos sus métodos private.

class Sopa{
        private Sopa(){}
        //(1) Creación sólo mediante este método:
        public static Sopa hazSopa(){
```

```
            return new Sopa();
            }
        //(2)Otra forma
        private static Sopa ps1=new Sopa();
        public static Sopa access(){
            return ps1;
            }
        public void f(){}
        }

    class Sandwich{
        void f(){ new Lunch();}
        }

    public class Lunch{
        void test(){
            //Imposible, constructor privado!
            //!Sopa priv1=new Sopa();
            Sopa priv2=Sopa.hazSopa();
            Sandwich f1= new Sandwich();
            Sopa.access().f();
            }
        }
```

Resumen

Control de acceso				
Especificadores	clase	subclase	paquete	todas
public	✓	✓	✓	✓
protected	✓	✓	✓	
private	✓			
friendly	✓	✓	✓	
private protected		✓		

7 Reutilización de Clases

Una de las ventajas más importantes del lenguaje de Java es la reutilización de código. Para la reutilización de código, es necesario crear nuevas clases, pero la ventaja es que no las vamos a crear de la nada, sino utilizando clases existentes que alguien más las ha creado.

Para la reutilización de código existen dos métodos importantes y muy utilizados. El primero consiste en crear simplemente objetos a partir de clases existentes. A este método se le conoce como "**composition**" (composición), porque nuestra nueva clase se compone de objetos de clases ya existentes.

El segundo método es más difícil de comprender. Consiste en crear una nueva clase como "tipo de otra clase". Literalmente se toma la forma de la clase existente y se implementa con nuevas funciones o atributos. A este método se le conoce como "**inheritance**" o **herencia**. La herencia es una de los porqués de la OOP.

En este capítulo aprenderemos a utilizar ambas técnicas.

Composición

Hasta ahora hemos estado utilizando la composición muy frecuentemente. Por ejemplo, supongamos que queremos un objeto que contenga muchos objetos String, dos datos primitivos y un objeto de otra clase:

```
//SistemaAlmacen.java
//Utilizando composición para reutilización de código.

class FuenteDeAgua{
        private String s;
        FuenteDeAgua(){
                System.out.println("Fuente de Agua!");
                s=new String("Construida");
                }
        public String toString(){return s;}
        }


public class SistemaAlmacen{
        private String valvula1,valvula2,valvula3,valvula4;
        FuenteDeAgua fuente;
        int i;
        float f;
        void print(){
                System.out.println("valvula1= "+ valvula1);
                System.out.println("valvula2= "+ valvula2);
                System.out.println("valvula3= "+ valvula3);
                System.out.println("valvula4= "+ valvula4);
```

```
            System.out.println("i= "+ i);
            System.out.println("f= "+ f);
            System.out.println("fuente= "+ fuente);
            }
      public static void main(String[] args){
            SistemaAlmacen x=new SistemaAlmacen();
            x.print();
            }
      }
```

El método dentro de **FuenteDeAgua** es nuevo: **toString**(). Todo objeto no primitivo tiene un método **toString**() y se utiliza en ocasiones especiales cuando el compilador quiere un **String** pero tiene un dato distinto.

Como vemos en este ejemplo, los datos primitivos son inicializados en cero, mientras que los objetos son inicializados en **null**. Para inicializar objetos podemos hacerlo en de tres maneras distintas:

1. **Cuando los objetos son definidos.**
2. **En el constructor de la clase.**
3. **Justo antes de utilizar el objeto, se le conoce como " lazy inicialization"(inicialización tardía) .**

En el siguiente ejemplo veremos el uso de las tres inicializaciones:

```
//Banio.java
//Inicialización de objetos con composición.

class Jabon{
      private String s;
      Jabon(){
            System.out.println("Jabon()");
            s=new String("Construido");
            }
      public String toString(){return s;}
      }

public class Banio{
      private String
      //Inicializando en el punto de definición
      s1=new String("Feliz"),
      s2="Feliz",
      s3,s4;
      Jabon castille;
      int i;
```

```
float juguete;
Banio(){
        System.out.println("Dentro de Banio()");
        s3=new String("Feliz");
        i=47;
        juguete=3.14f;
        castille=new Jabon();
        }
void print(){
        //Inicialización tardía.
        if(s4==null)
                s4=new String("Feliz");
        System.out.println("s1= "+s1);
        System.out.println("s2= "+s2);
        System.out.println("s3= "+s3);
        System.out.println("s4= "+s4);
        System.out.println("i= "+i);
        System.out.println("juguete= "+juguete);
        System.out.println("castile= "+ castille);
        }
public static void main(String[] args){
        Banio b=new Banio();
        b.print();
        }
}
```

Sintaxis de Herencia.

La herencia es una parte esencial de Java (de hecho, de todos los lenguajes OOP en general). Podría resultar algo sorprendente saber que hemos estado utilizando una herencia implícita, ya que cada vez que una clase que creamos no especifica que va a heredar de otra, heredamos implícitamente de la clase **Object**.

Cuando heredamos, decimos "Esta clase es como esa otra clase ". Hemos visto anteriormente la sintaxis de herencia, sin embargo veamos un ejemplo más completo:

```
//Detergente.java
//Heredando.

class Limpiador{
        private String s=new String("Limpiador");
        public void append(String a){s+=a;}
        public void diluir(){ append("diluir()");}
        public void aplicar(){ append("aplicar()");}
        public void tallar(){ append("tallar()");}
        public void print(){ System.out.println(s);}
```

```
        public static void main(String[] args){
                Limpiador x=new Limpiador();
                x.diluir();x.aplicar();x.tallar();x.print();
                }
        }

public class Detergente extends Limpiador {
        //Cambiando un método
        public void tallar(){
                append("Detergente.tallar()");
                super.tallar();//versión de clase padre
                }
        //Tiene métodos propios!
        public void otraCosa(){ append("otraCosa()");}
        //Probando la nueva clase
        public static void main(String[] args){
                Detergente x=new Detergente();
                x.diluir();
                x.aplicar();
                x.tallar();
                x.otraCosa();
                x.print();
                System.out.println("Probando clase padre");
                Limpiador.main(args);
                }
        }
```

Podemos observar que ambas clases tienen método **main**(). Cada clase puede tener su método **main**() sin que interfiera con la otra. Es importante que los métodos de Limpiador sean **public**, ya que de otra manera la clase hija no podrá acceder a ellos. Cuando se utiliza herencia, la regla general es que todos los atributos sean **private** y los métodos **public**. Otra cosa importante de este ejemplo es que la clase padre, **Limpiador**, tiene sus métodos propios. La clase hija hereda automáticamente de la clase padre sus métodos e incluso podrá tener sus propios métodos o mejorar los heredados. La expresión **super.tallar**() llama al método de la clase padre.

Inicializando las clases de herencia.

Como tenemos dos clases involucradas, la clase base y la clase derivada de ella, podría resultar un poco confuso imaginar el objeto resultante producido por la clase derivada. Cuando se crea un objeto de la clase que hereda, en realidad contiene un **subobjeto** de la clase padre. Es esencial que dicho subobjeto sea inicializado correctamente. Para cumplir con dicho objetivo, es necesario hacerlo media nte el constructor:

```
//Cartoon.java
//Comportamiento del constructor mediante herencia.
class Art{
        Art(){
                System.out.println("Constructor de Art()");
                }
        }
class Drawing extends Art{
        Drawing(){
                System.out.println("Constructor de Drawing");
                }
        }
public class Cartoon extends Drawing{
        Cartoon(){
                System.out.println("Constructor de Cartoon");
                }
        public static void main(String[] args){
                Cartoon x=new Cartoon();
                }
        }
```

Como podemos ver, la clase base es inicializada antes de las clases heredadas. Veamos ahora un ejemplo de constructores con argumentos:

```
//Chess.java
//Herencia, construtores y argumentos.

class Game{
        Game(int i){
                System.out.println("Constructor de Juego");
                }
        }
class Tablero extends Game{
        Tablero(int i){
                super(i);
                System.out.println("Constructor de Tablero");
                }
        }
public class Chess extends Tablero{
        Chess(){
                super(11);
                System.out.println("Constructor de Ajedrez");
                }
        public static void main(String[] args){
                Chess x=new Che ss();
                }
        }
```

En este ejemplo podemos observar varias situaciones interesantes. Primero, vemos que cuando nuestro constructor contiene argumentos y además pertenece a una clase que hereda, es **NECESARIO** hacer una referencia al constructor de la clase padre a través de la palabra reservada **super**. Además, debe notarse que es lo **primero** que debemos hacer en el constructor de la clase hija.

Combinando Composición y Herencia.

Es una práctica muy común combinar en una clase la composición y le herencia. El siguiente ejemplo muestra la creación de una clase más compleja, utilizando herencia y composición:

```
//PlaceSetting.java
//Combinando composición y herencia.

class Plato{
        Plato(int i){
                System.out.println("Constructor de platos!");
                }
        }

class PlatoDeCena extends Plato{
        PlatoDeCena(int i){
                super(i);
                System.out.println("Constructor de PlatoDeCena");
                }
        }



class Utensilio{
        Utensilio(int i){
                System.out.println("Constructor de utensilios");
                }
        }

class Cuchara extends Utensilio{
        Cuchara(int i){
                super(i);
                System.out.println("Constructor de Cucharas");
                }
        }

class Tenedor extends Utensilio{
        Tenedor(int i){
                super(i);
```

```
                System.out.println("Constructor de Tenedores");
                }
        }
class Cuchillo extends Utensilio{
                Cuchillo(int i){
                        super(i);
                        System.out.println("Constructor de cuchillos");
                        }
                }
class Custom{
        Custom(int i){
                System.out.println("Constructor Custom");
                }
        }

public class PlaceSetting extends Custom{
        Cuchara sp;
        Tenedor frk;
        Cuchillo kn;
        PlatoDeCena pl;
        PlaceSetting(int i){
                super(i+1);
                sp=new Cuchara(i+2);
                frk=new Tenedor(i+3);
                kn=new Cuchillo(i+4);
                pl=new PlatoDeCena(i+5);
                System.out.println("Constructor de PlaceSetting");
                }

        public static void main(String[] args){
                PlaceSetting x=new PlaceSetting(9);
                }
        }
```

Veamos ahora un ejemplo de sobrecarga de métodos:

```
//Oculta.java
//Sobrecagando el nombre de un método padre en una clase hija.
//No lo oculta!

class Homer{
        char doh(char c){
                System.out.println("doh(char)");
                return 'd';
                }
        float doh(float f){
```

```
                System.out.println("doh(float)");
                return 1.0f;
                }
        }
class Marge{}
class Bart extends Homer{
        void doh(Marge m){}
        }
class Oculta{
        public static void main(String[] args){
                Bart b=new Bart();
                b.doh(1);//usando doh(float)
                b.doh('x');
                b.doh(1.0f);
                b.doh(new Marge());
                }
        }
```

Escogiendo entre composición y herencia.

Como hemos visto, ambas técnicas nos permiten crear objetos dentro de nuestra nueva clase. La Composición es utilizada generalmente cuando queremos las features de la clase de la que creamos el nuevo objeto, pero no nos interesa su interfaz. Es decir, utilizamos las ventajas de dicho objeto dentro de nuestra clase. Veamos un ejemplo típico de composición:

```
//Carro.java
//Composición con objetos públicos

class Motor{
        public void arranca(){}
        public void reversa(){}
        public void frena(){}
        }
class Rueda{
        public void infla(int psi){}
        }
class Ventana{
        public void rollup(){}
        public void rolldown(){}
        }
class Puerta{
        public Ventana ventana=new Ventana();
        public void abre(){}
        public void cierra(){}
        }
public class Carro{
```

```
public Motor motor=new Motor();
public Rueda[] rueda=new Rueda[4];
public Puerta izq=new Puerta(),
              der=new Puerta();
public Carro(){
        for(int i=0;i<4;i++)
              rueda[i]=new Rueda();
        }
public static void main(String[] args){
        Carro carro=new Carro();
        carro.izq.ventana.rollup();
        carro.rueda[0].infla(72);
        }
}
```

Como vemos , la composición se utiliza más en el contexto de "Un carro tiene 4 ruedas, dos puertas, un motor, etc". Es decir , es una relación de pertenencia.

La herencia se enfoca más hacia otro sentido, su relación es " es un " o "se parece a". Por ejemplo, un perro es un animal, un gato es un animal, un auto es un vehículo. El análisis de herencia puede ser tan específico como se desee, pudiendo llegar desde clases abstractas ,que son más generales, hasta subclases más específicas.

*** En inglés, la relación de composición es conocida como "has a", mientras que la relación de herencia se le conoce como "is a".**

Revisión al Especificador Protected.

Ahora que conocemos la herencia, tiene sentido lo mencionado anteriormente acerca de dicho especificador. La palabra reservada **protected**, nos asegura una relación de herencia. En realidad dice "***Esto es privado para aquellos que no son de mi clase o subclase, y público para aquellos que me hereden o que pertenezcan a mi paquete".***

Como se ha mencionado anteriormente, se recomienda que los atributos de una clase sean **privados**, y que los métodos sean **públicos**. Sin embargo, con miembros **protegidos**, aseguramos un control sobre ellos a clases herederas. Veamos un ejemplo:

```
//Orc.java
//La palabra reservada protected.
import java.util.*;

class Villano{
        private int i;
        protected int lee(){return i;}
        protected void set(int ii){i=ii;}
        public Villano(int ii){i=ii;}
        public int valor(int m){return m*i ;}
        }
```

```
public class Orc extends Villano{
        private int j;
        public Orc(int jj){super(jj);j=jj;}
        public void cambia(int x){set(x);}
        }
```

Como podemos ver, aquí respetamos la regla de atributos **private** y métodos **public**. También observamos que el método **cambia**(), tiene acceso al método **set**(), porque es **protected**.

Upcasting

Hasta ahora, podemos pensar que el aspecto más importante acerca de la herencia, es que podemos heredar métodos a nuestra clase hija. Sin embargo, la herencia es importante por la relación que se crea entre la clase padre y la clase hija.

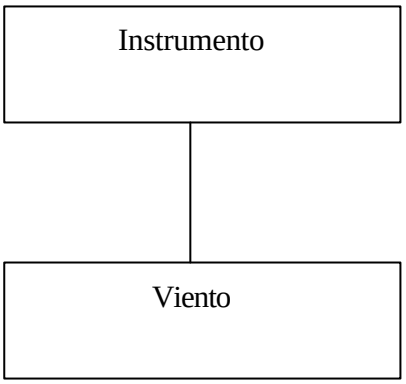
Veamos un ejemplo, consideremos como clase base **Instrumento** que representa instrumentos musicales, y una clase derivada **Viento**. Como sabemos, la herencia implica que todos los métodos de la clase padre estarán disponibles en la clase hija; entonces, cualquier mensaje que le enviemos a la clase padre podrá ser enviado a la clase hija.

```
//Viento.java
//Herencia y upcasting
import java.util.*;

class Instrumento{
        public void play(){}
        static void tune(Instrumento i){
                //…
                i.play();
                }
        }
//Los instrumentos de Viento son instrumentos!
//porque tienen la misma interfaz
class Viento extends Instrumento{
        public static void main(String[] args){
                Viento flauta=new Viento();
                Instrumento.tune(flauta); //Upcasting
                }
        }
```

Vemos en el ejemplo lo siguiente: el método **tune**() tiene como argumento un objeto **Instrumento**. Sin embargo, en el método **Viento.main**() el método **tune**() es llamado con una referencia de tipo **Viento**. Como podemos comprobar, los objetos de tipo **Viento** también son objetos de tipo **Instrumento**.

Veamos el diagrama de herencia de estas dos clases:



Vemos que el **upcasting** tiene sentido ya que partimos de una clase más específica a una más general, por lo que se cumple la regla. La clase hija podrá tener más métodos que la clase padre, sin embargo, debe contener por lo menos los métodos de la clase padre.

La palabra clave final.

Este palabra clave tiene diferentes significados dependiendo de su aplicación, pero de manera general significa " No puedes cambiar esto".

Datos final.

Muchos lenguajes de programación tienen alguna forma de decir al compilador que algún dato es constante. Una constante es útil por dos motivos:

1. **En tiempo de compilación nunca cambiará.**
2. **Puede ser inicializada en tiempo de ejecución sin que cambie nunca.**

En Java, dichas constantes deben ser primitivas y se expresan con la palabra clave **final**. En el caso de los objetos, **final** hace la referencia al objeto constante, sin embargo, el objeto en sí puede ser modificado. Veamos un ejemplo:

```
//FinalData.java
//El efecto de final

class Valor{
        int i=1;
        }
public class FinalData{
```

```
final int i1=9;
static final int VAL_TWO=99;
public static final int VAL_THREE=39;
final int i4=(int)(Math.random()*20);
static final int i5=(int)(Math.random()*20);

Valor v1=new Valor();
final Valor v2=new Valor();
static final Valor v3=new Valor();
final int[] a={1,2,3,4,5,6};
public void print(String id){
        System.out.println(id+": "+"i4= "+i4+",i5= "+i5);
        }
public static void main(String[] args){
        FinalData fd1=new FinalData();
        //!fd1.i1++;//Error: no podemos cambiar su valor!
        fd1.v2.i++;//El objeto no es constante
        fd1.v1=new Valor();
        for(int i=0;i<fd1.a.length;i++)
                fd1.a[i]++;//El objeto no es constante!
        //fd1.v2=new Valor();//Error: no podemos cambiar
        //fd1.v3=new Valor();//la referencia
        //fd1.a=new int[3];
        fd1.print("fd1");
        System.out.println("Creando datos Final");
        FinalData fd2=new FinalData();
        fd1.print("fd1");
        fd2.print("fd2");
        }
}
```

Java permite también la creación de datos **final** que son declarados como tales, pero que no son inicializados. De cualquier manera, deben ser inicializados antes de ser utilizados. Esto nos permite que cada campo **final** dentro de una clase tenga un valor distinto para cada objeto y además mantiene su calidad de **final**. Veamos un ejemplo:

```
//BlankFinal.java
//Uso de final en datos en blanco.

class Poppet{}

class BlankFinal{
        final int i=0; //Inicializado
        final int j; //Blank!
        final Poppet p; // Blank!
        //Inicializamos los datos blank en el constructor
```

```
BlankFinal(){
        j=1;
        p=new Poppet();
        }
BlankFinal(int x){
        j=x;
        p=new Poppet();
        }

public static void main(String[] args){
        BlankFinal bf=new BlankFinal();
        }
}
```

Como vemos, la inicialización puede garantizarse mediante el constructor.

Java permite también el uso de argumentos **final** para declararlos en nuestros métodos. Esto implica que el método **NO** puede cambiar el valor de dichos datos.

```
//FinalArgumentos.java
//Usando final como argumentos

class Gizmo{
        public void spin(){}
        }
public class FinalArgumentos{
        void with(final Gizmo g){
                //!g=new Gizmo();//illegal, g es final
                }
        void without(Gizmo g){
                g=new Gizmo();//OK g no es final
                g.spin();
                }
        //void f(final int i){i++;}//NOP
        int g(final int i){return i+1;}
        public static void main(String[] args){
                FinalArgumentos bf=new FinalArgumentos();
                bf.without(null);
                bf.with(null);
                }
        }
```

Existen dos razones de ser de métodos **final**. La primera es poner un candado al método para prevenir la herencia de la clase. La segunda razón es por eficiencia. Un método **final** es más rápido que un método normal, sin embargo, si su tarea es larga, dicha eficiencia no será considerable.

Otro detalle importante, es que cualquier método **private** es implícitamente **final**, ya que no podemos tener acceso a ellos ni sobrescribirlos.

Clases Final.

Cuando una clase es **final**, implica que no queremos que dicha clase herede. Esto se realiza rara vez cuando el diseño de nuestro modelo orientado a objetos nos lo indique. Veamos un ejemplo:

```
//Jurassic.java
//Haciendo una clase final.

class SmallBrain{}
final class Dinosaur{
        int i=7;
        int j=1;
        SmallBrain x=new SmallBrain();
        void f(){}
        }

//!class Further extends Dinosaur{}
//error: No podemos heredar una clase final

public class Jurassic{
        public static void main(String[] args){
                Dinosaur n=new Dinosaur();
                n.f();
                n.i=40;
                n.j++;
                }
        }
```

Inicialización con herencia.

Es de ayuda ver el proceso de inicialización, incluyendo cuando realizamos herencia. Veamos un ejemplo:

```
//Beetle.java
//El proceso de inicialización

class Insect{
        int i=9;
        int j;
        Insect(){
                prt("i= "+i+", j = "+j);
                j=39;
                }
```

```
        static int x1=prt("Static Insect.x1.inicializado");
        static int prt(String s){
                System.out.println(s);
                return 47;
                }
        }
public class Beetle extends Insect{
        int k=prt("Beetle.k inicializado");
        Beetle(){
                prt("k= "+k);
                prt("j= "+j);
                }
        static int x2=prt("static Beetle.x2 inicializado");
        public static void main(String[] args){
                prt("Constructor de Beetle");
                Beetle b=new Beetle();
                }
        }
```

Ejercicios.

1. **Crea dos clases A y B, con constructores sin argumentos que simplemente indiquen la creación del objeto. Hereda de una nueva clase C de la clase A, y crea un objeto de la clase B dentro de C. No construir constructor para C, observa los resultados.**
2. **Modifique la clase anterior de tal forma que A y B tengan constructores con argumentos. Construya un constructor para C e inicialice en éste.**
3. **Crear una clase Anfibio. A partir de ella, herede una clase llamada Rana. Cree los métodos adecuados en Anfibio, de tal manera que demuestre el upcasting.**

8 Objeto Array y String

Arreglos

La presentación acerca de la inicialización de los arreglos se vio anteriormente. Los arreglos son herramientas de gran uso dentro de un lenguaje de programación. Los arreglos nos proporcionan la manera más eficiente que nos proporciona Java de almacenar y acceder a objetos. Un arreglo es una secuencia lineal, que nos proporciona bastante velocidad en su manejo, pero pagamos el precio de dicha rapidez: cuando se crea un arreglo, su tamaño es fijo y no puede ser cambiado por otro.

Los arreglos permiten el almacenamiento de información. En Java existen otros objetos que también nos permiten almacenar información. Estos objetos son **List**, **Set** y **Map**.

Inicialización de Arreglos.

Veamos un ejemplo de inicialización de arreglos, a manera de recordatorio.

```
//Arreglos.java
//Inicialización.

class Weeble {} // Una criatura Mística!
public class Arreglos{
        public static void main(String args[]){
                //Arreglo de objetos.
                Weeble[] a; // Referencia nula
                Weeble[] b=new Weeble[5]; //Referencia nula
                Weeble[] c=new Weeble[4];
                for(int i=0;i<c.length;i++){
                        c[i]=new Weeble();
                //Inicializado
                Weeble[] d={ new Weeble(),new Weeble(), new Weeble()};
                a=new Weeble[] { new Weeble(),new Weeble()};
                System.out.println("a.length="+a.length);
                System.out.println("b.length="+b.length);
                //Las referencias son automáticamente inicializadas en null.
                for(int i=0;i<b.length;i++)
                        System.out.println("b["+i+"]="+b[i]);
                System.out.println("c.length="+c.length);
                System.out.println("d.length="+d.length);
                a=d;
                System.out.println("a.length="+a.length);

                //Arreglos de Primitivas.
```

```
        int[] e; //Referencia nula
        int[] f=new int[5];
        int[] g=new int[4];
        for(int i=0;i<g.length;i++)
                g[i]=i*I;
        int[] h={11,47,93};
        //Error de compilacion: "Variable e not initialiazed"
        //!System.out.println("e.length="+e.length);
        System.out.println("f.length="+f.length);
        //Los valores Primitivos son inicializados en cero
        for(int i=0;i<f.length;i++)
                System.out.println("f["+i+"]="+f[i]);
        System.out.println("g.length="+g.length);
        System.out.println("h.length="+h.length);
        e=h;
        System.out.println("e.length="+e.length);
        e=new int[] {1,2};
        System.out.println("e.length="+e.length);
        }
}
```

Como vemos, este ejemplo muestra las distintas maneras de manejar un arreglo. Sin embargo, hay que destacar que los arreglo de objetos que no son inicializados explícitamente, serán inicializados implícitamente en **null**. De manera similar, los arreglos de datos primitivos serán inicializados en **cero** en caso de valores numéricos, y en **false** en caso de bolean.

Regresando un arreglo.

Supongamos que escribimos un método donde queramos que el valor de retorno no sea un dato simple, sino un conjunto de datos. En lenguajes como C o C++ esta tarea es difícil ya que no podemos regresar un arreglo tal cual, sino que lo hacemos con la ayuda de un apuntador al arreglo. Esto provoca una serie de problemas ya que resulta muy fastidioso el control de la vida del arreglo, lo que genera errores de manejo de memoria.

Java permite regresar un arreglo dentro de un método. De hecho, se maneja la referencia a éste de manera similar a C ó C++, sin embargo no tenemos que preocuparnos de la memoria gracias al Recolector de Basura.

Veamos el siguiente ejemplo:

```
//Helado.java
//Regresando arreglos desde métodos.

public class Helado{
        static String[] sabor={ "Chocolate","Fresa","Vainilla",
                                "Menta","Limon"};
        static String[] saborSet(int n){
                n=Math.abs(n)%(sabor.length+1);
```

```
String[] resultados=new String[n];
boolean[] eleccion=new boolean[sabor.length];
for(int i=o;i<n;i++){
        int i;
        do
                t=(int)(Math.random()*sabor.length);
        while(eleccion[t]);
        resultados[i]=sabor[t];
        eleccion[t]=true;
        }
return resultados;
}

public static void main(String[] args){
        for(int i=0;i<20;i++){
                System.out.println("saborSet("+i+")=");
                String[] f1=saborSet(sabor.length);
                for(int j=0;j<f1.length;j++)
                        System.out.println("\t"+f1[j]);
                }
        }
}
```

La clase "Arrays".

En **java.util**, encontraremos la clase **Arrays**, la cual contiene varios métodos **static** para mejorar el manejo de nuestros arreglos en nuestras clases.

Existen cuatro funciones básicas:

1. **equals()**: Compara dos arreglos.
2. **fill()** : Llenar un arreglo con valores.
3. **sort():** Ordenar un arreglo.
4. **binarySearch():** Encontrar un elemento en un arreglo.

Además, existe el método **asList()** que transforma un arreglo en un contenedor de tipo **List.**

Antes de ver los ejemplos de dichos métodos, vamos a crear un generador para cado tipo de dato primitivo dentro de nuestras utilerías:

```
//com.serge.util.Generador.java
package com.serge.util;
public interface Generador{
        Object next();
        }
```

```
//com.serge.util.BooleanGenerador.java
package com.serge.util;
public interface BooleanGenerador{
        boolean next();
        }

//com.serge.util.ByteGenerador.java
package com.serge.util;
public interface ByteGenerador{
        byte next();
        }

//com.serge.util.CharGenerador.java
package com.serge.util;
public interface CharGenerador{
        char next();
        }
//com.serge.util.ShortGenerador.java
package com.serge.util;
public interface ShortGenerador{
        short next();
        }

//com.serge.util.IntGenerador.java
package com.serge.util;
public interface IntGenerador{
        int next();
        }

//com.serge.util.LongGenera dor.java
package com.serge.util;
public interface LongGenerador{
        long next();
        }

//com.serge.util.FloatGenerador.java
package com.serge.util;
public interface FloatGenerador{
        float next();
        }

//com.serge.util.DoubleGenerador.java
package com.serge.util;
public interface DoubleGenerador{
        double next();
        }
```

Ahora veamos la utilería de **Arreglos2** que contiene una diversa variedad de métodos **print()**, sobrecargados para cada tipo.

```
//com.serge.util.Arreglos2.java
package com.serge.util;
import java.util.*;

public class Arreglos2{
	private static void start(int desde, int hasta, int longitud){
		if(desde!=0 || hasta!=longitud)
			System.out.println("["+desde+":"+hasta+"] ");
		System.out.print("(");
		}
	private static void end(){
		System.out.println(")");
		}
	public static void print(Object[] a){
		print(a,0,a.length);
		}
	public static void print(String msg,Object[] a){
		System.out.println(msg+" ");
		print(a,o,a.length);
		}
	public static void print(Object[] a,int desde, int hasta){
		start(desde,hasta,a.length);
		for(int i=desde;i<hasta;i++){
			System.out.print(a[i]);
			if(i<desde -1)
				System.out.print(", ");
			}
		end();
		}
	public static void print(boolean[] a){
		print(a,0,a.length);
		}
	public static void print(String msg,boolean[] a){
		System.out.println(msg+" ");
		print(a,o,a.length);
		}

	public static void print(boolean[] a,int desde, int hasta){
		start(desde,hasta,a.length);
		for(int i=desde;i<hasta;i++){
			System.out.print(a[i]);
```

```
                if(i<desde -1)
                        System.out.print(“, “);
                }
        end();
        }

public static void print(byte[] a){
        print(a,0,a.length);
        }
public static void print(String msg,byte[] a){
        System.out.println(msg+” “);
        print(a,o,a.length);
        }

public static void print(byte[] a,int desde, int hasta){
        start(desde,hasta,a.length);
        for(int i=desde;i<hasta;i++){
                System.out.print(a[i]);
                if(i<desde -1)
                        System.out.print(“, “);
                }
        end();
        }

public static void print(char[] a){
        print(a,0,a.length);
        }
public static void print(String msg,char[] a){
        System.out.println(msg+” “);
        print(a,o,a.length);
        }

public static void print(char[] a,int desde, int hasta){
        start(desde,hasta,a.length);
        for(int i=desde;i<hasta;i++){
                System.out.print(a[i]);
                if(i<desde -1)
                        System.out.print(“, “);
                }
        end();
        }

public static void print(short[] a){
        print(a,0,a.length);
        }
public static void print(String msg,short[] a){
        System.out.println(msg+” “);
```

```
        print(a,o,a.length);
        }

public static void print(short[] a,int desde, int hasta){
        start(desde,hasta,a.length);
        for(int i=desde;i<hasta;i++){
                System.out.print(a[i]);
                if(i<desde -1)
                        System.out.print(“, “);
                }
        end();
        }

public static void print(int[] a){
        print(a,0,a.length);
        }
public static void print(String msg,int[] a){
        System.out.println(msg+” “);
        print(a,o,a.length);
        }

public static void print(int[] a,int desde, int hasta){
        start(desde,hasta,a.length);
        for(int i=desde;i<hasta;i++){
                System.out.print(a[i]);
                if(i<desde -1)
                        System.out.print(“, “);
                }
        end();
        }

public static void print(long[] a){
        print(a,0,a.length);
        }
public static void print(String msg,long[] a){
        System.out.println(msg+” “);
        print(a,o,a.length);
        }

public static void print(long[] a,int desde, int hasta){
        start(desde,hasta,a.length);
        for(int i=desde;i<hasta;i++){
                System.out.print(a[i]);
                if(i<desde -1)
                        System.out.print(“, “);
                }
        end();
```

```
        }

public static void print(float[] a){
        print(a,0,a.length);
        }
public static void print(String msg,float[] a){
        System.out.println(msg+" ");
        print(a,o,a.length);
        }

public static void print(float[] a,int desde, int hasta){
        start(desde,hasta,a.length);
        for(int i=desde;i<hasta;i++){
                System.out.print(a[i]);
                if(i<desde -1)
                        System.out.print(", ");
                }
        end();
        }

public static void print(double[] a){
        print(a,0,a.length);
        }
public static void print(String msg,double[] a){
        System.out.println(msg+" ");
        print(a,o,a.length);
        }

public static void print(double[] a,int desde, int hasta){
        start(desde,hasta,a.length);
        for(int i=desde;i<hasta;i++){
                System.out.print(a[i]);
                if(i<desde-1)
                        System.out.print(", ");
                }
        end();
        }
//Llenando el arreglo con el Generador
public static void fill(Object[] a,Generador gen){
        fill(a,0,a.length,gen);
        }
public static void fill(Object[] a,int desde,int hasta,Generador gen){
for(int i=desde;i<hasta;i++)
        a[i]=gen.next();
}
public static void fill(boolean[] a,BooleanGenerador gen){
        fill(a,0,a.length,gen);
```

```
        }
    public static void fill(boolean[] a,int desde,int hasta,BooleanGenerador
gen){
    for(int i=desde;i<hasta;i++)
        a[i]=gen.next();
    }
    public static void fill(byte[] a,ByteGenerador gen){
        fill(a,0,a.length,gen);
        }
    public static void fill(byte[] a,int desde,int hasta,ByteGenerador gen){
    for(int i=desde;i<hasta;i++)
        a[i]=gen.next();
    }
    public static void fill(char[] a,CharGenerador gen){
        fill(a,0,a.length,gen);
        }
    public static void fill(char[] a,int desde,int hasta,CharGenerador gen){
    for(int i=desde;i<hasta;i++)
        a[i]=gen.next();
    }
    public static void fill(short[] a,ShortGene rador gen){
        fill(a,0,a.length,gen);
        }
    public static void fill(short[] a,int desde,int hasta,ShortGenerador gen){
    for(int i=desde;i<hasta;i++)
        a[i]=gen.next();
    }
    public static void fill(int[] a,IntGenerador gen){
        fill(a,0,a.length,gen);
        }
    public static void fill(int[] a,int desde,int hasta,IntGenerador gen){
    for(int i=desde;i<hasta;i++)
        a[i]=gen.next();
    }
    public static void fill(long[] a,LongGenerador gen){
        fill(a,0,a.length,gen);
        }
    public static void fill(long[] a,int desde,int hasta,LongGenerador gen){
    for(int i=desde;i<hasta;i++)
        a[i]=gen.next();
    }
    public static void fill(float[] a,FloatGenerador gen){
        fill(a,0,a.length,gen);
        }
    public static void fill(float[] a,int desde,int hasta,FloatGenerador gen){
    for(int i=desde;i<hasta;i++)
        a[i]=gen.next();
```

```
        }
        public static void fill(double[] a,DoubleGenerador gen){
                fill(a,0,a.length,gen);
                }
        public static void fill(double[] a,int desde,int hasta,DoubleGenerador
gen){
        for(int i=desde;i<hasta;i++)
                a[i]=gen.next();
        }

        private static Random r=new Random();
        public static class RandBooleanGenerador implements
BooleanGenerador{
                public boolean next(){
                        return r.nextBoolean();
                        }
                }
        public static class RandByteGenerador implements ByteGenerador{
                public byte next(){
                        return (byte)r.nextInt();
                        }
                }
        static String
                sfuente="ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
                "abcdefghijklmnopqrstuvwxyz";
        static char[] src=sfuente.toCharArray();
        public static class RandCharGenerador implements CharGenerador{
                public char next(){
                        int pos=Math.abs(r.nextInt());
                        return src[pos%src.length];
                        }
                }
        public static class RandStringGenerador implements Generador{
                private int len;
                private RandCharGenerador cg=new RandCharGenerador();
                public RandStringGenerador(int length){
                        len=length;
                        }
                public Object next(){
                        char[] buf=new char(len);
                        for(int i=0;i<len;i++)
                                buf[i]=cg.next();
                        return new String(buf);
                        }
                }
        public static class RandShortGene rador implements ShortGenerador{
                public short next(){
```

```
                return (short)r.nextInt();
                }
        }
    public static class RandLongGenerador implements LongGenerador{
        public long next(){
                return r.nextLong();
                }
        }
    public static class RandFloatGenerador implements FloatGenerador{
        public float next(){
                return r.nextFloat();
                }
        }
    public static class RandDoubleGenerador implements
DoubleGenerador{
        public double next(){
                return r.nextDouble();
                }
        }

    public static class RandIntGenerador implements IntGenerador{
        private int mod=10000;
        public RandIntGenerador(){}
        public RandIntGenerador(int modulo){
                mod=modulo;
                }
        public int next(){
                return r.nextInt()%mod;
                }
        }
```

El siguiente ejemplo es de prueba de la utilería recién creada:

```
//TestArreglo2.java
import com.serge.util.*;

public class TestArreglo2{
    public static void main(String[] args){
        int tam=6;
        //o desde la línea de comandos.
        if(args.length!=0)
                tam=Integer.parseInt(args[0]);
        boolean[]a1= new boolean[tam];
        byte[] a2=new byte[tam];
        char[] a3=new char[tam];
        short[] a4=new short[tam];
```

```
        int[] a5=new int[tam];
        long[] a6=new long[tam];
        float[] a7=new float[tam];
        double[] a8=new double[tam];
        String[] a9=new String[tam];
        Arreglos2.fill(a1,new Arreglos2.RandBooleanGenerador());
        Arreglos2.print(a1);
        Arreglos2.print("a1= ",a1);
        Arreglos.print(a1,tam/3,tam/3+tam/3);
        Arreglos2.fill(a2,new Arreglos2.RandBooleanGenerador());
        Arreglos2.print(a2);
        Arreglos2.print("a2= ",a2);
        Arreglos.print(a2,tam/3,tam/3+tam/3);
        Arreglos2.fill(a3,new Arreglos2.RandBooleanGenerador());
        Arreglos2.print(a3);
        Arreglos2.print("a3= ",a3);
        Arreglos.print(a3,tam/3,tam/3+tam/3);
        Arreglos2.fill(a4,new Arreglos2.RandBooleanGenerador());
        Arreglos2.print(a4);
        Arreglos2.print("a4= ",a4);
        Arreglos.print(a4,tam/3,tam/3+tam/3);
        Arreglos2.fill(a5,new Arreglos2.RandBooleanGenerador());
        Arreglos2.print(a5);
        Arreglos2.print("a5= ",a5);
        Arreglos.print(a5,tam/3,tam/3+tam/3);
        Arreglos2.fill(a6,new Arreglos2.RandBooleanGenerador());
        Arreglos2.print(a6);
        Arreglos2.print("a6= ",a6);
        Arreglos.print(a6,tam/3,tam/3+tam/3);
        Arreglos2.fill(a7,new Arreglos2.RandBooleanGenerador());
        Arreglos2.print(a7);
        Arreglos2.print("a7= ",a7);
        Arreglos.print(a7,tam/3,tam/3+tam/3);
        Arreglos2.fill(a8,new Arreglos2.RandBooleanGenerador());
        Arreglos2.print(a8);
        Arreglos2.print("a8= ",a8);
        Arreglos.print(a8,tam/3,tam/3+tam/3);
        Arreglos2.fill(a9,new Arreglos2.RandBooleanGenerador());
        Arreglos2.print(a9);
        Arreglos2.print("a9= ",a9);
        Arreglos.print(a9,tam/3,tam/3+tam/3);

        }
}
```

Método Fill()

El método **fill()** consiste en copiar un valor en todo el arreglo o si es el caso, una referencia. Veamos un ejemplo:

```
//FillArreglo.java
//Uso de Arrays.fill()
import com.serge.util.*;
import java.util.*;

public class FillArreglo{
        public static void main(String[] args){
                int tam=6;
                //o desde la línea de comandos.
                if(args.length!=0)
                        tam=Integer.parseInt(args[0]);
                boolean[]a1= new boolean[tam];
                byte[] a2=new byte[tam];
                char[] a3=new char[tam];
                short[] a4=new short[tam];
                int[] a5=new int[tam];
                long[] a6=new long[tam];
                float[] a7=new float[tam];
                double[] a8=new double[tam];
                String[] a9=new String[tam];
                Arrays.fill(a1,true);
                Arreglos2.print("a1= ",a1);
                Arrays.fill(a2,(byte)11);
                Arreglos2.print("a2= ",a2);
                Arrays.fill(a3,'x');
                Arreglos2.print("a3= ",a3);
                Arrays.fill(a4,(short)17);
                Arreglos2.print("a4= ",a4);
                Arrays.fill(a5,19);
                Arreglos2.print("a5= ",a5);
                Arrays.fill(a6,23);
                Arreglos2.print("a6= ",a6);
                Arrays.fill(a7,29);
                Arreglos2.print("a7= ",a7);
                Arrays.fill(a8,47);
                Arreglos2.print("a8= ",a8);
                Arrays.fill(a9,"Hola");
                Arreglos2.print("a1= ",a9);
                Arrays.fill(a9,3,5,"Mundo");
```

```
            Arreglos2.print("a1= ",a9);
            }
}
```

Método arraycopy().

Dicho método permite la copia de arreglo de una manera rápida que con un ciclo for. El método System.arraycopy está sobrecargado para permitir manipular varios tipos de datos, veamos un ejemplo:

```
//CopiandoArreglos.java
import com.serge.util.*;

public class CopiandoArreglos{
    public static void main(String[] args){
            int[] i=new int[25];
            int[] j=new int[25];
            Arrays.fill(i,47);
            Arrays.fill(j,99);
            Arreglos2.print("i= ",i);
            Arreglos2.print("j= ",j);
            System.arraycopy(i,0,j,0,i.length);
            Arreglos2.print("j= ",j);
            int[] k=new int[10];
            Arrays.fill(k,103);
            System.arraycopy(i,0,k,0,k.length);
            Arreglos2.print("k= ",k);
            Arrays.fill(k,103);
            System.arraycopy(k,0,i,0,k.length);
            Arreglos2.print("i= ",i);
            //Objetos
            Integer[] u=new Integer[10];
            Integer[] v=new Integer[5];
            Arrays.fill(u,new Integer(47));
            Arrays.fill(v,new Integer(99));
            Arreglos2.print("u= ",u);
            Arreglos2.print("v= ",v);
            System.arraycopy(v,0,u,u.length/2,v.length);
            Arreglos2.print("u= ",u);
            }
}
```

Los argumentos del método son: el arreglo de origen, el origen desde donde se desea que se comience la copia, el arreglo destino, el punto desde donde se desea que se copie y el número de elementos a copiar.

Comparando Arreglos.

El método **equals()** compara los elementos de un arreglo. Para ser idénticos deben tener el mismo número de elementos , además que los elementos correspondientes deben ser los mismos. Veamos un ejemplo:

```
//ComparandoArreglos.java
import java.util.*;

public class ComparandoArreglos{
        public static void main(String[] args){
                int[] a1=new int[10];
                int[] a2=new int[10];
                Arrays.fill(a1,47);
                Arrays.fill(a2,47);
                System.out.println(Arrays.equals(a1,a2));
                a2[3]=11;
                System.out.println(Arrays.equals(a1,a2));
                String[] s1=new String[5];
                Arrays.fill(s1,"hola");
                String[] s2={"hola","hola","hola","hola","hola"};
                System.out.println(Arrays.equals(s1,s2));
                }
        }
```

Si los arreglos son iguales, entonces el método regresa el valor de **true**. En caso contrario será **false**.

Ordenando un arreglo.

Podemos ordenar arreglos de datos primitivos.

```
//StringSort.java
import com.serge.util.*;
import java.util.*;

public class StringSort{
        public static void main(String[] args){
                String[] sa=new String[30];
                Arreglos2.fill(sa,new Arreglos2.RandStringGenerador(5));
                Arreglos2.print("Antes de ordenar: ",sa);
                Arrays.sort(sa);
                Arreglos2.print("Después de ordenar: ",sa);
```

```
        }
    }
```

La herramienta de sort() resulta útil. Veamos una utilería de aplicación:

```
//ComparadorAlfabetico.java
package com.serge.util.*;
import java.util.*;

public class ComparadorAlfabetico implements Comparator{
    public int compare(Object o1,Object o2){
        String s1=(String)o1;
        String s2=(String)o2;
        Return s1.toLowerCase().compareTo(s2.toLowerCase());
        }
    }
```

Veamos un ejemplo de aplicación:

```
//OrdenamientoAlfabetico.java
import com.serge.util.*;
import java.util.*;

public class OrdenamientoAlfabetico{
    public static void main(String[] args){
        String[] sa=new String[30];
        Arreglos2.fill(sa,new Arreglos2.RandStringGenerador(5));
        Arreglos2.print("Antes de ordenar: ",sa);
        Arrays.sort(sa,new ComparadorAlfabetico());
        Arreglos2.print("Despues de ordenar: ",sa);
        }
    }
```

El método binarySearch().

Una vez que un arreglo está ordenado, podemos realizar una búsqueda rápida utilizando **Arrays.binarySearch().** Es **MUY** importante que el arreglo esté ordenado, de lo contrario los resultados de dicho método serán impredecibles.

```
//ArraySearching.java
import com.serge.util.*;
import java.util.*;
```

```
public class ArraySearching{
        public static void main(String[] args){
                int[] a=new int[100];
                Arreglos2.RandIntGenerador gen=
                        new Arreglos2.RandIntGenerador(1000);
                Arreglos2.fill(a,gen);
                Arrays.sort(a);
                Arreglos2.print("Arreglo ordenado: ",a);
                while(true){
                        int r=gen.next();
                        int locacion=Arrays.binarySearch(a,r);
                        if(location>=0){
                                System.out.println("locacion de
                                 "+r+"es"+locacion+" , a[ " + locacion+
                                "] = "+a[locacion]);
                                break;
                                }
                        }
                }
                }
```

La clase *String*

Dentro de un objeto de la clases *String* o *StringBuffer*, Java crea un array de caracteres de una forma similar a como lo hace el lenguaje C++. A este array se accede a través de las funciones miembro de la clase.

Los strings u objetos de la clase *String* se pueden crear explícitamente o implícitamente. Para crear un string implícitamente basta poner una cadena de caracteres entre comillas dobles. Por ejemplo, cuando se escribe

```
System.out.println("El primer programa");
```

Java crea un objeto de la clase *String* automáticamente.

Para crear un string explícitamente escribimos

```
String str=new String("El primer programa");
```

También se puede escribir, alternativamente

```
String str="El primer programa";
```

Para crear un string nulo se puede hacer de estas dos formas

```
String str="";
String str=new String();
```

Un string nulo es aquél que no contiene caracteres, pero es un objeto de la clase *String*. Sin embargo,

```
String str;
```

está declarando un objeto *str* de la clase *String*, pero aún no se ha creado ningún objeto de esta clase.

Cómo se obtiene información acerca del string

Una vez creado un objeto de la clase *String* podemos obtener información relevante acerca del objeto a través de las funciones miembro.

Para obtener la longitud, número de caracteres que guarda un string se llama a la función miembro *length*.

```
String str="El primer programa";
int longitud=str.length();
```

Podemos conocer si un string comienza con un determinado prefijo, llamando al método *startsWith*, que devuelve **true** o **false**, según que el string comience o no por dicho prefijo

```
String str="El primer programa";
boolean resultado=str.startsWith("El");
```

En este ejemplo la variable resultado tomará el valor **true**.

De modo similar, podemos saber si un string finaliza con un conjunto dado de caracteres, mediante la función miembro *endsWith.*

```
String str="El primer programa";
boolean resultado=str.endsWith("programa");
```

Si se quiere obtener la posición de la primera ocurrencia de la letra p, se usa la función *indexOf*.

```
String str="El primer programa";
int pos=str.indexOf('p');
```

Para obtener las sucesivas posiciones de la letra p, se llama a otra versión de la misma función

```
pos=str.indexOf('p', pos+1);
```

El segundo argumento le dice a la función *indexOf* que empiece a buscar la primera ocurrencia de la letra p a partir de la posición *pos+1*.

Otra versión de *indexOf* busca la primera ocurrencia de un substring dentro del string.

```
String str="El primer programa";
int pos=str.indexOf("pro");
```

Vemos que una clase puede definir varias funciones miembro con el mismo nombre pero que tienen distinto número de parámetros o de distinto tipo.

Comparación de strings

La comparación de strings nos da la oportunidad de distinguir entre el operador lógico == y la función miembro *equals* de la clase *String.* En el siguiente código

```
String str1="El lenguaje Java";
String str2=new String("El lenguaje Java");
    if(str1==str2){
            System.out.println("Los mismos objetos");
    }else{
            System.out.println("Distintos objetos");
    }
    if(str1.equals(str2)){
            System.out.println("El mismo contenido");
    }else{
            System.out.println("Distinto contenido");
    }
```

Esta porción de código devolverá que *str1* y *str2* son dstintos objetos pero con el mismo contenido. *str1* y *str2* ocupan posiciones distintas en memoria pero guradan los mismos datos.

Cambiemos la segunda sentencia y escribamos

```
String str1="El lenguaje Java";
String str2=str1;
    System.out.prinln("Son el mimso objeto "+(str1==str2);
```

Los objetos *str1* y *str2* guardan la misma referencia al objeto de la clase *String* creado. La expresión *(str1==str2*) devolverá **true**.

Asñi pues, el método *equals* compara un string con un objeto cualquiera que puede ser otro string, y devuelve **true** cuando dos strings son iguales o **false** si son distintos.

```
String str="El lenguaje Java";
    boolean resultado=str.equals("El lenguaje Java");
```

La variable *resultado* tomará el valor **true.**

La función miembro *comapareTo* devuelve un entero menor que cero si el objeto string es menor (en orden alfabético) que el string dado, cero si son iguales, y mayor que cero si el objeto string es mayor que el string dado.

```
String str="Tomás";
int resultado=str.compareTo("Alberto");
```

La variable entera *resultado* tomará un valor mayor que cero, ya que Tomás está después de Alberto en orden alfabético.

```
String str="Alberto";
int resultado=str.compareTo("Tomás");
```

La variable entera *resultado* tomará un valor menor que cero, ya que Alberto está antes que Tomás en orden alfabético.

Extraer un substring de un string

En muchas ocasiones es necesario extraer una porción o substring de un string dado. Para este propósito hay una función miembro de la clase *String* denominada *substring*.

Para estraer un substring desde una posición determinada hasta el final del string escribimos

```
String str="El lenguaje Java";
    String subStr=str.substring(12);
```

Se obtendrá el substring "Java".

Una segunda versión de la función miembro *substring*, nos permite extraer un substring especificando la posición de comienzo y la el final.

```
String str="El lenguaje Java";
    String subStr=str.substring(3, 11);
```

Se obtendrá el substring "lenguaje". Recuérdese, que las posiciones se empiezan a contar desde cero.

Convertir un número a string

Para convertir un número en string se emplea la función miembro estática *valueOf* (más adelante explicaremos este tipo de funciones).

```
int valor=10;
String str=String.valueOf(valor);
```

La clase *String* proporciona versiones de *valueOf* para convertir los datos primitivos: **int**, **long**, **float**, **double**.

Esta función se emplea mucho cuando programamos applets, por ejemplo, cuando queremos mostrar el resultado de un cálculo en el área de trabajo de la ventana o en un control de edición.

Convertir un string en número

Cuando introducimos caracteres en un control de edición a veces es inevitable que aparezcan espacios ya sea al comienzo o al final. Para eliminar estos espacios tenemos la función miembro *trim*

```
String str="  12 ";
String str1=str.trim();
```

Para convertir un string en número entero, primero quitamos los espacios en blanco al principio y al final y luego, llamamos a la función miembro estática *parseInt* de la clase *Integer* (clase envolvente que describe los números enteros)

```
String str="  12 ";
int numero=Integer.parseInt(str.trim());
```

Para convertir un string en número decimal (**double**) se requieren dos pasos: convertir el string en un objeto de la clase envolvente *Double*, mediante la función miembro estática *valueOf*, y a continuación convertir el objeto de la clase *Double* en un tipo primitivo **double** mediante la función *doubleValue*

```
String str="12.35 ";
double numero=Double.valueOf(str).doubleValue();
```

Se puede hacer el mismo procedimiento para convertir un string a número entero

```
String str="12";
int numero=Integer.valueOf(str).intValue();
```

La clase *StringBuffer*

En la sección dedicada a los operadores hemos visto que es posible concatenar cadenas de caracteres es, decir, objetos de la clase *String*. Ahora bien, los objetos de la clase *String* son constantes lo cual significa que por defecto, solamente se pueden crear y leer pero no se pueden modificar.

Imaginemos una función miembro a la cual se le pasa un array de cadenas de caracteres. Los arrays se estudiarán en la siguiente página.

```
public class CrearMensaje{
        public String getMensaje(String[] palabras){
                String mensaje="";
                for(int i=0; i<palabras.length; i++){
                        mensaje+=" "+palabras[i];
                }
                return mensaje;
        }
//...

}
```

Cada vez que se añade una nueva palabra, se reserva una nueva porción de memoria y se desecha la vieja porción de memoria que es más pequeña (una palabra menos) para que sea liberada por el recolector de basura (grabage collector). Si el bucle se realiza 1000 veces, habrá 1000 porciones de memoria que el recolector de basura ha de identificar y liberar.

Para evitar este trabajo extra al recolector de basura, se puede emplear la clase *StringBuffer* que nos permite crear objetos dinámicos, que pueden modificarse.

```
public class CrearMensaje{
        public String getMensaje(String[] palabras){
                StringBuffer mensaje=new StringBuffer();
                for(int i=0; i<palabras.length; i++){
                        mensaje.append(" ");
                        mensaje.append(palabras[i]);
                }
                return mensaje.toString();
        }
}
```

La función *append* incrementa la memoria reservada para el objeto *mensaje* con una palabra más sin crear nueva memoria, cada vez que se ejecuta el bucle. La función *toString*, que veremos más adelante, convierte un objeto en una cadena de caracteres.

A continuación se agregan algunas tablas relacionadas con el objeto String:

Declaración y Constructores

Declaración de instancias de la clase *string*	String s="abc";
a partir de un array de caracteres	char cC[ ]={'a','b','c'}; String sC=new String(cC); System.out.println(sC); // abc

indicando parte del array	char cC[ ]={'a','b','c','d','e','f'};
	Strig sC=new String(cC,2,3);
	// tres caracteres a partir del índice 2
	System.out.println(sC); // cde

forzando utilización de ASCII	char cC[ ]={'a','b','c','d','e','f'};
	String sC1=new String(cC,0)
	String sC2=new String(cC,0,2,3);
	System.out.println(sC1); //abcdef
	System.out.println(sC2); //cde

a partir de un literal u otro string	String sC="Hola Mundo"; String sM=new String(sC); System.out.println(sC); //Hola Mundo

StringBuffer	StringBuffer sbMibuffer=new StringBuffer()
	// vacío
	StringBuffer sbMibuffer=new
	StringBuffer("Hola Mundo")
	// Copia un string
	StringBuffer sbMibuffer=new StringBuffer(50)
	// de una determinada longitud

Es posible convertir cualesquiera objeto a un String mediante el método ***toString***.	Object.toString()
Igualmente el método ***valueOf*** de String esta sobrecargado en todas las clases y mediante llamada a toString de la clase correspondiente realiza el mismo trabajo de conversión a string del	

Objeto

Extracción de Información

Información General

Los siguientes métodos retornan la longitud de la cadena (***length()***) o la capacidad del buffer(***StringBuffer.capacity***).	public int String.length() public int StringBuffer..length() public int StringBuffer.capacity();

Búsquedas

Comparan el carácter dado por ***ch*** y retornan su posición en la cadena, o -1 si no se encuentra, las versiones de dos parámetros comienzan la búsqueda desde la posición dada por ***Index***.	public int String.indexOf(int ch);
	public int String indexOf(int ch, int Index);
	public int String.lastIndexOf(int ch);
	public int String lastIndexOf(int ch, int Index);

Estos métodos se suelen sobrecargar para comparar cadenas de caracteres:, aunque otros métodos (***regiónMaches***) realizan esta tarea de forma mas completa.	public int String.indexOf(String chs);
	public int String indexOf(String chs, int Index);
	public int String.lastIndexOf(String chs);
	public int String lastIndexOf(String chs, int Index);

Extracción de Caracteres

Los métodos ***charAt*** permiten consultar el carácter en una determinada posición dado por el índice ***Index***. Los métodos getCharts permiten disponer en un array de caracteres parte de los caracteres del String, la posición de comienzo en al array es la indicada por ***PrimIndex***.

```
String.charAt(int Index);
public void String.getCharts
      (int iniIndex, int endIndex, char dst[ ], int
PrimIndex);
public chart StringBuffer.charAt(int Index);
Public void StringBuffer.getCharts
    (int iniIndex, int endIndex, char dst[ ], int
PrimIndex);
```

```
class getChartsDemo {
  public static void main(String args[ ]) {
  String sC="Esto es una demostración";
  int start=12;
  int end=16;
  char buf[ ]=new char[end-start];
  sC.getChars(start,end,buf,0)
  System.out.println(buf); //demo
  }
}
```

Otro método, ***toCharArray*** convierte el String en un array de caracteres con la cadena completa

```
public char[ ] String.toCharArray()
```

El método *getBytes* opera igual que getCharts, pero disponiendo el resultado en un array de Bytes:

```
public void String.getBytes
  (int iniIndex, int endIndex,
  byte dst[ ], int PrimIndex);
```

Comparación

Esta colección de métodos permite comparar objetos string y comprobar su igualdad en longitud, igualdad de caracteres y su situación (***equals***) con o sin importar mayúsculas y minúsculas (***equalsIgnoreCase***),

```
public boolean String.equals(Object obj)
public boolean String.equalsIgnoreCase(String
str)
public int String.compareTo(String str)
public boolean String.regionMatches
    (int iniIndex, String str, intIndexOtr, int
Leng)
public boolean String.regionMatches
    (boolean IgnCase,int iniIndex,
```

permiten la comparación de Strings (***compareTo*** retorna 0 en caso de igualdad, < 0 si str es mayor y > 0 si es menor). También se dispone de métodos que permiten comprobar regiones particulares (***regionMatche***s) y el comienzo o final (***StartsWith***) y (*EndWith*).

```
String str, intIndexOtr, int Leng)
public boolean String.startsWith(String Prefix)
public boolean String.endsWith(String Subfix)
```

El operador ==, permite la comparación de String, este retornara *true* solo si los dos objetos comparados son el mismo

```
class equalsDemo {
 public static void main(String args[ ]) {
   String s1="Hola";
   String s2="Hola";
   String s3="Adios";
   String s4="HOLA";
   String s5 =new String(s1);
   System.out.println("s1 equals s2 -
>",s1.equals(s2));
                //s1 equals s2 -> true
   System.out.println("s1 equals s3 -
>",s1.equals(s3));
                //s1 equals s3 -> false
   System.out.println("s1 equals s4 -
>",s1.equals(s4));
                //s1 equals s4 -> false
   System.out.println
        ("s1 equalsIgnoreCase s2 -
>",s1.equalsIgnoreCase(s4));
              //s1 equalsIgnoreCase s4 -> true
   System.out.println("s1 equals s5 -
>",s1.equals(s5));
            //s1 equals s5 -> true
   System.out.println("s1 == s5 ->",s1==s5); //s1
== s5 -> false
  }
}
```

Ordenación

Este método permiten la comparación de

```
public int String.compareTo(String str)
```

Strings (***compareTo*** retorna 0 en caso de igualdad, < 0 si str es mayor y > 0 si es menor).

```
class SortStringDemo {
 static String arr[] = { "Ahora", "es", "el", "momento"};
 public static void main(String args[ ]) {
    for (int j=0; j<arr.length; j++) {
       for (int i=j+1;i<arr.length; i++) {
         if (arr[i].compareTo(arr[j]<0) {
            String t=arr[j];
           arr[j]=arr[i];
           arr[i]=t;
         }
        }
        System.out.println(arr[j]);
     }
   }
}
```

Modificaciones

Concatenación de Cadenas

Operador +

```
String sM="Hola "+"Mundo"

String sC="El tiene "+2+" años"
System.out.println(sC); //El Tiene 2 años
String sC="cuatro= "+2+2
System.out.println(sC); //cuatro=22
String sC="cuatro= "+(2+2)
System.out.println(sC); //cuatro=4
```

Cuidado con la precedeencia de operadores, la concatenación se realiza antes de la suma de entero

Método *Appen*

El método ***append*** se aplica a objetos del tipo ***StringBuffer***, su forma general es ***StringBuffer.append(Object obj)***; donde la forma String del *obj* (String.ValueOf) se añade al final del

```
StringBuffer sbM=nw StringBuffer("Hola");
sbM.append(" Mundo ");
sbM.append(2);
System.out.println(sbM); //Hola Mundo2
```

StringBuffer

```
class appendDemo {
  public static void main (String args[
]) {
    String s;
    int a=42;
    StringBuffer sbB= new
StringBuffer(40);
   s=sbB.append("a=").append(a).app
end('!').toString();
    System.out.println(s);
            // la salida será : a=42!
  }
}
```

El string vacío s, creado al principio se borrara y se creara uno nuevo con el valor de la expresión dada.

Inserciones

El método ***insert*** se aplica a objetos del tipo ***StringBuffer***, su forma general es ***StringBuffer.insert(int Index, Object obj)***;
la forma String del ***obj*** (String.ValueOf) se inserta en el índice ***Index***, desplazando el resto hacia la derecha.

```
class insertDemo {
  public static void main(String args[
]) {
    StringBuffer sb=new
StringBuffer("hola mundo!");
    sb.insert(5,"buen ");
    System.out.println(sb) //hola buen
mundo
  }
}
```

Modificaciones de caracteres

Se pueden extraer caracteres particulares y modificarles con los métodos ***StringBuffer.charAt, StringBuffer.charAt*** y ***StringBuffer.setCharAt***:

```
String.charAt(int Index)
StringBuffer.charAt(int Index)
```

El carácter en la posición *Index* es retornado o es cambiado al valor dado por *ch.*

```
StringBuffer.setCharAt(int Index,
char ch)
```

```
class setCharDemo {
```

```
 public static void main(String args[
]) {
   StringBuffer sb=new
StringBuffer("Hola");
   System.out.println("Antes: "+sb);
       //Antes: Hola
   System.out.println("Carácter 1:
"+sb.charAt(1)); // o
   sb.setCharAt(1,'i');
   System.out.println("Después:
"+sb);
     //Antes: Hila
   System.out.println("Carácter 1:
"+sb.charAt(1)); // i
 }
}
```

Modificaciones de Tamaño

El método *StringBuffer.setLength* permite cambiar el tamaño de un String. El tamaño del string cambia al valor dado por *newLength*.

```
StringBuffer.setLength(int
newLength)
```

```
class setLengthDemo {
 public static void main(String args[
]) {
   StringBuffer sb=new
StringBuffer("Hola");
   System.out.println("Antes: "+sb);
       //Antes: Hola
   sb.setLength(2);
   System.out.println("Después:
"+sb);
       //Antes: Ho
 }
}
```

el método *StringBuffer.ensureCapacity*, permite cambiar la capacidad de un String después de creado y asegura un tamaño mínimo de *MinCapacity*.

```
StringBuffer.ensureCapacity(int
MinCapacity);
```

Creación de String por modificaciones

en otro String

Una serie de métodos de String permiten crear String a partir de los datos del original, por modificación. en todos los casos se aplica la operación indicada y se crea un nuevo String para disponer el resultado.

```
String.substring(int IndexIni, int
IndexEnd);
String.concat(String str);
String.replace(Char OldChar, Char
NewChar);
String.toLowercase();
String.toUpperCase();
String.trim();
```

Implementación del método *toString*

9 Polimorfismo e Interfaces

El polimorfismo es la tercera característica esencial de un lenguaje orientado a objetos. Es una herramienta que provee otra dimensión de interfaz, capaz de separar el "qué" del "cómo".

En el capítulo pasado aprendimos que con la herencia podemos tratar un objeto como “tipo de” su clase padre. Esta habilidad es esencial porque permite que objetos de varios tipos puedan ser tratados como objetos de un solo tipo, todo en el mismo código para trabajar con todos los distintos tipos de manera similar. El polimorfismo nos permite que distinguir entre los tipos de objetos que son similares. Dicha diferencia se expresa a través del comportamiento de los métodos que se heredan de la clase padre.

Recordatorio de Upcasting.

Veamos un ejemplo para recordar el concepto de upcasting:

```
//Musica.java
//Herencia y upcasting.

class Nota{
        private int valor;
        private Nota(int val){ valor=val;}
        public static final Nota
                MIDDLE_C=new Nota(0),
                C_SHARP=new Nota(1),
                B_FLAT=new Nota(2);
                }
class Instrumento{
        public void play(Nota n){
                System.out.println(“Instrumento.play()”);
                }
        }
//Los instrumentos de viento son también tipo Instrumento!
//porque tienen el mismo comportamiento.
class Viento extends Instrumento{
        //Redefiniendo un método
        public void play(Nota n){
                System.out.println(“Viento.play()”);
                }
        }
public class Musica{
        public static void tonada(Instrumento i){
                i.play(Nota.MIDDLE_C);
                }
        public static void main(String[] args){
                Viento flauta=new Viento();
                tonada(flauta); //Upcasting
                }
        }
```

El método **Musica.tonada**() acepta un objeto **Instrumento** como referencia, pero también objetos derivado de éste. Por eso es que cualquier objeto **Viento** es aceptable para dicho método, ya que **Viento** hereda de **Instrumento**.

Ahora, veamos el siguiente programa:

```
//Musica2.java
//Sobrecargando en vez de hacer upcasting

class Nota{
        private int valor;
        private Nota(int val){ valor=val;}
        public static final Nota
                MIDDLE_C=new Nota(0),
                C_SHARP=new Nota(1),
                B_FLAT=new Nota(2);
                }

class Instrumento{
        public void play(Nota n){
                System.out.println("Instrumento.play()");
                }
        }
class Viento extends Instrumento{
        //Redefiniendo un método
        public void play(Nota n){
                System.out.println("Viento.play()");
                }
        }
class Cuerda extends Instrumento{
        //Redefiniendo un método
        public void play(Nota n){
                System.out.println("Cuerda.play()");
                }
        }
class Percusion extends Instrumento{
        //Redefiniendo un método
        public void play(Nota n){
                System.out.println("Percusion.play()");
                }
        }
public class Musica2{
        public static void tonada(Viento v){
                v.play(Nota.MIDDLE_C);
                }
        public static void tonada(Cuerda c){
                c.play(Nota.MIDDLE_C);
```

```
        }
public static void tonada(Percusion p){
        p.play(Nota.MIDDLE_C);
        }
public static void main(String[] args){
        Viento f=new Viento();
        Cuerda c=new Cuerda();
        Percusion p=new Percusion();
        tonada(f);
        tonada( c );
        tonada( p );
        }
}
```

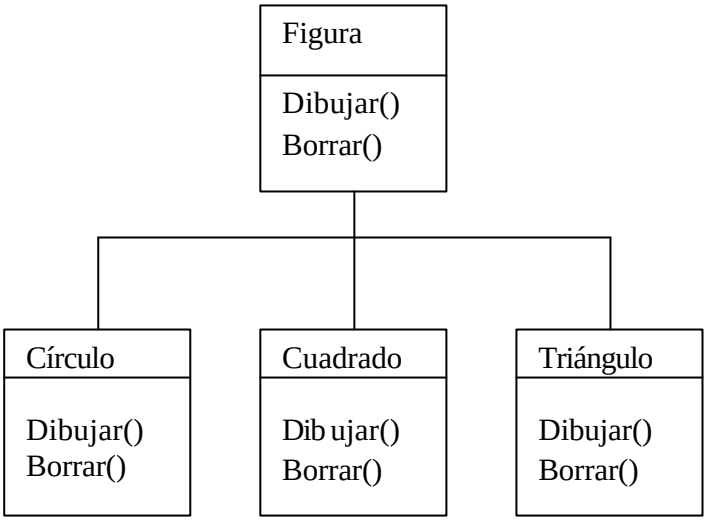
Como podemos observar, en este ejemplo creamos un método para cada tipo de objeto distinto. Nuestra primera impresión es que dicho código resulta bastante largo debido a que repetimos una y otra vez la misma función. Además resulta muy estático, dado que una vez que creemos un nuevo objeto que herede de la clase **Instrumento**, tendremos que agregar una nueva función. ¿ No sería más sencillo escribir un solo método que tome como argumento a la clase base sin especificar el tipo de objeto en particular ?.

Regresemos al ejemplo de **Musica.java**. Es claro que la salida del programa es **Viento.play**(), ya que el método **tonada**() recibe como argumento un objeto **Instrumento**. Pero, ¿ cómo puede saber el compilador que el objeto que recibe es **Viento** y no de otro tipo? La respuesta es la manera en que trabaja Java: ***binding*** (atadura).

Expliquemos a grandes rasgos de en qué consiste este concepto. Ligar la llamada de un método con el cuerpo de un método se llama atar (bind). Cuando el compilador realiza dicha acción antes de ejecutar un código, se le conoce como **early binding**. Java utiliza una alternativa llamada **late binding**, que significa que dicha atadura ocurre en tiempo de compilación. A esta alternativa de **binding** se le conoce también como **dynamic binding**. Todos los métodos en Java utilizan dicha técnica a menos que sean declarados final. Dicho de otra forma, Java es capaz de reconocer el tipo de objeto que será utilizado en un método gracias al **dynamic binding**.

Produciendo el comportamiento correcto.

Veamos un ejemplo clásico de la OOP. Consideremos la clase **Figura** y sus clases hijas **Círculo**, **Cuadrado** y **Triángulo**. Este ejemplo es bastante explícito ya que es bastante lógico decir: " *un círculo es una figura*". El siguiente diagrama muestra las relaciones:



La frase anterior puede ser codificada correctamente de la siguiente manera:

Figura f = new Cuadrado();

En este ejemplo, un objeto **Cuadrado** es creado y el resultado es una referencia asignada a **Figura**, lo cual parece incorrecto, sin embargo, y desde la perspectiva anterior, "*un Cuadrado es una figura*", por lo que en realidad la sentencia es correcta.

Ahora, supongamos que llamamos un método de la clase padre:

f.Dibujar();

podría suponerse que el método **Dibujar**() de **Figura** es llamado, sin embargo, es el método de **Cuadrado** el que es llamado gracias al polimorfismo. Ahora veamos el ejemplo en código Java:

```
//Figuras.java
//Polimorfismo en Java

class Figura{
        void dibujar(){}
        void borrar(){}
        }
class Circulo extends Figura{
        void dibujar(){
                System.out.println("Circulo.dibujar()");
                }
        void borrar(){
                System.out.println("Circulo.borrar()");
                }
```

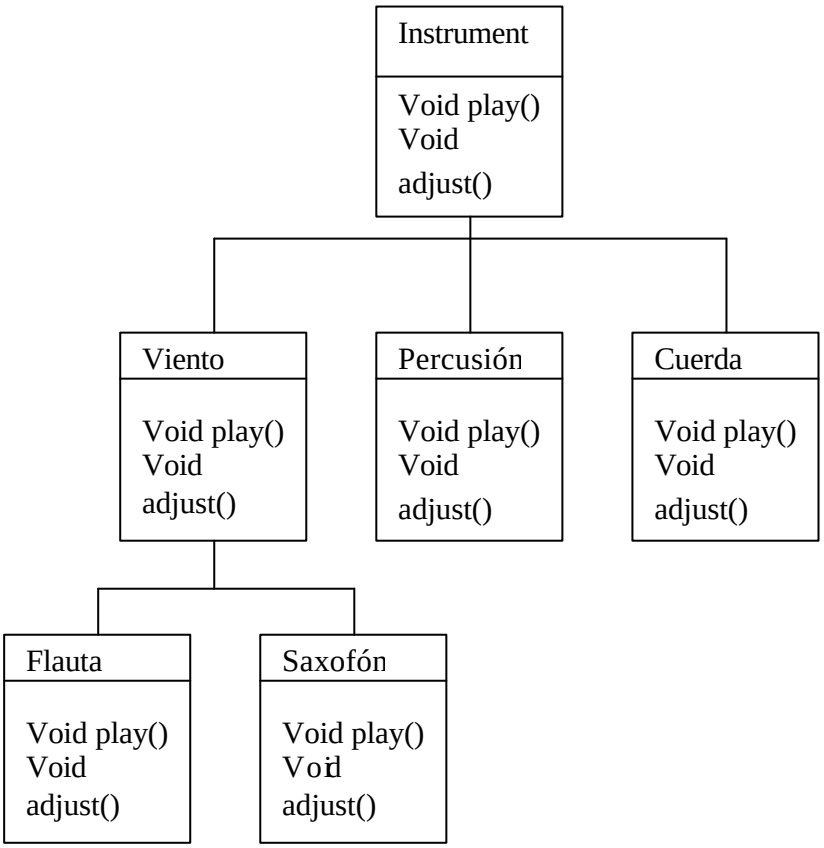
```
        }
class Triangulo extends Figura{
        void dibujar(){
                System.out.println("Triangulo.dibujar()");
                }
        void borrar(){
                System.out.println("Triangulo.borrar()");
                }
        }
class Cuadrado extends Figura{
        void dibujar(){
                System.out.println("Cuadrado.dibujar()");
                }
        void borrar(){
                System.out.println("Cuadrado.borrar()");
                }
        }

public class Figuras{
        public static Figura randFigura(){
                switch((int)(Math.random()*3)){
                        default:
                                case 0:
                                        return new Circulo();
                                case 1:
                                        return new Cuadrado();
                                case 2:
                                        return new Triangulo();
                                }
                }
        public static void main(String[] args){
                Figura[] s=new Figura[9];
                //llenamos el arreglo de figures!
                for(int i=0;i<s.length;i++)
                        s[i]=randFigura();
                //Llamadas polimorficas
                for(int i=0;i<s.length;i++)
                        s[i].dibujar();
                }
        }
```

La clase padre establece el comportamiento de las clases que la heredan. Es decir, todas las subclases pueden borrar y dibujar. Cada subclase establece su comportamiento particular para dichos métodos. En el ejemplo, se crean objetos de manera aleatoria y se almacenan en un arreglo de objetos **Figura**.

Extensibilidad.

Gracias al polimorfismo, se pueden agregar cuantos tipos de clases nuevas queramos al sistema sin necesidad de redefinir métodos. A dicho modelo se le conoce con el nombre de extensible, ya que podemos agregar nuevos tipos de clases heredando de las existentes sin necesidad de cambiar los métodos de las clases padre. Veamos el ejemplo de Música:



Las nuevas clases funcionan de manera correcta, gracias al comportamiento de herencia y de polimorfismo. Cada una podrá tener su comportamiento propio para sus funciones específicas. Ahora veamos le implementación en código del programa:

```
//Musica3.java
//Un programa extensible
import java.util.*;
```

```
class Instrumento{
        public void play(){
                System.out.println("Instrumento.play()");
                }
        public String quienSoy(){
                return "Instrumento";
                }
        public void adjust(){}
        }
class Viento extends Instrumento{
        public void play(){
                System.out.println("Viento.play()");
                }
        public String quienSoy(){
                return "Viento";
                }
        public void adjust(){}
        }
class Percusion extends Instrumento{
        public void play(){
                System.out.println("Percusion.play()");
                }
        public String quienSoy(){
                return "Percusion";
                }
        public void adjust(){}
        }
class Cuerda extends Instrumento{
        public void play(){
                System.out.println("Cuerda.play()");
                }
        public String quienSoy(){
                return "Cuerda";
                }
        public void adjust(){}
        }
class Flauta extends Viento{
        public void play(){
                System.out.println("Flauta.play()");
                }
        public String quienSoy(){
                return "Flauta";
                }
        public void adjust(){ System.out.println("Flauta.adjust()");}
        }
class Saxofon extends Viento{
        public void play(){
```

```
            System.out.println("Saxofon.play()");
            }
        public String quienSoy(){
            return "Saxofon";
            }
        }

    public class Musica3{
        static void tune(Instrumento i){
            i.play();
            }
        static void tuneAll(Instrumento[] e){
            for(int i=0;i<e.length;i++)
                tune(e[i]);
            }
        public static void main(String[] args){
            Instrumento[] orquesta=new Instrumento[5];
            int i=0;
            orquesta[i++]=new Viento();
            orquesta[i++]=new Percusion();
            orquesta[i++]=new Cuerda();
            orquesta[i++]=new Flauta();
            orquesta[i++]=new Saxofon();
            tuneAll(orquesta);
            }
        }
```

En este caso el ejemplo demuestra que es más que evidente que el método tuneAll() es capaz de identificar que objeto está recibiendo cada vez.

El polimorfismo es una de las técnicas más importantes que permiten separar "las cosas que cambien de aquellas que no lo hacen".

Overloading.

Como sabemos, Java nos permite sobrecargar métodos sin ningún problema. Sin embargo, el comportamiento puede resultar inadecuado:

```
//ErrorDeViento.java
//Cambiando el comportamiento!

class NotaX{
    public static final int MIDDLE_C=0,C_SHARP=1,C_FLAT=2;
}
class InstrumentoX{
    public void play(int NotaX){
        System.out.println("InstrumentoX.play()");
        }
```

```
        }
class VientoX extends InstrumentoX{
        //OOPS, cambiamos la interfaz!
        public void    play(NotaX n){
                System.out.println("VientoX.play()");
                }
        }
public class ErrorDeViento{
        public static void tune(InstrumentoX i){
                i.play(NotaX.MIDDLE_C);
                }

        public static void main(String[] args){
                VientoX flauta=new VientoX();
                tune(flauta);
        }
}
```

La clase **InstrumentoX** tiene un método **play**() que toma un entero con el nombre de **NotaX**. Sin embargo, el error radica en que la clase **Viento** tiene el mismo método pero con argumento un objeto **NotaX**.

Clases Abstractas.

En los ejemplos anteriores, podemos observar que los métodos de las clases base no tienen función alguna. Esto significa que sólo establecen el comportamiento de las clases derivadas. Por ejemplo, la clase **Instrumento** sólo nos especifica **que** un instrumento puede tocarse y ajustarse, pero no nos especifica **cómo**. Es decir, Instrumento es una ***clase abstracta***.

Creamos una clase abstracta cuando queremos especificar un comportamiento muy general, para después especificar con cada subclase. Es decir, sólo creamos la interfaz sin ninguna implementación en particular.

Java provee un mecanismo para crear un método abstracto. Un método abstracto es aquel que sólo es declarado y no contiene sintaxis para el cuerpo del método. La sintaxis es:

```
abstract void f();
```

Si una clase posee una o más funciones abstractas, entonces la clase también es abstracta (si no el compilador envía un mensaje de error).

Cuando se hereda de una clase abstracta, debe implementarse el body para cada uno de los métodos de la clase padre, de lo contrario, la clase hija también será abstracta. Veamos el ejemplo de la orquesta con clases abstractas:

***Nota: Una clase abstracta puede no tener ningún método abstracto.**

```
//Musica4.java
//Clases abstractas y métodos.
```

```
import java.util.*;
abstract class Instrumento{
        int i;
        public abstract void play();
        public String quienSoy(){
                return “Instrumento”;
                }
        public abstract void adjust();
        }
class Viento extends Instrumento{
        public void play(){
                System.out.println(“Viento.play()”);
                }
        public String quienSoy(){
                return “Viento”;
                }
        public void adjust(){}//Se implementa el body
        }
class Percusion extends Instrumento{
        public void play(){
                System.out.println(“Percusion.play()”);
                }
        public String quienSoy(){
                return “Percusion”;
                }
        public void adjust(){}
        }
class Cuerda extends Instrumento{
        public void play(){
                System.out.println(“Cuerda.play()”);
                }
        public String quienSoy(){
                return “Cuerda”;
                }
        public void adjust(){}
        }
class Flauta extends Viento{
        public void play(){
                System.out.println(“Flauta.play()”);
                }
        public String quienSoy(){
                return “Flauta”;
                }
        public void adjust(){ System.out.println(“Flauta.adjust()”);}
        }
class Saxofon extends Viento{
```

```
        public void play(){
                System.out.println("Saxofon.play()");
                }
        public String quienSoy(){
                return "Saxofon";
                }
        }

public class Musica4{
        static void tune(Instrumento i){
                i.play();
                }
        static void tuneAll(Instrumento[] e){
                for(int i=0;i<e.length;i++)
                        tune(e[i]);
                }
        public static void main(String[] args){
                Instrumento[] orquesta=new Instrumento[5];
                int i=0;
                orquesta[i++]=new Viento();
                orquesta[i++]=new Percusion();
                orquesta[i++]=new Cuerda();
                orquesta[i++]=new Flauta();
                orquesta[i++]=new Saxofon();
                tuneAll(orquesta);
                }
        }
```

Realmente no existe mucho cambio en el código, a excepción de lo mencionado anteriormente sobre clases y métodos abstractos.

Diseñando con Herencia.

Una vez que hemos visto este nuevo concepto de Polimorfismo parecería que todo debe de hacerse con los conceptos de herencia, polimorfismo, etc. Sin embargo, muchas veces estos conceptos nos complican demasiado la solución de un problema innecesariamente. Una mejor alternativa es utilizar composición, ya que es más flexible porque nos permite dinámicamente escoger el tipo de nuestro objeto. Veamos un ejemplo:

```
//Cambiando.java
//Cambiando dinámicamente el comportamiento de un objeto

abstract class Actor{
        abstract void act();
        }
class ActorFeliz extends Actor{
        public void act(){
                System.out.println("=)");
                }
        }
class ActorTriste extends Actor{
        public void act(){
                System.out.println("=(");
                }
        }
class Escenario{
        Actor a=new ActorFeliz ();
        void cambia(){ a=new ActorTriste ();}
        void go(){a.act();}
        }

public class Cambiando{
        public static void main(String[] args){
                Escenario s=new Escenario();
                s.go();
                s.cambia();
                s.go();
                }
        }
```

Downcasting.

Hemos visto que con el **upcasting** somos capaces de manipular objetos de manera segura ya que la clase hija siempre tiene al menos el comportamiento de la clase padre. Por ejemplo, podemos asegurar que un *"círculo es una figura"*, sin embargo el **downcasting** no siempre es lógico, no podemos afirmar que *"una figura es un círculo"*.

En algunos lenguajes como C++ debes efectuar una operación especial para saber si el **downcast** es seguro, pero en Java cada **cast** es verificado. Esta acción recibe el nombre de *"**Run time type Identification**"* o también conocido como **RTTI**. Veamos un ejemplo de **RTTI**:

```
//RTTI.java
//Downcasting y RTTI
import java.util.*;

class Util{
```

```
        public void f(){}
public void g(){}
}
class MasUtil extends Util{
        public void f(){}
        public void g(){}
        public void u(){}
        public void v(){}
        public void w(){}
        }
public class RTTI{
        public static void main(String[] args){
                Util[] x={ new Util(), new MasUtil()};
                x[0].f();
                x[1].g();
                ((MasUtil)x[1]).u();//Downcast
                ((MasUtil)x[0]).u();//Excepcion
                }
        }
```

Ejercicios.

1. **Crear la siguiente jerarquía de herencia. Clase padre: Roedor. Clases hijas: Ratón , Hamster, Ardilla , etc. Cree métodos de comportamiento para la clase padre de tal manera que sea abstracta, para que las clases hijas puedan implementar dicho comportamiento.**
2. **Modifique el ejemplo de Música3.java de tal manera que genere objetos Instrumento al azar , tal como lo hace la clase Figuras.**
3. **Añada un nuevo tipo a Música3.java y verifique la validez del polimorfismo.**
4. **Añada un nuevo tipo a Figuras.java y verifique la validez del polimorfismo.**

Interfaces y Clases internas.

Las interfaces y clases internas proven un mecanismo más sofisticado de organizar y controlar los objetos de nuestro sistema.

En el capítulo anterior vimos lo que significa el concepto de abstracción, el cual nos permite crear uno o más métodos en una clase sin definición. Cuando creamos una "**interface**" , en realidad creamos una clase completamente abstracta, que no tiene implementación alguna. Aprenderemos que una interfaz nos permite en realidad realizar muchas más cosas, como la llamada "***herencia múltiple***".

Por otra parte, veremos el comportamiento de clases internas, aunque no profundizaremos en dicho tema. A grandes rasgos , una clase interna nos permite crear códigos más elegantes y claros.

Interfaces.

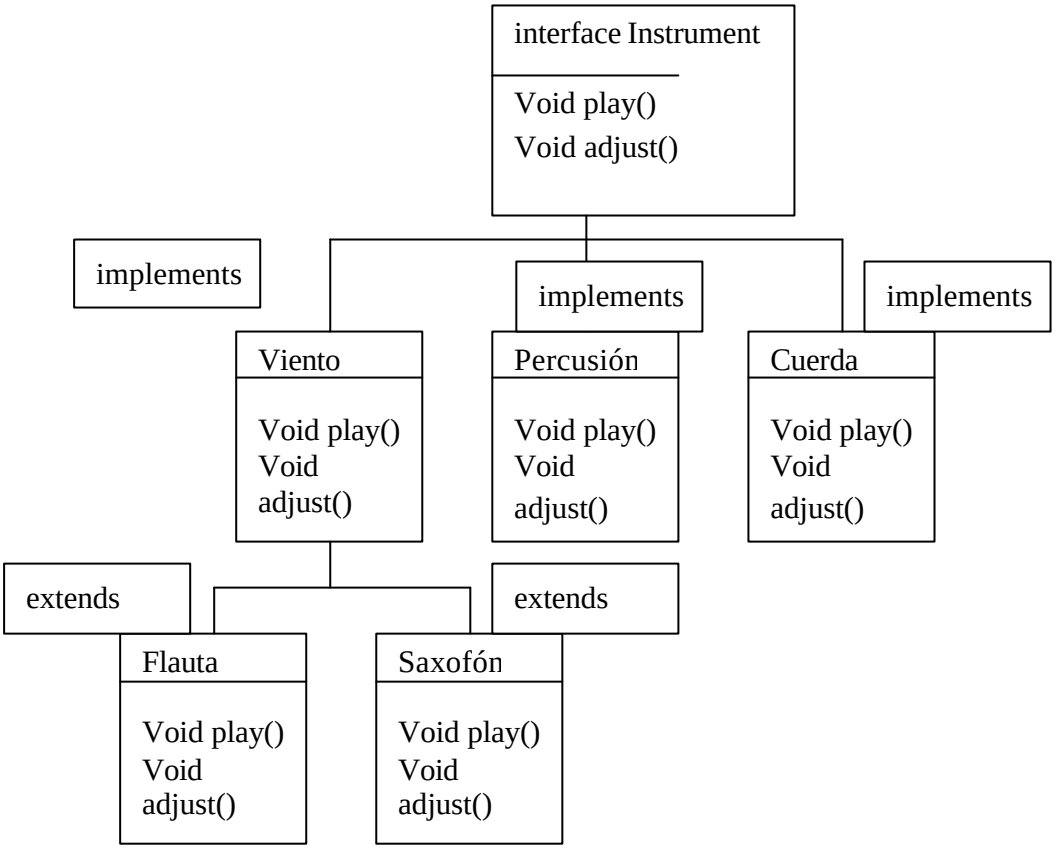
Pensemos en el concepto de una clase "***abstracta pura***". Tal clase nos permite establecer la "**forma**" para una clase: nombres de métodos, listas de argumentos, tipos de retorno, pero no los cuerpos de los métodos. Una interfaz puede contener campos, pero estos son implícitamente **static** y **final**. En pocas palabras: una interfaz nos da la forma, pero no el comportamiento o implementación.

En realidad una interfaz puede entenderse mejor como" Así es como se verán todas estas clases que implementen mi interfaz". También a una interfaz se le conoce como un mecanismo de "protocolo" de clases.

Para crear una interfaz, simplemente usaremos la palabra clave **interface** en lugar de la palabra clave **class**. Para indicar que otra clase quiere utilizar dicha interfaz, la sintaxis es:

Class X implements InterfazY; //implementación de una interfaz
interface NombreDeLaInterfaz{ //crear interfaces.

Veamos el diagrama de instrumentos implementado como interfaz:



Una clase que implementa a una interfaz es en realidad una clase común y corriente.

Podemos declarar explícitamente los métodos de una interfaz como públicos, aunque lo sean implícitamente. Veamos un ejemplo de interfaces:

```
//Musica5.java
//Interfaces

import java.util.*;

interface Instrumento{
        int i=5;
        //Los métodos no debenestar definidos.
        void play();
        String quienSoy();
        void adjust();
        }
class Viento implements Instrumento{
        public void play(){
```

```
                System.out.println("Viento.play()");
                }
        public String quienSoy(){return "Viento";}
        public void adjust(){}
        }
class Percusion implements Instrumento{
        public void play(){
                System.out.println("Percusion.play()");
                }
        public String quienSoy(){return "Percusion";}
        public void adjust(){}
        }
class Cuerda implements Instrumento{
        public void play(){
                System. out.println("Viento.play()");
                }
        public String quienSoy(){return "Viento";}
        public void adjust(){}
        }
class Flauta extends Viento{
        public void play(){
                System.out.println("Flauta.play()");
                }
        public void adjust(){
                System.out.println("Fla uta.adjust()");
                }
        }
class Saxofon extends Viento{
        public void play(){
                System.out.println("Saxofon.play()");
                }
        public String quienSoy(){return "Saxofon";}
        }
public class Musica5{
        static void tune(Instrumento i){
                i.play();
                }
        static void tuneAll(Instrumento[] e){
                for(int i=0;i<e.length;i++)
                        tune(e[i]);
                }
        public static void main(String[] args){
                Instrumento[] orquesta=new Instrumento[5];
                int i=0;
                orquesta[i++]=new Viento();
                orquesta[i++]=new Percusion();
                orquesta[i++]=new Cuerda();
```

```
                orquesta[i++]=new Flauta();
                orquesta[i++]=new Saxofon();
                tuneAll(orquesta);
                }
        }
```

El código en sí es muy parecido al visto en abstracción. El comportamiento es el mismo, lo único que cambia es que estamos utilizando interfaces .

Herencia Múltiple en Java.

Una Interfaz no sólo nos sirve para definir la forma de una clase. Tiene en realidad un propósito más grande. Ya que una interfaz no tiene implementación alguna, entonces nada nos prohíbe crear muchas interfaces y combinarlas. A este acto de combinar interfaces en una clase es llamado herencia múltiple. A veces nos es muy útil la herencia múltiple, ya que podemos tener algún problema del tipo "***x es a y b y c***."

El siguiente ejemplo ilustra mejor la herencia múltiple:

```
//Aventura.java
//Multiples interfaces.
import java.util.*;

interface PuedePelear{
        void pelea();
        }
interface PuedeNadar{
        void nada();
        }
interface PuedeVolar{
        void vuela();
        }
class PersonajeDeAccion{
        public void pelea(){}
        }
class Heroe extends PersonajeDeAccion implements
PuedeNadar,PuedeVolar,PuedePelear{
        public void nada(){}
        public void vuela(){}
        }
public class Aventura{
        static void t(PuedePelear x){ x.pelea();}
        static void u(PuedeNadar x){ x.nada();}
        static void v(PuedeVolar x){ x.vuela();}
        static void w(PersonajeDeAccion x){ x.pelea();}
        public static void main(String[] args){
                Heroe h=new Heroe();
```

```
            t(h); //lo trata como objeto PuedePelear
            u(h);//lo trata como objeto PuedeNadar
            v(h);//lo trata como objeto PuedeVolar
            w(h);//lo trata como objeto PersonajeDeAccion
            }
    }
```

En el ejemplo vemos que **Heroe** combina la clase **PersonajeDeAccion** con las interfaces **PuedePelear**, **PuedeNadar** y **PuedeVolar**. Cuando combinamos herencia con implementación, la clase heredada debe estar presente en primer lugar y luego las implementaciones.

Colisión de Nombres al combinar interfaces.

El pequeño problema de implementar interfaces es la colisión de nombres. Del ejemplo anterior, **PuedePelea** y **PersonajeDeAcción** tienen un método idéntico llamado **void pelea**(). En este caso no hay problema porque el método es el mismo, pero si no lo fuera:

```
//ColisionDeInterfaz.java

interface I1{ void f();}
interface I2{ int f(int i);}
interface I3{int f();}
class C{ public int f() { return 1;} }
class C2 implements I1,I2 {
        public void f(){}
        public int f(int i){ return 1;} //sobrecarga
        }
class C3 extends C implements I2{
        public int f(int i){ return 1; }
        }
class C4 extends C implements I3{
        //Metodo identico no problem!
        public int f(){ return 1;}
        }

//Los metodos difieren solo por el tipo de retorno:
//! class C5    extends C implements I1{}
//! interface I4 extends I1,I3{}
```

Cuando realizamos una sobrecarga de métodos, no pueden diferir por el tipo de dato de retorno. Cuando las dos últimas líneas son descomentadas, el mensaje de error dice:

ColisionDeInterfaz.java:20: f() in C cannot implement f() in I1; attempting to use incompatible return type
found : int
required: void
class C5 extends C implements I1{}
^
ColisionDeInterfaz.java:21: interfaces I3 and I1 are incompatible; both define f(), but with different return type
interface I4 extends I1,I3{}

Extendiendo una interfaz con herencia.

Podemos fácilmente añadir un nuevo método a una interfaz usando herencia, y también podemos combinar muchas interfaces con una nueva interfaz con ayuda de la herencia:

```
//ShowDeHorror.java
//Extendiendo una interfaz con herencia

interface Monstruo{
        void amenaza();
        }
interface MonstruoPeligroso extends Monstruo{
        void destruye();
        }
interface Letal{
        void mata();
        }
class DragonZilla implements MonstruoPeligroso{
        public void amenaza(){}
        public void destruye(){}
        }
interface Vampiro extends MonstruoPeligroso,Letal{
        void ChupaSangre();
        }
class ShowDeHorror {
        static void u(Monstruo b){ b.amenaza();}
        static void v(MonstruoPeligroso d){
                d.amenaza();
                d.destruye();
                }
        public static void main(String[] args){
```

```
        DragonZilla barney=new DragonZilla();
        u(barney);
        v(barney);
        }
}
```

MonstruoPeligroso es una simple extensión de **Monstruo** que produce una nueva interfaz, la cual es implementada en **DragonZilla**. La sintaxis que se ve en **Vampiro sólo** se utiliza cuando **heredamos interfaces**.

Agrupando Constantes.

Dado que todos los datos que pongamos dentro de una interfaz son automáticamente **static** y **final**, la interfaz es una herramienta conveniente para crear grupos de valores constantes, muy parecido al **enum** de **C** ó **C++.**

```
//Meses.java
//Usando interfaces para crear grupos de constantes

package com.serge.tools;
public interface Meses{
    int ENERO=1, FEBRERO=2, MARZO=3, ABRIL=4, MAYO=5,
        JUNIO=6, JULIO=7, AGOSTO=8, SEPTIEMBRE= 9,
        OCTUBRE=10, NOVIEMBRE=11, DICIEMBRE=12 ;
    }
```

La notación de constantes es con letras mayúscula.

Clases internas.

Es posible crear clases dentro de otra clase. A esto se le llama clases internas. Esto nos permite agrupar clases que por lógica pertenecen al mismo funcionamiento. Es importante mencionar que las clases internas no son lo mismo que la composición. Como mencionamos anteriormente, no vamos a profundizar en este tema.

Como es de esperarse, una clase interna se define dentro de la clase:

```
//Parcela.java
//Creando clases internas

public class Parcela{
    class Contenidos{
        private int i=11;
        public int valor(){ return i;}
        }
    class Destino{
        private String etiqueta;
```

```
        Destino(String adonde){
                etiqueta=adonde;
                }
        String leerEtiqueta(){ return etiqueta;}
        }
public void ship(String dest){
        Contenidos c=new Contenidos();
        Destino d=new Destino(dest);
        System.out.println(d.leerEstiqueta());
        }
public static void main(String[] args){
        Parcela p=new Parcela();
        p.ship("Tanzania");
        }
}
```

Como vemos, la manera de construir las clases internas es bastante obvia. El comportamiento es bastante comprensible, sin embargo, para profundizar recomendamos una lectura más extendida y dedicada a este tema.

***Nota: Para crear un objeto de una clase interior FUERA de dicha clase la sintaxis es: ClaseExterna.ClaseInterna. Por ejemplo Parcela.Destino d=new ...**

Ejercicios

1. **A partir de la clase Sándwich.java del capítulo anterior, crear una interfaz llamada ComidaRapida y modificar Sándwich para que implemente a ComidaRapida.**
2. **En Aventura.java, agregue una interfaz llamada PuedeEscalar siguiendo la forma de las interfaces existentes.**

10 Threads

Los objetos nos proveen de una manera de dividir un programa en secciones independientes, ya que frecuentemente es necesario separar un programa en partes independientes que sean ejecutables, cada una de estas partes es conocida como un thread (hilo o tarea) y se programan como si cada thread se ejecutara a sí mismo como si contara con el CPU sólo para él. Lo que sucede realmente es que un mecanismo fundamental esta dividiendo el timpo del CPU por ti, pero generalmente no es necesario que se piense en esto, lo que hace que la programación con múltiples threads sea un aspecto mucho más fácil.

Un proceso es un programa auto ejecutable con su propia dirección de memoria. Un sistema operativo multitareas es capaz de ejecutar más de un proceso (programa) al mismo tiempo, asignando periódicamente ciclos de CPU a cada uno.

Un thread es un control de flujo secuencial y simple dentro de un proceso. Un proceso simple puede entonces tener múltiples threads ejecutables concurrentes. Existen múltiples usos posibles para multithreading (multitareas), pero en general, nuestros programas estarán enfocados en un recurso o evento en particular y seguramente no estaremos dispuestos a modificar el resto del programa por esta razón. Por lo tanto, se crea un thread asociado con dicho recurso o evento y se le deja correr independientemente del programa principal. En resumen, una de las razones enmediatas para el empleo de multithreading es el hecho de producir una interfazis de usuario responsiva

Interfaces de usuario responsivas

Como un punto de partida, considera un programa que realiza una operación intensa (CPU) que termina ignorando los datos de entrada del usuario, siendo irresponsivo.

Esta es una combinación applet/aplicación, que simplemente desplegará un contador ejecutándose:

```
//Counter1.java
// Una interfaz de usuario no responsiva
// <applet code=Counter1 width=300 height=100>
// </applet>

impo rt javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class Counter1 extends JApplet {
        private int count = 0;
        private JButton
        start = new JButton("Start"),
        onOff = new JButton("Toggle");
        private JTextField t = new JTextField(10);
```

```
        private boolean runFlag = true;
        public void init() {
                Container cp = getContentPane();
                cp.setLayout(new FlowLayout());
                cp.add(t);
                start.addActionListener(new StartL());
                cp.add(start);
                onOff.addActionListener(new OnOffL());
                cp.add(onOff);
        }
        public void go() {
                while (true) {
                        try {
                                Thread.sleep(100);
                        } catch(InterruptedException e) {
                        System.err.println("Interrupted");
                }
                if (runFlag)
                t.setText(Integer.toString(count++));
        }
}

class StartL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                go();
        }
}
class OnOffL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
                runFlag = !runFlag;
        }
}
public static void main(String[] args) {
        Console.run(new Counter1(), 300, 100);
        }
} ///:~
```

La siguiente clase es necesaria para ejecutar algunos de los ejemplos.

```
import javax.swing.*;
import java.awt.event.*;
public class Console{
// Creando una cadena de título a partir del nombre de la clase:
        public static String title(Object o) {
                String t = o.getClass().toString();
// Remueve la palabra class
                if(t.indexOf("class") != -1)
```

```
            t = t.substring(6);
            return t;
        }
        public static void setupClosing(JFrame frame) {
// La solución para el JDK 1.2 como una clase interna anónima
            frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
// La solución improvisada en JDK 1.3:
// frame.setDefaultCloseOperation(
// EXIT_ON_CLOSE);
        }
        public static void run(JFrame frame, int width, int height) {
            setupClosing(frame);
            frame.setSize(width, height);
            frame.setVisible(true);
        }
        public static void run(JApplet applet, int width, int height) {
            JFrame frame = new JFrame(title(applet));
            setupClosing(frame);
            frame.getContentPane().add(applet);
            frame.setSize(width, height);
            applet.init();
            applet.start();
            frame.setVisible(true);
        }
        public static void run(JPanel panel, int width, int height) {
            JFrame frame = new JFrame(title(panel));
            setupClosing(frame);
            frame.getContentPane().add(panel);
            frame.setSize(width, height);
            frame.setVisible(true);
        }
} ///:~
```

El método **go( )** es la parte en la cual el programa se mantiene ocupado: imprime el valor de **count** dentro del **JTextField** t, después incrementa **count**. Parte del ciclo infinitodentro de **go( )** es la llamada a **sleep( )**. **sleep( )** debe ser asociada con un objeto **Thread**, a partir de lo cual se puede concluir que toda aplicación tiene un thread asociado a esta. (Además, Java esta basado en threads por lo que siempre hay algunos corriendo durante tus aplicaciones), puedes producir el thread actual usado por el programa con **Thread** y el método **static sleep( ).** Nota que **sleep( )** puede generar una **InterruptedException**, además de que la generación de una excepción es considerado como un modo hostil de salir de un thread, lo cual debe ser considerado (las excepciones son para condiciones excepcionales, no para

control de flujo). La interrupción de un sleeping thread es incluida para soportar alguna característica del lenguaje en el futuro.

Cuando el botón **start** es presionado, **go( )** es invocado. Examinando **go(**), se pensará incredulamente (como lo hice yo) que permitirá multithreading porque se dormirá. Esto es, mientras el método se encuentra en dicho estado, todo parece indicar que el CPU estaría ocupado monitoreando los eventos sobre otro botón. Pero entonces se puede percibir que el verdadero problema es que **go( )** nunca vuelve, debido a que es un ciclo infinito, lo que a la vez significa que **actionPerformed( )** nunca vuelve. Desde el momento en que te encuentras dentro de **actionPerformed( )** por medio del primer accionamiento de un botón, el programa no puede manejar ningún otro evento. (Para salir, debes terminar -matar- de alguna manera el proceso; la manera más fácil de hacerlo es tecleando Control-C en la ventana de la consola, si es el caso de que lo iniciaste en la consola. Si lo iniciaste via navegador, debes cerrar la ventana del navegador.)

El problema básico aquí es que **go( )** necesita mejorar continuamente sus operaciones, y al mismo tiempo necesita regresar para que **actionPerformed( )** pueda terminar y la interface pueda continuar respondiéndole al usuario. Pero en un método convencional como **go( )** no puede continuar y al mismo tiempo devolver el control al resto del programa. Esto suena como un situación imposible de concretar, como si el CPU se encontrara en dos lugares al mismo tiempo, pero esta es precisamente la idea que el uso de threads da.

El modelo thread (y su soporte de programación en Java) es un convención de programación para simplificar el uso de numerosas operaciones dentro de un programa al mismo tiempo. Con threads, el CPU tomará el entorno y le dará a cada thread una parte de su tiempo. Cada thread funciona bajo la conciencia de que mantener al CPU para sí, pero el tiempo del CPU realmente es dividido entre todos los threads. La excepción en todo esto es el caso de que el programa este corriendo en múltiples CPUs. Pero una de las ventajas de emplear threads es que te encuentras abstraído de esta capa, así tu código no necesita saber si es que realmente esta corriendo en un CPU o en varios. De esta forma, los threads nos dan una forma transparente y escalable de crear programas. Pero el mejoramiento de una red in el diseño de un programa, el balance de los recursos y la conveniencia del usuario es muy valuable también. Desde luego, si se tiene más de un CPU, entonces el Sistema Operativo puede dedicar cada CPU a un conjunto de threads o incluso sólo uno y el resto del programa podrá correr más rápido.

Es más razonable emplear multitareas y multithreading en sistemas con varios procesadores.

Heredando de Thread

La manera más sencilla de crear un thread es heredar de la clase **Thread**, la cual tiene lo necesario para crear y ejecutar threads. El método más importante para el **Thread** es **run(**), que debes sobrescribir para que realice lo que tu quieres. Entonces, **run( )** es el código que será ejecutado “simultaneamente” con los otros threads en un programa.

El siguiente programa crea un número de threads de los cuales lleva el control asignándoles un número único generado por medio de una variable **static**. El método **run()** de **Thread** es manipulado para que realice un conteo decreciente a lo largo de su ciclo y finalizar cuando el conteo llegue a cero (momento en el cual **run( )** devuelve, el thread ha terminado).

```
// SimpleThread.java
// Ejemplo simple de threads.
public class SimpleThread extends Thread {
        private int countDown = 5;
        private static int threadCount = 0;
        private int threadNumber = ++threadCount;
        public SimpleThread() {
                System.out.println("Making " + threadNumber);
        }
        public void run() {
                while(true) {
                        System.out.println("Thread " +
                        threadNumber + "(" + countDown + ")");
                        if(--countDown == 0) return;
                }
        }
        public static void main(String[] args) {
                for(int i = 0; i < 5; i++)
                        new SimpleThread().start();
                System.out.println("All Threads Started");
        }
} ///:~
```

Un método **run( )** virtualmente siempre tiene una especie de ciclo que continua hasta que el thread ya no sea necesario, así pues es necesario establecer la condición que dé fin a dicho ciclo (o, como en el caso anterior, simplemente **return** desde **run(**)). Es común que **run( )** sea establecido como un ciclo infinito, lo que significa que a menos de que ocurra algún factor externo que finalice con él, este nunca terminará.

En **main( )** se puede ver un número de threads siendo creados y ejecutados. El método **start( )** en la clase **Thread** realiza una inicialización especial para el thread y luego llama a **run(**). Así los pasos son: el constructor es llamado para construir el objeto, luego **start()** configura al thread y llama a **run(**). Si no llamas a **start( )** (lo cual puedes hacer en el constructor, si es apropiado) el thread nunca será iniciado.
La salida cuando se corre una vez este programa (será diferente para cada ejecución) es:

```
Making 1
Making 2
Making 3
Making 4
Making 5
```

Thread 1(5)
Thread 1(4)
Thread 1(3)
Thread 1(2)
Thread 2(5)
Thread 2(4)
Thread 2(3)
Thread 2(2)
Thread 2(1)
Thread 1(1)
All Threads Started
Thread 3(5)
Thread 4(5)
Thread 4(4)
Thread 4(3)
Thread 4(2)
Thread 4(1)
Thread 5(5)
Thread 5(4)
Thread 5(3)
Thread 5(2)
Thread 5(1)
Thread 3(4)
Thread 3(3)
Thread 3(2)
Thread 3(1)

Notarás que en este ejemplo en ningún lado se llama a **sleep( )**, y la salida del programa indica que cada thread obtiene una porción del tiempo del CPU para ser ejecutado. Esto muestra que **sleep()**, mientras se establece la existencia de un thread en orden de ejecutarlo, no esta envuelto en el hecho de habilitar o deshabilitar el threading. Esto es simplemente otro método.

Se puede observar que los threads no son ejecutados en el orden en que son creados. De hecho, el orden en que el CPU atiende un conjunto de threads es indeterminado, a menos de que se determinen sus prioridades usando el método **setPriority( )** de la clase **Thread**.

Cuando **main( )** crea objetos **Thread** objects no captura las referencias de ninguno de ellos. Un objeto ordinario será objeto del recolector de basura, pero no un **Threa**d. Cada **Thread** "se registra" a si mismo de tal modo que existe una referencia a él en algún lugar de modo que el recolector no lo puede desechar.

Empleo de Threading para una interfaz responsiva

Ahora es posible solucionar el problema de **Counter1.java** con un thread.
El truco es colocar un ciclo dentro de **go(),** dentro del método **run( )** de un thread. Cuando el usuario oprime el botón **start**, el thread es iniciado, pero entonces la creación del thread se completa, de modo que aunque se encuentre en ejecución, el trabajo principal del programa (esperar y responder los eventos de la interfaz de usuario) puede continuar. A continuación se presenta la solución:

```
// Counter2.java
// Interfaz de usuario responsiva con threads.
// <applet code=Counter2 width=300 height=100>
// </applet>
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Counter2 extends JApplet {
	private class SeparateSubTask extends Thread {
		private int count = 0;
		private boolean runFlag = true;
		SeparateSubTask() { start(); }
		void invertFlag() { runFlag = !runFlag; }
		public void run() {
			while (true) {
				try {
					sleep(100);
				} catch(InterruptedException e) {
				System.err.println("Interrupted");
				}
				if(runFlag)
				t.setText(Integer.toString(count++));
			}
		}
	}
	private SeparateSubTask sp = null;
	private JTextField t = new JTextField(10);
	private JButton
	start = new JButton("Start"),
	onOff = new JButton("Toggle");
	class StartL implements ActionListener {
	public void actionPerformed(ActionEvent e) {
		if(sp == null)
		sp = new SeparateSubTask();
	}
}
class OnOffL implements ActionListener {
	public void actionPerformed(ActionEvent e) {
```

```
                if(sp != null)
                sp.invertFlag();
        }
}
public void init() {
        Container cp = getContentPane();
        cp.setLayout(new FlowLayout());
        cp.add(t);
        start.addActionListener(new StartL());
        cp.add(start);
        onOff.addActionListener(new OnOffL());
        cp.add(onOff);
}
public static void main(String[] args) {
        Console.run(new Counter2 (), 300, 100);
}
} ///:~
```

Counter2 es un programa que lleva un curso definido, cuya tarea única es el conformar y mantener la interfaz de usuario. Pero ahora, cuando el usuario presiona el botón **start**, el código encargado del manejo de eventos no llama a un método. En lugar de un thread de la clase es creada una subinstancia llamada **SeparateSubTask**, para que después continue el ciclo de eventos de **Counter2**.

La clase **SeparateSubTask** es una seimple extensión de **Thread** con un constructor que devuelve el thread llamando a **start(**), y a **run( )** que esencialmente contiene al código de **"go(**)" de **Counter1.java**.

Debido a que **SeparateSubTask** es una clase interna, puede acceder directamente al **JTextField t** en **Counter2**; esto se puede observar dentro de **run(**). El campo **t** en la clase más externa es **private** desde que **SeparateSubTask** puede accederlo sin algún permiso especial —y siempre es bueno hacer que los campos sean "tan privados como sea posible", de modo que no puedan ser modificados accidentalmente por fuerzas externas a nuestra clase.

Cuando oprimes el botón **onOff** este oculta a **runFlag** dentro del objeto **SeparateSubTask**. Este thread (cuando evalúa la bandera) puede iniciar y detenerse a si mismo. Oprimir el botón **onOff** produce aparentemente una respuesta instantánea. Desde luego, la respuesta no es instantánea, como aquel sistema que es manejado por interrupciones. El contador se detiene solamente cuando el thread tiene al CPU y nota que la bandera ha cambiado.

Puedes observar que la clase interna **SeparateSubTask** is **private**, lo que significa que a sus campos se les puede dar acceso por default (excepto para **run(**), el cual debe ser **public** desde que este es **public** en su clase original). La clase privada interna no puede ser accesible por ninguna otra clase, excepto por **Counter2**, y las dos clases estan claramente relacionadas. Todo el tiempo encontramos clases muy relacionadas unas con las otras, lo

que implica que para lograr el mejoramiento y el buen mantenimiento del código es importante considerarlas.