WorkShop

Capacitación Java 2012

Módulo: Java Básico

Calle Salta 1056 Planta Alta
San Salvador de Jujuy
Provincia de Jujuy (4600)
Argentina
+54 (0388) 423-7549
www.softlogia.com
info@softlogia.com

Fecha de Impresión: 21/08/2012

Contenido

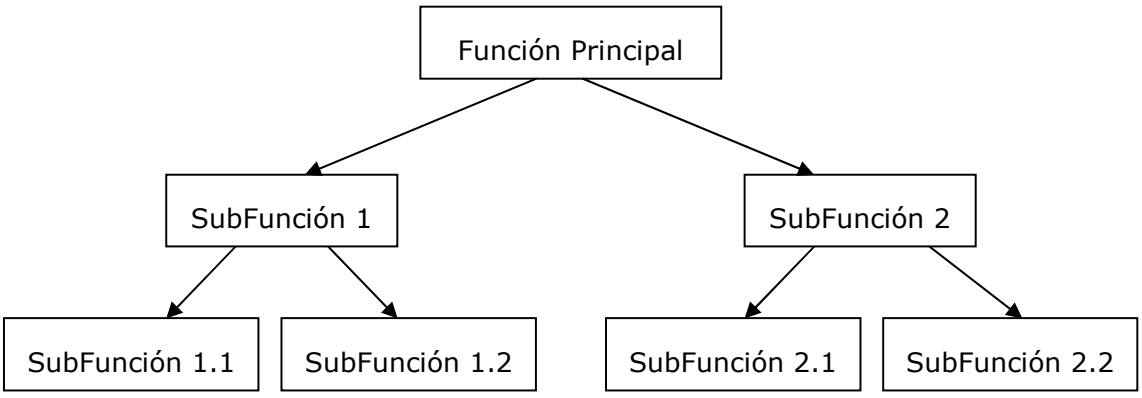
DIA 1

1. Introducción a la programación orientada a objetos

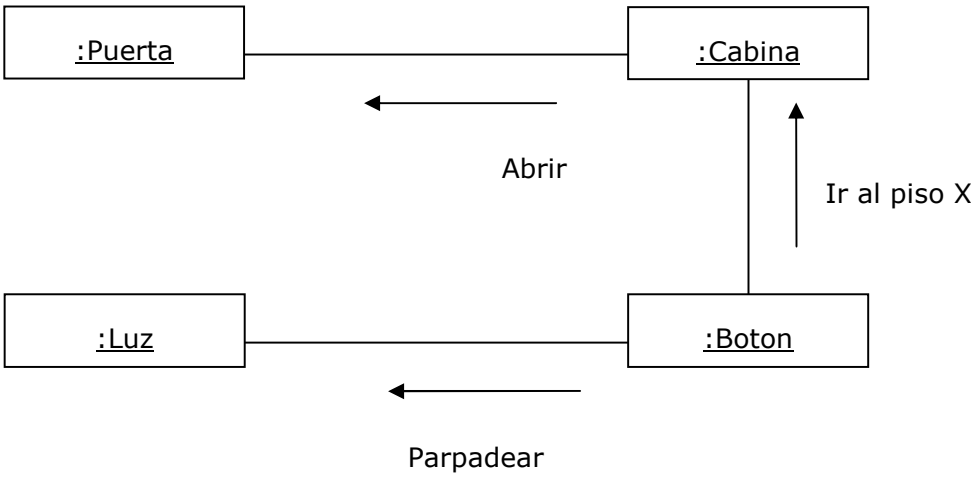
La OOP (Object Oriented Programming, Programación Orientada a Objetos) es el paradigma de programación dominante actualmente, que está reemplazando al paradigma procedural basado en procedimientos y funciones.

La construcción de un programa es una serie de iteraciones del tipo división-unión. Hay que descomponer – dividir – para comprender, y hay que reunir – unión – para construir.

En el enfoque estructurado el proceso de descomposición se ha dirigido tradicionalmente por un criterio funcional. Se identifican las funciones del sistema, luego se descomponen en sub funciones, y así sucesivamente hasta la obtención de elementos simples, directamente representables en los lenguajes de programación por las funciones y procedimientos.



Este método, cuyos elementos integradores son la función y la jerarquía de funciones, aporta resultados satisfactorios cuando las funciones están bien identificadas y son estables en el tiempo. Sin embargo dado que la función induce a la estructura, las evoluciones funcionales pueden involucrar modificaciones estructurales pesadas, debido al acoplamiento estático entre estructura y funciones.



El enfoque orientado a objetos, propone un método de descomposición no basado en lo que hace el sistema, sino más bien en lo que el sistema es y hace. Se identifican los objetos que constituyen el sistema. Los objetos tienen ciertas propiedades y pueden llevar a cabo determinadas operaciones. Las funciones se representan entonces por formas de colaboración entre los objetos que componen el sistema. El acoplamiento se hace dinámico y las evoluciones funcionales del programa no cuestionan ya la estructura estática del programa. Consideremos a manera de ejemplo un sistema de elevador.

La orientación a objetos obtiene su fuerza de su capacidad de agrupar lo que se ha separado, construir lo complejo a partir de lo elemental y sobre todo, de integrar estáticamente y dinámicamente los constituyentes de un sistema.

2. Los objetos

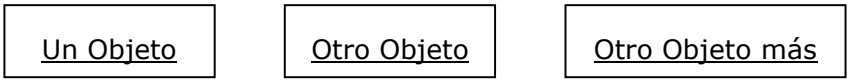
Un objeto es una unidad atómica formada por la unión de un estado y de un comportamiento. Proporciona una relación de encapsulación que asegura a la vez una cohesión interna muy fuerte y un débil acoplamiento con el exterior.

El mundo en el que vivimos está constituido por objetos materiales de todo tipo. Su tamaño es variable: algunos son pequeños, como los granos de arena, y otros son muy grandes como las estrellas. Nuestra percepción intuitiva de lo que constituye un objeto se basa en el concepto de masa, es decir, en una dimensión que caracteriza la cantidad de materia de un cuerpo.

Es perfectamente posible definir otros objetos sin masa, como las cuentas bancarias, las pólizas de seguro o bien las ecuaciones matemáticas. Estos objetos corresponden entonces a conceptos en lugar de entidades físicas.

Los objetos pueden pertenecer también a mundos virtuales, por ejemplo, en asociación con Internet, para crear comunidades de personas q no están situadas en el mismo punto geográfico.

Los *objetos informáticos* definen una representación abstracta de las entidades de un mundo real o virtual, con el objetivo de controlarlos o simularlos. Esta representación abstracta puede ser vista como una especie de espejo informático, que devuelve una imagen simplificada de un objeto que existe en el mundo percibido por el usuario.



Representación de objetos en UML

2.1. Características fundamentales de un objeto

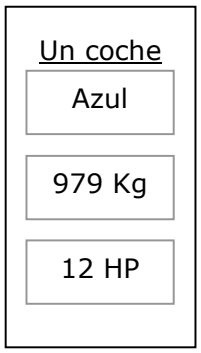
Todo objeto presenta las 3 características siguientes: un estado, un comportamiento, y una identidad.

Objeto = Estado + Comportamiento + Identidad

El estado

Agrupa los valores instantáneos de todos los atributos de un objeto sabiendo que un atributo es una información que cualifica al objeto que la contiene. El estado de un objeto, en un instante dado, corresponde a una selección de valores, entre todos los valores posibles de los diferentes atributos.

El diagrama siguiente muestra un objeto coche que contiene los valores de 3 atributos diferentes: el color, el peso y la potencia física.



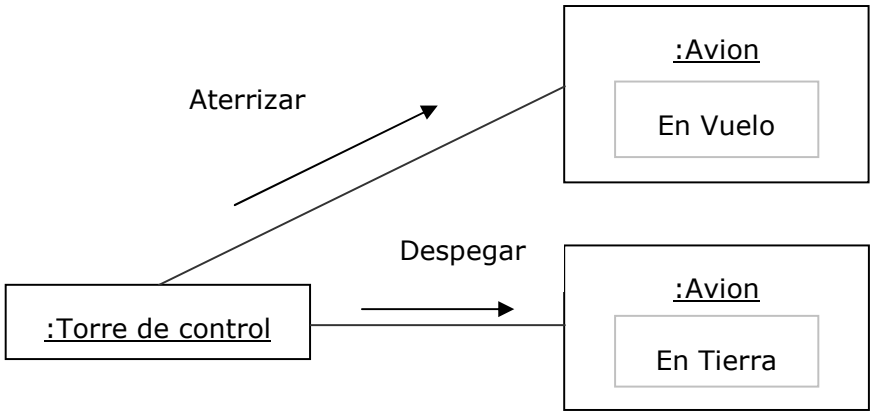
El estado agrupa los valores de los diferentes atributos que caracterizan a un objeto

El estado evoluciona con el tiempo; así cuando un coche corre, la cantidad de combustible disminuye, los neumáticos se gastan y la temperatura en el habitáculo varía. Ciertos componentes del estado pueden ser constantes; es el caso de la marca del coche, o el modelo. Sin embargo el estado de un objeto es variable y puede verse como la consecuencia de sus comportamientos pasados.

El comportamiento

Agrupa todas las competencias de un objeto y describe las acciones de ese objeto. Cada átomo de comportamiento se llama operación. Las operaciones de un objeto se desencadenan como consecuencia de un estimulo externo, representado en forma de un mensaje enviado por otro objeto.

El estado y el comportamiento están relacionados: el comportamiento en un instante dado depende del estado actual, y el estado puede ser modificado por el comportamiento. Por ejemplo, no es posible aterrizar un avión, más que cuando está volando, es decir, el comportamiento "Aterrizar" solo es válido si la información "En vuelo" es válida. Tras el aterrizaje, la información "En vuelo" pasa a ser no valida, y la operación "Aterrizar" deja de tener sentido.

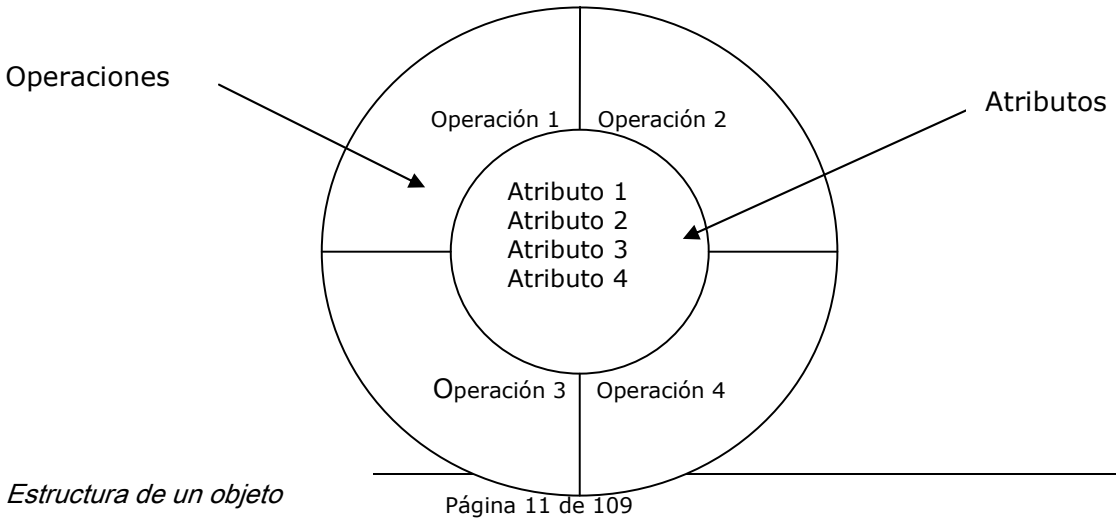


La identidad

Además de su estado, un objeto posee una identidad que caracteriza su propia existencia. La identidad permite distinguir los objetos de forma no ambigua, independientemente de su estado. Ello permite entre otras cosas, distinguir dos objetos en los que todos los valores de atributos son idénticos. Los objetos por lo general poseen un identificador, llamado también clave natural, por ejemplo para un coche es su matrícula, para una persona es su DNI. Esta clave natural puede añadirse al estado del objeto, sin embargo esto es simplemente un artificio de realización, porque el concepto de identidad es independiente del concepto de estado.

Estructura de un objeto

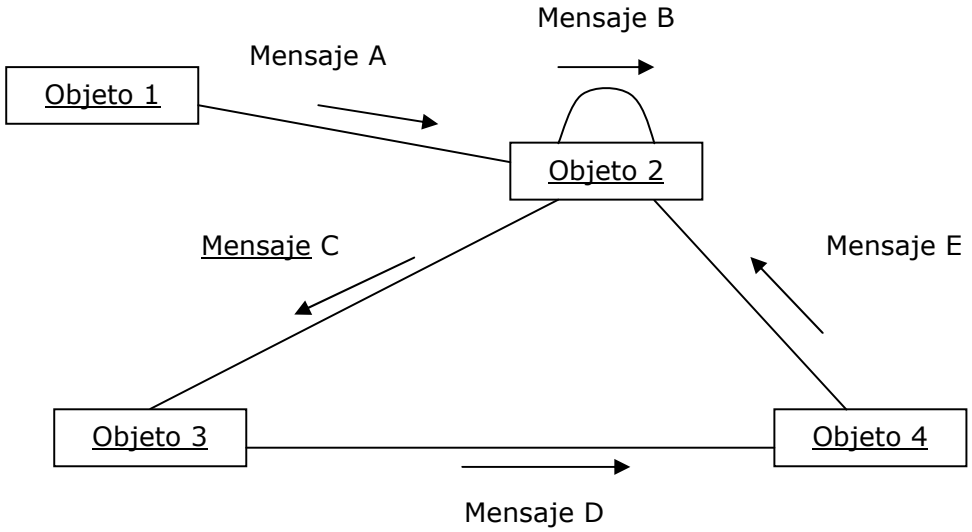
La estructura de un objeto está definida por los atributos que lo definen y las operaciones que puede realizar el mismo.



Estructura de un objeto

2.2. Comunicación entre objetos

Los sistemas pueden verse como sociedades de objetos que trabajan en sinergia a fin de realizar las funciones de la aplicación. El comportamiento global de una aplicación se basa pues en la comunicación entre los objetos que la componen. La gran diferencia entre la aproximación funcional y la orientación a objetos reside precisamente en esta articulación que reduce el acoplamiento entre la estructura y la función.



Los objetos se comunican intercambiando mensajes

El concepto de mensaje

La unidad de comunicación entre objetos se llama mensaje. El mensaje es el soporte de una relación de comunicación que vincula, de forma dinámica, los objetos que han sido separados por el proceso de descomposición.

El mensaje es un integrador dinámico que permite reconstruir una función de la aplicación por la puesta en marcha de la colaboración en un grupo de objetos.

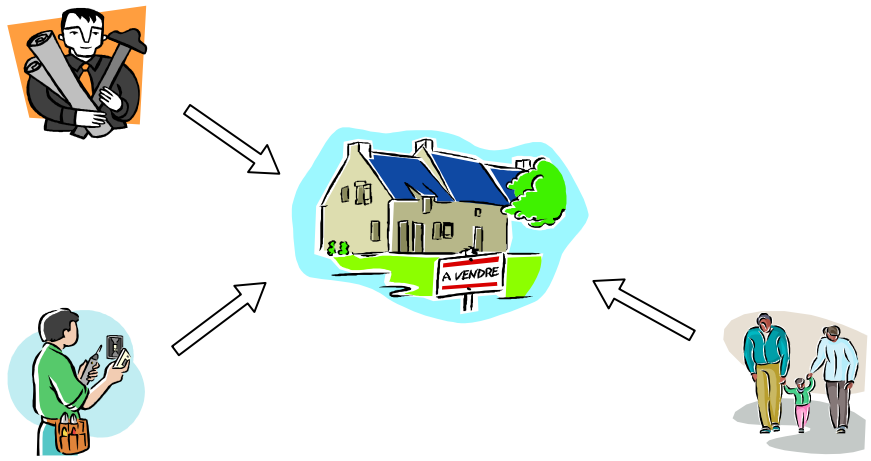
3. Las clases

Estamos rodeados por numerosos objetos en interacción, como hemos visto estos objetos pueden pertenecer al mundo real o a mundos virtuales. A menudo estos objetos son demasiados complejos para ser comprendidos en su integridad a primera vista. Para reducir esta complejidad – o al menos para dominarla – agrupamos los elementos que se parecen y distinguimos las estructuras generales, dejando de lado detalles inútiles, para ello utilizamos la abstracción.

3.1. El método de abstracción

La abstracción es una operación intelectual, que consiste en concentrarnos en un elemento de representación o de noción determinado, centrando la atención especialmente en su esencia o en un conjunto de sus características. El método de abstracción es arbitrario, es decir se define respecto a un punto de vista. Así un objeto puede ser visto a través de abstracciones diferentes.

Para ilustrar el concepto de abstracción consideremos el siguiente ejemplo. Consideramos el objeto casa, esta es vista de diferentes maneras por la familia que la habita, por un arquitecto, y por un electricista. La familia considerará los espacios de la casa, para ver como distribuye sus posesiones y como se ubican ellos mismos. El arquitecto va a centrar su atención en los cimientos, la estructura, la calidad de los materiales que se utilizaron para su construcción, etc. El electricista se enfocara en la instalación eléctrica, los materiales utilizados, el consumo de energía eléctrica, los mecanismos de seguridad, etc.

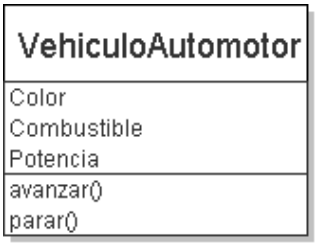


El método de abstracción depende del punto de vista

Podemos ver que el método de abstracción depende del punto de vista que se adopte.

Una clase describe el ámbito de definición de un conjunto de objetos. Los objetos informáticos se construyen a partir de la clase por un proceso llamado instanciación. De este modo todo objeto es una instancia de una clase.

Los lenguajes de programación orientados a objetos permiten describir y manipular clases y sus instancias. Esto significa que el programador puede crear en su máquina una representación informática de las abstracciones que suele manipular mentalmente, esto mediante la creación de clases.



Representación de una clase en UML

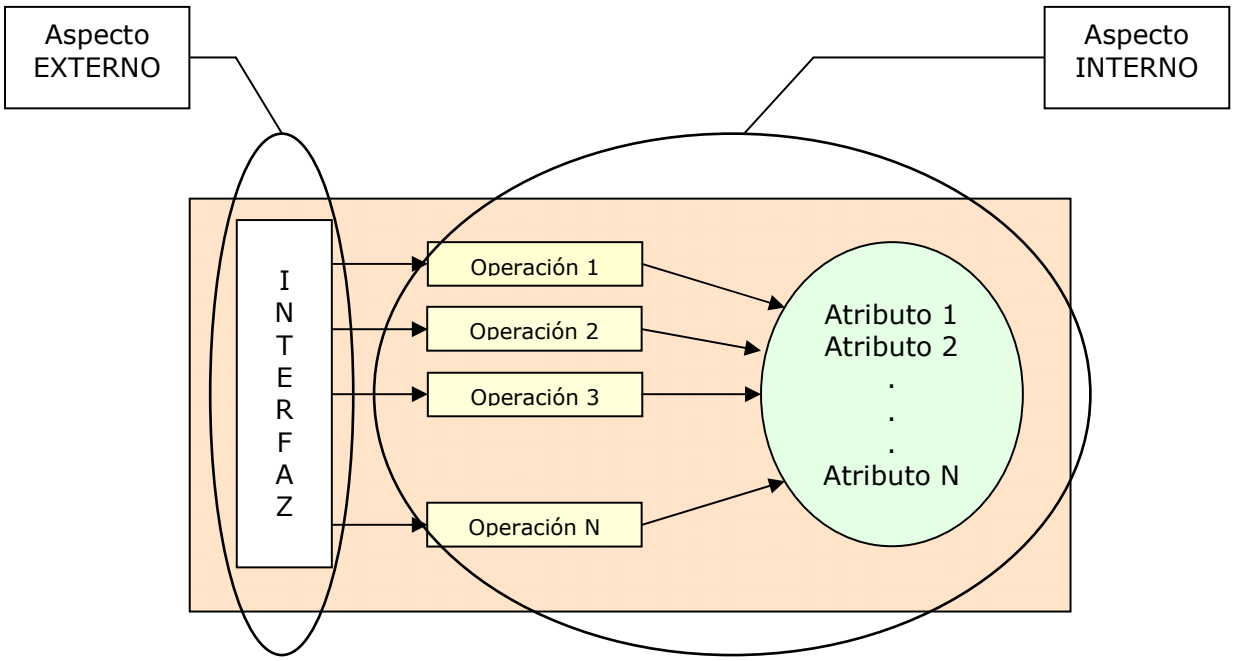
4. Principios de la programación orientada a objetos

Antes de ver los conceptos básicos de la programación orientado a objetos hagamos un breve repaso de lo que hemos visto hasta ahora.

Hemos visto que el paradigma orientado objetos basa la descomposición de un problema en lo que es y lo que hace el sistema, es decir en el conjunto de objetos que constituyen el sistema y en su comportamiento. Los objetos definen una representación abstracta de las entidades de un mundo real o virtual. Los lenguajes de programación orientados a objetos proveen el manejo de objetos mediante el concepto de clase. Una clase describe la estructura de un objeto. Los objetos son creados a partir de una clase mediante un proceso de instanciación. Habiendo recordado esto veamos los principios de la programación orientada a objetos.

4.1. Encapsulación

Las clases y los objetos derivados de ellas encapsulan los atributos y operaciones de las mismas. Una clase encapsula los atributos y operaciones de los objetos que define. Esto debido a que los atributos de un objeto son definidos dentro del contexto de la clase que lo describe y además los atributos deben ser accedidos solo a través de las operaciones del objeto, también definidas en la clase. Podemos ilustrar este concepto con el siguiente gráfico:



OBJETO

De esta manera cuando algún objeto necesite de algún servicio de otro objeto solo deberá conocer su aspecto externo, es decir su interfaz.

Esto presenta ciertas ventajas:

- Los datos encapsulados en los objetos se protegen de accesos improcedentes, lo cual permite garantizar su integridad.
- Los usuarios de una abstracción no dependen de la implementación de la abstracción sino solamente de su especificación. Es decir que los usuarios de un objeto de una clase determinada, no deben preocuparse por su implementación, sino solo de la interfaz que proporciona el objeto para poder comunicarse con él.

Los 3 niveles de encapsulamiento habitualmente manejados por los lenguajes de programación orientados a objetos son:

- Privado
- Protegido
- Publico

La encapsulación actúa como el blindaje de confinamiento de una central nuclear: los defectos quedan encerrados en la clase afectada, no se propagan.

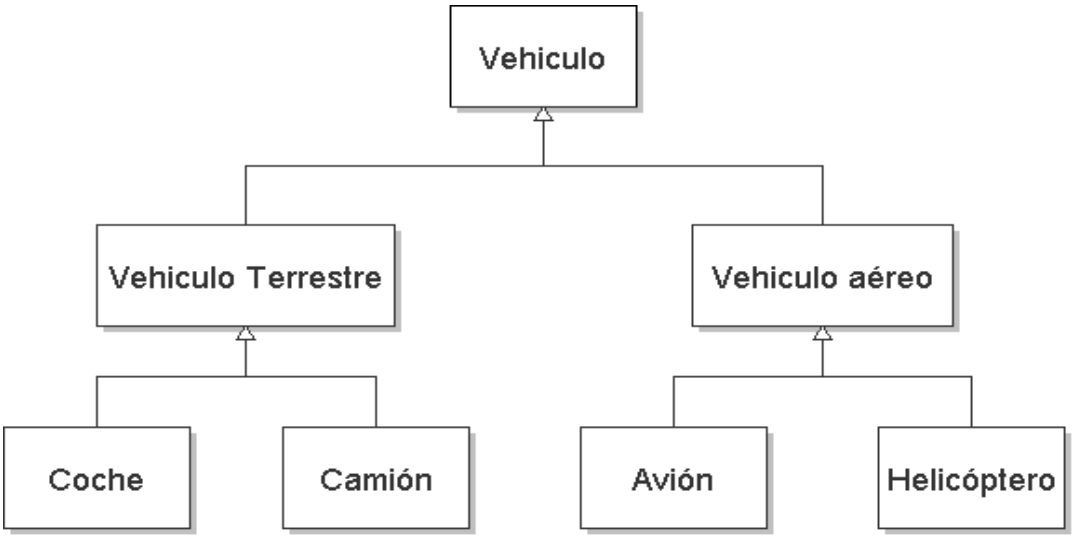
Los criterios de encapsulación se basan en la fuerte coherencia interna en el interior de una clase y en la baja vinculación entre las clases.

4.2. Herencia

Se basa en la relación de generalización y especialización que existe entre las clases.

La generalización consiste en factorizar los elementos comunes (atributos, operaciones, y restricciones) de un conjunto de clases en una clase mas general llamada superclase. Las clases se ordenan según una jerarquía. Las jerarquías de clases permiten gestionar la complejidad ordenando los objetos dentro de árboles de clases de abstracción creciente. La depuración de una jerarquía óptima es delicada e iterativa. Los árboles de clases no crecen a partir de su raíz. Por el contrario, se determinan partiendo de las hojas porque estas pertenecen al mundo real mientras que los niveles superiores son abstracciones construidas para ordenar y comprender.

El ejemplo siguiente muestra una jerarquía de medios de transporte.



La especialización permite capturar las particularidades de un conjunto de objetos no discriminados por las clases ya identificadas. Las nuevas clases se representan por una nueva clase, subclase de una de las clases existentes. La especialización es una técnica eficaz para la extensión coherente de un conjunto de clases.

La generalización y la especialización son dos puntos de vistas antagonistas del concepto de clasificación: expresan en qué sentido se utiliza jerarquía de clases. En toda aplicación real, los dos puntos de vistas se implementan simultáneamente. La generalización se emplea generalmente una vez que los elementos del ámbito han sido identificados a fin de aislar una descripción separada de las soluciones. La especialización, por su parte, se encuentra en la base de la programación por extensión y de la reutilización. Las nuevas necesidades se encapsulan en subclases que extienden armoniosamente de clases existentes.

La herencia es una técnica ofrecida por los lenguajes de programación orientado a objetos para construir una jerarquía de clases. La herencia se utiliza para satisfacer dos necesidades distintas: la clasificación y la construcción.

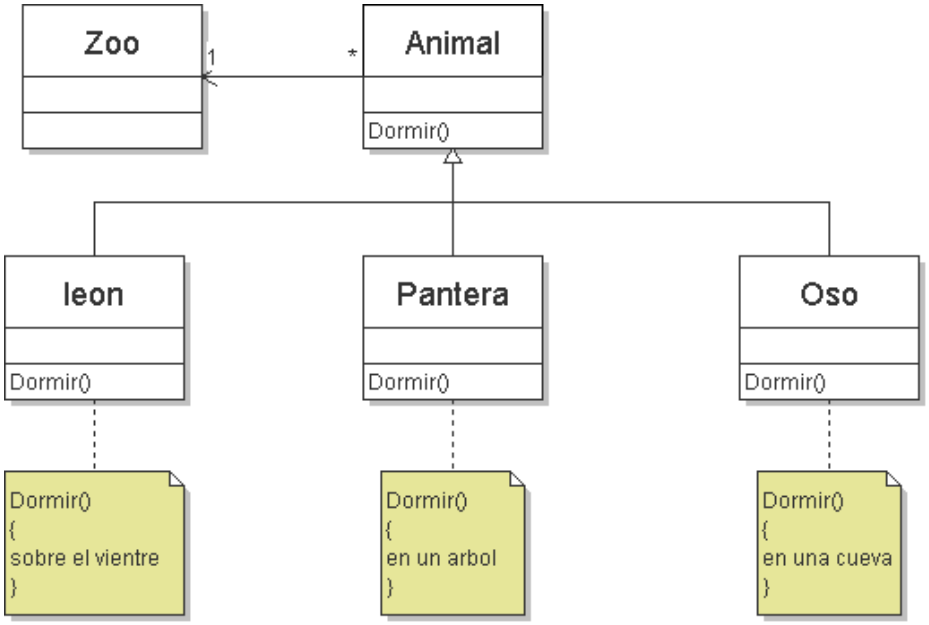
4.3. Polimorfismo

El termino polimorfismo describe la característica de un elemento que puede tomar varias formas, como el agua que se encuentra en estado sólido, liquido o gaseoso, o como el personaje de los X-Men (Mystic).

El termino polimorfismo se refieren en este caso en particular al polimorfismo de operación, es decir, la posibilidad de desencadenar operaciones diferentes en respuesta a un mismo mensaje. Cada subclase hereda las operaciones de sus superclases, pero tiene la posibilidad de modificar localmente el comportamiento de estas operaciones, a fin de tener en cuenta mejor las peculiaridades relacionadas con un nivel de abstracción dado. Desde este punto de vista, una operación dada es polimorfa porque su implementación puede tomar varias formas.

El polimorfismo es un mecanismo de desacoplamiento que actúa en el tiempo. Los beneficios del polimorfismo se recogen principalmente durante el mantenimiento. La implementación eficaz del polimorfismo se basa en la identificación de mecanismos abstractos, aplicables de manera uniforme a objetos instancias de subclases diferentes. La abstracción hace posible el polimorfismo, este debe ser un efecto secundario de la abstracción.

Para ilustrar el concepto de polimorfismo consideremos el siguiente ejemplo. El zoo contiene numerosos animales que pueden ser leones, o panteras u osos. La clase Animal describe colectivamente a todos los animales. El programa escrito a nivel de zoo, no necesita conocer los detalles propios de cada animal. Todos los animales saben dormir, pero cada raza tiene sus costumbres particulares. Las subclases particularizan la operación dormir() según los gustos de cada raza. El mecanismo que duerme a los animales del zoo consiste en informar a cada animal que es hora de dormir. Las subclases de animal implementan de forma diferente esta operación, según la costumbre del animal. Vemos que el mecanismo que duerme a los animales es independiente de los animales que se encuentran realmente en el zoo en un momento dado. Tampoco depende de la subclase precisa del animal actual: si se añaden nuevos animales al zoo, no es necesario modificar el código que duerme a los animales existentes para dormir a los recién llegados.



5. Java Virtual Machine

Una de las características de la tecnología Java que la hace tan especial es ser de Arquitectura Neutral. El compilador Java genera un fichero objeto que es independiente de la arquitectura (El código compilado se ejecuta en cualquier procesador en el que corra un sistema de ejecución Java). El compilador de java genera instrucciones bytecode que no están relacionadas con una arquitectura de computadora en particular. La Java Virtual Machine (Maquina Java Virtual) es quien interpreta el código neutro de los ficheros objeto generados por el compilador de java, generalmente llamados ficheros "bytecodes", convirtiéndolos en código particular de la arquitectura utilizada.

Esto significa que cualquier sistema de computación con la JVM instalada puede correr un programa java sin tener en cuenta el sistema de computación en el cual la aplicación fue originalmente desarrollada.

6. El lenguaje de programación Java

Java es un buen lenguaje de programación. No hay duda de que es uno de los mejores lenguajes de programación para programadores serios. Aunque con las nuevas versiones se espera que haya algunas mejoras, básicamente, la estructura del lenguaje de programación java de mañana, será más o menos la misma que hoy.

Habiendo dicho esto podemos preguntarnos, ¿De dónde han venido las grandes mejoras de Java? La respuesta es que no han venido de grandes cambios subyacentes en el lenguaje de

programación, sino de los grandes cambios de las bibliotecas de Java. Con el tiempo Sun MicroSystems cambio los nombres de muchas funciones de bibliotecas (para hacerlas más coherentes), la forma en que funcionan los gráficos, (cambiando el manejo de eventos y volviendo a escribir algunas partes desde el principio). El resultado es una plataforma de programación mucho más útil que ha llegado a ser mucho más potente que las versiones anteriores de java.

El lenguaje de programación java nos permite escribir programas poderosos y útiles para empresas que pueden correr en el browser, en el escritorio, en un servidor, o en un dispositivo.

7. J2SE (Java 2 Standard Edition)

La tecnología java es un portafolio de productos que está basado en el poder de las redes y en la idea de que el mismo software debería correr en muchos tipos diferentes de sistemas y dispositivos.

La tecnología Java está organizada en las siguientes áreas:

- J2SE (Core/Desktop)
- J2EE (Enterprise/Services)
- J2ME (Mobile/Wireless)
- Java Card
- Java Web Services
- Java Business Integration

En este curso nos centraremos en la J2SE 1.4.2

La plataforma Java 2 Standard Edition (J2SE) provee un entorno completo para el desarrollo de aplicaciones de escritorio (o aplicaciones cliente) y aplicaciones de servidor, y para el despliegue en entornos embebidos. También sirve como base para la plataforma Java 2 Enterprise Edition y Java Web Services.

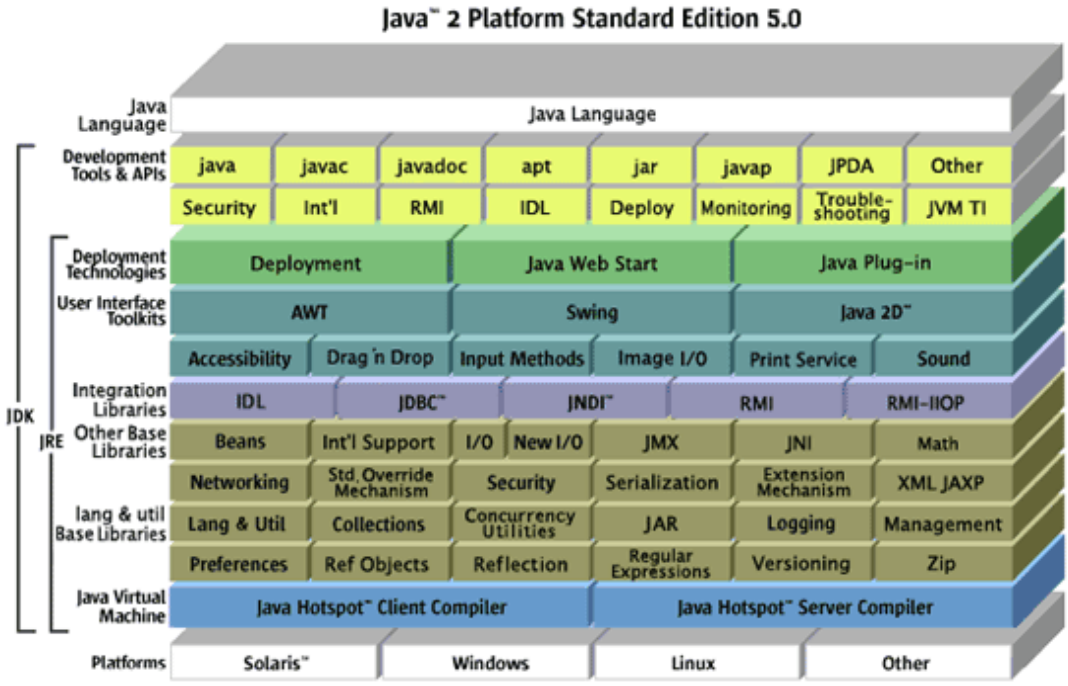
Hay 2 productos principales que componen la J2SE:

- J2SE Runtime Environment (JRE)
- J2SE Development Kit (JDK).

El Entorno de Ejecución Java (JRE) provee las interfaces de programación de aplicaciones de Java (API Application Program Interface), la JVM y otros componentes necesarios para correr applets y aplicaciones escritas en lenguaje de programación Java. El JRE no contiene herramientas ni utilidades tales como compiladores o debuggers para el desarrollo de applets y aplicaciones.

EL JDK contiene al JRE mas un conjunto de herramientas tales como compiladores y debuggers necesarias para el desarrollo de applets y aplicaciones.

El siguiente diagrama conceptual ilustra todas las tecnologías que comprenden J2SE y como se relacionan.



Sun Microsystems distribuye la plataforma J2SE en la forma de J2SE Development Kit (JDK), a veces también llamado Software Development Kit (SDK).

8. El Garbage Collector

El Garbage Collector o Recolector de Residuos es un proceso que dispara la JVM y se ocupa de revisar el estado de asignación de memoria buscando bloques de memoria des referenciados, más precisamente objetos que no se usan más. Cuando encuentra un bloque que no está siendo referenciado por ningún puntero simplemente lo libera, es decir elimina el objeto. En otros lenguajes el programador debe estar muy atento para ver en que situaciones puede estar dejando memoria des referenciada y entonces liberarla explícitamente. El garbage collector le permite al programador olvidarse de este problema de memoria, y escribir menos código.

9. Entornos de desarrollo

Si su experiencia como programador proviene de Visual Basic o Visual C++, estará acostumbrado a un entorno de desarrollo con un editor de textos y menús para compilar y lanzar un programa, sin olvidarnos de un depurador o debugger. El JDK no contiene nada ni remotamente similar. Todo se realiza escribiendo comandos en una ventana de la consola. Creemos que los entornos integrados de desarrollo no facilitan necesariamente el aprendizaje de java, porque son bastantes complejos y ocultan muchos detalles interesantes al programador.

Eventualmente podrá elegir uno para desarrollar proyectos de envergadura, entre la gran cantidad de IDE para java podemos nombrar: Eclipse, JDeveloper, JBuilder, Net Beans, Studio Creador, y Web Sphere. Una alternativa intermedia entre las herramientas de la línea de comandos y los IDE, apropiada para programas sencillos, son los editores de texto que se integran con el JDK, desde ellos es posible compilar y correr los programas, algunos de estos editores son: Emacs, TextPad, y JEdit.

Resumiendo, tiene tres posibilidades entre las que elegir como entorno de desarrollo:

- Usar el JDK y su editor de texto favorito.
- Usar el JDK y un editor que integre JDK.
- Usar un IDE.

10. Instalación del JDK

La más completa y actualizada versión del J2SE de Sun Microsystems se encuentra disponible para Solaris, Linux y Windows.

Si utiliza Solaris, Linux o Windows, puede descargar el JDK de Java desde la dirección http://java.sun.com/j2se. El CD provisto por este curso incluye el JDK para Windows y Linux.

Establecimiento de la ruta de ejecución

Una vez Instalado el JDK, es necesario llevar a cabo un paso más: añadir el directorio jdk/bin a la ruta de ejecución, es decir, a la lista de directorios que recorre el sistema operativo para localizar los ficheros ejecutables. Este paso varía según el Sistema Operativo.

- En Unix (Incluyendo Solaris o Linux)

El procedimiento para añadir una ruta de ejecución depende del shell que estemos utilizando.

En caso de utilizar el shell C (configuración predeterminada en Solaris)

Set path=(/usr/local/jdk/bin $path)

En caso de utilizar el shell Bourne Again (configuración predeterminada en Linux)

Export PATH=/usr/local/jdk/bin:$PATH

- En Windows 95/98/Me

SET PATH=C:\jdk\bin;%PATH%

- En Windows NT/2000/XP

Abra el panel de control, seleccione Sistema y después avanzado. Haga click en el botón Variables de Entorno y localice la variable PATH en la lista superior. Después añada el directorio jdk/bin al comienzo de la misma. Guarde la configuración. Cualquier nueva ventana de la consola que abra tendrá la ruta correcta.

Para determinar si todo ha salido bien, siga estas indicaciones:

Inicie una nueva ventana de la consola. La forma de hacerlo depende de su sistema operativo. Escriba la línea:

java - versión

y pulse enter. Debería obtener un mensaje que indique la versión de Java, de no ser así deberá volver atrás y revisar su instalación.

Instalación de los ficheros fuente de las bibliotecas y documentación

Los ficheros fuente de las bibliotecas se distribuyen en el JDK de Java como un fichero comprimido src.jar, por lo que tendrá que descomprimirlo para tener acceso al código fuente. Le recomendamos que lo haga.

La documentación se encuentra en otro fichero comprimido separado del JDK. Puede descargarlo desde la dirección http://java.sun.com/docs y está disponible en varios formatos (.zip, .gz y .Z). Seleccione el que más le convenga.

Uso de Herramientas de la línea de comandos

Javac.exe: esta herramienta es utilizada para compilar ficheros fuente Java

Java.exe: esta herramienta es utilizada para correr los ficheros compilados java.

Jar.exe: esta herramienta se utiliza para la manipulación de ficheros *.jar

CLASSPATH: variable de entorno que se utiliza para indicar donde se encuentran las clases o librerías de Java que no vienen incluidas en el JDK. Puede incluir la ruta de directorios o ficheros *.zip o *.jar en los que se encuentren los ficheros *.class. Los ficheros *.jar son ficheros *.jar comprimidos.

Día 2: Aprendiendo a programar con Java

11. Un programa Java Sencillo

Veamos de cerca el programa más sencillo que se puede escribir en Java; uno que simplemente imprima un mensaje en la ventana de la consola:

```
public class PrimerEjemplo {
   public static void main(String[] args)   {
      System.out.println("Para no romper la costumbre digamos: Hola Mundo!");
   }
}
```

Vale la pena que analice este ejemplo hasta que se sienta cómodo con el contenido del mismo, ya que las partes de las que consta este programa se repetirán en todas las aplicaciones.

Antes que nada Java hace distinción entre mayúsculas y minúsculas. Si hay algún error en la combinación de mayúsculas y minúsculas (como escribir Main en lugar de main), el programa no se ejecutara.

Veamos el código línea a línea. A la palabra reservada public se la llama modificador de acceso. Los modificadores de acceso se utilizan para establecer el nivel de encapsulamiento, controlan que otras partes del programa pueden utilizar este código. Veremos más sobre los modificadores de acceso más adelante. La palabra clave class está ahí para recordarle que todo lo que hay en un programa Java reside dentro de una clase. Aunque veremos con detenimiento las clases más adelante, por ahora piense en una clase como contenedor para la lógica del programa que define el comportamiento de una aplicación. Todo lo que hay en un programa Java debe estar dentro de una clase.

A continuación de la palabra clave class aparece el nombre de la clase. Hay unas cuantas reglas para los nombres de las clases en Java. Los nombres deben empezar con una letra y después pueden tener cualquier combinación de letras y dígitos. La longitud es prácticamente ilimitada. No puede utilizar una palabra reservada de Java para nombrar una clase. Como puede ver por el nombre PrimerEjemplo, la convención es que los nombres de clases sean nombres que comiencen con una letra en mayúscula. Si un nombre tiene varias palabras, utilice una mayúscula al principio de cada palabra. El nombre del fichero debe tener el mismo nombre que la clase publica, con la extensión .java añadida. Así deberá almacenar este código en un fichero con el nombre PrimerEjemplo.java. Si no lo hace así, verá un mensaje de error más que evidente cuando intente ejecutar este código con el compilador Java ("Public class PrimerEjemplo debe ser definida en un archivo llamado PrimerEjemplo.java").

Si ha escrito correctamente el nombre del fichero y no ha cometido errores al introducir el código fuente, al compilar este código obtendrá un fichero con los bytecode de esta clase. El compilador Java automáticamente llama a este fichero bytecode PrimerEjemplo.class y lo guarda en el mismo directorio que en el fichero fuente. Finalmente ejecute el fichero bytecode con el intérprete Java con el siguiente comando:

java PrimerEjemplo

Recuerde que no hay que poner la extensión .class, cuando utilice:

java NombreDeClase

Para ejecutar un programa compilado, el intérprete Java siempre comienza la ejecución con el código del método main de la clase que le indique. De este modo, debe tener un método main en el fichero fuente de la clase para que se ejecute su código. Puede, por supuesto, añadir sus propios métodos a una clase y llamarlos desde el método main.

Fíjese en las llaves {} del código fuente. En Java, las llaves se utilizan para delimitar los bloques de un programa. En Java, el código de cualquier programa debe comenzar con una llave de apertura { y acabar con una llave de cierre }.

Los estilos que tienen que ver con las llaves es un tema de discusión. Por lo general se utiliza un estilo que alinea las llaves relacionadas. Puede utilizar el estilo de llaves que quiera.

Por ahora, no se preocupe de las palabras clave static void; solo piense en ellas como parte de lo que necesita para que se compile un programa Java. Debe recordar que todo programa Java debe tener un método main con una cabecera idéntica a la que se muestra aquí.

```
public class NombreClase {
   public static void main(String[] args) {
      instrucciones del programa
  }
}
```

A continuación, analicemos el siguiente fragmento de código

```
{
   System.out.println(”Para no romper la costumbre digamos: Hola Mundo!”);
}
```

Las llaves señalan el principio y el final del cuerpo del método. Este método tiene una sola sentencia. Como en la mayoría de los lenguajes de programación, piense en las sentencias de Java como oraciones del lenguaje. En Java, todas las sentencias deben finalizar con punto y coma. Los retornos de carro no marcan el final de una sentencia, por lo que las sentencias pueden ocupar varias líneas si hace falta.

El cuerpo del método main contiene una sentencia que muestra una línea de texto en la consola.

En este caso estamos utilizando el objeto System.out y llamando a su método println. Fíjese en los puntos que se usan para invocar un método. Java utiliza esta sintaxis genérica:

```
objeto.metodo(parametros)
```

Para llamar a las funciones.

En este caso llamamos al método println y le pasamos una cadena de caracteres como parámetro. El método muestra la cadena de caracteres en la consola. No escribe nada más en esa línea, de tal forma que cada llamada a println muestra su salida en una nueva línea. Observe que Java utiliza las comillas dobles para delimitar cadenas de caracteres.

Los métodos en Java, como las funciones en cualquier otro lenguaje de programación, pueden tener cero, uno o más parámetros (en algunos lenguajes se llaman argumentos). Aunque en un método no haya parámetros deberá utilizar los paréntesis vacíos. Por ejemplo hay una variante del método println sin parámetros que solamente imprime una línea en blanco. Se invoca con la siguiente llamada:

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

12. Comentarios en Java

Los comentarios en Java, como en la mayoría de los lenguajes de programación, no aparecen en el programa ejecutable. Así, puede añadir tantos comentarios como necesite sin miedo a inflar el código. Java tiene 3 formas de incluir comentarios.

La forma más común es con //. El comentario ira desde // hasta el final de la línea.

También puede usar los delimitadores /* */ que le permiten poner varias líneas de comentarios.

La tercera forma de comentario se utiliza para generar documentación automáticamente. Este comentario utiliza /** para comenzar y */ para finalizar.

13. Palabras reservadas de Java

A continuación se da una lista de las palabras reservadas de Java para que se vaya familiarizando con la sintaxis del lenguaje.

PALABRAS RESERVADAS				
abstract	do	implements	private	throw
boolean	double	import	protected	throws
Break	else	instanceof	public	transient
Byte	extends	int	return	true
Case	false	interface	short	try
Catch	final	long	static	void
Char	finally	native	super	while
Class	float	new	switch	

continue	for	null	synchronized	
default	If	package	this	

14. Tipos de datos

Java es un lenguaje con un potente control de los tipos de datos. Esto quiere decir que hay que especificar el tipo de todas las variables. Hay ocho tipos primitivos en Java. Cuatro de ellos son tipos enteros; dos son tipos numéricos en punto flotante, uno es del tipo carácter char, que se utiliza para la codificación de los caracteres Unicode, y hay un tipo boolean para los valores lógicos.

Enteros

Enteros	Longitud (bits)	Rango
Byte	8	-2^7 ... $+2^7-1$
Short	16	-2^{15} ... $+2^{15}-1$
Int	32	-2^{31} ... $+2^{31}-1$
Long	64	-2^{63} ... $+2^{63}-1$

Tipos en punto flotante

Flotantes	Longitud (bits)
float	32
double	64

El tipo carácter (character)

En primer lugar las comillas simples se utilizan para delimitar las constantes char. Por ejemplo 'H' es un carácter. Es distinto de "H", una cadena de caracteres que contiene un solo carácter. En segundo lugar, el tipo char es el que se utiliza en el sistema de codificación Unicode. Como aclaración diremos que el código Unicode se diseño para manejar los caracteres de todas las lenguas escritas del mundo, es un sistema de 2 bytes.

El tipo bolean

El tipo bolean tiene dos valores, false (false) y true (verdadero). Se utiliza para evaluar condiciones lógicas. No se puede realizar la conversión entre valores enteros y boolean.

15. Variables

En Java todas las variables tienen un tipo. Una variable se declara colocando el tipo en primer lugar, seguido del nombre de la variable. A continuación tiene algunos ejemplos:

```
double salario;
int diasVacaciones;
long poblacionTierra
char sexo;
boolean bandera ;
```

Observe el punto y coma al final de cada declaración. El punto y coma es necesario porque una declaración es una sentencia completa de Java.

15.1. Reglas para la formación de los nombres de las variables

Un nombre de una variable debe comenzar con una letra, y debe ser una secuencia de letras o dígitos. Observe que los términos "letra" y "digito" son más amplios en Java que en la mayoría de los lenguajes, esto se debe a que utiliza el código Unicode. Todos los caracteres del nombre de una variable son significativos y también se distingue entre mayúsculas y minúsculas. La longitud del nombre de una variable es prácticamente ilimitada. Se sigue la siguiente convención para la construcción de nombres de variables: el nombre debe ser representativo, comenzar con una letra en minúscula, separando las palabras que componen el nombre de la variable escribiendo la letra de la siguiente palabra en mayúsculas. Esto puede observarse en las declaraciones de arriba.

No se puede utilizar una palabra reservada de Java para nombrar una variable.

15.2. Asignaciones e Inicializaciones

Después de haber declarado una variable, deberá darle un valor inicial de manera explícita con una sentencia de asignación (no podrá utilizar nunca los valores de variables sin inicializar). Para asignar un valor a una variable previamente declarada, ponga el nombre de la variable a la izquierda, un signo igual (=) y, a continuación una expresión Java que tenga sentido a la derecha. A continuación tiene algunos ejemplos:

```
int diasVacaciones;  //Esto es una declaración
diasVacaciones = 12;  //Esto es una asignación
char sexo;
sexo = 'M';
```

Una característica de Java es que puede declararse e inicializarse una variable en una misma sentencia.

```
int diasVacaciones = 12; //Esto es una declaración e inicialización
```

Otra característica de Java es que puede declarar variables en cualquier parte del código.

16. Constantes

En Java debe utilizar la palabra clave **final** para describir una constante.

```
final double CM_PER_INCH = 2.54;
```

La palabra clave final indica que se puede realizar la asignación de una variable una sola vez. Es costumbre poner los nombres de las constantes en mayúsculas.

17. Operadores

Los operadores aritméticos más corrientes +-*/ se utilizan en Java para la suma, resta, multiplicación, y la división.

El operador / indica una división entera si los dos operandos son enteros, y una división en punto flotante en cualquier otro caso. El resto de una división se indica con el operador %. Por ejemplo, 15/2 es 7, 15%2 es 1, y 15.0/2 es 7.5

La división entera por 0 genera una excepción, mientras que la división en punto flotante por 0 da como resultado infinito o un NaN.

Hay una manera abreviada de utilizar los operadores aritméticos binarios con una asignación.

Por ejemplo:

```
x += 4;  //Equivale a lo siguiente
x = x + 4;
```

Como regla general, se debe poner el operador a la izquierda del signo =.

17.1. Operadores de incremento y decremento

Los programadores saben que una de las operaciones más comunes con un valor numérico es sumar o restar 1. Java siguiendo los pasos de C y C++, tiene los operadores de incremento y decremento, ++ y – respectivamente. Estos operadores tienen 2 formas de aplicación postfija y prefija

```
int m = 7;
int n = 7;
int a = 2 *  ++m;  //Ahora a vale 16 y m vale 8
int b = 2 *  n++;  //Ahora b vale 14 y n vale 8
```

La forma prefija hace la suma primero, la postfija evalúa la expresión con el valor anterior de la variable.

Le recomendamos que no utilice los operadores ++ y – dentro de otras expresiones puesto que puede causar confusión en el código y errores molestos.

17.2. Operadores relacionales y boléanos

Los operadores relacionales de Java son los siguientes

El operador de igualdad es el doble signo igual, ==.

```
3 == 7 // da como resultado false
```

El operador de desigualdad es !=

```
3 != 7 // da como resultado true
```

Tiene los operadores habituales

< (menor que)

> (mayor que)

<= (menor o igual que)

>= (mayor o igual que)

Java utiliza && como operador lógico AND, y || como operador lógico OR. Como recordara fácilmente del operador != , la exclamación ! es el operador lógico de negación. Los operadores && y || se evalúan en modo "cortocircuito". Esto quiere decir que con una expresión como la siguiente:

```
A && B
```

Una vez que el valor de la expresión A se ha determinado como false, el valor de la expresión B no se calcula. De forma parecida para la expresión

```
A || B
```

Si el valor de la expresión A es true, entonces la expresión A || B será automáticamente true, sin necesidad de evaluar B.

17.3. Operadores de bits

Al trabajar con cualquier dato entero, tenemos operadores que funcionan con los bits que componen el entero. Esto quiere decir que puede utilizar técnicas de enmascaramiento para averiguar el valor de un bit individual en un número.

Los operadores de bits son los siguientes:

& ("and")

| ("or")

^("xor")

- (“not”)

Estos operadores actúan sobre patrones de bits.

También tenemos los operadores >> y <<, que desplazan un patrón de bits a la derecha o a la izquierda.

18. Los paréntesis y la jerarquía de operadores

Como en todos los lenguajes de programación, es recomendable que utilice los paréntesis para indicar el orden en el que quiere llevar a cabo las operaciones.

La jerarquía de operadores en Java se indica en la siguiente Tabla:

Operadores	Asociación
[] . () (llamada a un método)	Izquierda a derecha
! ++ -- +(unario) –(unario) () (moldeado) new	Derecha a izquierda
* / %	Izquierda a derecha
+-	Izquierda a derecha
<< >> >>>	Izquierda a derecha
< <= >= > instanceof	Izquierda a derecha
== !=	Izquierda a derecha
&	Izquierda a derecha
^	Izquierda a derecha
|	Izquierda a derecha
&&	Izquierda a derecha
||	Izquierda a derecha
?:	Izquierda a derecha
= += -= *= /= %= &= |= ^= <<= >>= >>>=	Derecha a izquierda

19. Estructuras de control

Java, como cualquier lenguaje de programación, tiene tanto estructuras de control condicionales como estructuras de control de repetitivas para determinar el control del flujo del programa.

19.1. Ámbito de los bloques

Antes de ver las estructuras de control, debe saber algo más sobre los bloques. Un bloque o sentencia compuesta es cualquier número de sentencias simples de Java que están

comprendidas entre dos llaves. Los bloques definen el alcance de las variables, y pueden estar anidados. He aquí un bloque que esta anidado dentro de otro:

```
{
  int n;
  . . .
  {
     int k;
     . . .
   } // La definición de K llega hasta aquí}
```

Sin embargo, no se pueden declarar variables con el mismo nombre en dos bloques anidados. Por ejemplo, el código siguiente dará error y no se compilará.

```
{
  int n;
  . . .
  {
     int k;
     int n;  // Error, no se puede redefinir en un bloque interior
     . . .
   }
}
```

Estructuras de Decisión

Estructuras de Decisión

```
if( condición )
{
        :
}
else
{
        :
}
```

```
switch( variable )
{
        case c1:
                :
                break;
        case c2:
                :
                break;
        case cn:
                :
                break;
        default:
                :
                break;
}
```

Estructuras de Repetición

Estructuras de Repetición	

while(*condición*)
{
:
}
	do
	{
	:
	} while(*condición*)

for(int i=0 ; *condición* ; i++)
{
:
}	

20. Uso de las clases existentes

Ya que no es posible hacer nada en Java sin las clases, vamos a ver algunas de ellas en funcionamiento. Puede utilizar los métodos de estas clases sin necesidad de saber cómo están implementados, todo lo que necesita saber es el nombre y los parámetros (si son necesarios). Este es el objetivo de la encapsulación y realmente funciona en todas las clases.

A continuación veremos una clase típica, Date, y aprenderá a crear objetos e invocar métodos de esta clase.

20.1. Objetos y variables objetos

Para trabajar con objetos, es necesario construirlos primero e indicar su estado inicial. Después, se pueden aplicar métodos a los objetos.

En Java, se utilizan constructores para crear nuevas instancias. Un constructor es un método especial cuyo propósito es crear e inicializar objetos. Veamos un ejemplo. La biblioteca Java Standard contiene una clase Date. Sus objetos describen intervalos en el tiempo, como "December 31, 2004, 23:59:59 GMT".

Los constructores siempre tienen el mismo nombre que la clase. Es decir que el constructor de la clase Date se llama Date. Para construir un objeto de la clase Date se debe combinar el constructor con el operador new.

```
new Date();
```

Esta expresión construye un nuevo objeto de la clase Date, que en este caso se inicializa con la fecha y hora actuales.

Si lo desea puede pasar el objeto a un método

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

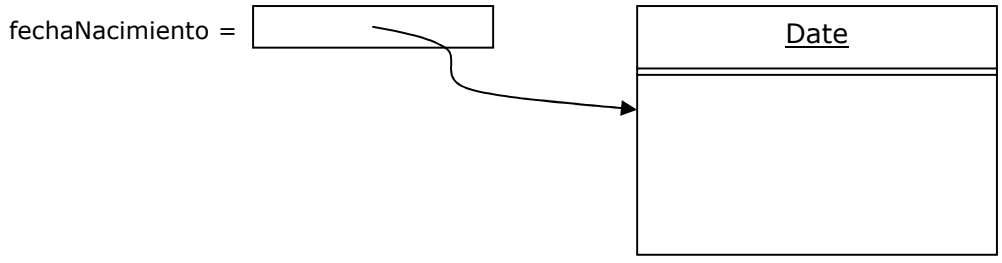
Alternativamente puede aplicar un método al objeto que acaba de construir. Uno de los métodos de la clase Date es toString. El método toString obtiene una representación de tipo cadena de la fecha. Aquí tiene la forma en la que puede aplicar toString al nuevo objeto Date construido:

```
String s = new Date().toString();
```

En estos dos ejemplos, el objeto construido solo es utilizado una vez. Sin embargo, lo normal es mantener esos objetos para su utilización posterior. Para ello, basta con almacenarlos en una variable:

```
Date fechaNacimiento = new Date();
```

La siguiente figura muestra la variable objeto fechaNacimiento que hace referencia al objeto creado recientemente.



Aunque por lo general nos referimos a fechaNacimiento como un objeto de tipo Date, es importante notar que esto realmente no es exacto.

Existe una importante diferencia entre objetos y variables objeto. Por ejemplo, la sentencia:

```
Date fechaEntrega;  //Aquí fechaEntrega no hace referencia a ningún objeto.
```

Define una variable objeto, fechaEntrega, que puede hacer referencia a objetos de tipo Date. Es importante resaltar que la variable fechaEntrega no es un objeto y, de hecho, aun no hace referencia a ninguno. Por lo tanto, no puede utilizar ninguno de los métodos Date en esta variable. La instrucción:

```
String s = fechaEntrega.toString(); //Todavía no
```

Generará un error de compilación.

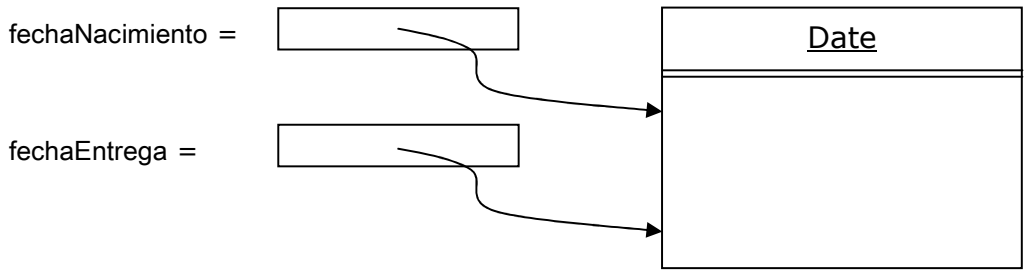
Lo primero que debe hacer es inicializar la variable fechaEntrega. Tiene dos posibilidades:

La primera es hacerlo a la hora de crear el objeto:

```
fechaEntrega = new Date();
```

y la segunda es referenciar a un objeto ya existente:

```
fechaEntrega = fechaNacimiento;
```



Ahora ambas variables hacen referencia al mismo objeto.

Es importante reseñar que una variable objeto no contiene un objeto sino que hace referencia a un objeto.

En Java, el valor de cualquier variable objeto es una referencia a un objeto almacenado en alguna otra parte. El valor devuelto por el operador new también es una referencia. Una sentencia como esta

```
Date fechaEntrega = new Date();
```

Tiene dos partes. La expresión new Date() crea un objeto nuevo de tipo Date, y su valor es una referencia al objeto creado. Dicha referencia se almacena después en la variable fechaEntrega.

Es posible establecer explícitamente una variable objeto con un valor nulo para indicar que, actualmente, no hace referencia a ningún objeto:

```
Date fechaEntrega = null
. . .
if (fechaEntrega !=null ) {
   System.out.println(fechaEntrega);
}
```

Si se aplica un método a una variable que contenga null, se genera un error en tiempo de ejecución.

```
Date fechaNacimiento = null;
String s = fechaNacimiento.toString();  // ¡error de ejecución!
```

21. Construcción de clases propias

A continuación aprenderá a construir sus propias clases, del tipo "clases trabajadoras" necesarias para aplicaciones sofisticadas. Por lo general este tipo de clases no disponen de un método main. En su lugar, tienen sus propios atributos y sus métodos. Para construir un programa completo, es necesario combinar varias clases, una de las cuales dispone de un método main.

21.1. Una clase Empleado

La forma más sencilla para la definición de una clase Java es la siguiente:

```
class NombreDeClase {
   //atributos
   atributo1
   atributo2
   . . .
   atributoN

   //constructores
   constructor1
   constructor2

   //metodos
   metodo1
   metodo2
   . . .
   metodoN
}
```

Ejemplo 2.1

Considere la siguiente, y muy simplificada, versión de una clase Empleado que podría utilizarse en el sistema de nominas de una empresa:

Ejemplo 2.1

```
package com.empresa.cursos.java.basico.ejemplos.dia2;


public class Empleado {
                //atributos o campos instanciados
                private String nombre;
                private double salario;

                //constructor
                public Empleado(String nom, double sal) {
                   nombre = nom;
                   salario = sal;
                }

                //método 1
                public String getNombre() {
```

```
        return nombre;
    }

    //metodo 2
    public double getSalario() {
        return salario;
    }

    //método 3
    public void incrementarSalario(double porcentaje) {
        double incremento = salario * porcentaje / 100;
        salario += incremento;
    }
}
```

Análisis de la clase Empleado

A continuación vamos a diseccionar la clase Empleado. Empezaremos con los métodos de dicha clase. Si se examina dicha clase podrá ver que esta clase dispone de un constructor y de cuatro métodos.

Todos los métodos de esta clase están etiquetados como public. La palabra reservada public, denota el nivel de acceso, significa que cualquier método de cualquier clase puede invocar ese método public. Observe que existen 2 atributos instanciados que contendrán los datos que manipularemos dentro de una instancia de la clase Empleado.

La palabra reservada private, denota el nivel de acceso, garantiza que los únicos métodos que podrán acceder a estos campos son los de la propia clase Empleado. Ningún otro método podrá leer o escribir en ellos.

Por último observe que uno de los atributos instanciados es, por si mismo una referencia a un objeto: el atributo nombre es una referencia a un objeto String. Esto es bastante habitual: las clases suelen contener con frecuencia atributos instanciados de un tipo de clase.

21.2. Método Constructor

Existe una importante diferencia entre los métodos constructores y el resto de los métodos: un constructor solo puede ser invocado en conjunción con el operador new, y no es posible aplicarlo a un objeto ya existente para reinicializar los campos instanciados. Debe tener presente lo siguiente:

- Un constructor tiene el mismo nombre que la clase
- Una clase puede tener más de un constructor
- Un constructor puede tener cualquier numero de parámetros
- Un constructor no devuelve ningún valor
- Un constructor siempre se invoca con el parámetro new.

Ejemplo 2.2

El siguiente ejemplo muestra un programa que pone a trabajar a la clase Empleado.

En este programa, construimos un array de Empleados y lo rellenamos con tres objetos Empleado

A continuación, utilizamos el método incrementarSalario de la clase Empleado para aumentar el salario de cada Empleado un 5%.

Por último, mostramos la información de cada empleado invocando los métodos getNombre y getSalario.

Ejemplo 2.2

```
package com.empresa.cursos.java.basico.ejemplos.dia2;

public class EmpleadoTest {
    public static void main(String[] args) {
        //Rellena el array staff con tres objetos Empleado
        Empleado[] staff = new Empleado[3];
        staff[0] = new Empleado("Roberto Cracker", 75000);
        staff[1] = new Empleado("Carlos Hacker", 50000);
        staff[2] = new Empleado("Antonio Tester", 40000);

        //Incrementa el salario de cada uno un 5%
        for (int i=0; i<staff.length; i++) {
            staff[i].incrementarSalario(5);
        }

        //muestra la información de todos los objetos Empleado
        for (int i=0;i<staff.length; i++) {
            Empleado e = staff[i];
            System.out.println("Nombre: " + e.getNombre());
            System.out.println("Salario: " + e.getSalario());
        }
    }
```

```
}

class Empleado {
  //atributos o campos instanciados
  private String nombre;
  private double salario;

  //constructor
  public Empleado(String nom, double sal) {
    nombre = nom;
    salario = sal;
  }

  //método 1
  public String getNombre() {
    return nombre;
  }

  //metodo 2
  public double getSalario() {
    return salario;
  }

  //método 3
  public void incrementarSalario(double porcentaje) {
    double incremento = salario * porcentaje / 100;
    salario += incremento;
  }
}
```

Observe que este programa consta de dos clases: Empleado y EmpleadoTest, que tiene el especificador de acceso public. El método main está contenido en la clase EmpleadoTest.

El nombre del fichero fuente es EmpleadoTest.java ya que debe coincidir con el nombre de la clase public. Solo se puede tener una clase pública en un fichero fuente, pero se puede tener un número indeterminado de clases no públicas.

El programa se inicia facilitando al intérprete de bytecodes el nombre de la clase que contiene el método main:

java EmpleadoTest

El intérprete comienza ejecutando el código de este método main. Dichas sentencias por orden, crean tres nuevos objetos Empleado y muestran su estado.

21.3. Uso de varios ficheros fuente

El programa del ejemplo anterior contiene dos clases en un mismo fichero. Sin embargo, muchos programadores prefieren colocar cada una de sus clases en su propio fichero fuente. Por ejemplo podría situar la clase Empleado en el fichero Empleado.java y EmpleadoTest en EmpleadoTest.java.

22. Paquetes

Java le permite agrupar clases en una colección llamada paquete. Los paquetes son adecuados para organizar su trabajo y para separarlo en bibliotecas de código.

La biblioteca Standard de Java esta suministrada a través de muchos paquetes, como java.lang, java.util, java.net, etc. Los paquetes Java Standard son ejemplos de paquetes jerárquicos. Al igual que disponemos de subdirectorios anidados en nuestro disco duro, podemos organizar los paquetes usando niveles de anidamiento. Todos los paquetes Java Standard se encuentran dentro de los paquetes java y javax.

La principal razón para el uso de paquetes es garantizar la singularidad de los nombres de las clases. Suponga que dos programadores tienen la feliz idea de suministrar la clase Empleado. Si cada uno de ellos sitúa su clase en un paquete diferente, no habrá ningún problema. De hecho, para garantizar completamente un nombre de paquete único, Sun recomienda que utilice el nombre del dominio Internet de su empresa (que, como todos sabemos, es único) escrito de forma inversa. Entonces podrá usar subpaquetes en diferentes proyectos. Por ejemplo empresa.com es uno de los dominios que nuestra empresa tiene registrado. Escrito de forma inversa, se convierte en el paquete com.empresa, que a su vez puede ser subdividido en subpaquetes como com.empresa.cursos

El único propósito para anidar un paquete es el de gestionar nombres únicos. Desde el punto de vista del compilador, no existe relación alguna entre los paquetes anidados. Por ejemplo, java.util y java.util.jar no tienen nada que ver uno con otro. Cada uno de ellos es una colección independiente de clases.

22.1. Uso de paquetes

Una clase puede utilizar todas las clases incluidas en su paquete y las declaradas como publicas en otros paquetes. Existen dos formas de acceder a las clases públicas de otro paquete:

La primera es añadir el nombre completo de ese paquete al principio de cada nombre de clase. Por ejemplo:

```
java.util.Date hoy = new java.util.Date();
```

Esto es bastante tedioso. El acercamiento más sencillo, y también el más común, es usar la palabra reservada import. El objetivo de esta sentencia es, simplemente, ofrecerle un atajo

para referirse a las clases de otros paquetes. Una vez utilizada import, ya no será necesario que emplee los nombres completos en las clases.

Puede importar una clase específica, o un paquete completo. Las sentencias import se sitúan en la parte superior de los ficheros fuente, pero debajo de cualquier instrucción package. Por ejemplo puede importar todas las clases del paquete java.util con la siguiente instrucción:

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

Después de esto, la siguiente sentencia será correcta

```
Date hoy = new Date();
```

También puede importar una clase específica de un paquete.

```
import java.util.Date;
```

La importación de todas las clases de un paquete es fácil. No tiene ningún efecto negativo sobre el código, por lo que no suele haber ninguna razón lógica para no hacerlo.

Debe tener cuidado cuando importa las clases de 2 paquetes distintos, y estos paquetes tienen una clase con el mismo nombre. Al usarse la clase con el mismo nombre se produce un error de compilación. En estos casos debe utilizar el nombre completo del paquete en cada una de las clases con el mismo nombre.

22.2. Adición de una clase a un paquete

Para situar clases en un paquete, debe situar el nombre de dicho paquete al comienzo de su fichero fuente, antes del código que define las clases de ese paquete. Por ejemplo el fichero Empleado.java se inicia de la siguiente forma:

```
package com.empresa.cursos.java.basico.ejemplos.dia2;

public class Empleado {

  . . .

}
```

Si no se sitúa la sentencia package en el fichero fuente, las clases de dicho fichero se incluirán en un paquete predeterminado que no tiene nombre.

Los ficheros de un paquete deben situarse en un subdirectorio que coincida con el nombre completo de ese paquete. Por ejemplo todas las clases del paquete com. empresa.cursos.java.basico.ejemplos.dia2 deben situarse en un subdirectorio com/empresa /cursos/java/basico/ ejemplos/dia2

Ejemplo 2.3

A continuación veremos un ejemplo que ilustra el uso de paquetes

Ejemplo 2.3

```
//No se define un paquete, por lo tanto este fichero fuente pertenece al paquete
//predeterminado
        import com.empresa.cursos.java.basico.ejemplos.dia2.*;


        public class PackageTest {
                public static void main(String[] args) {
                    //Debido a la sentencia import, no tenemos que usar
                    //com.softlogia.cursos.java.basico.Empleado
                    Empleado marcelo = new Empleado("Marcelo Hacker", 50000);


                    //aumenta el salario un 5%
                    marcelo.incrementarSalario(5);


                    //muestra la información sobre marcelo
                    System.out.println("Nombre: " + marcelo.getNombre() + ", Salario: " +
marcelo.getSalario() );
                }
        }
```

La estructura de directorios debe ser la siguiente:

```
.(Directorio base)
  PackageTest.java
  PackageTest.class
  com/
      empresa /
          cursos/
              java/
                  basico/
                      ejemplos/
                          dia2/
                              Empleado.java
                              Empleado.class
```

Cabe notar que por lo general un IDE maneja por separado los ficheros fuentes y los ficheros objetos. Así se tendrá la misma estructura pero una estará en un directorio source y la otra en un directorio classes.

Es importante que la ruta de las clases compiladas, es decir los ficheros bytecodes coincida con el nombre del paquete.

22.3. Ámbito de un paquete

Ya hemos visto los modificadores de acceso public y private. Los modificadores de acceso se aplican a las clases, a sus variables y métodos. Una etiqueta private indica que el elemento al cual califica solo puede ser utilizado por la clase en la cual está definido. Una etiqueta public indica que el elemento al cual califica puede ser utilizado en cualquier clase. Si no se especifica ni public ni private entonces un elemento puede ser accedido por los métodos que se encuentran en el mismo paquete. Considere el programa del ejemplo 2.2. La clase Empleado no fue definida como clase publica. Por lo tanto otras clases del mismo paquete, como por ejemplo EmpleadoTest, podrán acceder a ella. Para las clases este planteamiento es razonable. Sin embargo para las variables este comportamiento no es muy adecuado. Estas deben ser marcadas explícitamente como privadas, o podrán ser vistas por cualquier paquete. Esto rompe la encapsulación.

23. Acceso Protegido

Como ya sabe, los campos de una clase es mejor que estén etiquetados como private, y los métodos como public.

A continuación puede ver un resumen de los cuatro modificadores de acceso en Java que controlan la visibilidad:

- Visible solo para la clase (private)
- Visible para todo el mundo (public)
- Visible para el paquete y todas las subclases (protected)
- Visible para el paquete (es el valor predeterminado). No se necesita ningún modificador

24. Consejos para el diseño de clases

Sin querer ser pesados, les damos algunos consejos que harán que sus clases sean más aceptables en los "privilegiados" círculos de la OOP.

- Mantenga siempre los datos privados
- Inicialice siempre los datos
- No utilice demasiado tipos básicos en una clase
- No todos los campos necesitan campos de acceso y modificación individuales
- Utilice un formato standard para las definiciones de clase

- Divida las clases con demasiadas responsabilidades
- Haga que los nombres de las clases y sus métodos reflejen lo que hacen

Día 3: Características avanzadas de programación Java

25. Herencia de clases

La herencia es un concepto fundamental de la programación orientada a objetos. La idea que se oculta tras la herencia es que puede crear nuevas clases que están basadas en otras que ya existen. Cuando se hereda de otra clase se reutilizan (o heredan) sus métodos y campos, además de poder definir nuevos métodos y campos para adaptar esta nueva clase a la nueva situación.

Vamos a continuar con la clase Empleado que vimos en la sección anterior. Suponga que trabaja en una empresa en la que los directivos son tratados de forma diferente al resto de empleados. Por supuesto, los directivos son empleados en muchos aspectos, y ambos ganan su salario. Sin embargo, mientras que los empleados deben completar sus tareas para recibir su sueldo mensual, los directivos obtienen gratificaciones si alcanzan los objetivos previstos. Este es el tipo de situación ideal para la herencia. ¿Por qué? Bien, tiene que definir una nueva clase, Directivo, y añadirle funcionalidad. Pero puede reutilizar parte de lo que ya tiene programado para la clase Empleado, y todos los campos de la clase original pueden ser preservados. De forma más abstracta, existe una obvia relación "es-una" entre Directivo y Empleado. Cada Directivo es un empleado: esta relación "es-una" es el sello de identidad de la herencia.

Aquí tiene la forma de definir una clase Directivo que herede de la clase Empleado. Para denotar la herencia se utiliza la palabra reservada extends:

```
class Directivo extends Empleado
{
   //campos y métodos
}
```

extends indica que se está construyendo una nueva clase que deriva de otra clase que ya existe. Esta última recibe el nombre de superclase o clase padre. La nueva clase recibe el nombre de subclase o clase hija.

La clase Empleado es una superclase, pero no porque sea superior a su subclase o porque tenga más funcionalidad. De hecho, es todo lo contrario: las subclases tienen más funcionalidad que sus superclases.

Ejemplo 3.1

El código para nuestra clase Directivo es el siguiente

Ejemplo 3.1

```
package com.empresa.cursos.java.basico.ejemplos.dia3.herencia;

public class Directivo extends Empleado{
    private double bonus;

      public Directivo(String nomb, double sal)  {
        super(nomb, sal);
        bonus = 0;
      }

      public double getSalario() {
         double salarioBasico = super.getSalario();
         return salarioBasico + bonus;
      }

      public void setBonus(double b) {
         bonus = b;
      }
}
```

Desde luego, si tiene un objeto Empleado, no es posible aplicar el método setBonus, ya que no se encuentra entre los métodos definidos para la clase Empleado. Sin embargo es posible utilizar los métodos getNombre, getSalario con objetos Directivo. Cada uno de estos métodos no están declarados explícitamente en la clase Directivo, pero son heredados automáticamente de la superclase Empleado.

De forma análoga los campos nombre y salario se heredan también de la superclase. Por lo tanto cada objeto Directivo tiene tres campos: nombre, salario y bonus.

Cuando se define una subclase que hereda de su superclase, solo es necesario indicar las diferencias entre ambas. Cuando se diseñan clases, los métodos más generales se sitúan en la superclase, mientras que los más especializados van en la subclase. La extracción de la funcionalidad más común moviéndola a una superclase es muy común en la programación orientada a objetos.

Sin embargo, algunos de los métodos de la superclase no son adecuados para la subclase Directivo. En particular getSalario que debe devolver la suma del salario básico más las gratificaciones. Por lo tanto es necesario suministrar un nuevo método sustituyendo al definido

en la superclase. El método getSalario de Directivo no tiene acceso directo a los campos privados de la superclase, por ello no puede utilizar simplemente el campo salario. Solo los métodos de la clase Empleado tienen acceso a los campos privados. Para indicar una llamada al método getSalario de la superclase Empleado utilizamos la palabra reservada super.

La llamada

```
super.getSalario();
```

Invoca al método getSalario de la clase Empleado. Si no se utiliza super, entonces el método getSalario de la clase Directivo se llamaría a si mismo infinitamente. En el constructor la palabra super tiene un significado diferente. La instrucción:

```
super(nomb, sal);
```

es una forma de decir "llama al constructor de la superclase Empleado con los parámetros nomb, sal".

Ya que el constructor de la clase Directivo no puede acceder a los campos privados de la clase Empleado, debe inicializarlos a través de un constructor, el cual es invocado con la sintaxis especial super. La llamada usando super debe ser la primera sentencia usando el constructor de la subclase.

Si dicho constructor no invoca explícitamente el constructor de la superclase, esta usa su constructor predeterminado (sin parámetros). En caso de que la superclase no lo tenga definido, y que el de la subclase no invoque explícitamente a ningún otro constructor de la superclase, el compilador Java generara un error.

Ejemplo 3.2

Aquí tiene un ejemplo de funcionamiento. Vamos a crear un nuevo directivo y asignar su gratificación. Construiremos un array de 3 empleados, y rellenaremos el array con una mezcla de directivos y empleados, para luego mostrarlos.

Ejemplo 3.2

```
package com.empresa.cursos.java.basico.ejemplos.dia3.herencia;


public class DirectivoTest {
   public static void main (String[] args) {
            //Construye un objeto directivo
            Directivo jefe = new Directivo("Marcelo Craker", 80000);
            jefe.setBonus(5000);


            Empleado[] staff = new Empleado[3];
            //Rellena el array staff con objetos Directivo y Empleado
```

```
            staff[0] = jefe;
            staff[1] = new Empleado("Carina Cracker", 5000);
            staff[2] = new Empleado("Roberto Tester", 4000);

            //Muestra la información de todos los objetos Empleado
            for(int i=0; i<staff.length; i++) {
              Empleado e = staff[i];
              System.out.println("Nombre: " + e.getNombre());
              System.out.println("Salario: " + e.getSalario());
            }
          }
}
```

Lo más importante de este ejemplo es la llamada

```
e.getSalario();
```

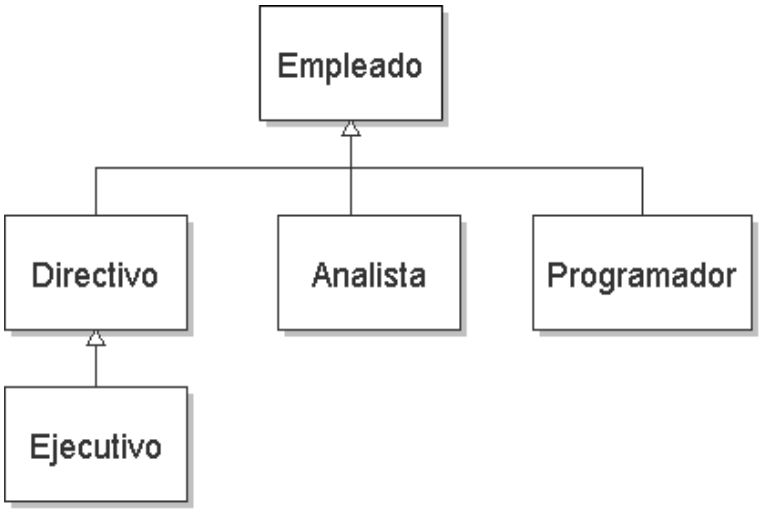
Ya que toma el método getSalario correcto. Observe que el tipo declarado e es Empleado, mientras que el tipo actual del objeto al cual hace referencia e puede ser Empleado (cuando i sea 1 o 2) o Directivo (cuando i sea 0).

Cuando e se refiere a un objeto Empleado, la llamada e.getSalario() invoca el método getSalario de la clase Empleado. Sin embargo, cuando e apunta a un objeto Directivo, el método getSalario invocado es el de la clase Directivo. La maquina Virtual sabe cuál es el tipo actual del objeto al cual se refiere e, y por lo tanto, es capaz de invocar el método correcto.

El hecho de que una variable objeto (como e) pueda hacer referencia a múltiples tipos diferentes, se conoce también con el nombre de polimorfismo. La selección automática del método apropiado durante la ejecución, se llama enlace dinámico (dynamic binding).

25.1. Jerarquías de Herencia

La herencia no tiene porque detenerse en un nivel de clases. Por ejemplo podríamos tener una clase Ejecutivo que derivara de Directivo. La colección de todas las clases derivadas de una superclase se llama jerarquía de herencia o jerarquía de clase.



26. Moldeado o Casting

El moldeado es el proceso de forzar una conversión de un tipo a otro. El lenguaje Java dispones de una notación especial para estas operaciones. Por ejemplo:

```
double x = 3.405

int nx = (int)x;
```

convierte el valor de la expresión x a un entero, descartando para ello la parte fraccionaria.

Lo mismo que ocasionalmente puede necesitar convertir un número en punto flotante a entero, puede necesitar transformar una referencia de un objeto de una clase a otra. Para realizar esta operación se utiliza una sintaxis similar a la empleada con los valores numéricos. Envuelva el nombre de la clase destino entre paréntesis y sitúe esta expresión delante del objeto a moldear. Por ejemplo:

```
Directivo jefe = (Directivo)staff[0];
```

Solo existe una razón por la cual puede ser necesaria una operación de moldeado: usar un objeto en toda su extensión una vez que su tipo actual ha sido olvidado temporalmente. Por ejemplo, en la clase DirectivoTest, staff tenía que ser un arreglo de objetos Empleado, ya que algunas de sus entradas eran de empleados normales. Puede que necesitemos moldear los objetos directivos del array a Directivo para utilizar algunos de sus métodos específicos, no heredados.

Como ya sabe, en Java, cada variable objeto hace referencia a un objeto de una determinada clase, que describe campos y métodos. Por ejemplo, staff[i] hace referencia a un objeto Empleado (por lo que también puede referirse a un objeto Directivo, ya que hereda los campos y métodos de Empleado).

Usted especifica estas descripciones en su código, y el compilador comprueba que no se está ofreciendo más de lo que describe una variable, es decir que no se esté asignando a una variable de una superclase a una variable de una subclase. Si asigna un objeto de una subclase a una variable de una superclase, estará por debajo de ese límite y el compilador le dejara hacerlo. Sin embargo, el proceso contrario implicaría sobrepasar el límite impuesto, por lo que debe confirmar al compilador esta operación mediante una notación de moldeado (subclase).

Si intenta moldear hacia abajo una cadena de herencia y está mintiendo, entonces ocurrirá un error.

```
Directivo jefe = (Directivo) staff[i]  //ERROR!!
```

Cuando el programa se ejecuta, el sistema de ejecución Java observa la ruptura del límite antes mencionado, y genera una excepción, veremos el concepto de excepción más adelante. Por lo tanto, es una buena práctica de programación determinar si puede llevarse a cabo una operación de moldeado antes de intentarlo. Para ello basta con usar el operador instanceof. Por ejemplo

```
if (staff[i] instanteof Directivo) {
   jefe = (Directivo) staff[i]
   …
}
```

Resumiendo

- Solo se puede moldear dentro de una jerarquía de herencia
- Use instanceof para comprobar antes de moldear desde una superclase a una subclase

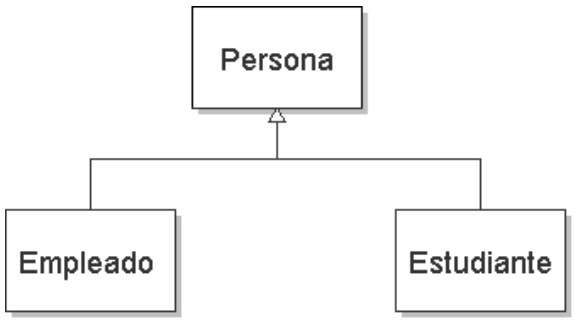
En la actualidad, convertir el tipo de un objeto efectuando un moldeado no es una buena idea. En nuestro ejemplo, no es necesario moldear un objeto Empleado como Directivo en la mayoría de las ocasiones. El método getSalario funcionara correctamente en ambas situaciones, porque el enlace dinámico que hace que funcione el polimorfismo localiza automáticamente el método adecuado.

La única razón para llevar a cabo una operación de moldeado es usar un método que es exclusivo de los directivos como setBonus. Si por alguna razón se ve forzado a usar setBonus en un objeto Empleado, pregúntese si esto es una indicación del flujo del diseño de la superclase. Podría tener sentido rediseñar la superclase y añadir un método setBonus. Recuerde que un solo moldeado erróneo es más que suficiente para finalizar un programa. En general conviene reducir al máximo estas operaciones y el uso del operador instanceof.

27. Clases Abstractas

Según avanzamos en la jerarquía de herencia, las clases se vuelven más generales y, probablemente más abstractas. En algún punto, la clase antecesora se vuelve tan general que puede pensarse en ella mas como una base para otras clases que en una clase con instancias especificas que querrá usar. Considere por ejemplo una abstracción de nuestra clase Empleado.

Un empleado es una persona, y también un estudiante. Ampliemos nuestra jerarquía de clases incluyendo Persona y Estudiante. La siguiente figura muestra las relaciones jerárquicas entre las clases.



¿Por qué molestarse con un nivel de abstracción tan alto? Existen algunos atributos que tienen sentido en cada persona, como el nombre. Pero los estudiantes y los empleados tienen nombres, e introduciendo una superclase común podemos extraer el método getNombre a un nivel superior en la jerarquía de herencia. Añadiremos un nuevo método, getDescripcion, cuyo propósito es devolver una pequeña descripción de la persona:

Un empleado con salario de 50000 pesos.

Un Estudiante concentrado en programación.

Ejemplo 3.3

A Continuación analizaremos la clase abstracta Persona.

Ejemplo 3.3

```
package com.empresa.cursos.java.basico.ejemplos.dia3.abstractas;

abstract class Persona {
      private String nombre;

      public Persona(String nom) {
        nombre = nom;
      }

      public abstract String getDescripcion();
```

```
    public String getNombre() {
      return nombre;
    }

}
```

El método getDescripcion() es abstracto. Un método declarado como abstracto no necesita ser implementado. Una clase con uno o más métodos abstractos debe ser declarado también como abstracta. Una clase abstracta puede contener métodos no abstractos, por ejemplo el método getNombre.

Los métodos abstractos actúan como receptáculos que serán implementados en las subclases. Cuando se hereda de una clase abstracta tiene 2 posibilidades. Puede dejar definidos todos, o parte, de los métodos abstractos, lo que le obligara también a declarar esa subclase como abstracta, o puede definir todos los métodos con lo que no es necesario hacer abstracta la subclase.

Ejemplo 3.4

Por ejemplo, definiremos una clase Estudiante que herede de la clase Persona y que implemente el método getDescripcion. Ya que ninguno de los métodos de Estudiante es abstracto, no es necesario declarar a dicha clase como tal.

Ejemplo 3.4

```
package com.empresa.cursos.java.basico.ejemplos.dia3.abstractas;


public class Estudiante extends Persona{
    private String materia;


    public Estudiante(String nom, String mat)  {
      super(nom);
      materia = mat;
    }


    public String getDescripcion() {
      return "Un Estudiante concentrado en " + materia;
    }
}
```

Una clase puede ser declarada como abstract incluso sino tiene métodos abstractos.

Las clases abstractas no pueden instanciarse. Esto es, si una clase es declarada como abstract, no podrían crearse objetos de esa clase. Por ejemplo la expresión:

new Persona("Albert Einstein");

es un error. Sin embargo, si se pueden crear objetos de subclases concretas.

Observe que incluso es posible crear variables objeto de una clase abstracta, pero una variable de este tipo debe ser referenciada a un objeto de una subclase no abstracta. Por ejemplo:

Persona p = new Estudiante("Newton", "Física");

Aquí p es una variable del tipo abstracto Persona que se refiere a una instancia de la subclase no abstracta Estudiante.

Ejemplo 3.5

El siguiente ejemplo es un programa que rellena un array de referencias a Persona con empleados y estudiantes. Después muestra los nombres y descripciones de esos objetos.

Ejemplo 3.5

```
package com.empresa.cursos.java.basico.ejemplos.dia3.abstractas;


public class PersonaTest {
	public static void main(String[] args) {
		Persona[] gente = new Persona[2];
		//Rellena el arreglo gente con objetos Estudiante y Empleado
		gente[0] = new Empleado("Martin Craker", 50000);
		gente[1] = new Estudiante("Newton", "Fisica");

		//muestra todos los nombres y descripciones de todos los objetos Persona
		for(int i=0; i < gente.length; i++){
		  Persona p = gente[i];
		  System.out.println(p.getNombre() + ", " + p.getDescripcion());
		}
	      }
}
```

Tenga en cuenta que la variable p nunca hace referencia a un objeto Persona ya que es imposible construirlo. p siempre se refiere a un objeto de una clase concreta como Empleado o Estudiante. Para estos objetos, el método getDescripcion está definido.

¿Podría haberse omitido por completo el método abstracto de la clase Persona y definir los métodos getDescripcion en las subclases Empleado y Estudiantes? En caso de hacerlo, no sería posible invocar el método getDescripcion en la variable p. El compilador se asegura de que solo se invoquen métodos que están declarados en la clase.

Los métodos abstractos son un importante concepto del lenguaje Java. Los encontrara con mucha frecuencia dentro de las interfaces.

28. Consejos de diseño para la herencia

- Sitúe las operaciones y los campos comunes en la superclase
- No utilice campos protegidos
- Utilice la herencia para modelar una relación "es-una"
- No utilice la herencia a menos que todos los métodos heredados tengan sentido
- Utilice el polimorfismo, no realice acciones averiguando la clase.

29. Interfaces

Una Interfaz es una forma de describir que debe hacer una clase, pero sin especificar cómo debe hacerlo. Una interfaz no es una clase, sino una secuencia de requisitos para las clases que deseen implementar esa interfaz.

Veamos un ejemplo concreto. El método sort de la clase Arrays dice que ordena un array de objetos, pero bajo la condición de que los objetos deben pertenecer a clases que implementen la interfaz comparable.

El aspecto de la interfaz Comparable es el siguiente:

```
public interface Comparable {
  int compareTo(Object otroObjeto);
}
```

Esto significa que cualquier clase que implemente esta interfaz debe tener un método compareTo, y que dicho método debe tomar un parámetro Object y devolver un entero.

Todos los métodos de una interfaz son automáticamente public. Por esto, no es necesario indicar la palabra reservada public a la hora de declarar dichos métodos.

Desde luego, existe un requisito adicional que una interfaz no puede revisar: cuando se invoca x.compareTo(y), el método compareTo debe ser capaz de comparar dos objetos, y devolver una indicación en función del cual de esos dos objetos sea el mayor. Se asume que el

método devuelve un numero negativo si x es menor que y, cero si son iguales, y un valor positivo en el resto de las situaciones.

La interface Comparable solo tiene un método, pero existen otras que disponen de más de uno. Las interfaces no pueden tener métodos instanciados y sus métodos jamás se encuentran implementados dentro de ellas. Estas dos tareas son responsabilidad de las clases que implementen dicha interfaz.

Ahora suponga que quiere usar el método de la clase Arrays para ordenar un array de objetos Empleado. Por lo tanto, dicha clase debe implementar la interfaz Comparable.

Para que una clase implemente una interfaz, se deben llevar a cabo dos pasos:

1. Declarar que su clase tiene la intención de implementar la interfaz
2. Suministrar definiciones para todos los métodos de dicha interfaz

Para indicar que una clase implementa una interfaz se usa la palabra reservada implements:

Esto implica que la clase Empleado debe suministrar un método compareTo. Vamos a suponer que queremos comparar empleados por sus salarios. Consideraremos lo siguiente para el método compareTo de Empleado, devolverá -1 si el salario del primer empleado es menor que el del segundo, 0 si son iguales, y 1 en el resto de los casos.

Ejemplo 3.6

El siguiente ejemplo muestra la ordenación de un array de empleados.

Ejemplo 3.6

```
package com.empresa.cursos.java.basico.ejemplos.dia3.interfaces;

import java.util.Arrays;

public class EmpleadoSortTest {

  public static void main(String[] args) {
    Empleado[] staff = new Empleado[3];

    staff[0] = new Empleado("Andres Hacker", 35000);
    staff[1] = new Empleado("Carlos Cracker", 75000);
    staff[2] = new Empleado("Vanesa Tester", 38000);

    Arrays.sort(staff);

    //Muestra la información de todos los objetos Empleado
    for (int i=0; i < staff.length; i++) {
      Empleado e = staff[i];
      System.out.println("Nombre: " + e.getNombre() + ", Salario:" + e.getSalario());
```

```
   }

 }
}
```

Vimos que si una clase quiere beneficiarse del servicio de ordenación, debe implementar un método compareTo. Esto resulta bastante razonable. Necesitará alguna forma de que el método sort compare objetos. Pero, ¿Por qué no bastaría con que la clase Empleado simplemente ofreciera un método compareTo, sin necesidad de implementar la interfaz Comparable?

La razón de las interfaces es que Java es un lenguaje estrictamente tipado. Cuando se realiza la invocación de un método, el compilador debe ser capaz de determinar que ese método existe. En alguna parte del método sort podrían existir sentencias como estas:

```
if (a[i].compareTo(a[j])>0) {
  //reestructura a[i] y a[j]
  . . .
}
```

El compilador debe saber que a[i] dispone de un método compareTo. Si a es un array de objetos Comparable, se asegura la existencia del método porque cada clase que implemente la interfaz Comparable debe suministrarlo.

29.1. Propiedades de las interfaces

Las interfaces no son clases. Por lo tanto, nunca podrá usar el operador new para instanciar una interfaz:

```
x = new Comparable(…) ; //ERROR
```

sin embargo, aunque no pueda construir objetos de interfaz, puede declarar variables de interfaz:

```
Comparable x;  //OK
```

Una variable de interfaz debe hacer referencia a un objeto de una clase que implemente esa interfaz:

```
x = new Empleado(…);  //CORRECTO, ya que Empleado implementa Comparable
```

El operador instanceof también puede utilizarse para averiguar si una clase implementa una interfaz determinada.

Al igual que se pueden construir jerarquías de clases, se pueden derivar interfaces. Esto permite definir cadenas de interfaces que van desde un grado superior de generalidad a un nivel mucho más especializado.

Aunque no se puedan poner campos instanciados ni métodos estáticos en una interfaz, si se pueden suministrar constantes en ella.

29.2. Interfaces y clases abstractas

Habiendo visto el concepto de clases abstractas, podría preguntarse porque los diseñadores del lenguaje Java se molestaron en incluir el concepto de interfaz. Por ejemplo: ¿por qué Comparable no podría ser implementada como una clase abstracta?

```
abstract class Comparable { //Porque no?
  public abstract  int compareTo(Object otro);
}
```

Esto permitiría que la clase Empleado derivase de esta clase abstracta y que suministrase el método compareTo

```
class Empleado extends Comparable { //¿Por qué no?
  public int compareTo(Object otro) {…}
}
```

La razón es que una clase solo puede derivar de una única clase padre. En java no existe la herencia múltiple. Por ejemplo la clase Empleado no podría derivar de Persona y Comparable al mismo tiempo. Sin embargo una clase puede implementar tantas interfaces como desee

```
class Empleado extends Persona implements Comparable, OtraInterface  //OK
```

Los diseñadores de Java eligieron no soportar la herencia múltiple porque hacia que el lenguaje fuera más complejo o menos eficiente. En su lugar, las interfaces ofrecen muchos de los beneficios de la herencia múltiple, evitando también dificultades e incoherencias.

30. Atributos y métodos de clase

30.1. Atributos estáticos

Los atributos estáticos, también llamados atributos de clase, son atributos que solo existen para la clase y no para todos los objetos que derivan de ella. Vamos a explicarlo mejor con un ejemplo, suponga que queremos asignar un identificador único a cada Empleado. Añadimos un campo instanciado id y otro campo static nextId a la clase Empleado:

```
class Empleado {
  private int id;
  private static int nextId;
}
```

Ahora, cada objeto Empleado tiene su propio atributo id, pero solo existe un nextId que es compartido por todas las instancias de la clase. Vamos a explicarlo de otro modo. Si existen mil objetos de la clase Empleado, existirán mil atributos instanciados id, uno por cada objeto.

Sin embargo, solo hay un atributo estático nextId. Incluso aunque no existan objetos Empleado, nextId está presente, ya que pertenece a la clase y no a los objetos individuales.

30.2. Constantes

Las variables estáticas son algo raras. Pero las constantes estáticas son bastante más habituales. Por ejemplo, la clase Math define la constante estática PI

```
public class Math {
   public static final double PI = 3.14159265...;
}
```

Puede acceder a ella en sus programas mediante la instrucción Math.PI.

Si se omite la palabra reservada static, PI podría ser un campo instanciado de la clase Math. Esto implicaría tener un objeto de la clase Math para acceder a PI, y cada objeto de esta clase tendría su propia copia de este valor.

30.3. Métodos Estáticos

Los métodos estáticos, también llamados métodos de clase, son métodos que no operan sobre objetos. Por ejemplo pow es un método estático de la clase Math. La expresión:

```
Math.pow(x, y);
```

Obtiene la potencia de x elevado a la y. Este método no utiliza ningún objeto Math para llevar a cabo la operación. En otras palabras, no tiene parámetro implícito. Ya que los métodos estáticos no operan sobre objetos, no se puede acceder a los campos instanciados desde ellos. Sin embargo, si pueden hacerlo a los campos estáticos de su clase.

Los métodos estáticos se utilizan en 2 situaciones:

1. Cuando un método no necesita acceder al estado del objeto porque todos los parámetros necesarios se suministran de forma explícita.
2. Cuando un método solo necesita acceder a los campos estáticos de la clase.

Ejemplo 3.7

A continuación veremos un ejemplo que aplica campos, constantes y métodos estáticos.

Ejemplo 3.7

```
com.empresa.cursos.java.basico.ejemplos.dia3.estaticos

public class StaticTest {

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

```
      //Rellena el array staff con tres objetos Empleado
      Empleado[] staff = new Empleado[3];
      staff[0] = new Empleado("Andres Hacker", 35000);
      staff[1] = new Empleado("Carlos Cracker", 75000);
      staff[2] = new Empleado("Vanesa Tester", 38000);

      //Muestra información sobre todos los objetos Empleado
      for (int i=0; i < staff.length; i++) {
        Empleado e = staff[i];
        e.setId(i);
        System.out.println("Nombre: " + e.getNombre() + ", Salario:" + e.getSalario());
      }

      int n = Empleado.getNextId(); //Llamada al método estático
      System.out.println("Siguiente Id disponible: " + n);
    }

}
```

31. Parámetros de un método

Vamos a recordar un poco el concepto de pasajes de parámetros. El término **paso por valor** significa que el método solo toma el valor que se pasa desde la llamada. Por el contrario, **el paso por referencia** implica que ese método obtiene la localización de la variable pasada en la llamada. De esta manera, un método puede modificar el valor almacenado en dicha variable, cosa que no puede hacerse en el paso por valor.

Java siempre utiliza el paso por valor. Esto significa que el método siempre obtiene una copia de los valores de todos los parámetros. Por lo tanto, no podrá modificar el contenido de ninguna de las variables que recibe.

Por ejemplo, considere la siguiente llamada:

```
double porcentaje = 10;

Roberto.incrementarSalario(porcentaje)
```

Independientemente de cómo este implementado el método, y los cambios que se realicen al parámetro dentro del cuerpo del método, el valor de porcentaje seguirá siendo 10.

Sin embargo existen dos tipos de parámetros de un método

- Tipos primitivos (números, valores lógicos, etc.)
- Referencias a objetos

Ya ha visto que resulta imposible para un método modificar un parámetro de tipo primitivo. La situación cambia con los parámetros que son objetos (esto es, parámetros objetos). Veamos un ejemplo.

```
public static void aumentarSalario(Empleado x) {
   x.incrementarSalario(20);
}
```

Si invoca :

```
carlos = new Empleado(…);
aumentarSalario(carlos);
```

sucede lo siguiente:

1. x se inicializa con una copia del valor de carlos, es decir, una referencia a un objeto
2. el método incrementarSalario se aplica a dicha referencia. El objeto Empleado al cual se refieren tanto x como carlos se aumenta un 20%
3. El método finaliza, y la variable parámetro x ya no se utiliza mas. Por supuesto, la variable objeto carlos sigue apuntando al objeto cuyo salario ha sido incrementado.

Como ha podido ver, resulta posible(y de hecho, es muy común) implementar métodos que cambien el estado de un parámetro objeto. La razón es simple. El método obtiene una copia de la referencia del objeto, y tanto el original como la copia hacen referencia al mismo objeto. Java no utiliza el paso por referencia con los objetos, por el contrario las referencias a objetos son pasadas por valor.

Aquí tiene un resumen de que puede hacer, y que no puede hacer, con los parámetros de un método en Java:

- Un método no puede modificar un parámetro de tipo primitivo (es decir, números o valores lógicos).
- Un método no puede cambiar el estado de un parámetro objeto.
- Un método no puede crear un parámetro objeto que se refiera a un nuevo objeto.

32. Sobrecarga de métodos

La sobrecarga de métodos se produce cuando una clase tiene más de un método con el mismo nombre, pero con distintos parámetros. El compilador debe determinar cuál de estos métodos es el que se está invocando. Para ello, busca entre los diferentes tipos de parámetros de las cabeceras de los métodos que dispone, aquellos que coinciden con los de los valores usados en la llamada al método. En caso de no localizar ninguna coincidencia, o si se dan varias de ellas, se genera un error de compilación (este proceso se denomina resolución de la sobrecarga).

Java le permite sobrecargar cualquier método, no solo los constructores. De este modo, para describir completamente un método, es necesario especificar su nombre junto con sus

tipos de parámetros. Esto es lo que se conoce como firma de un método. Por ejemplo la clase String tiene cuatro métodos IndexOf. Estas son sus firmas:

indexOf(int)

indexOf(int, int)

indexOf(String)

indexOf(String, int)

El valor devuelto no forma parte de la firma del método, es decir no puede tener dos métodos con los mismos nombres y tipos de parámetros, pero con tipos de devolución distintos.

33. Excepciones

En un mundo perfecto, los usuarios nunca introducirían datos de forma errónea, los ficheros a abrir siempre existirían y el código nunca tendría fallos. Por el momento todo el código que hemos visto se ha escrito como si estuviéramos en ese mundo perfecto. Vamos a ver los mecanismos de los que dispone Java para tratar con el mundo real de datos incorrectos y código erróneo.

33.1. Manipulación de excepciones

Suponga que se produce un error mientras un programa Java se está ejecutando. Dicho error podría ser causado por una conexión de red inestable, o por el uso incorrecto de un índice de un arreglo, o un intento de usar una referencia a un objeto que aun no haya sido asignada. Los usuarios esperan que un programa actúe en consecuencia con los errores que se produce. Si una operación no puede ser completada por un error, el programa debería:

- Volver a un estado seguro y permitir al usuario que ejecute otros comandos. O bien
- Permitir al usuario guardar todo su trabajo y finalizar el programa de una manera agradable

La misión de la manipulación de excepciones es transferir el control desde el lugar en el que se produce el error hasta el manipulador que se encarga de lidiar con la situación. Para controlar situaciones excepcionales en su programa, debe tener en cuenta los errores y problemas que pueden surgir.

Java permite a todo método un camino de salida distinto al especificado con la palabra reservada return en caso de que no pueda completar satisfactoriamente su tarea. En esta situación el método lanza un objeto (Exception) que encapsula la información de error. Esto hace que el método finalice inmediatamente, por lo que no devuelve su valor normal. Por otra parte la ejecución no vuelve a la instrucción siguiente a la que llamo al método; en su lugar, el mecanismo de manipulación de excepciones empieza a buscar un manipulador de excepciones que pueda administrar esa condición de error particular.

Las excepciones tienen su propia sintaxis y forman parte de una jerarquía de herencia especial

Veamos un ejemplo de la vida real para ilustrar el uso de las excepciones. Suponga que en este momento me da sed. Muy bien, las acciones que debo realizar para saciar mi sed son las siguientes:

- caminaraHastaLaPuerta()
- abrirLaPuerta()
- caminarHastaLaCocina()
- servirmeAguaEnUnVaso()

Es obvio que si mientras camino hasta la puerta me tropiezo y me rompo una pierna entonces no voy a intentar abrir la puerta ni mucho menos seguir caminando hasta la cocina. Voy a tratar de tomar el teléfono (como pueda) para llamar al médico. Pero si logro llegar con éxito hasta la puerta entonces la intentare abrir. Claro que puede estar trabada entonces ni intento caminar hasta la cocina porque será inútil. Mi problema será lograr abrir la puerta.

Así, generalmente se da que para una sucesión de métodos, el método i+1 debe realizarse si y solo sí el método i finalizo con éxito. A su vez, el método i debió haberse ejecutado si y solo si el método i-1 resulto exitoso. Veamos un código que ilustre este ejemplo.

```
try {
  caminarHastaLaPuerta();
  abrirLaPuerta();
  caminarHastaLaMaquinaDeAgua();
  servirAgua();
}
catch(TropezonException ex) {
  System.out.println("Me tropeze !!!");
}
catch(PuertaNoAbreException ex) {
  System.out.println("Ohh nooo !!!");
}
catch(NoHayAguaException ex) {
  System.out.println("No hay agua en el bidón...");
}
catch(Exception ex) {
  System.out.println("Error inesperado");
  ex.printStackTrace();
}
```

33.2. Captura o Propagación de excepciones

Para tratar con excepciones tenemos dos posibilidades: una es capturar la excepción que lanza el método utilizando la sentencia catch. Pero muchas veces no es nuestra responsabilidad emprender las acciones correctivas. En ese caso simplemente debemos propagar el error, con el uso de la palabra reservada throws para que se haga cargo el llamador del método. A su vez, si él puede tratar la excepción lo hará mediante un bloque try-catch pero si no puede tratarlo entonces propagará el error y así sucesivamente.

Ejemplo 3.8

El siguiente ejemplo es una clase con tres métodos: metodo1(), metodo2() y metodo3(). Si el método metodo1() falla entonces disparara una excepción del tipo M1Exception. Análogamente si metodo2() o metodo3() fallan dispararan M2Exception y M3Exception respectivamente.

Los métodos reciben un boolean error con el que indicaremos si queremos que el método falle o no. Si error=true entonces el método lanzara la excepción que le corresponde. Si no finalizara con éxito.

Un método que eventualmente pueda lanzar una excepción debe especificarse con la palabra reservada throws. Esto forzara al llamador del método a encerrarlo en un bloque try-catch o bien a propagar el error.

Ejemplo 3.8

```
package com.empresa.cursos.java.basico.ejemplos.dia3.exceptions;
public class ExceptionTest
{
  public static void main(String[] args) {
    try {
        boolean errorM1;
        boolean errorM2;
        boolean errorM3;
        errorM1= Boolean.valueOf(args[0]).booleanValue();
        errorM2= Boolean.valueOf(args[1]).booleanValue();
        errorM3= Boolean.valueOf(args[2]).booleanValue();
        ExceptionTest se = new ExceptionTest();
        se.metodo1(errorM1);
        se.metodo2(errorM2);
        se.metodo3(errorM3);
    }
        catch(M1Exception ex) {
          ex.printStackTrace();
          System.out.println("Error en el método metodo1()");
       }
       catch(M2Exception ex) {
```

```
            ex.printStackTrace();
            System.out.println("Error en el método metodo2()");
        }
        catch(M3Exception ex) {
            ex.printStackTrace();
            System.out.println("Error en el método metodo3()");
        }
        catch(Exception ex) {
            ex.printStackTrace();
             System.out.println( "Error inesperado. Probablemente estén mal pasados los
argumentos");
        }

    }
    public void metodo1(boolean error) throws M1Exception {
      System.out.println("Comienza metodo1()");
      if(error) {
          throw new M1Exception();
      }
      System.out.println("Termina metodo1() ok!");
    }


    public void metodo2(boolean error) throws M2Exception{
       System.out.println("Comienza metodo2()");
       if(error) {
           throw new M2Exception();
       }
       System.out.println("Termina metodo2() ok!");
    }

    public void metodo3(boolean error) throws M3Exception {
       System.out.println("Comienza metodo3()");
       if(error){
          throw new M3Exception();
       }
       System.out.println("Termina metodo3() ok!");
    }
}

class M1Exception extends Exception {
}

class M2Exception extends Exception {
```

```
}

class M3Exception extends Exception {
}
```

33.3. Algunos consejos para el uso de excepciones

- La manipulación de excepciones no debe sustituir a una simple comprobación
- No suprima las excepciones
- Propagar excepciones no es ninguna deshonra

Día 4: Clases básicas de Java

34. String

Las cadenas de caracteres son secuencias de caracteres como "Hola". Java no tiene un tipo cadena definido. En vez de eso la biblioteca estándar de Java tiene una clase predefinida llamada String. Toda cadena de caracteres encerrada entre comillas es una instancia de la clase String:

```
String e = ""; //una cadena de caracteres vacía

String saludo = "Hola";
```

34.1. Concatenación

Java utilizar el operador + para realizar la concatenación entre cadenas de caracteres

```
String localidad = "San Antonio" ;

String nombre = "Spurs";

String nombreEquipo = localidad + " " + nombre;
```

Este código pone en la variable de cadena nombreEquipo "San Antonio Spurs".

Al concatenar una cadena de caracteres con un valor que no lo es, este último se convierte en una cadena de caracteres. Por ejemplo:

```
int edad = 13;

String apto = "Para mayores de " + edad;
```

Hace que apto tenga la cadena de caracteres "Para mayores de 13". Esta característica se utiliza en sentencias de impresión. Por ejemplo:

```
System.out.println("Película apta para mayores de " + edad + " años");
```

34.2. Subcadenas de caracteres

Puede extraer una subcadena de una cadena mayor con el método substring de la clase String. Por ejemplo:

```
String saludo = “Hola”;

String s  = saludo.substring(0, 2);
```

Crea una cadena con los caracteres “Ho”. Java cuenta los caracteres en cadena de una forma peculiar: el primer carácter de la cadena tiene la posición 0. Por ejemplo el carácter ‘H’ está en la posición 0 de la cadena “Hola”, y el carácter ‘a’ está en la posición 3. El segundo parámetro de substring es la primera posición que no quiere copiar. En nuestro caso, queremos copiar los caracteres de las posiciones 0 y 1. Tal como cuenta substring desde la posición 0 inclusive hasta la posición 2 exclusive.

Debido a esta forma de funcionar de substring es fácil contar la longitud de una subcadena de caracteres. La cadena s.substring(a, b) tiene la longitud b – a.

34.3. Edición de cadenas de caracteres

Para saber la longitud de una cadena de caracteres utilice el método length. Por ejemplo:

```
String saludo = “Hola”;

int n = saludo.length();
```

De la misma forma que char indica un carácter Unicode, String indica una secuencia de caracteres Unicode. Es posible recuperar un carácter individual de una cadena con el método charAt(n). Este método devuelve el carácter Unicode de la posición n, donde n esta entre 0 y n - 1. Por ejemplo:

```
char ultimo = saludo.charAt(3); //El tercero vale ‘a’.
```

Sin embargo la clase String no tiene métodos que le permitan cambiar un carácter de una cadena. Si desea cambiar saludo a “Hol!”, no puede cambiar directamente la última posición de saludo a ‘!’. En Java para modificar una cadena se debe obtener la subcadena que quiere mantener y agregar los caracteres que quiere reemplazar.

```
saludo = saludo.substring(0, 3) + ‘!’;
```

Esto cambia el valor de la cadena saludo a “Hol!”

Dado que no puede cambiar los caracteres individuales de una cadena en Java los objetos de la clase String se conocen como inmutables. Esta característica de cadenas inmutables se utiliza para que el compilador pueda organizar estas cadenas para que sean compartidas.

34.4. Comprobación de igualdad entre cadenas

Para comprobar si dos cadenas de caracteres son iguales o no, utilice el método equals. La expresión:

```
s.equals(t)
```

devuelve true si las cadenas s y t son iguales, y false si son distintas. Tenga en cuenta que s y t pueden ser variables de cadena o constantes. Por ejemplo:

```
"Hola".equals(comando);
```

Es perfectamente válida. Para comprobar si dos cadenas son iguales sin tener en cuenta mayúsculas/minúsculas, utilice el método equalsIgnoreCase.

```
"Hola".equalsIgnoreCase("holA");
```

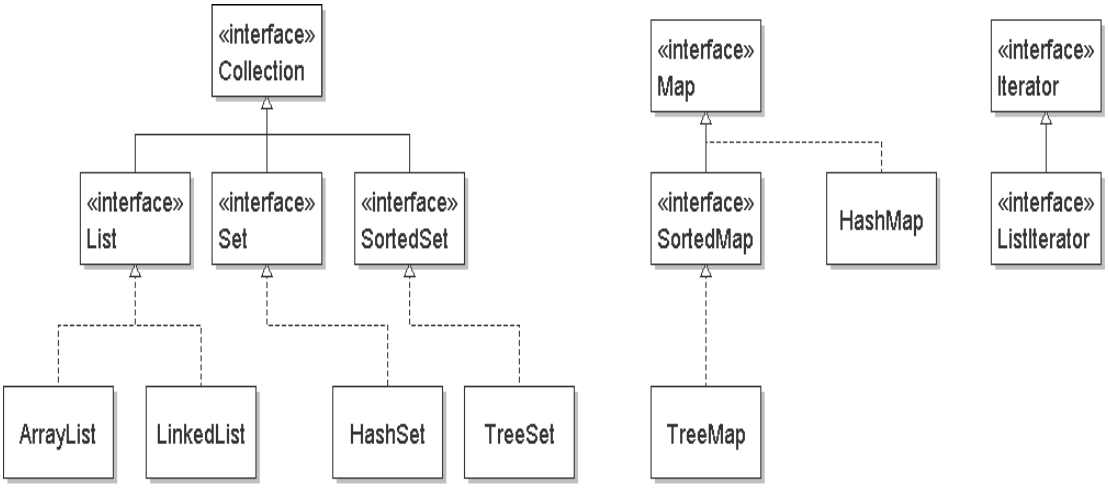
No utilice el operador == para comprobar que dos cadenas sean iguales. Este operador solamente le dirá si dos cadenas se almacenan en la misma localización. Por supuesto si las dos cadenas están en la misma localización deben ser iguales. Pero es perfectamente posible almacenar múltiples copias de cadenas idénticas en sitios diferentes.

Recuerde, nunca utilice == para comparar cadenas de caracteres o tendrá un programa con la peor clase de error (un error intermitente que parece que ocurre de forma aleatoria).

35. Colecciones de Objetos

En general, cuando usted programe, siempre creará nuevos objetos basándose en algún criterio que conocerá solo en tiempo de ejecución. Usted no sabrá la cantidad o el tipo exacto de objetos que necesita hasta que el programa se esté ejecutando. Para resolver este problema de programación, necesitara ser capaz de crear un número indeterminado de objetos en cualquier momento. Por lo tanto no puede contar con referencias a objetos para mantener cada uno de estos objetos creados en tiempo de ejecución, ya que nunca sabrá en tiempo de diseño que cantidad de estos objetos necesitara.

Java proporciona varias formas para enfrentar este problema frecuente de programación. Una técnica bastante común es la utilización de un array de una determinada clase de objeto, o de un tipo primitivo. Pero Java también proporciona una serie de clases e interfaces, conocido como framework de colecciones para mantener y manipular sus objetos. Veamos la arquitectura básica de este framework.



Las clases son implementaciones de estructuras de datos que seguramente usted ya conoce, como ser: Lista enlazada, Árbol, etc. Vea a las instancias de estas clases como colecciones de objetos que tienen una estructura de datos determinada. Las interfaces proveen métodos que son implementados por estas clases.

La razón de esta estructura se debe a que es posible independizarse de la implementación de la estructura de datos y utilizar directamente la interfaz. Por ejemplo es posible utilizar la interfaz List cuando en realidad los métodos de esta interfaz pueden estar implementados por la clase ArrayList o bien LinkedList.

A continuación nos enfocaremos en la interfaces Collection, e Iterator. Y veremos la implementación ArrayList.

35.1. Interfaz Collection

La interfaz fundamental del framework de colecciones es la interfaz Collection. Esta interfaz tiene dos métodos fundamentales:

```
boolean add (Object objeto)

Iterator iterator()
```

Además de estos dos existen otros métodos. El método add añade un objeto a la colección, y devuelve true si dicho objeto cambia la colección, o false en caso contrario. Por ejemplo, si intenta añadir un objeto a un conjunto y dicho objeto ya está presente en el, se rechaza la petición, ya que un conjunto no posee elementos repetidos, en este caso add devuelve false. El método iterator devuelve un objeto que implementa la interfaz Iterator.

35.2. Interfaz Iterator

La interfaz Iterator dispone de tres métodos fundamentales

```
Object next()

boolean hasNext()

void remove()
```

Las llamadas repetidas a next le permiten recorrer los elementos de una colección uno a uno. Sin embargo cuando se alcanza el final de la colección, next lanza una NoSuchElementException. Por lo tanto es necesario invocar al método hasNext antes que a next. El método hasNext devuelve true si el objeto iterador aun dispone de más elementos que visitar. Si quiere inspeccionar todos los elementos de una colección, debe crear un iterador y llamar a next mientras hasNext devuelva true.

```
Iterator iter = c.iterador();
while (iter.hasNext()) {
   Object obj = iter.next();
   //hacer algo con obj
}
```

El método remove elimina el elemento que ha sido devuelto por la última llamada al método next. Debe ser cuidadoso cuando utilice este método, ya que la llamada al mismo elimina el elemento que fue devuelto por la última llamada a next. Debe tener en cuenta que si desea realizar una eliminación en función de una posición, deberá primero localizar ese elemento mediante next, no es posible otra forma con esta interfaz. Por ejemplo aquí tiene la forma de eliminar el primer elemento de una colección.

```
Iterator iter = c.iterator();
iter.next(); //Se posiciona en el primer elemento
iter.remove();
```

También debe tener en cuenta que existe una correspondencia entre las llamadas a los métodos next y remove. Es ilegal realizar una llamada a remove si no ha existido una llamada a next anteriormente. Si lo intenta obtendrá IlegalStateException.

Si quiere borrar dos elementos adyacentes no podrá hacer algo como:

```
iter.remove();
iter.remove();  //ERROR!!
```

35.3. Clase ArrayList

La clase ArrayList se utiliza para construir un arreglo de objetos redimensionable. Es un arreglo de objetos, que a diferencia de un array común, puede aumentar su tamaño conforme se agregan elementos. Esto se logra gracias a que internamente maneja una lista de objetos y de allí su nombre. Cuando se crea un ArrayList es posible especificar la cantidad de elementos iniciales que tendrá, si no se especifica nada toma la dimensión por defecto(10):

```
ArrayList arreglo1 = new ArrayList(); //El arreglo tiene la dimensión por defecto, 10 elementos

ArrayList arreglo2 = new ArrayList(2); //El arreglo tiene una dimensión de 2
```

Para agregar un elemento al arreglo es posible agregarlo al final de la lista, o bien en una posición determinada del arreglo

```
arreglo1.add(new Empleado("Marcelo", 50000));

arreglo1.add(2, new Empleado("Marcelo", 50000));
```

Para quitar un elemento del arreglo, o bien para quitar todos los elementos del arreglo se utilizan los métodos:

```
arreglo2.remove(2); //Quita el elemento ubicado en la posición 2 del arreglo

arreglo2.clear(), //Quita todos los elementos del arreglo
```

Para averiguar si el arreglo tiene elementos se utiliza el método isEmpty();

```
arreglo1.isEmpty()

//Devuelve true si el arreglo no contiene elementos, o false en caso contrario
```

Ejemplo 4.1

Podemos ver la utilización de estos métodos en el siguiente ejemplo. Se crean 2 arreglos, uno especificando la dimensión, otro sin especificar la dimensión. Vemos como crece un arreglo cuando la adición de un elemento provoca el crecimiento del mismo. También vemos la eliminación de elementos del arreglo.

Ejemplo 4.1

```
package com.empresa.cursos.java.basico.ejemplos.dia4.colecciones;
import java.util.ArrayList;

public class ArrayListTest
{
  public static void main(String[] args)  {
    //Creacion de ArrayLists
    ArrayList arreglo1 = new ArrayList();
    ArrayList arreglo2 = new ArrayList(2);
    arreglo1.add(new Empleado("Roberto Hacker", 50000));
```

```
arreglo1.add(new Empleado("Emanuel Ginobili", 53000));
arreglo1.add(new Empleado("Steve Muench", 53000));
Empleado carlos = new Empleado("Carlos Santana", 50000);
arreglo2.add(carlos);
arreglo2.add(carlos);
arreglo2.add(carlos);

//Mostrar los elementos del arreglo1
for (int i=0; i<arreglo1.size(); i++) {
  Empleado e = (Empleado)arreglo1.get(i);
  System.out.println("Nombre: " + e.getNombre());
  System.out.println("Salario: " + e.getSalario());
}

//Mostrar los elementos del arreglo2
for (int i=0; i<arreglo2.size(); i++) {
  Empleado e = (Empleado)arreglo2.get(i);
  System.out.println("Nombre: " + e.getNombre());
  System.out.println("Salario: " + e.getSalario());
}

//Se guitan objetos de los arreglos
arreglo1.remove(1);
arreglo2.clear();

//Mostrar los elementos del arreglo1 después del remove
System.out.println("arreglo1 después del remove");
for (int i=0; i<arreglo1.size(); i++) {
  Empleado e = (Empleado)arreglo1.get(i);
  System.out.println("Nombre: " + e.getNombre());
  System.out.println("Salario: " + e.getSalario());
}

//Mostrar los elementos del arreglo2
System.out.println("arreglo2 después del remove");
for (int i=0; i<arreglo2.size(); i++) {
  Empleado e = (Empleado)arreglo2.get(i);
  System.out.println("Nombre: " + e.getNombre());
  System.out.println("Salario: " + e.getSalario());
}
```

```
  }
}
```

Ejemplo 4.2

El ejemplo anterior también se podría haber realizado utilizando las interfaces Collection e Iterator, como se ve en el siguiente ejemplo:

Ejemplo 4.2

```
package com.empresa.cursos.java.basico.ejemplos.dia4.colecciones;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;


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

    //Creacion de ArrayLists
    Collection collection = new ArrayList();
    collection.add(new Empleado("Roberto Hacker", 50000));
    collection.add(new Empleado("Emanuel Ginobili", 53000));
    collection.add(new Empleado("Steve Muench", 53000));

    //Mostrar los elementos del arreglo1
    Iterator iter1 = collection.iterator();
    while(iter1.hasNext()) {
      Empleado e = (Empleado)iter1.next();
      System.out.println("Nombre: " + e.getNombre());
      System.out.println("Salario: " + e.getSalario());
    }

    //Eliminar el empleado llamado Steve Muench
    Iterator iter2 = collection.iterator();
    while(iter2.hasNext()) {
      Empleado e = (Empleado)iter2.next();
      if (e.getNombre().equals("Steve Muench")) {
        iter2.remove();
      }
    }

    //Mostrar los elementos del arreglo1 después de la eliminación
    Iterator iter3 = collection.iterator();
```

```
    System.out.println("collection despise del remove");
    while(iter3.hasNext()) {
      Empleado e = (Empleado)iter3.next();
      System.out.println("Nombre: " + e.getNombre());
      System.out.println("Salario: " + e.getSalario());
    }

  }
}
```

36. Interfaces Gráficas AWT y SWING

Para la construcción de interfaces graficas de Usuario utilizaremos las bibliotecas Swing y Awt de Java. En un principio, solo existía Awt, utilizaba un enfoque de iguales (peers). Esto consistía en delegar la creación y el comportamiento de los componentes de GUI a las herramientas nativas de la GUI de la plataforma destino (Windows, Solaris, etc.). Esto trajo muchos problemas que se vieron solucionados con la aparición de Swing, que delega solo las responsabilidades más básicas y comunes a la plataforma destino, como la creación de una ventana. Swing no reemplaza totalmente a Awt, se complementan. En la actualidad se utiliza Swing cuando se trabaja con componentes de GUI que no siguen el enfoque de iguales, y con Awt para el manejo de eventos.

36.1. Creación de un Marco

Un marco en Java es una ventana del nivel más alto(es decir, una ventana que no está contenida en otra). La biblioteca AWT tiene una clase, llamada Frame, para este nivel superior. La versión Swing se llama JFrame, y deriva de la clase Frame. La clase JFrame es uno de los pocos componentes Swing que no son dibujadas por Swing, sino que las dibuja el sistema de ventanas del usuario.

La mayoría de las clases Swing comienzan con una J. JButton, JFrame, etc.

Los marcos son ejemplos de contenedores. Esto quiere decir que un marco puede contener otros componentes de la interfaz del usuario como botones y campos de texto.

En esta sección veremos los métodos más comunes para trabajar con un JFrame de Swing y comentaremos el manejo de eventos para los componentes GUI.

Ejemplo 4.3

Ejemplo 4.3

```
import javax.swing.*;

public class SimpleFrameTest {
```

```
  public static void main(String[] args) {
    SimpleFrame frame = new SimpleFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.show();
  }

}

class SimpleFrame extends JFrame {
  public static final int DEFAULT_WIDTH = 300;
  public static final int DEFAULT_HEIGHT = 200;

  public SimpleFrame() {
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
  }

}
```

Analicemos el programa línea a línea.

Las clases de Swing están en el paquete javax.swing. Por defecto un marco tiene un tamaño de 0x0 pixeles, que no sirve para nada. Definimos una subclase SimpleFrame cuyo constructor establece el tamaño a 300x200 pixeles. En el método main de la clase SimpleFrameTest, empezamos construyendo un objeto SimpleFrame.

A continuación definimos que debería pasar cuando el usuario cierra el marco. En este programa en particular, queremos que el programa finalice. Para hacer que se comporte así, se coloca la línea que puede ver en el ejemplo. En programas con muchos marcos no querrá que el programa finalice al cerrarse un marco. Por defecto, un marco se oculta cuando el usuario lo cierra, pero el programa no termina.

La construcción de un marco no hace que se muestre de forma automática. Los marcos son, en principio, invisibles. Esto le da al programador la oportunidad de añadir componentes en el marco antes de mostrarlo por primera vez. Para mostrar el marco, el método main llama al método show del marco.

Después, el método main acaba. El hecho de que acabe main no hace que finalice el programa, solamente el thread principal. Al mostrar el marco se activa un thread de la interfaz del usuario que hace que el programa se mantenga vivo.

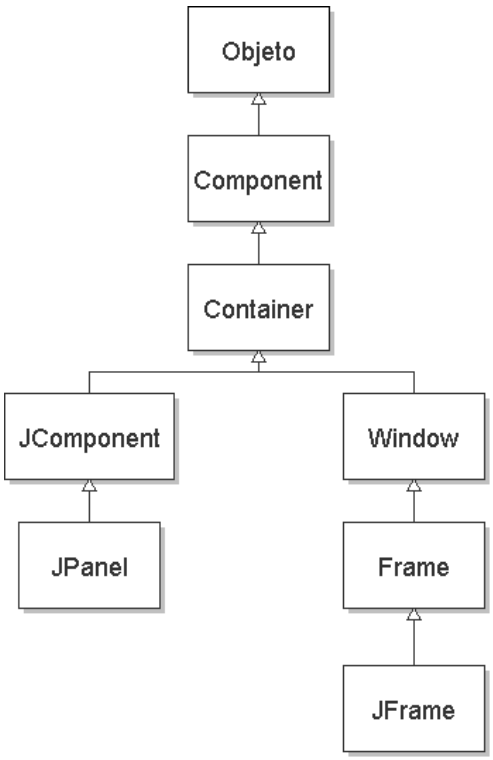
Este programa cuando se ejecuta muestra una ventana sin nada, solo una barra de titulo y lo que la acompaña, como el redimensionamiento de las esquinas. Esto lo dibuja el sistema operativo y no la biblioteca Swing. La biblioteca Swing dibuja lo que hay dentro del marco, en el caso del programa anterior, lo único que hace Swing es poner en el marco un color de fondo.

36.2. Posicionamiento de un marco

La clase JFrame por si misma tiene unos pocos métodos para cambiar el aspecto del marco. Pero debido a la herencia, la mayoría de los métodos para trabajar con el tamaño y la posición del marco vienen de varias superclases de JFrame. Entre los métodos más importantes se encuentran los siguientes:

- El método dispose, que cierra la ventana y recupera los recursos del sistema que se han utilizado para crearla
- El método setIconImage, que toma un objeto Image para usarlo como icono cuando la ventana es iconizada
- El método setTitle, para cambiar el texto de la barra de titulo
- El método setResizable, que toma un valor boolean para determinar si un marco podrá ser redimensionado por el usuario.

Para saber más acerca de los métodos de los marcos puede consultar las superclases de la clase JFrame. A continuación se muestra la jerarquía de clases para los componentes GUI.



Como podrá imaginarse la biblioteca Swing es muy extensa y una descripción de sus componentes llevaría un libro completo. Completaremos esta sección comentándole como es que Java maneja los eventos que se realizan sobre los componentes GUI.

36.3. Introducción a la manipulación de Eventos

Cualquier entorno que soporte GUI constantemente monitoriza eventos tales como las pulsaciones del teclado o los clicks del ratón. El entorno informa de estos eventos a los programas que se están ejecutando. Cada programa decide que hay que hacer, si es que hay que hacer algo, en respuesta a esos eventos. En lenguajes como Visual Basic, la correspondencia entre los eventos y el código es obvia. Se escribe el código para cada evento específico de interés y se pone el código en lo que se llama un procedimiento de evento. Por otra parte, si utiliza un lenguaje como C, para programar eventos se tendrá que escribir código que controle continuamente la cola de eventos, para ver la información que está generando el entorno.

Java hace una aproximación entre Visual Basic y C en términos de potencia y, por lo tanto, en complejidad. Dentro de los límites de los eventos que AWT conoce, se controla la forma en la que estos eventos se transmiten desde los orígenes de los eventos (tales como botones, barras de desplazamiento, ventanas, etc.) a los oyentes de los eventos. Puede hacer que cualquier objeto sea un oyente de eventos. En la práctica elija un objeto que pueda cumplir convenientemente la respuesta deseada al evento.

Los orígenes de los eventos tienen métodos que le permiten registrar oyentes de eventos. Cuando ocurre un evento en el origen, el origen envía una notificación de ese evento a todos los objetos oyentes que se registraron para ese evento.

Como se estará imaginando, la información del evento se encapsula en un objeto evento. En java, todos los objetos de eventos derivan de la clase java.util.EventObject. Por supuesto hay subclases para cada tipo de evento, tales como ActionEvent y WindowEvent.

Diferentes orígenes de eventos pueden producir diferentes tipos de eventos. Por ejemplo, un botón puede generar un objeto ActionEvent, mientras que una ventana puede enviar objetos WindowEvent.

Para resumir mostramos cómo funciona la manipulación de eventos en el AWT:

- Un objeto oyente es una instancia de una clase que implementa una interfaz especial llamada interfaz oyente (listener)
- Un origen de eventos es un objeto (por lo general un componente GUI) que puede registrar objetos oyentes y enviarles objetos de eventos.
- El origen de eventos envía objetos de eventos a todos los oyentes registrados cuando ocurre ese evento.

A continuación, los objetos oyentes usaran la información del objeto evento para determinar su reacción al evento.

Se registra el objeto oyente con el objeto origen del evento con las líneas de código que siguen este modelo:

objetoOrigenEvento.add*Evento*Listener(*objetoOyenteEvento*);

Por ejemplo:

```
ActionListener listener = …;
```

```
JButton boton = new JButton("Ok");
boton.addActionListener(listener);
```

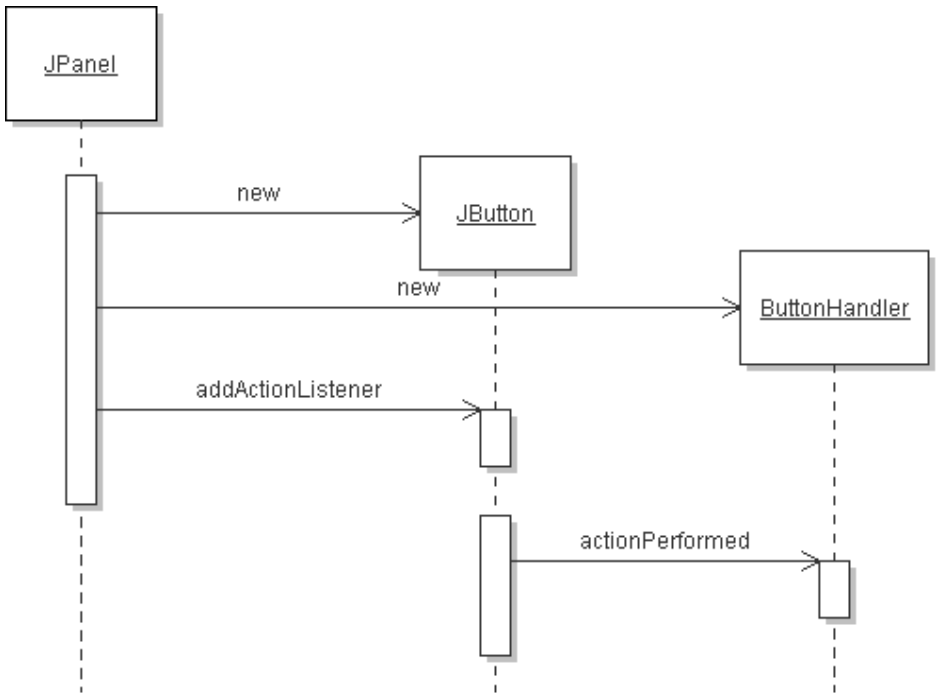
Ahora al objeto listener se le notifica cada vez que ocurre un "evento de acción" en el botón. Para los botones, como sería de esperar un evento de acción es un click en el botón.

Un código como el anterior requiere que la clase a la que pertenece el objeto oyente implemente la interfaz apropiada (que en este caso es la interfaz ActionListener). Como con todas las interfaces en Java, implementar una interfaz significa facilitar los métodos con las firmas apropiadas. Para implementar la interfaz ActionListener, la clase oyente debe tener un método llamado actionPerformed que reciba un objeto ActionEvent como parámetro:

```
class buttonHandler implements ActionListener {
  …
  public void actionPerformed(ActionEvent event) {
    //La reacción al click del botón va aquí.
    …
  }
}
```

Cuando el usuario haga clic sobre el botón, el objeto JButton creará un objeto ActionEvent y llamará a listener.actionPerformed(event) pasando ese objeto de evento. Es posible añadir diversos objetos como oyentes a un origen de eventos, como por ejemplo un botón. En ese caso, el botón llamará a los métodos actionPerformed para todos los oyentes cuando el usuario haga clic en el botón.

La siguiente figura muestra la interacción entre el origen de eventos, el oyente de eventos, y el objeto evento.


JPanel
new
JButton
new
ButtonHandler
addActionListener
actionPerformed

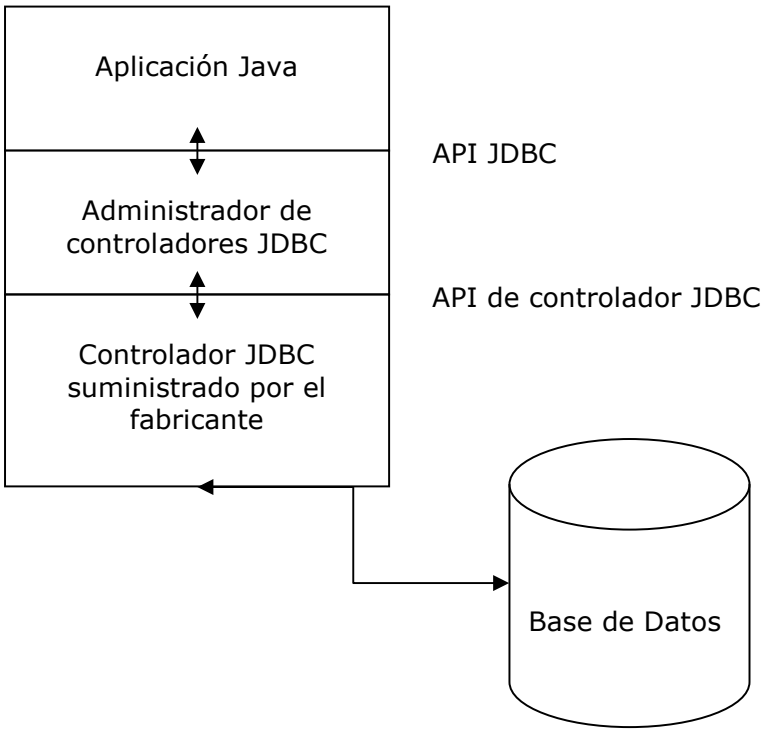
Día 5: JDBC Acceso a Bases de Datos

37. Introducción a JDBC

Ya hemos visto que Java es multiplataforma. Con la API JDBC (Java DataBase Connectivity) Java permite que su aplicación pueda interactuar con cualquier base de Datos, Oracle, MySql, Postgree, etc, que provea un driver para Java.

El funcionamiento se basa en un enfoque por capas. En la capa superior tenemos la aplicación Java, aquí haremos uso de todos los servicios proporcionados por la capa inferior enviándole distintas sentencias SQL, para ello utilizaremos las clases e interfaces proporcionados por la API JDBC. La capa de Administrador de Controlador de JDBC es transparente para el desarrollador. Se comunica con los controladores proporcionados por los fabricantes de Base de Datos. Esta es la capa que permite la pluridad de Base de Datos con la cuales puede trabajar Java. Para comunicarse con el controlador utiliza la API de Controlador JDBC. Los controladores suministrados por los fabricantes de Base de Datos deben ser construidos respetando la interfaz de controlador JDBC, esto es de interés solo para distribuidores de Bases de Datos y de herramientas.

La siguiente figura ilustra el funcionamiento:



El objetivo final de JDBC es hacer posible lo siguiente:

- Los programadores pueden escribir aplicaciones en lenguaje Java para acceder a cualquier Base de Datos usando sentencias SQL estándar (o, incluso, extensiones especializadas del mismo)
- Los distribuidores de Bases de Datos y de herramientas pueden ofrecer los controladores de Bajo nivel. De esta forma, puede optimizar esos controladores para sus productos específicos.

38. Conceptos básicos de programación JDBC

La programación con las clases JDBC no difiere en exceso de la que se realiza con las clases habituales de Java: se construyen objetos de las clases JDBC básicas, heredándolas jerárquicamente en caso de ser necesario.

38.1. El URL de la base de datos

Cuando se conecta con una base de datos, se debe especificar el origen de los datos y, en algunos casos, otros parámetros. JDBC utiliza una sintaxis similar a un URL corriente para describir las fuentes de datos. Aquí tiene un ejemplo de dicha sintaxis:

jdbc:db2:COREJAVA

jdbc:oracle:thin:@192.168.0.100:1521:SION

La sintaxis general es:

jdbc:nombre_subprotocolo:otros_elementos

donde se utiliza nombre_suprotocolo para seleccionar un controlador concreto para la conexión con la Base de Datos. El formato del parámetro otros elementos depende del subprotocolo empleado. Debe estudiar la documentación del fabricante para obtener el formato específico.

38.2. Realización de la conexión

Lo primero es conocer los nombres de las clases de los controladores JDBC utilizados por el fabricante. Algunos ejemplos son los siguientes:

COM.ibm.db2.jdbc.app.DB2Driver

oracle.jdbc.driver.OracleDriver

A continuación, es necesario encontrar la librería donde se encuentra el controlador. Es necesario indicar la ruta completa a ese controlador en la ruta de clases. Para ello use uno de estos mecanismos:

- Lance sus programas de bases de datos con el argumento de línea de comandos –classpath
- Modifique la variable de entorno CLASSPATH
- Copie la librería de base de datos en el directorio jre/lib/ext

DriverManager es la clase responsable de la selección de los controladores de base de datos y de la creación de las conexiones pertinentes. Sin embargo, antes de que el administrador de controladores (DriverManager) pueda activarlo, ese controlador debe estar registrado.

Existen dos formas de llevar a cabo esta operación de registro.

La propiedad jdbc.drivers contiene una lista de los nombres de clases de los controladores que el administrador deberá registrar en el arranque. Dichos nombres deben separarse por el caracter dos puntos

Se puede especificar la propiedad con un argumento de la linea de comandos:

Java –D jdbc.drivers= oracle.jdbc.driver.OracleDriver MiProgram

O hacer que su aplicación lea un archivo de propiedades con la linea

jdbc.drivers= oracle.jdbc.driver.OracleDriver

Y que añada estos parámetros a las propiedades del sistema.

Este último planteamiento es bastante utilizado, permite a los usuarios de su aplicación que instalen los controladores apropiados modificando el archivo de propiedades. De esta

manera, los programas leen todos los parámetros de la base de datos desde un archivo database.properties. Cuando utilice este enfoque asegúrese de editar el archivo de propiedades para localizar su base de datos.

Un procedimiento simple para establecer la conexión con la base de datos, sin utilizar un archivo de propiedades, es el siguiente:

```
String driver = "oracle.jdbc.driver.OracleDriver";
String url = "jdbc:oracle:thin:@192.168.0.105:1521:SION";
String username = "test";
String password = "test";
Class.forName(driver);
return DriverManager.getConnection(url, username, password);
```

Class.forName(driver) es una forma alternativa de cargar el controlador. Sin embargo, se debe especificar la ruta completa de la clase del controlador en el CLASSPATH.

El administrador de controladores tratara de encontrar un controlador que use el protocolo especificado en la url de la base de datos, recorriendo todos los controladores registrados en el.

Antes de utilizar JDBC es recomendable comprobar el funcionamiento de la base de datos sin JDBC. Una vez verificado el funcionamiento de la base de datos, asegúrese de que cuenta con la siguiente información, necesaria para la configuración de JDBC

- El nombre de usuario y al contraseña de la base de datos
- El nombre de la base de datos a utilizar
- El formato del URL JDBC
- El nombre del controlador JDBC
- La localización de los archivos de librería que contienen el código del controlador.

puede utilizar el siguiente ejemplo como programa de prueba para comprobar su configuración JDBC. En el ejemplo se utiliza una Base de Datos Oracle, se podría utilizar cualquier otra, teniendo en cuenta las respectivas variaciones de configuración.

Ejemplo 5.1

Ejemplo 5.1

```
package com.empresa.cursos.java.basico.ejemplos.dia5.jdbc;

import java.io.FileInputStream;
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import java.sql.Statement;
import java.util.Properties;


/**
 * Este programa controla si la Base de Datos y el controlador JDBC
 * estan correctamente configurados
 */

public class DataBaseTest {
	public static void main(String[] args) throws Exception {
		try {
		   Connection conn = getConnection();
		   Statement stat = conn.createStatement();
		   stat.execute("CREATE TABLE Saludo (Name CHAR(20))");
		   stat.execute("INSERT INTO Saludo VALUES ('Hola Mundo!')");
		   ResultSet result = stat.executeQuery("SELECT * FROM Saludo");
		   result.next();
		   System.out.println(result.getString(1));
		   result.close();
		   stat.execute("DROP TABLE Saludo");
		   stat.close();
		   conn.close();

		  }
		  catch (SQLException ex) {
		    while (ex !=null) {
		      ex.printStackTrace();
		      ex = ex.getNextException();
		    }
		  }
		  catch (IOException ex) {
		    ex.printStackTrace();
		  }

		}


		/**
		 * Obtiene una conexion a partir de las propiedades especificadas en el archivo
		 * database.properties
		 * @return la conexion con la base de datos
		 */
		public static Connection getConnection() throws Exception {
		  String driver = "oracle.jdbc.driver.OracleDriver";
		  String url = "jdbc:oracle:thin:@192.168.80.164:1521:xe";
		  String username = "CURSO_JAVA";
		  String password = "CURSO_JAVA";
```

```
            Class.forName(driver);
            return DriverManager.getConnection(url, username, password);
        }

}
```

38.3. Ejecución de Comandos SQL

Para ejecutar comandos SQL, primero debo crear un objeto Statement. Para ello, puede utilizar el objeto Connection.

```
Statement stat = conn.createStatement();
```

A continuación escriba la sentencia SQL que desee ejecutar en una cadena, por ejemplo

```
String comandoSQL = "INSERT INTO EMPLEADOS"
```

Una vez definida la sentencia deberá determinar si la sentencia se debe ejecutar con un método executeUpdate o executeQuery.

El método executeUpdate puede ejecutar acciones del tipo INSERT, UPDATE, y DELETE, así como comandos DDL como CREATE TABLE, y DROP TABLE. Este método devuelve el número de filas que han sido afectadas por la sentencia SQL

El método executeQuery se utiliza para ejecutar sentencias SELECT. Este método devuelve un objeto del tipo ResultSet que se utiliza para procesar las filas de un resultado. Por ejemplo:

El bucle básico para analizar un conjunto de resultados tiene este aspecto:

```
while (rs.next()) {
  //Inspeccionar una fila del resultado
}
```

Cuando se inspecciona una fila querrá conocer el contenido de cada columna. Existen varios métodos para obtener esta información:

```
String  nombre = rs.getString("Nombre");
String empleadoId = rs.get(1);
```

Existen métodos de este tipo para cada uno de los tipos de datos del lenguaje Java. Cada método tiene dos formas, una que toma un argumento numérico y otra que toma una cadena. En el primer caso, se hace referencia a una columna que se encuentra en la posición marcada por el número. Por ejemplo. rs.getString(3) devuelve el valor de la tercera columna de la fila actual.

Cuando se suministra una cadena como argumento, se hace referencia a la columna del conjunto de resultados que tenga ese nombre. Por ejemplo rs.getString("Nombre") devuelve el valor de la columna nombre.

El uso de argumentos numéricos es un poco más eficientes, pero las cadenas hacen que el código sea más legible y fácil de mantener.

Cada método get realizara la conversión adecuada cuando no coincidan el tipo del método y el de la columna. Por ejemplo, la llamada a rs.getString("Salario") convierte el valor en punto flotante de la columna salario en una cadena.

Ejemplo 5.2

En el siguiente ejemplo vemos la ejecución de sentencias SQL con los métodos executeUpdate y executeQuery.

Ejemplo 5.2

```
package com.empresa.cursos.java.basico.ejemplos.dia5.jdbc;

import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class DataBaseQuery {
        private Connection conn;
        private Statement stat;
        private String empleadosQuery = "SELECT Empleados.Empleado_Id,
Empleados.Nombre, Empleados.Salario, Empleados.Domicilio, Empleados.Fecha_Ingreso,
Empleados.Departamento_Id FROM Empleados";
        private String departamentosQuery = "SELECT Departamentos.Departamento_Id,
Departamentos.Nombre, Departamentos.Localidad_Id FROM Departamentos";
        private String empleadoQuery = "SELECT Empleados.Empleado_Id,
Empleados.Nombre, Empleados.Departamento_Id FROM Empleados WHERE Empleado_id =
?";
        private String departamentoQuery = "SELECT Departamentos.Departamento_Id,
Departamentos.Nombre FROM Departamentos WHERE Departamento_id = ?";
        private String empleadoUpdate = "INSERT INTO Empleados (Empleado_Id, Nombre,
Salario, Domicilio, Fecha_Ingreso, Departamento_id) VALUES (20, 'Martin', 2000, 'Belgrano
324', sysdate, 1)";
        private String departamentoUpdate = "INSERT INTO Departamentos
(Departamento_Id, Nombre, Localidad_Id) VALUES (10, 'Ventas', 3)";
        private PreparedStatement empleadoQueryStmt;
        private PreparedStatement departamentoQueryStmt;

        public DataBaseQuery() {
          try {
            conn = getConnection();
            stat = conn.createStatement();
```

```
    }
    catch (SQLException ex) {
      while (ex !=null) {
        ex.printStackTrace();
        ex = ex.getNextException();
      }
    }
    catch (IOException ex) {
      ex.printStackTrace();
    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
  }


  public static void main(String[] args) {
    DataBaseQuery dbq = new DataBaseQuery();
    dbq.consultaTablas();
    int empleadoId = 1;
    dbq.executeEmpleadoQuery(empleadoId);
    int departamentoId = 2;
    dbq.executeDepartamentoQuery(departamentoId);
    dbq.executeEmpleadoUpdate();
    dbq.executeDepartamentoUpdate();

  }

  /**
   * Cierra el Statement y la Conexion
   */
  private void cerrar() {
    try {
       stat.close();
       conn.close();
    } catch (SQLException e) {
        while (e != null) {
          e.printStackTrace();
          e = e.getNextException();
        }
    }
  }

  /**
   * Realiza consultas generales a las tablas Empleado y Departamento
   */
  private void consultaTablas() {
    empleadosQuery();
```

```
    departamentosQuery();
  }

  /**
   * Realiza una consulta a la tabla Empleado
   */
  private void empleadosQuery() {
    try {
      ResultSet result = stat.executeQuery(empleadosQuery);
      String empleadoId;
      String nombre;
      String salario;
      String domicilio;
      String fechaIngreso;
      String departamentoId;
      while (result.next()) {
        empleadoId = result.getString("Empleado_Id");
        nombre = result.getString("Nombre");
        salario = result.getString("Salario");
        domicilio = result.getString("Domicilio");
        fechaIngreso = result.getString("Fecha_Ingreso");
        departamentoId = result.getString("Departamento_Id");
        System.out.println("EmpleadoId:" + empleadoId);
        System.out.println("Nombre:" + nombre);
        System.out.println("Salario:" + salario);
        System.out.println("Domicilio:" + domicilio);
        System.out.println("Fecha Ingreso:" + fechaIngreso);
        System.out.println("DepartamentoId:" + departamentoId);
      }
      result.close();
    }
    catch (SQLException e)
    {
      while (e != null) {
        e.printStackTrace();
        e = e.getNextException();
      }
    }
  }

  /**
   * Realiza una consulta a la tabla Departamento
   */
  private void departamentosQuery() {
    try {
      System.out.println("Pasa por departamentosQuery");
      ResultSet result = stat.executeQuery(departamentosQuery);
      String departamentoId;
```

```
        String nombre;
        String localizacion;
        while (result.next()) {
          departamentoId = result.getString(1);
          nombre = result.getString(2);
          localizacion = result.getString(3);
          System.out.println("DepartamentoId:" + departamentoId);
          System.out.println("Nombre:" + nombre);
          System.out.println("Localizacion:" + localizacion);
        }
        result.close();
      }
      catch (SQLException e)  {
        while (e != null) {
          e.printStackTrace();
          e = e.getNextException();
        }
      }

  }

  /**
   * Realiza una Consulta a la tabla Empleado para un empleadoId especifico
   * @param empleadoId
   */
  private void executeEmpleadoQuery(int empleadoId) {
    try {
        empleadoQueryStmt = conn.prepareStatement(empleadoQuery);
        empleadoQueryStmt.setInt(1, empleadoId);
        ResultSet rs = empleadoQueryStmt.executeQuery();
        if (rs.next()) {
          System.out.println("EmpleadoId: " + rs.getString("Empleado_Id"));
          System.out.println("Nombre: " + rs.getString("Nombre"));
        }
      } catch (SQLException e) {
        while (e != null) {
          e.printStackTrace();
          e = e.getNextException();
        }
      }
  }

  /**
   * Realiza una consulta a la tabla Departamnento para un departamentoId especifico
   * @param departamentoId
   */
  private void executeDepartamentoQuery(int departamentoId) {
    try {
```

```
      departamentoQueryStmt = conn.prepareStatement(departamentoQuery);
      departamentoQueryStmt.setInt(1, departamentoId);
      ResultSet rs = departamentoQueryStmt.executeQuery();
      if (rs.next()) {
        System.out.println("DepartamentoId: " + rs.getString("Departamento_Id"));
        System.out.println("Nombre: " + rs.getString("Nombre"));
      }
    } catch (SQLException e) {
      while (e != null) {
        e.printStackTrace();
        e = e.getNextException();
      }
    }
  }

  /**
   * Realiza un insert en la tabla Empleado
   */
  private void executeEmpleadoUpdate() {
    try {
      stat.executeUpdate(empleadoUpdate);
    } catch (SQLException e) {
      while (e != null) {
        e.printStackTrace();
        e = e.getNextException();
      }
    }
  }

  /**
   * Realiza un insert en la tabla Departamento
   */
  private void executeDepartamentoUpdate() {
    try {
      stat.executeUpdate(departamentoUpdate);
    } catch (SQLException e) {
      while (e != null) {
        e.printStackTrace();
        e = e.getNextException();
      }
    }
  }

  /**
   * Obtiene la conexion
   */
  private static Connection getConnection() throws Exception {
    Configuration configuracion = new Configuration();
```

```
            String driver = configuracion.getDriver();
            String url = configuracion.getUrl();
            String username = configuracion.getUsername();
            String password = configuracion.getPassword();
            Class.forName(driver);
            return DriverManager.getConnection(url, username, password);
        }

}
```

El ejemplo es bastante sencillo, primero se hacen dos consultas generales bastante simples, una consulta a la tabla de Empleados y otra consulta a la tabla de Departamentos.

Luego se realizan consultas para empleados y departamentos específicos con el uso de sentencias predefinidas.

Una sentencia SQL puede ser almacenada y precompilada en un objeto PreparedStatement o sentencia predefinida. Las sentencias predefinidas nos evitan el tener una sentencia SELECT por cada empleado o departamento que se desee consultar. Esta técnica nos ofrece una ganancia en el rendimiento. PreparedStatement proporciona una serie de métodos setTipo(indice_variable, valor_del_Tipo) para setear los valores para las variables se hallan especificado en la sentencia predefinida. Así el método setInt(1, empleadoId) setea para la primera variable que aparece en la sentencia predefinida el valor entero de empleadoId.

Finalmente en el ejemplo se ve el uso del método executeUpdate para realizar inserts en las tablas.

Los errores que podrían ocurrir durante la ejecución de estas sentencias son tratados mediante Excepciones y todas son mostradas en la consola.

39. Ejecución de Store Procedures de Bases de Datos

39.1. Creación de un Store Procedure

La sintaxis para la definición de un stored procedure varía de acuerdo a la Base de Datos que se utilice. En algunas bases de datos se usa la siguiente sentencia SQL para crear un store procedure.

```
String createProcedure = “CREATE OR REPLACE procedure CANTIDAD_EMPLEADOS is “+
                         “cantidadEmpleados Number(5) := null;” +
                         “begin  “ +
                         “  select count(*) “+
                         “  into cantidadEmpleados “ +
                         “  from Empleado" +
```

```
“  order by Empleado.Nombre; “ +
“  dbms_output.put_line('cantidad de “+
“empleados:'||cantidadEmpleados);” +
"end;";
```

Este objeto String es una sentencia para una Base de Datos Oracle.

El siguiente fragmento de código usa el objeto Connection conn para crear un objeto Statement, el cual es usado para ejecutar una sentencia SQL, que cree el store procedure en la base de datos

```
Statement stmt = conn.createStatement();
stmt.executeUpdate(createProcedure);
```

El procedimiento CANTIDAD_EMPLEADOS será compilado y almacenado en la base de datos como un objeto de la base de datos que luego podrá ser llamado.

39.2. Llamada a un Store Procedure desde JDBC

JDBC le permite llamar un store procedure de la base de datos desde una aplicación Java. El primer paso para crear un objeto CallableStatement, así como con los objetos Statement y PreparedStatement es crear un Objeto Connection. Un objeto CallabaleStatement contiene una llamada a un store procedure, no contiene el storeProcedure. La primer línea de código de abajo crea una llamada para un stored procedure CANTIDAD_EMPLEADOS usando la conexión conn. La parte que está encerrada entre llaves es la sintaxis para stored procedures. Cuando el driver encuentra CANTIDAD_EMPLEADOS, trasladara esta sintaxis de escape al SQL nativo usado por la Base de Datos para llamar al stored procedure llamado CANTIDAD_EMPLEADOS

```
CallableStatement cs = conn.prepareCall("{call CANTIDAD_EMPLEADOS}");
cs.execute();
```

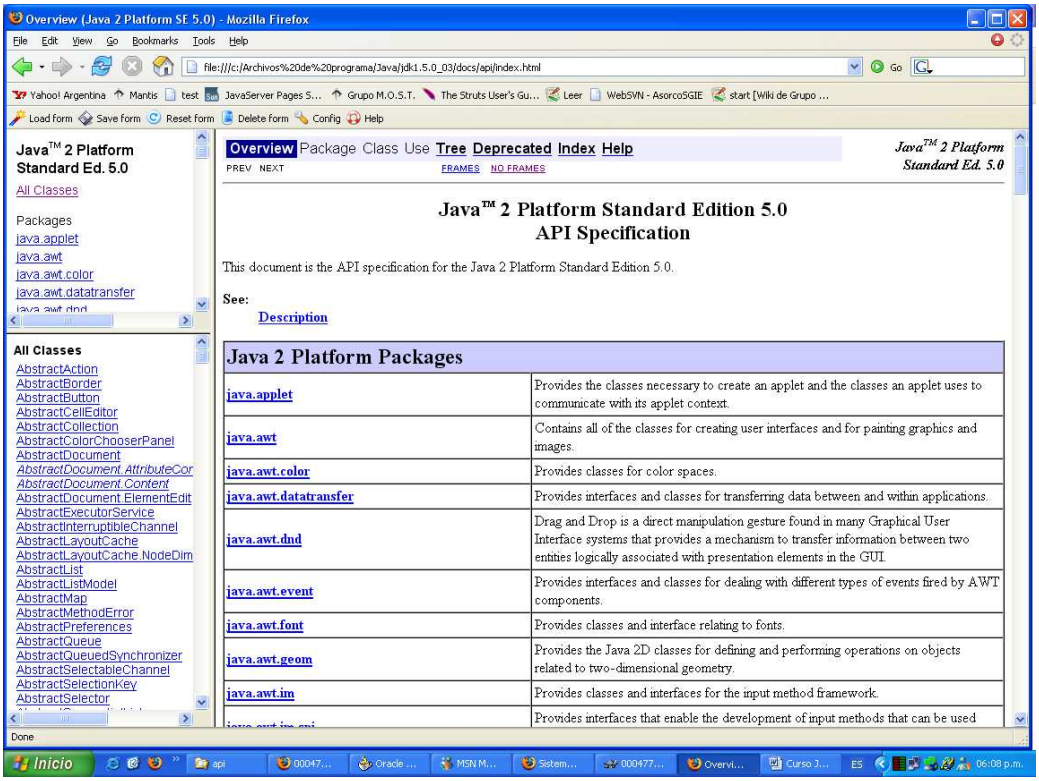
La clase CallableStatement es una subclase de PreparedStatement, así que un objeto CallableStatement puede tomar parámetros de entrada como un objeto PreparedStatement.

40. Agradecimientos

Este manual fue posible gracias a la colaboración de Martín Baspineiro quien aportó su dedicación y tiempo para poder confeccionarlo.

41. Apéndice A: Lectura de la documentación de la API en línea

Las bibliotecas estándar de java contienen cientos de clases, y cada una de estas clases contiene varios métodos. Es imposible recordar esta información en todo momento, para ello contamos con la documentación de la API que se puede consultar en línea. Esto le permitirá buscar todas las clases y métodos de la biblioteca estándar. La documentación de la API forma parte del SDK de Java. Esta en formato html. Apunte su navegador al subdirectorio docs/api/index.html de su instalación del SDK de Java. Observara una pantalla como la siguiente:



La pantalla está organizada en tres ventanas. Una ventana pequeña en la parte superior izquierda muestra todos los paquetes disponibles. Por debajo de ella, una ventana más grande muestra todas las clases. Haga click en cualquier clase, y la documentación de esa clase se mostrara en la ventana de la derecha. Le recomendamos, y seguramente usted estará de acuerdo, en agregar a la lista de favoritos de su browser la pagina docs/api/index.html. Le será de utilidad mientras realice cualquier tipo de desarrollo.

42. Apéndice B: Mejoras introducidas en el JDK 1.5

42.1. Generics

Generics tiene principalmente dos usos, el primero y principal de estos usos es el de tipar a las *collections.* Antes de la versión 1.5 solamente los arrays eran tipados, se podía declarar como se realiza en el siguiente ejemplo:

```
Persona[] agenda = new Persona[10]
```

Donde el compilador solamente aceptará *Persona* (o alguna subclase de *Persona)* como elementos de *agenda,* si intentamos meter cualquier otro tipo de objeto obtendremos una excepción.

Este tipado no existía para las *collections*, en ellas no podíamos definir que las estructuras fueran de un solo tipo, cualquier objeto de tipo collection podía albergar cualquier objeto que extendiese de Object (es decir CUALQUIER clase menos las primitivas), con lo cual para, por ejemplo obtener la primera entrada de nuestra agenda si fuera un ArrayList tendríamos que hacer:

```
Persona p = (Persona)ArrayList.get(0);
```

Es decir había que hacer un casting manual cada vez que obteníamos un elemento, porque no podíamos asegurar que los elementos fueran del tipo *Persona*; eso sin contar el tratamiento de errores que habría que hacer por el casting.

Para solucionar aparecieron los *generics*, que proporcionan la forma de solventar este problema, para ello solo tenemos que escribir inmediatamente después de la clase entre los signos < y > el tipo que van a tener los elementos de la estructura, quedaría algo como:

```
List<Persona> agenda = new ArrayList<Persona>();

agenda.add(new Persona("Pedro"));

Persona p = agenda.get(0);
```

Como se puede ver en la linea anterior, no hace falta realizar el casting para sacar el elemento de *agenda,* ya que la hemos declarado como tipada y con eso aseguramos que dentro de ella solo habrá elementos de tipo Persona.

Un detalle con el que hay que tener cuidado es que los tipos de las clases no admiten poliformismo, es decir un *ArrayList<Persona>* no admite un *ArrayList<Amigo>,* aunque *Amigo* extienda de *Persona,* si podemos hacer que un Array*List<Persona>* admita un *ArrayList<Persona>,* pero el tipo debe ser exactamente el mismo. Bueno, podemos estirar un poco este concepto usando el símbolo "*?"* y *extends* o *super* para crear poliformismo (por ejemplo un *ArrayList<? extends Persona>*admite cualquier subclase de *Persona,* incluida *Amigo,* y *ArrayList<? super Amigo>* admite cualquier superclase de *Amigo,*incluida *Persona*).

Hasta aquí todo parece estupendo, ahora tenemos la posibilidad de tipar collections, pero aqui vienen las complicaciones, que pasa cuando mezclamos código con *generics* y código *pre-generics,* por ejemplo en el siguiente ejemplo:

```
ArrayList <Persona> agenda = new ArrayList<Persona>();
agenda = llenarAgenda (agenda);
Persona p = agenda.get(0);

public static List llenarAgenda(List agenda){
agenda.add(new Persona("Maria"));
agenda.add(new Persona("Juan"));
return agenda;
}
```

Desde nuestra clase en la que usamos generics llamamos a otra confiando que manipulará nuestro objeto y nos los devolverá. ¿Y cuál es el problema? Bien, en el código de arriba vemos como la clase externa añadía dos elementos de tipo *Persona*, pero y si en lugar de eso hubiera añadido un elemento de tipo *Perro?* El compilador daría un error si se metiera un objeto de tipo *Perro* en un *ArrayList* tipado con *Persona,* pero no es el caso, la clase *llenarAgenda* recibe como parámetro un Array*List* sin tipar, y mientras está dentro de su código se comporta como antes de 1.5, pero una vez devuelto a nuestro código tipado, la máquina virtual da por sentado de que el objeto agenda sigue estando correctamente formado, pero cuando saquemos el elemento *Perro* que ha insertado la clase externa se producirá una excepción.

¿Y por qué ocurre esto?¿Por qué se permite que el código tipado se pueda usar con código sin tipar? Es para mantener la compatibilidad con versiones anteriores de Java. La opción de permitir solo código tipado hubiera eliminado este problema pero hubiera obligado a modificar todo el código hecho en Java hasta la fecha, si, casi nada. Así que para permitir la convivencia de ambos códigos se tomó una difícil y a veces no muy popular decisión: el tipado que ofrece generics es solo a nivel de compilador, es decir que cuando se compila el código, la anotación de generics desaparece y se sustituye por los castings correspondientes, por tanto cuando el código llega a la máquina virtual de java no hay ninguna diferencia entre una colección tipada y otra que no lo sea.

La única pista que nos dará el compilador será un warning que nos advertirá que estamos usando operaciones inseguras. Así que la moraleja es prestar máxima atención cuando mezclemos código con generics y código normal, bien sea código antiguo nuestro, librerías externas, etc…

Bueno, después de todo esto, todavía queda comentar la segunda utilidad que tienen los generics, aunque realmente se use poco. Imaginemos por un momento que tenemos una clase que sirve para manejar un negocio genérico de alquiler, algo como esto:

```
public class Alquiler {
private List inventario;
public Alquiler(List inventario){
```

```
this.inventario = inventario;
}
public Object alquiler(){
return inventario.get(0);
}
public void devolver(Object o){
inventario.add(o);
}
}
```

Ahora, si quisiéramos crear una subclase de alquiler para bicicletas tendríamos que sobrecargar los métodos para cambiar el inventario por uno tipado y hacer los castings al devolver el objeto. Y si quisiéramos hacer otra clase para el alquiler de coches necesitaríamos otros tantos cambios.

Pero hay otro camino, podemos crear un plantilla de nuestra clase *Alquiler*, y que dependiendo del parámetro con el que se construya adapte su estructura, para ello usamos el símbolo <*T*> (por convención, siempre podemos usar otro), con lo que la clase anterior quedaría de la siguiente manera:

```
public class Alquiler<T> {
private List<T> inventario;
public Alquiler(List<T> inventario){
this.inventario = inventario;
}
public T alquiler(){
return inventario.get(0);
}
public void devolver(T o){
inventario.add(o);
}
}
```

Con esto le decimos a la clase que es una clase genérica, que dependiendo del tipo <*T*> que reciba como argumento en el constructor debe adaptar sus métodos y atributos. Por tanto para construir una clase que gestione el alquiler de bicicletas tan solo deberíamos llamar al constructor con una lista de bicicletas como parámetro.

También podemos aplicar este mismo concepto para los métodos, creando métodos genéricos, y además podemos usar el símbolo <*?*> para crear plantillas más complejas.

Ejemplo de Generics

```
package com.empresa.cursos.java.basico.ejemplos.jdk15;
import java.util.ArrayList;
import java.util.List;

public class GenericsExample {
	public static void main(String[] args) {
		List<Empleado> empleados = new ArrayList<Empleado>();
		List<Empleado> empleadosLleno = new ArrayList<Empleado>();
		empleadosLleno = llenarAgenda(empleados);
		Empleado emp = empleadosLleno.get(0);
		System.out.println("Nombre Empleado : "+emp.getNombre());
		System.out.println("Nombre Salario : "+emp.getSalario());
	}
	public GenericsExample() {
	}
	public static List<Empleado> llenarAgenda(List<Empleado> empleados){
		empleados.add(new Empleado("Maria",12));
		empleados.add(new Empleado("Juan",14));
	return empleados;
	}
}
```

42.2. Anotaciones en código fuente - Meta Datos

Las anotaciones en código fuente es otra de las funcionalidades incorporadas en Java 5. A través de anotaciones se tiene la posibilidad de utilizar información complementaria colocada directamente en código fuente (meta datos) al momento de compilar o ejecutar una clase Java.

El mecanismo de colocar información dentro del código fuente para fines distintos al de ejecución, no es algo nuevo para Java, el proceso para generar documentación de clases emplea este mismo mecanismo y ha estado disponible desde las primeras versiones Java, sin embargo, lo que resulta importante de esta funcionalidad en Java 5 es la capacidad de utilizar dicha información para indicar instrucciones al compilador o JVM ("Java Virtual Machine").

En el JDK 5 se encuentran tres anotaciones que pueden ser utilizadas directamente:

Anotación	Función
Override	Utilizado para indicar que el método ha modificado su comportamiento ("override") sobre su superclase.
Deprecated	Utilizado para indicar que el uso de determinado elemento no es recomendable o ha dejado de ser actualizado.
SuppressWarnings	Permite suprimir mensajes del compilador relacionados con advertencias/avisos.

Debido a la misma gama de anotaciones que pueden ser requeridas en distintas circunstancias, el JDK ofrece los mecanismos para crear anotaciones de cualquier tipo e integrarles lógica personalizada, inclusive ya han empezado a surgir librerías de anotaciones producidas por terceros para uso en proyectos con requerimientos particulares, a continuación se ilustra cómo se deben utilizar las anotaciones.

```
import java.util.*;


class Instrumento {
   public void tocar() {
        System.out.println("Instrumento.tocar()");
   }
   public String tipo() {
        return "Instrumento";
   }
   /**
    * Metodo suplantado ("deprecated") en favor de afinarDeFabrica
    * @deprecated Utilizar afinarDeFabrica()
    */
```

```
    @Deprecated public void afinar() {}
    public void afinarDeFabrica() {}
}

class Guitarra extends Instrumento {
    @Override
    public void tocar() {
        System.out.println("Guitarra.tocar()");
    }
    @Override
    public String tipo() { return "Guitarra"; }
    @Override
    public void afinar() {}
}
```

El grupo de clases anteriores demuestra como la anotación @Override nos ofrece una solución para detectar si la clase derivada efectivamente ha re implementado ("override") correctamente los métodos de su superclase. Al colocar esta anotación en los métodos de la clase derivada, el compilador verifica que dichos métodos existan en la superclase, de esta manera, si se escribe incorrectamente el método en la clase derivada o se tiene una jerarquía de herencia compleja, este tipo de error que puede ser difícil de encontrar una vez en ejecución el programa, es detectado al momento de compilación.

La anotación @Deprecated permite marcar un método como obsoleto o antiguo y que el usuario de dicha superclase sea notificado de esta circunstancia al momento de compilación. En versiones anteriores a Java 5, la única manera en que un usuario podía percatarse de utilizar una librería o método marcado como "deprecated" era mediante la documentación de dicha librería, ahora a través de la anotación @Deprecated es posible detectar este hecho al momento de compilar.

Ejemplo de Anotaciones

```
package com.empresa.cursos.java.basico.ejemplos.jdk15;
public class InstrumentoAnotaciones {
    public void tocar() {
        System.out.println("Instrumento.tocar()");
    }
    public String tipo() {
        return "Instrumento";
```

```
    }
    /**
    * Metodo suplantado ("deprecated") en favor de afinarDeFabrica
    * @deprecated Utilizar afinarDeFabrica()
    */
    @Deprecated public void afinar() {}
    public void afinarDeFabrica() {}
}

package com.empresa.cursos.java.basico.ejemplos.jdk15;
public class GuitarraAnotaciones extends InstrumentoAnotaciones {
    @Override
    public void tocar() {
        System.out.println("Guitarra.tocar()");
    }
    @Override
    public String tipo() { return "Guitarra"; }
    @Override
    public void afinar() {
        System.out.println("Guitarra.afinar()");
    }
    public static void main(String[] args) {
        GuitarraAnotaciones guitarra = new GuitarraAnotaciones();
        String strTipo;
        String strTipoInstrumentacion;
        guitarra.tocar();
        strTipo = guitarra.tipo();
        System.out.println("strTipo : "+strTipo);
        guitarra.afinar();
        InstrumentoAnotaciones instrumentacion = new InstrumentoAnotaciones();
        instrumentacion.tocar();
        strTipoInstrumentacion = instrumentacion.tipo();
```

```
            System.out.println("strTipoInstrumentacion : "+strTipoInstrumentacion);
        }
    }
```

42.3. Autoboxing

No todo en java es un objeto, los tipos primitivos, si bien tienen wrappers siguen siendo primitivos y no pueden mezclarse con objetos....hasta ahora, a partir de Java 1.5 tenemos el boxing y unboxing automático.

Que es el autoboxing? Es la propiedad de los tipos primitivos de convertirse en sus wrappers objetos y viceversa.

Un ejemplo, en Java 1.4 teníamos que escribir el siguiente código:

ArrayList l = new ArrayList();

l.add(new Integer (1));

a partir de Java 1.5, el mismo código quedaría:

ArrayList<Integer> l = new ArrayList<Integer>();

l.add(1); // autoboxing!

Para obtener int de un arrayList, en java 1.4 habríamos hecho:

Integer entero = l.get(1);

int i = entero.intValue();

En Java 1.5, podemos hacer de la siguiente manera (supongo que l es un ArrayList<Integer>):

int i = l.get(1); //Unboxing!

Gracias al autoboxing, a partir de Java 1.5 no necesitamos mas pasar a tipos primitivos si queremos hacer alguna suma de, por ejemplo, dos Integer.

Anteriormente, en Java1.4, hubiera sido:

Integer entero1 = new Integer(1);

Integer entero2 = new Integer(2);

int valor = entero1.intValue() + entero2.intValue();

Ahora, usando Java1.5, escribimos:

Integer entero1 = new Integer(1);

Integer entero2 = new Integer(2);

int valor = entero1 + entero2;

Visto desde otro punto de vista se podría decir que se extendió la sobrecarga de los operadores aritméticos.

Ejemplo de AutoBoxing

```
package com.empresa.cursos.java.basico.ejemplos.jdk15;
import java.util.ArrayList;
public class Autoboxing {
   public static void main(String[] args) {
        int valorLista;
        //De esta forma se realizaba con el JDK 1.4
        //ArrayList l = new ArrayList();
        //l.add(new Integer (1) );
        //Con el JDK 1.5
        ArrayList<Integer> l = new ArrayList<Integer>();
        l.add(1); // autoboxing!
        valorLista = l.get(0); //Unboxing!
        System.out.println("Valor en Lista : "+valorLista);
   }
}
```

42.4. Uso de Enumeraciones.

Una enumeración es una estructura de datos diseñada para agrupar información estática y facilitar su acceso de otras clases. Esta funcionalidad ya presente en una serie de lenguajes de programación, comoC++ y Perl, esta siendo debutada en el mundo Java hasta esta versión 5.

Como sería normal esperar, ante la falta de apoyo directo en el JDK para enumeraciones, muchos desarrolladores familiarizados con esta técnica han creado sus propias maneras de simular el comportamiento.

```
public enum Autos { FORD, CHEVY, VW, BMW, NISSAN };
public enum Sexo  { M, F };
```

La característica común de los datos contenidos en las enumeraciones anteriores (y cualquier otra desde luego) es que son grupos de datos relacionados entre sí que no son modificados a lo largo de la ejecución de un programa.

Si ha trabajado ampliamente con Java, una enumeración es el equivalente grupal a definir un campo como: public static final, esto es, un dato con acceso publico, con una sola instancia (static) y que no cambia su valor al ejecutarse el programa (final).

Vale mencionar que una enumeración posee el mismo comportamiento que cualquier otra estructura Java de primer nivel como lo sería una clase o interfase: puede ser declarada en un archivo por si sola y tiene la capacidad de pertenecer a una librería ("package") de manera independiente.

A pesar de la aparente similitud que tiene una enumeración con un arreglo o colección, difieren considerablemente, veamos un ejemplo práctico donde sea utilizada una enumeración:

```
public class Piloto {
   private Auto auto;
   private Sexo sexo;
   public Piloto(Auto auto, Sexo sexo) {
        this.auto = auto;
        this.sexo = sexo;
   }
   public static void main(String[] args) {
        Piloto piloto1 = new Piloto(Auto.FORD,Sexo.M);
        Piloto piloto2 = new Piloto(Auto.CHEVY,Sexo.M);
        Piloto piloto3 = new Piloto(Auto.BMW,Sexo.F);
   }
}
```

La clase anterior define dos campos: auto y sexo, basados en las enumeraciones Auto y Sexo respectivamente. Posteriormente, se define un constructor que asigna los valores de estos campos a cada instancia de la clase.

Dentro del método principal (main) de la clase, se lleva acabo la generación de instancias donde los respectivos campos son inicializados a partir de los valores en la enumeración. Note la sintaxis de asignación: Nombre_Enumeracion.Valor_Explicito.

Como puede observar, este tipo de asignación llevado acabo a través de enumeraciones, permite salvaguardar la asignación de datos estáticos en clases Java, de intentarse asignar un valor no encontrado dentro de la enumeración el compilador generaría un error. Además de este apoyo para detección de errores al momento de compilar, una enumeración también permite concentrar listas de datos que pueden ser reutilizadas en diversos programas, tales como los datos enunciados anteriormente, listas de ciudades, nombres, países u otro tipo información.

Ejemplo de Enumeraciones

```
package com.empresa.cursos.java.basico.ejemplos.jdk15;
public class PilotoEnumeracion {
        public enum Autos { FORD, CHEVY, VW, BMW, NISSAN };
        public enum Sexo  { M, F };
        private Autos auto;
   private Sexo sexo;
      public PilotoEnumeracion(Autos auto, Sexo sexo) {
        this.auto = auto;
        this.sexo = sexo;
   }
   public static void main(String[] args) {
        PilotoEnumeracion piloto1 = new PilotoEnumeracion(Autos.FORD,Sexo.M);
        PilotoEnumeracion piloto2 = new PilotoEnumeracion(Autos.CHEVY,Sexo.M);
        PilotoEnumeracion piloto3 = new PilotoEnumeracion(Autos.BMW,Sexo.F);
        System.out.println("Piloto 1 Auto : "+piloto1.auto);
        System.out.println("Piloto 1 Sexo : "+piloto1.sexo);
        System.out.println("Piloto 2 Auto : "+piloto2.auto);
        System.out.println("Piloto 2 Sexo : "+piloto2.sexo);
        System.out.println("Piloto 3 Auto : "+piloto3.auto);
        System.out.println("Piloto 4 Sexo : "+piloto3.sexo);
   }
}
```

42.5. Varargs

En versiones anteriores, un método que necesitaba un número arbitrario de valores requería crear un array y colocar los valores en el array antes de invocar el método. Por ejemplo, así se utiliza la clase MessageFormat para dar formato a un mensaje:

```
Object[] arguments = {
   new Integer(7),
   new Date(),
```

```
    "a disturbance in the Force"
};

        String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "
     + "{0,number,integer}.", arguments);
```

Es cierto que varios argumentos deben ser pasados en un array, pero la característica varargs automatiza y oculta el proceso. Así, por ejemplo, el método MessageFormat.format ahora tiene esta declaración:

```
public static String format(String pattern,   Object... arguments);
```

Los tres puntos después de escribir el último parámetro indican que el argumento final puede pasar como un array o como una secuencia de argumentos. Varargs sólo se puede utilizar en la posición final del argumento. Teniendo en cuenta la nueva declaración varargs para MessageFormat.format, la invocacion anterior podrá ser sustituida por la siguiente petición más corta y más simple:

```
String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "
    + "{0,number,integer}.",
    7, new Date(), "a disturbance in the Force");
```

Existe una fuerte sinergia entre autoboxing y varargs, que se ilustra en el siguiente programa utilizando reflexión:

```
public class Test {
    public static void main(String[] args) {
        int passed = 0;
        int failed = 0;
        for (String className : args) {
            try {
                Class c = Class.forName(className);
                c.getMethod("test").invoke(c.newInstance());
                passed++;
            } catch (Exception ex) {
                System.out.printf("%s failed: %s%n", className, ex);
                failed++;
            }
        }
        System.out.printf("passed=%d; failed=%d%n", passed, failed);
    }
}
```

Ejemplo de VarArgs
package com.empresa.cursos.java.basico.ejemplos.jdk15; public class VarArgsExample {

```
            /**
             * @param args - Se debe pasar por parametro una serie de cadenas de texto y el programa se encargara de imprimirlas en la consola
             */
            public static void main(String[] args) {
          for (String className : args) {
            System.out.println("Parametro en Main() : "+className);
          }
            }
      }
```

42.6. The For-Each Loop

Consideremos el siguiente método, que tiene una colección de tareas TimerTask y las cancela:

```
void cancelAll(Collection<TimerTask> c) {
   for (Iterator<TimerTask> i = c.iterator(); i.hasNext(); )
      i.next().cancel();
}
```

Así es como se ve el ejemplo con el for-each:

```
void cancelAll(Collection<TimerTask> c) {
   for (TimerTask t : c)
      t.cancel();
}
```

Cuando usted ve los dos puntos (:) se lee como "en". El bucle de arriba se lee como "para cada 't' de tipo TimerTask en c" Como puede ver, el for-each combina muy bien con los genéricos.
Este es un error común que las personas cometen cuando están tratando de hacer iteración anidada en dos colecciones:

```
List suits = ...;
List ranks = ...;
List sortedDeck = new ArrayList();
// throws NoSuchElementException
for (Iterator i = suits.iterator(); i.hasNext(); )
   for (Iterator j = ranks.iterator(); j.hasNext(); )
      sortedDeck.add(new Card(i.next(), j.next()));
```

¿Puedes ver el error? Muchos programadores expertos han cometido este error en un momento u otro. El problema es que el método next está siendo llamado muchas veces en el

"metodo exterior" colección (suits). Es llamado en el bucle interior, tanto para las colecciones del exterior como del interior. Con el fin de solucionarlo, hay que añadir una variable en el ámbito del bucle externte de la aplicación:

```
// Fixed, though a bit ugly
for (Iterator i = suits.iterator(); i.hasNext(); ) {
   Suit suit = (Suit) i.next();
   for (Iterator j = ranks.iterator(); j.hasNext(); )
      sortedDeck.add(new Card(suit, j.next()));
```

Entonces, ¿qué tiene todo esto que ver con la construcción for-each? Esto está hecho a medida para la iteración anidada!

```
for (Suit suit : suits)
   for (Rank rank : ranks)
      sortedDeck.add(new Card(suit, rank));
```

La construcción del for-each es también aplicable a las array, donde se esconde la variable índice en lugar del iterador. El siguiente método devuelve la suma de los valores en un array:

```
// Returns the sum of the elements of a
int sum(int[] a) {
   int result = 0;
   for (int i : a)
      result += i;
   return result;
}
```

Ejemplo de For Each

```
package com.empresa.cursos.java.basico.ejemplos.jdk15;
public class ForEachExample {
	public static void main(String[] args) {
		int[] valores = new int[3];
		int valoresResueltos = 0;
		valores[0]=0;
		valores[1]=1;
		valores[2]=2;
		valoresResueltos = sum(valores);
		System.out.println("Resultado de los Valores : "+valoresResueltos);
	}
```

```
        static int sum(int[] a) {
            int result = 0;
            for (int i : a)
                result += i;
            return result;
        }
    }
```

42.7. Static Import

Para acceder a los miembros estáticos, es necesario calificar las referencias a la clase de la que proviene. Por ejemplo, hay que decir:

```
double r = Math.cos(Math.PI * theta);
```

Con el fin de evitar esto, la gente a veces pone los miembros estáticos en una interfaz y heredar de la interfaz. Esta es una mala idea, de hecho, es tan mala idea que hay un nombre para él: "la constante interfaz Antipatrones". El problema de que si una clase utiliza los miembros estáticos de otra clase es un mero detalle de implementación. Cuando una clase implementa una interfaz, se convierte en parte de la API pública de la clase. Los detalles de implementación no deberían filtrarse en las API públicas.
La construcción de la importación estática permite el acceso incondicional a los miembros estáticos sin heredar del tipo que contiene los miembros estáticos. En cambio, el programa importa los miembros, ya sea de forma individual:

```
import static java.lang.Math.PI;
```

o en masa:

```
import static java.lang.Math.*;
```

Una vez que los miembros estáticos han sido importados, se los puede utilizar sin evaluación:

```
double r = cos(PI * theta);
```

La declaración de importación estática es análoga a la declaración de importación

normal. Donde la declaración normal de importación, importa clases de paquetes, la declaración de importación estática, importa miembros estáticos de las clases.

Ejemplo de Static Imports

```
package com.empresa.cursos.java.basico.ejemplos.jdk15;
import static java.lang.Math.PI;
public class StaticImportExample {
	/**
	 * @param args
	 */
	public static void main(String[] args) {
		int theta = 10;
		double r = (PI * theta);
		System.out.println("Resultado : "+r);
	}
}
```

43. Bibliografía

43.1. Bibliografía

- CAY S. HORSTMANN, GARY CORNELL, JAVA 2 Fundamentos, Prentice Hall.
- CAY S. HORSTMANN, GARY CORNELL, JAVA 2 Características Avanzadas, Prentice Hall.
- MULLER PIERRE ALAIN, Modelado de Objetos con UML, Ediciones Gestión 2000.
- BRUCE ECKEL, Thinking in Java, Prentice Hall.

43.2. Sitios de Internet consultados

- www.java.sun.com
- www.javahispano.org
- www.programacion.com