

Programación AV I

JAVA

Matias Ezequiel Ramos

En este apunte se encuentra toda la teoría necesaria para aprobar la cursada de Programación Avanzada I. Sobre el final del apunte se encuentran las prácticas que realizaremos en el transcurso de la cursada de Programación AV I y Laboratorio V

Índice

• Introducción a JAVA	3
• La clase Object	8
• La clase String	9
• Clases envoltorio - Wrappers	10
• Documentación	11
• Excepciones	14
• Test Unitario	20
- JUnit.....	21
• Manejo de archivos.	23
- Serializacion	26
• Colecciones	29
• Tipos genéricos.....	34
• Patrones	37
- Patrones de Diseño	39
- Antipatrón	47
• Conexión de Base de Datos	49
• Reflexión Informática	54
- Reflexión en Java.....	53
- Annotations.....	62
• Hilos.....	64
- Clase Thread	64
- Ciclo de vida	67
- Interrupción.....	68
- Sincronización de hilos.....	70
- Pool de hilos	72
• Guía de Ejercicios	68

- Trabajo Practico Reflexión 76
- Trabajo Practico Conexión a BD 78

Introducción a Java

Lo primero que vamos a ver es la historia de Java y cuales son sus principales características. Además veremos cuales son las herramientas de desarrollo para programar en Java.

Historia. El perfil de Java

Los padres de Java son James Gosling (emacs) y Bill Joy (Sun). Java descende de un lenguaje llamado Oak cuyo propósito era la creación de software para la televisión interactiva. Las características de Oak eran:

- Pequeño.
- Robusto.
- Independiente de la máquina.
- Orientado a objetos.

El proyecto de televisión interactiva fracasó y el interés de los creadores de Oak se dirigió a Internet bajo el lema «La red es la computadora».

Los criterios de diseño de Java fueron:

- Independiente de la máquina.
- Seguro para trabajar en red.
- Potente para sustituir código nativo.

Características del lenguaje

La principal característica de Java es la de ser un lenguaje compilado e interpretado. Todo programa en Java ha de compilarse y el código que se genera “bytecodes” es interpretado por una máquina virtual. De este modo se consigue la independencia de la máquina, el código compilado se ejecuta en máquinas virtuales que si son dependientes de la plataforma.

Java es un lenguaje orientado a objetos de propósito general. Aunque Java comenzará a ser conocido como un lenguaje de programación de applets que se ejecutan en el entorno de un navegador web, se puede utilizar para construir cualquier tipo de proyecto.

Su sintaxis es muy parecida a la de C y C++ pero hasta ahí llega el parecido. Java no es una evolución ni de C++ ni un C++ mejorado.

En el diseño de Java se prestó especial atención a la seguridad. Existen varios niveles de seguridad en Java, desde el ámbito del programador, hasta el ámbito de la ejecución en la máquina virtual.

Con respecto al programador, Java realiza comprobación estricta de tipos durante la compilación, evitando con ello problemas tales como el desbordamiento de la pila. Pero, es durante la ejecución donde se encuentra el método adecuado según el tipo de la clase receptora del mensaje; aunque siempre es posible forzar un enlace estático declarando un método como final.

Todas las instancias de una clase se crean con el operador new(), de manera que un recolector de basura se encarga de liberar la memoria ocupada por los objetos que ya no están referenciados. La máquina virtual de Java gestiona la memoria dinámicamente.

Una fuente común de errores en programación proviene del uso de punteros. En Java se han eliminado los punteros, el acceso a las instancias de clase se hace a través de referencias.

Además, el programador siempre está obligado a tratar las posibles excepciones que se produzcan en tiempo de ejecución. Java define procedimientos para tratar estos errores.

Java también posee mecanismos para garantizar la seguridad durante la ejecución comprobando, antes de ejecutar código, que este no viola ninguna restricción de seguridad del sistema donde se va a ejecutar.

También cuenta con un cargador de clases, de modo que todas las clases cargadas a través de la red tienen su propio espacio de nombres para no interferir con las clases locales.

Otra característica de Java es que está preparado para la programación concurrente sin necesidad de utilizar ningún tipo de biblioteca.

Finalmente, Java posee un gestor de seguridad con el que poder restringir el acceso a los recursos del sistema.

A menudo se argumenta que Java es un lenguaje lento porque debe interpretar los bytecodes a código nativo antes de poder ejecutar un método, pero gracias a la tecnología JIT (Just In Time), este proceso se lleva a cabo una única vez, después el código en código nativo se almacena de tal modo que está disponible para la siguiente vez que se llame.

Herramientas de desarrollo

Las herramientas de desarrollo de Java se conocen como Java Development Kit(JDK). En el momento de escribir este apunte las herramientas de desarrollo van por la versión 8. Estas herramientas se pueden descargar gratuitamente de

<http://www.oracle.com/technetwork/es/java/javase/downloads/index.html>.

Este conjunto de herramientas cuenta entre otros con un compilador de línea de comandos `javac`; la máquina virtual de Java con la que poder ejecutar aplicaciones java; una herramienta de documentación `javadoc`; y una herramienta para empaquetar proyectos `jar`. La utilidad de estas herramientas la iremos viendo con detalle más adelante.

El compilador Java

Cuando usted programa para la plataforma Java, escribe el código de origen en archivos `.java` y luego los compila. El compilador verifica su código con las reglas de sintaxis del lenguaje, luego escribe los códigos byte en archivos `.class`. Los códigos byte son instrucciones estándar destinadas a ejecutarse en una Java Virtual Machine (JVM). Al agregar este nivel de abstracción, el compilador Java difiere de los otros compiladores de lenguaje, que escriben instrucciones apropiadas para el chipset de la CPU en el que el programa se ejecutará.

La JVM

Al momento de la ejecución, la JVM lee e interpreta archivos `.class` y ejecuta las instrucciones del programa en la plataforma de hardware nativo para la que se escribió la JVM. La JVM interpreta los códigos byte del mismo modo en que una CPU interpretaría las instrucciones del lenguaje del conjunto. La diferencia es que la JVM es un software escrito específicamente para una plataforma particular. La JVM es el corazón del principio "escrito una vez, ejecutado en cualquier lugar" del lenguaje Java. Su código se puede ejecutar en cualquier chipset para el cual una implementación apropiada de la JVM está disponible. Las JVM están disponibles para plataformas principales como Linux y Windows y se han implementado subconjuntos del lenguaje Java en las JVM para teléfonos móviles y aficionados de chips.

El Java Runtime Environment

El Java Runtime Environment (JRE, también conocido como el Java Runtime) incluye las bibliotecas de códigos de la JVM y los componentes que son necesarios para programas en ejecución escritos en el lenguaje Java. Está disponible para múltiples plataformas. Puede redistribuir libremente el JRE con sus aplicaciones, de acuerdo a los términos de la licencia del JRE, para darles a los usuarios de la aplicación una plataforma en la cual ejecutar su software. El JRE se incluye en el JDK.

El recolector de basura

En lugar de forzarlo a mantenerse a la par con la asignación de memoria (o usar una biblioteca de terceros para hacer esto), la plataforma Java proporciona una gestión de memoria lista para usar. Cuando su aplicación Java crea una instancia de objeto al momento de ejecución, la JVM asigna automáticamente espacio de memoria para ese objeto desde el almacenamiento dinámico, que es una agrupación de memoria reservada para que use su programa. El recolector de basura Java se ejecuta en segundo plano y realiza un seguimiento de cuáles son los objetos que la aplicación ya no necesita y recupera la memoria que ellos ocupan. Este abordaje al manejo de la memoria se llama gestión de la memoria implícita porque no le exige que escriba cualquier código de manejo de la memoria. La recolección de basura es una de las funciones esenciales del rendimiento de la plataforma Java.

Ide

Un entorno de desarrollo integrado, llamado también IDE (sigla en inglés de integrated development environment), es un programa informático compuesto por un conjunto de herramientas de programación. Puede dedicarse en exclusiva a un solo lenguaje de programación o bien puede utilizarse para varios.

Un IDE es un entorno de programación que ha sido empaquetado como un programa de aplicación; es decir, que consiste en un editor de código, un compilador, un depurador y un constructor de interfaz gráfica (GUI).

Existen diversos IDEs para Java. Vamos a citar algunos de ellos:

- Eclipse: software libre que se puede descargar en <http://www.eclipse.org>. Es uno de los entornos Java más utilizados a nivel profesional. El paquete básico de Eclipse se puede expandir mediante la instalación de plugins para añadir funcionalidades a medida que se vayan necesitando.
- NetBeans: software libre que se puede descargar en <http://www.netbeans.org>. Otro de los entornos Java muy utilizados, también expandible mediante plugins. Facilita bastante el diseño gráfico asociado a aplicaciones Java, por esta razón lo vamos a utilizar durante la cursada.
- BlueJ: software libre que se puede descargar en <http://bluej.org>. Es un entorno de desarrollo dirigido al aprendizaje de Java (entorno académico) y sin uso a nivel profesional. Es utilizado en distintas universidades para la enseñanza de Java. Destaca por ser sencillo e incluir algunas funcionalidades dirigidas a que las personas que estén aprendiendo tengan mayor facilidad para comprender aspectos clave de la programación orientada a objetos.
- JBuilder: software comercial. Se pueden obtener versiones de prueba o versiones simplificadas gratuitas en la web <http://www.embarcadero.com> buscando en la sección de productos y desarrollo de aplicaciones. Permite desarrollos gráficos.
- JCreator: software comercial. Se pueden obtener versiones de prueba o versiones simplificadas gratuitas en la web <http://www.jcreator.com>. Este IDE está escrito en

C++ y omite herramientas para desarrollos gráficos, lo cual lo hace más rápido y eficiente que otros IDEs.

Debes pensar en Java no solamente como un lenguaje de programación si no como un conjunto de tecnologías basadas en el mismo lenguaje. Este conjunto de tecnologías te permite escribir aplicaciones para gráficos, multimedia, la web, programación distribuida, bases de datos y un largo etcétera.

Recomendaciones

En Java existen ciertas reglas de codificación que son comúnmente utilizadas por los programadores. Conviene conocer y seguir estas reglas.

Los nombres de las clases deben empezar por mayúscula.

Los atributos y métodos de las clases deben empezar por minúsculas y si están formadas por varias palabras, se escriben sin espacios y la primera letra de cada palabra en mayúscula.

Las instancias de las clases siguen la misma recomendación que los métodos y atributos.

Las constantes se escriben en mayúsculas y si están formadas por varias palabras, se separan con guión bajo.

Palabras reservadas

El conjunto de palabras reservadas en Java:

abstract	continue	for	new	switch
boolean	default	goto	null	synchronized
break	do	if	package	this
byte	double	implements	private	threadsafe
byvalue	else	import	protected	throw
case	extends	instanceof	public	transient
catch	false	int	return	true
char	final	interface	short	try
class	finally	long	static	void
const	float	native	super	while

Ningún identificador puede llevar el nombre de una palabra reservada.

Tipos de datos primitivos (no son clases)

Tipo	Tamaño	Rango
byte	8 bits	-128 .. 127
short	16 bits	-32768 .. 32767
int	32 bits	-2147483648 .. 2147483647
long	64 bits	-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807
float	32 bits	-3.4x10 ³⁸ .. 3.4x10 ³⁸ (mínimo 1.4x10 ⁻⁴⁵)
double	64 bits	-1.8x10 ³⁰⁸ .. 1.8x10 ³⁰⁸ (mínimo 4.9x10 ⁻³²⁴)
boolean		true o false

char	16 bits	unicode
------	---------	---------

Tipos de datos compuestos (son clases)

- String
- Vectores y matrices
- Colecciones
- Clases
- wrappers/envolventes

```
<tipo_datos_compuesto> <nombre_variable> =  
new <tipo_datos> (<valor_inicial>);
```


La clase Object

Todas las clases son en realidad subclases de una clase más amplia: la clase Object. Cuando decimos esto estamos diciendo que todas las clases que utilicemos hasta las que creamos se extienden de la clase Object.

Los métodos de la clase Object

- `public boolean equals(Object obj)`: compara si dos objetos son iguales, por defecto un objeto es igual solamente a sí mismo.
- `public int hashCode()`: Devuelve un valor de código hash para el objeto. Este método se apoya en beneficio de tablas hash tales como los proporcionados por `java.util.Hashtable`.
- `protected Object clone() throws CloneNotSupportedException`: devuelve una copia binaria del objeto, al hacer la copia hace referencia a una nueva posición de memoria.
- `public final Class getClass()`: devuelve el objeto del tipo Class que representa dicha clase durante la ejecución, es decir devuelve el tipo de clase al que pertenece.
- `protected void finalize() throws Throwable`: se usa para finalizar el objeto, es decir, se avisa al administrador de la memoria que ya no se usa dicho objeto, y se puede ejecutar código especial antes de que se libere la memoria.
- `public String toString()`: devuelve una cadena describiendo el objeto.
- `void nativa public final notify()`: Se despierta un solo hilo que está esperando en el monitor de este objeto. Un subproceso espera en el monitor de un objeto llamando a uno de los de espera métodos.
- `final public native void notifyAll ()`: Se despierta todos los temas que están en espera en el monitor de este objeto. Un subproceso espera en el monitor de un objeto llamando a uno de los de espera métodos.
- `final public void wait ()`: Espera que se le notifique por otro subproceso de un cambio en este objeto.

Las clases derivadas deben sobrescribir los métodos adecuadamente, por ejemplo el método `equals`, `toString` y `hashCode`. También habría que utilizar la implementación de la Interface `Cloneable`, en el caso que queramos hacer un clonador del Objeto.

La clase String

La clase String está orientada a manejar cadenas de caracteres y proporciona métodos para el tratamiento de las cadenas de caracteres: acceso a caracteres individuales, buscar y extraer una subcadena, copiar cadenas, convertir cadenas a mayúsculas o minúsculas, etc.

Métodos	Descripción
length()	Devuelve la longitud de la cadena
indexOf('carácter')	Devuelve la posición de la primera aparición de carácter
lastIndexOf('carácter')	Devuelve la posición de la última aparición de carácter
charAt(n)	Devuelve el carácter que está en la posición n
substring(n1,n2)	Devuelve la subcadena comprendida entre las posiciones n1 y n2-1
toUpperCase()	Devuelve la cadena convertida a mayúsculas
toLowerCase()	Devuelve la cadena convertida a minúsculas
equals("cad")	Compara dos cadenas y devuelve true si son iguales
equalsIgnoreCase("cad")	Igual que equals pero sin considerar mayúsculas y minúsculas
compareTo(OtroString)	Devuelve 0 si las dos cadenas son iguales. <0 si la primera es alfabéticamente menor que la segunda ó >0 si la primera es alfabéticamente mayor que la segunda.
compareToIgnoreCase(OtroString)	Igual que compareTo pero sin considerar mayúsculas y minúsculas.
startsWith("cad")	Devuelve true si el String comienza con el String entregado como parámetro.
replaceAll("aReemplazar", "reemplazo")	Reemplaza todas las ocurrencias de aReemplazar por reemplazo.
valueOf(N)	Método estático. Convierte el valor N a String. N puede ser de cualquier tipo.

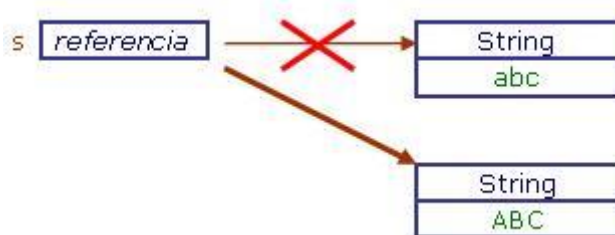
Debemos saber que los objetos String no son modificables. Por lo tanto, los métodos que actúan sobre un String con la intención de modificarlo lo que hacen es crear un nuevo String a partir del original y devolverlo modificado.

Por ejemplo: Una operación como convertir a mayúsculas o minúsculas un String no lo modificará sino que creará y devolverá un nuevo String con el resultado de la operación.

String s = "abc";



s = s.toUpperCase(); //convertir a mayúsculas el contenido del String s



Clases envoltorio (Wrapper)

En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, etc.) como objetos. Por ejemplo, los contenedores definidos por el API java.util (Arrays dinámicos, listas enlazadas, colecciones, conjuntos, etc.) utilizan como unidad de almacenamiento la clase Object. Dado que Object es la raíz de toda la jerarquía de objetos en Java, estos contenedores pueden almacenar cualquier tipo de objetos. Pero los datos primitivos no son objetos, con lo que quedan en principio excluidos de estas posibilidades.

Las clases envoltorio proporcionan métodos de utilidad para la manipulación de datos primitivos (conversiones de / hacia datos primitivos, conversiones a String, etc.)

Las clases envoltorio existentes son:

- Byte para byte.
- Short para short.
- Integer para int.
- Long para long.
- Boolean para boolean.
- Float para float.
- Double para double.
- Character para char.

Las clases envoltorio proporcionan también métodos de utilidad para la manipulación de datos primitivos. La siguiente tabla muestra un resumen de los métodos disponibles para la clase Integer.

Método	Descripción
Integer(int valor) Integer(String valor)	Constructores a partir de int y String
int intValue() / byte byteValue() / float floatValue() . . .	Devuelve el valor en distintos formatos, int, long, float, etc. (Sobrescriptos de Number)
Boolean equals(Object obj)	Devuelve true si el objeto con el que se compara es un Integer y su valor es el mismo.
static Integer getInteger(String s)	Devuelve un Integer a partir de una cadena de caracteres. Estático
static int parseInt(String s)	Devuelve un int a partir de un String. Estático.
static String toBinaryString(int i) static String toOctalString(int i) static String toHexString(int i) static String toString(int i)	Convierte un entero a su representación en String en binario, octal, hexadecimal, etc. Estáticos
String toString()	
static Integer valueOf(String s)	Devuelve un Integer a partir de un String. Estático.

Documentar proyectos JAVA con javaDoc

Documentar un proyecto es algo fundamental de cara a su futuro mantenimiento. Cuando programamos una clase, debemos generar documentación lo suficientemente detallada sobre ella como para que otros programadores sean capaces de usarla sólo con su interfaz. No debe existir necesidad de leer o estudiar su implementación, lo mismo que nosotros para usar una clase del API Java no leemos ni estudiamos su código fuente.

Javadoc es una utilidad de Oracle para la generación de documentación de APIs en formato HTML a partir de código fuente Java. Javadoc es el estándar para documentar clases de Java. La mayoría de los IDEs utilizan javadoc para generar de forma automática documentación de clases. BlueJ también utiliza javadoc y permite la generación automática de documentación, y mostrarla de forma completa para todas las clases de un proyecto, o bien de forma particular para una de las clases de un proyecto.

Veamos en primer lugar qué se debe incluir al documentar una clase:

- Nombre de la clase, descripción general, número de versión, nombre de autores.
- Documentación de cada constructor o método (especialmente los públicos) incluyendo: nombre del constructor o método, tipo de retorno, nombres y tipos de parámetros si los hay, descripción general, descripción de parámetros (si los hay), descripción del valor que devuelve.

Las variables de instancia o de clase no se suelen documentar a nivel de javadoc.

Para que javadoc sea capaz de generar documentación automáticamente han de seguirse estas reglas:

- La documentación para javadoc debe incluirse entre símbolos de comentario que tienen que empezar con una barra y doble asterisco, y terminar con un asterisco y barra.

```
/**
```

```
Esto es un comentario para javadoc ejemplo aprenderaprogramar.com
```

```
*/
```

- La ubicación le define a javadoc qué representa el comentario: si está incluido justo antes de la declaración de clase se considerará un comentario de clase, y si está incluido justo antes de un constructor o método se considerará un comentario de ese constructor o método.
- Para alimentar javadoc se usan ciertas palabras reservadas (tags) precedidas por el carácter "@", dentro de los símbolos de comentario javadoc. Si no existe al menos una línea que comience con @ no se reconocerá el comentario para la documentación de la clase.

En la tabla siguiente mostramos algunas de las palabras reservadas (tags), aunque hay más de las que aquí incluimos. Si necesitas una lista completa de las tags con su correspondiente uso consulta el libro en la biblioteca.

TAG	DESCRIPCIÓN	COMPRENDE
@author	Nombre del desarrollador.	Nombre autor o autores

@deprecated	Indica que el método o clase es obsoleto (propio de versiones anteriores) y que no se recomienda su uso.	Descripción
@param	Definición de un parámetro de un método, es requerido para todos los parámetros del método.	Nombre de parámetro y descripción
@return	Informa de lo que devuelve el método, no se aplica en constructores o métodos "void".	Descripción del valor de retorno
@see	Asocia con otro método o clase.	Referencia cruzada referencia (#método(); clase#método(); paquete.clase; paquete.clase#método()).
@throws	Excepción lanzada por el método	Nombre de la clase y descripción
@version	Versión del método o clase.	Versión

Las etiquetas `@author` y `@version` se usan para documentar clases e interfaces. Por tanto no son válidas en cabecera de constructores ni métodos. La etiqueta `@param` se usa para documentar constructores y métodos. La etiqueta `@return` se usa solo en métodos con valores de retorno.

Dentro de los comentarios se admiten etiquetas HTML, por ejemplo con `@see` se puede referenciar una página web como link para recomendar su visita de cara a ampliar información.

Ejemplo:

```
/**
 * @author matramos
 */
public interface IBeneficioService {
/**
 * Trae un Beneficio por número, año y abreviatura de plan que
 *no esté en estado BAJA <br>
 *
 * @param anio : año del beneficiario
 * @param numero : número del beneficiario
 * @param abreviatura : abreviatura del plan
 * @return BeneficioBean con todos sus datos cargados
 * @throws AccesoDatosException: Error al conectarse a la base de datos
 */
public BeneficioBean
obtenerBeneficioPorNumeroAnioBeneficiarioAbreviaturaActivo(Integer
numero, Integer anio,String abreviaturaPlan) throws
AccesoDatosException;
}
```

CodeTags

codetags son etiquetas que están presentes en todos los lenguajes, en todos los códigos y en los comentarios. Es el uso de tres palabras *TODO*, *FIXME* o *XXX* con igual significado: *arregla esto o tarea pendiente*.

Imagina que tienes la especificación de un código que tienes que hacer. Tienes el diseño ya pensado (o en UML) y tienes que ponerte a codificarlo. Llegado el momento de

implementar una función o método que es bastante *profundo* (usa otros métodos y estos a otros que aún no están implementados) podemos hacer una implementación rápida de las funciones no implementadas agregado un retorno por defecto y el comentario *TODO* con el texto de qué debe de hacer esa función.

```
public Producto obtenerProducto(String nombre) {
    //TODO buscar el producto por el nombre en la bd y retornarlo
    return null;
}
```

También se usan en caso de que implementando un código, nos damos cuenta de que no es todo lo eficiente que debería, pero no tenemos tiempo para dedicar a esa parte del código. En ese caso, podemos agregar un *FIXME* o *TODO*, depende si puede convertirse en problema o no.

```
public Producto obtenerProducto(String nombre) {
    /*FIXME En vez de obtener todos los productos de la bd y recorrerlos
    * ir a buscar el producto por nombre a la bd*/
    List<Producto> listaProductos = this.obtenerProductos();
    for (Producto p : listaProductos) {
        if (p.nombre.equals(nombre)) {
            return p;
        }
    }
    return null;
}
```

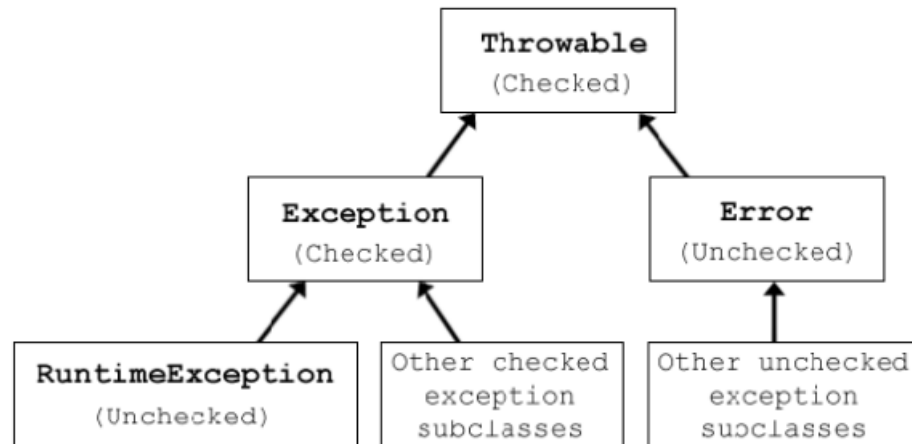
Por ultimo podríamos utilizar XXX (rara vez la encontraremos) donde detectamos que nuestro código está un poco “sucio” y podríamos arreglarlo para que sea más legible. El uso de estos *codetags* facilita encontrar entre las clases lo pendiente para poder seguir trabajando. Una ayuda inestimable para no perder el foco, mantener limpio nuestro código y poder pasar de una tarea a la siguiente sin demora.

Excepciones en Java

Las excepciones en Java son objetos.

Todos los tipos de excepciones deben extender de la clase Throwable o una de sus subclases.

Por convención, los nuevos tipos de excepciones extienden de la subclase Exception.



Las excepciones en Java son principalmente excepciones comprobadas (checked exception), lo que significa que el compilador comprueba que nuestros métodos lanzan las excepciones que ellos mismos han declarado que pueden lanzar.

Las excepciones de tiempo de ejecución (runTime) estándar y los errores graves son extensiones de las clases RuntimeException y Error, y ese tipo de errores son unchecked exceptions. Todas las excepciones que crea el usuario deberían ser extensiones de la clase Exception, esto es Checked Exceptions.

Ejemplo: reemplazar el valor de un atributo nombrado. Se debe controlar que exista el atributo.

```

public class AtributoNoEncontradoExcepcion extend Exception{
    public String nombreAtributo;
    public Object nuevoValor;

    AtributoNoEncontradoExcepcion(String nombre, Object valor){
        super("El atributo \""+nombre+"\" no se encontró");
        nombreAtributo = nombre;
        nuevoValor = valor;
    }
}
  
```

Throw

Las excepciones son lanzadas usando la cláusula throw.

Ejemplo

```

public void reemplazarValor(String nombre, Object nuevoValor) throw
AtributoNoEncontradoExcepcion{
  
```

```

    Atributo atributo = buscar(nombre); // busca el atributo
    if(atributo == null) //no lo encontró
        throw new AtributoNoEncontradoExcepcion(nombre, nuevoValor); //se lanza la
    excepción
    atributo.valueOf(nuevoValor);
  
```

```
}
```

El método primero busca el atributo a través de su nombre. Si no lo encuentra lanza un nuevo objeto de tipo `AtributoNoEncontradoExcepcion`, proveyendo al constructor con los datos descriptivos. Una excepción es un objeto, por lo tanto debe ser creada antes de ser lanzada. Si el atributo existe, su valor es reemplazado por un nuevo valor.

La cláusula throws

La primer cosa que el método de arriba hace es declarar cuales excepciones chequeadas serán lanzadas. Java requiere la declaración de excepciones chequeadas que el método lance, debido a que los programadores que la utilicen deben saber las posibles excepciones que lance el método así como el tipo de valor que retorne.

Se puede lanzar excepciones que son extensiones del tipo de excepción en la cláusula `throws`, debido a que se puede usar una clase en forma polimórfica en cualquier lugar que su superclase es esperada. Esto puede ocultar información al programador. Para propósitos de documentación la cláusula `throw` debería ser tan completa y específica como sea posible.

El contrato definido por la cláusula `throws` es estrictamente obligatorio, se puede usar solamente un tipo de excepción que ha sido declarada en la cláusula `throw`. Lanzar otro tipo de excepción es inválido. Si un método no tiene una cláusula `throws`, significa que no puede lanzarse ninguna excepción.

Todas las excepciones estándar de runtime (tales como `ClassCastException` y `ArithmeticException`) son extensiones de la clase `RuntimeException`. Errores más serios son señalados por excepciones que son extensiones de `Error`, y estas excepciones pueden ocurrir en cualquier momento del código. `RuntimeException` y `Error` son las únicas excepciones que no se necesita listar en la cláusula `throws`.

Inicializadores y bloques de código estáticos de inicialización no pueden lanzar excepciones chequeadas. Cuando se inicializan campos, la solución es inicializarlos dentro de un constructor, el cual puede lanzar excepciones. Para inicializadores estáticos, la solución es poner la inicialización dentro un bloque estático que catches y maneje la excepción. Los bloques estáticos no pueden lanzar excepciones, pero si pueden "cacharlas" (capturarlas).

Si se invoca un método que lista una excepción chequeada en la cláusula `throws`, hay tres elecciones:

- Capturar (`catch`) la excepción y manejarla.
- Capturar (`catch`) la excepción y mapearla dentro de una de sus excepciones lanzando una excepción de un tipo declarado en su propia cláusula `throws`.
- Declarar la excepción en su cláusula `throws` y dejar que la excepción pase a través de su método (aunque tiene que tener una cláusula `finally` que limpie primero, ver luego)

Para hacer estas cosas, se necesita tomar las excepciones lanzadas por otros métodos, lo cual se verá a continuación.

Try, catch y finally

El cuerpo de `try` es ejecutado hasta ya sea que una excepción es lanzada o que finalice satisfactoriamente. Si una excepción es lanzada, las cláusulas `catch` son examinadas para encontrar una para una excepción de la misma clase (o superclase). Puede haber cero o más

cláusulas catch. Si no hay ninguna, la excepción es lanzada hacia el código que invocó a este método.

Si una cláusula finally está presente, su código es ejecutado después de que se procesó completamente el try.

Ejemplo

```
Object valor = new Integer(8);
try{
    objetoAtribuido.sustituirValor("Edad", valor);
}catch(AtributoNoEncontradoExcepcion e){
    //no debería suceder, pero si suceder
    Atributo atributo = new Atributo(e.nombreAtributo, valor);
    objetoAtribuido.añadir(atributo);
}
```

Debido a que las cláusulas catch son examinadas en el orden en que fueron escritas, poner una cláusula catch que tenga una clase de excepción antes de otra que tenga una clase que la extiende, es un error.

Las excepciones son lanzadas por uno de estos posibles escenarios:

- Se llama un método que lanza una excepción, por ejemplo, el método `readline()` de `DataInputStream`.
- Se detecta un error y lanza una excepción con la sentencia `throw`
- Se produce un error de programa.
- Un error interno ocurre en Java.

Se declara la posibilidad de que su método puede lanzar una excepción con una especificación de excepción en la cabecera del método.

```
class Animacion{
    ....
    public Imagen cargarImagen(String s) throws IOException {
        ....
    }
}
```

Si el método debe lidiar con más de una excepción, se indicarán en la cabecera, separadas por coma.

Un método debe declarar todas las excepciones que lanza.

Si se sobrescribe un método de una clase padre, el método de la clase hija no puede lanzar más excepciones explícitas que el método de la clase padre (sí puede menos). En particular, si el método de la clase padre no lanza ninguna excepción explícita, tampoco lo hará la hija.

Creación de una clase Exception

```
class FileFormatException extends IOException{

    public FileFormatException()
    { }

    public FileFormatException(String g){
        super(g);
    }
}
```

¿Cómo lanzar una excepción?

Ejemplo

```
String readData(DataInput in) throws EOFException{
    while(...){
        if(ch == -1){
            if(n < len)
                throw new EOFException();
            //al lanzarla se crea una nueva excepción con new.
        }
    }
    ....
    return s;
}
```

Sintéticamente es:

- encontrar una clase de excepción apropiada.
- hacer un objeto de esa clase
- lanzarlo (dispararlo o activarlo)

Una vez que Java lanza una excepción, el método no retorna a quien lo llamó. Esto significa que no hay que preocuparse por un valor de retorno por defecto o un código de error.

Capturando excepciones

Para capturar una excepción se debe establecer un bloque try/catch.

```
try{
    código
}
catch(ExceptionType e){
    manejador para este tipo
    //al atraparla no se crea, se recibe como parámetro
}
```

Si el código dentro del bloque try/catch lanza una excepción de la misma clase que la dada en la cláusula catch, entonces:

- Java saltea el resto de código en el bloque try.
- ejecuta el código manejador dentro de la cláusula catch.

Si no se produce una excepción dentro del bloque try, entonces Java saltea la cláusula catch.

Si cualquier código en el método lanza una excepción de un tipo distinto que el contemplado en la cláusula catch. Java termina este método inmediatamente. Con la esperanza que uno de sus llamadores haya codificado una cláusula catch para este tipo.

Ejemplo

```
public static String readString(){
    int ch;
    String r = " ";
    boolean done = false;
    while(!done) {
        try{
            ch = System.in.read(); //puede lanzar una excepción
            if(ch < 0 || (char)ch == "\n")
                done = true;
        }
    }
}
```

```

else
    r = r + (char)ch;
//el problema en read() no es tenido en cuenta y el llamante sólo se
//se interesa por la cadena acumulada hasta este momento.
}catch(IOException e){
    done = true;
}
}
return r;
}

```

A veces la mejor opción es no hacer nada. Si un error ocurre en el método `read()`, se deja que se ocupe de ello al llamador del método `readString()`. Si se toma esta opción, entonces tenemos el hecho de que este método puede lanzar una `IOException`.

```

public static String readString() throws IOException{
    int ch;
    String r = "";
    boolean done = false;
    while(!done)
    {
        ch = System.in.read(); //puede lanzar una excepción
        if(ch < 0 || (char)ch == "\n")
            done = true;
        else
            r = r + (char)ch;
    }
    return r;
}

```

El compilador obliga la especificación `throws`. Si se llama un método que lanza una excepción explícita, se debe manipular o propagar.

Cuando se propaga una excepción, se debe agregar un modificador `throws` para alertar al llamante que una excepción puede ocurrir.

No está permitido agregar más especificaciones `throws` a un método de una clase hija que los que están presentes en el método de la clase padre.

Relanzando excepciones

Ocasionalmente se necesita relanzar una excepción que ha sido capturada por una cláusula `catch`.

```

Graphics g = image.getGraphics();
try
{
    código que puede lanzar una excepción
}
catch(MalformedURLException e)
{
    g.dispose();
    //se necesita hacer esto en forma urgente
    //y luego lanzar la excepción estrictamente
    throw e;
}

```

Cláusula Finally

Se utiliza para limpiar el estado interno o para liberar recursos que no son de objetos. Una mejor solución a relanzar excepciones es, por ejemplo:

```
Graphics g = image.getGraphics();
try{

    código que puede lanzar excepciones
}
catch(IOException e)
done = true;
finally
g.dispose();
```

El código en la cláusula finally es ejecutado ya sea que una excepción es capturada o no.

Try con recursos

A partir de java 7 se incluyeron los try con recursos, para solucionar el inconveniente de tener que cerrar archivos o conexiones a base de datos en el finally. Como los métodos close lanzan una excepción, nos obliga a tener que poner un try y catch vacíos dentro de los bloques finally.

Antes de java 7	A partir de java 7
<pre>public static void main(String[] args) { BufferedReader buffer = null; try { String linea; buffer = new BufferedReader(new FileReader("C:test.txt")); while ((linea = buffer.readLine()) != null) { System.out.println(linea); } } catch (IOException e) { e.printStackTrace(); } finally { try { if (buffer != null) { buffer.close(); } } catch (IOException ex) { ex.printStackTrace(); } } }</pre>	<pre>public static void main(String[] args) { try (BufferedReader buffer = new BufferedReader(new FileReader("C:test.txt"))){ String linea; while ((linea = buffer.readLine()) != null) { System.out.println(linea); } } catch (IOException e) { e.printStackTrace(); } }</pre>

Como se puede observar el código es más sencillo y más fácil de leer, reduciendo bastante el número de líneas y también se elimina el bloque finally para el cierre de recursos.

En este ejemplo, el recurso declarado en la sentencia try-with-resources es un `BufferedReader`. La declaración aparece entre paréntesis a continuación del try, de esta manera la instancia declarada en la sentencia try-with-resource será cerrada automáticamente cuando el bloque de sentencias del try sean ejecutadas completamente o cuando se produzca una excepción.

En la sentencia se pueden declarar tantos recursos como sea necesario, cerrándose de manera inversa a como han sido declarados

Test Unitario

Una prueba unitaria, en programación, es una forma de probar el correcto funcionamiento de un módulo de código.

Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado. Luego, con las Pruebas de Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

Es un test de caja negra, que quiere decir esto, que solo me importan los resultados, si yo tengo que testear una calculadora no me importa ver como hace las cuentas, solo me importa que si yo pongo $1 + 1$ el resultado sea 2. Hay otros tipos de test que evalúan el funcionamiento (test de caja blanca), pero los Test Unitarios solo evalúan los output.

Características

- **Automatizable:** No debería requerirse una intervención manual, los test unitarios deben ejecutarse de forma automática.
- **Completas:** Deben cubrir la mayor cantidad de código. Para que estos nos rindan al 100% debemos intentar testear todo nuestro código unitariamente.
- **Repetibles o Reutilizables:** No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. Porque lo que queremos es testear nuestra aplicación de forma automática cada vez que realizamos un cambio.
- **Independientes:** La ejecución de una prueba no debe afectar a la ejecución de otra, ya que si un test falla, el resto podrían fallar debido a que estaban conectados entre si.
- **Profesionales:** Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc. Nosotros tenemos que programar y testear de forma tal que cualquier persona pueda entender que es lo que estamos haciendo y porque.

Ventajas

Una de las principales ventajas que tenemos es fomentar el cambio y el orden, esto se da por dos motivos, al tener todo nuestro código testeado se hace muy simple incluir modificaciones en nuestro programa sin arruinar funcionalidades previas y fomenta el orden ya que al programar pensando en realizar test unitarios, vamos a ir comentando el código para no olvidarnos los posibles test que debemos realizar.

Otra de las ventajas que nos brinda es simplificar las implementaciones, por ejemplo si vamos a realizar un login de un sistema, dos programadores podrían dividirse las tareas de diseño de la interfaz grafica y métodos de validación de usuarios (esto se denomina front-end y back-end, respectivamente). El programador que va a realizar los métodos para validar, podría trabajar sin la necesidad de que la intefaz este programada. También va a poder testear su codigo con muchas combinatorias diferentes antes de integrar las tareas.

Al realizar test de nuestra aplicación, de cierta manera, estamos documentando nuestro código. Porque digo esto, yo leyendo todos los test sobre una funcionalidad puedo entender

perfectamente su funcionalidad sin ver todo el código. Por ejemplo si yo les digo tengo un método que le envié (2 , 3) y de retorno debería obtener un 5 y si le paso (3 -4) y de retorno debería obtener un -1 uds. ya puede entender que hace ese método sin necesidad de ver que hago con esos números internamente.

También nos facilita la detección de errores y la localización en nuestro sistema, cuando veamos el Framework JUnit veremos que simple es encontrar un error en nuestra aplicación.

JUnit

Ya que nombre a JUnit, voy a explicar muy brevemente que es. Es un framework, como bien dije antes, que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.

Funciona de la siguiente manera:

En función de algún valor de entrada se evalúa el valor de retorno esperado. Si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba. En caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

El Framework posee una clase llamada Assert que provee un conjunto de métodos útiles para escribir pruebas.

- assertEquals: iguala dos objetos.
- assertNotNull: verifica que no sea null.
- assertNull: verifica que sea null.
- assertFalse: verifica que una condición sea false.
- assertTrue: verifica que una condición sea true
- assertEquals: iguala dos arrays
- assertNotSame: verifica que dos referencias no sean iguales.
- assertEquals: verifica que dos referencias sean iguales.
- fail: hace fallar el test.

También incluye algunas anotaciones como @Test para indicar que un método debe ser testados, @Before y @After para indicar que hacer antes y después de iniciar un test y @Ignore para deshabilitar tests.

En la actualidad las IDE como NetBeans y Eclipse cuentan con plug-ins que permiten que la generación de las plantillas necesarias para la creación de las pruebas de una clase Java se realice de manera automática, facilitando al programador enfocarse en la prueba y el resultado esperado, y dejando a la herramienta la creación de las clases que permiten coordinar las pruebas.

Problemas a la hora de realizar los test

No es fácil escribir buenos tests; de hecho, según el tío Bob (Robert Cecil Martin, consultor de software y autor estadounidense), no todo el mundo es capaz de hacerlo: es una cualidad con la que algunos programadores nacen y que a otros les cuesta un gran esfuerzo aprender. La idea detrás de esta premisa es que podemos ser buenísimos desarrolladores, con años de experiencia, pero no por ello tenemos que saber escribir buenos tests unitarios sin la práctica necesaria. La razón es simple, es un tipo diferente de codificación y caeríamos en un error si asumiéramos que solo por nuestra experiencia previa, poseemos la habilidad innata para escribirlos.

Muchos de los tests unitarios que encontramos en quienes se inician en esta metodología son francamente inútiles. No es culpa de los desarrolladores; para la gran mayoría, el primer paso es instalar un framework del tipo JUnit y lanzarse a escribir tests previos para aquellas clases o métodos que necesitan. Una vez el ciclo en marcha, puede parecer que alternar las luces rojas y verdes es el buen camino: escribimos un test que falla, hacemos lo necesario para superarlo y avanzamos. ¿Todo correcto? No!.

Si la calidad de nuestros tests no es la suficiente, entonces éstos no aportan ningún valor añadido, sino todo lo contrario.

Es importante darse cuenta de que las pruebas unitarias no descubrirán todos los errores del código. Por definición, sólo prueban las unidades por sí solas. Por lo tanto, no descubrirán errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto. Además, podemos no anticipar todos los casos especiales de entradas que puede recibir la unidad de programa a testear. Las pruebas unitarias sólo son efectivas si se usan en conjunto con otras pruebas de software.

Sin lugar a dudas, los tests unitarios pueden mejorar significativamente la calidad de nuestros proyectos a la vez que aumentan nuestra productividad de forma indirecta. Sin embargo, no hay que caer en la trampa de que un código que presenta tests unitario es mejor que otro que no; ésto es sólo relativo a la calidad del diseño y al uso que hacemos de los mismos.

Ficheros Input / Output

La programación de operaciones de **entrada/salida (input/output o I/O)** se resuelve a través de clases que provee la plataforma Java:

- Clases básicas, o "Standard I/O", en el paquete **java.io**, que existe desde los comienzos, en JDK 1.0, y es para el modo llamado "stream-based".
- La llamada "New I/O", en los paquetes **java.nio**, introducida en J2SE 1.4, que introduce los conceptos de buffer.
- En J2SE 5.0, se incluye el "Formatted text-I/O", a través de las clases **java.util.Scanner** (lectura) y **java.util.Formatter** (escritura con método **format**).
- En Java SE 7, se incluye el llamado **NIO.2 (non-blocking I/O)**, en el paquete **java.nio.file**, con nuevas funcionalidades de manejo de archivos.

Esta materia, describe el uso de Standard I/O.

La clase File

- La clase **java.io.File** puede representar un archivo o un directorio.
- Se construye a partir del nombre y opcionalmente un path (ruta), que si no se especifica utiliza la default, que es aquella donde se ejecuta el programa.
- Tiene asociado un path para localizarlo.
- Ejemplos de creación de un objeto **File**:

```
//Archivo en ubicación actual (raíz del proyecto, poco utilizado)
File file = new File("proj.txt");
//Archivo en ubicación absoluta, nomenclatura Windows. Un '\' se escapa con un '\'
File textFile = new File("c:\\data\\data.txt");
//Directorio, ubicación absoluta, nomenclatura Linux/Unix/Mac
File tempDir = new File("/tmp");
//Archivo, especificando directorio en primer parámetro.
File tempFile = new File("/tmp","info.bak");
```

En Java SE 7, la clase **java.nio.file.Path** cubre las limitaciones que tiene la clase **File**.
path y sistema operativo

Los path tienen una nomenclatura **dependiente del sistema operativo**:

- Separador de **directorios**:
 - Windows utiliza backslash ('\').
 - Linux/Unix/Mac utilizan slash ('/').
- Separador de **path**:
 - Windows utiliza punto y coma (;').
 - Linux/Unix/Mac utilizan dos puntos (':').
- **Delimitador** de línea:
 - Windows utiliza un retorno de carro y nueva línea ('\r\n').
 - Linux/Unix utilizan sólo nueva línea ('\n').
 - Mac sólo retorno de carro ('\r').
- **Disco**:
 - Windows mapea el disco con una unidad ('c:', 'd:').
 - Linux/Unix/Mac utilizan una única raíz ('/').

Los valores de sistema operativo se pueden obtener programáticamente, lo que permite que el código sea portable:

- Carácter separador de directorios: con `File.separator` (String) y `File.separatorChar` (char).
- Carácter separador de rutas (path): `File.pathSeparator` y `File.pathSeparatorChar`.
- Delimitador de línea: Se puede obtener desde una propiedad de sistema, con `System.getProperty("line.separator")`, como String.

La clase `File` se puede utilizar para operaciones sobre directorios y archivos:

- Si un archivo o directorio existe se utiliza el método **`exists()`**
- Saber si un objeto `File` es archivo o directorio se utilizar el método **`isFile()`** o **`isDirectory()`**
- En caso de directorio, si no existe crearlo. El método **`mkdir()`** crea un directorio y **`makedirs()`** crea la ruta completa.
- El método **`list()`** lista de todos los nombres de archivos del directorio.
- El método **`listFiles()`** lista de todos los archivos del directorio.

Escritura y lectura de texto

Para escribir a un destino con un formato de texto, se utiliza alguna clase que hereda de `Writer`. Las más utilizadas son:

- **`OutputStreamWriter`**: Puente en escritura de caracteres a bytes, utilizando charset.
- **`PrintWriter`**: Escribe texto con formato. `System.out` es un `PrintWriter`.
- **`StringWriter`**: Escritura hacia un String, en memoria, para ejecución incremental.
- **`BufferedWriter`**: Optimiza escritura en destino de datos, almacenando en un buffer.

Para leer desde un origen con un formato de texto, se utiliza alguna clase que hereda de `Reader`. Las más utilizadas son:

- **`InputStreamReader`**: Puente en lectura de bytes a caracteres, utilizando charset.
- **`StringReader`**: Lectura desde un String, en memoria, para ejecución incremental.
- **`BufferedReader`**: Optimiza acceso a fuente de datos, almacenando en un buffer.

A continuación se muestra un ejemplo de lectura de un String y escritura en un archivo de texto:

```
String dh = " Toda persona tiene los derechos y libertades " +
    System.getProperty("line.separator") +
    "proclamados en esta Declaración, sin distinción alguna de ...";

File dhFile = new File("files", "DDHH.txt");

StringReader sr = new StringReader(dh); //lector de String
BufferedReader br = new BufferedReader(sr); //Permite lectura línea a línea
FileWriter fw = new FileWriter(dhFile); //Escribe en archivo definido en dhFile
BufferedWriter bw = new BufferedWriter(fw); //Optimiza acceso a disco en escritura

/** Lee línea a línea el String, a través del BufferedReader,
 * y escribe línea a línea a través del BufferedWriter */
String line = br.readLine();
while (line != null) {
    bw.write(line);
    line = br.readLine();
    if (line != null) {
        bw.newLine();
    }
}
bw.flush(); //Fuerza finalización de escritura del buffer acumulado
bw.close(); //Cierre del BufferedWriter, que cierra el FileWriter.
//El reader no es necesario cerrarlo, porque sólo trabaja en memoria.
```

En el siguiente ejemplo se lee un archivo, y se escribe en un String, utilizando una copia por bloque de caracteres, lo que permite lecturas de tamaño acotado:

```
File dhFile = new File("files", "DDHH.txt");

FileReader fr = new FileReader(dhFile); //Lee el archivo definido en dhFile
BufferedReader br = new BufferedReader(fr); //Optimiza acceso a disco en lectura
StringWriter sw = new StringWriter(); //Escribe en un String. No necesita buffer.

/** Lee en bloques de 100 caracteres desde el reader, y los copia en el writer. */
int size = 100;
char[] buffer = new char[size];
int len;
while ((len = br.read(buffer, 0, size)) > -1) {
    sw.write(buffer, 0, len);
}
//Cierre del BufferedReader, que cierra el FileReader.
br.close();
//El writer no es necesario cerrarlo, porque sólo trabaja en memoria.
//Extrae el String desde el StringWriter
System.out.println(sw.toString());
```

Escritura y lectura binaria

Para escribir a un destino con un formato binario, se utiliza alguna clase que hereda de `OutputStream`. Las más utilizadas son:

- **FileOutputStream:** Escritura hacia archivo. Se construye normalmente desde un `String` o `File`.
- **ByteArrayOutputStream:** Escritura hacia un `byte[]` en memoria. Para recorrido incrementa.

- **ObjectOutputStream:** Escritura hacia objetos serializados, (convertidos a cadena de bytes). Se complementa con otro OutputStream.
- **FilterOutputStream:** Sólo wrapper de otro OutputStream.
- **DataInputStream:** Extiende de FilterOutputStream, escritura de datos en una forma independiente de la plataforma.
- **BufferedInputStream:** Extiende de FilterOutputStream, optimiza escritura en destino de datos, almacenando en un buffer.

Para leer desde un origen con un formato binario, se utiliza alguna clase que hereda de InputStream. Las más utilizadas son:

- **FileInputStream:** Lectura desde archivo. Se construye normalmente desde un String o File.
- **ByteArrayInputStream:** Lectura desde un byte[] en memoria. Para recorrido incremental.
- **ObjectInputStream:** Lectura desde objetos serializados, (convertidos a cadena de bytes). Se complementa con otro InputStream.
- **FilterInputStream:** Sólo wrapper de otro InputStream.
- **DataInputStream:** Lectura de datos en una forma independiente de la plataforma.
- **BufferedInputStream:** Optimiza acceso a fuente de datos, almacenando en un buffer.

Por ejemplo, en el siguiente caso se lee un archivo binario, y se copia en otro, utilizando bloques de 1024. También es aplicable a archivos de texto:

```
File orig = new File("files", "orig.pdf");
File dest = new File("files", "dest.pdf");

FileInputStream fis = new FileInputStream(orig); //Lee el archivo definido en orig
BufferedInputStream bis = new BufferedInputStream(fis); //Optimiza lectura de disco
FileOutputStream fos = new FileOutputStream(dest); //Escribe en archivo dest
BufferedOutputStream bos = new BufferedOutputStream(fos); //Optimiza escritura a disco
/** Lee bloques de 1024 bytes desde el input stream, y los copia en el output stream. */
int size = 1024;
byte[] buf = new byte[size];
int len;
while ((len = bis.read(buf, 0, size)) > -1){
    bos.write(buf, 0, len);
}
//Fuerza escritura buffer
bos.flush();
//Se deben cerrar ambos, por ir a disco.
bis.close();
bos.close();
```

Obsérvese que en ningún momento el archivo se carga completo a memoria, sino sólo pequeños trozos, lo que permite manejar archivos muy grandes.

Serialización

La serialización es un caso de entrada/salida, en la cual un objeto es convertido en una cadena de bytes para ser enviado a un destino. La deserialización es el proceso inverso, es decir, desde una cadena de bytes se puede recuperar el objeto.

Para que un objeto se pueda serializar, debe cumplir con una de las dos siguientes condiciones:

- Ser marcado con la interfaz **Serializable**. Es la opción comúnmente utilizada. Serializa el objeto y sus atributos de instancia. Opcionalmente, se pueden implementar los métodos `writeObject` y `readObject` para modificar el comportamiento de la serialización y deserialización.
- Implementar la interfaz **Externalizable**. Permite construir, a través de los métodos `writeExternal` y `readExternal`, una serialización y deserialización personalizadas. Tiene mayor flexibilidad que la forma default de `Serializable`, para los casos especializados, aunque en muchos casos no es necesaria.

Para que se pueda ejecutar la serialización de una clase `Serializable`, se deben dar las siguientes condiciones:

- Si **no** se implementa el método `writeObject`:
 - Todos sus atributos de instancia (no static) deben ser serializables.
 - Si algún atributo no es serializable, puede ser marcado con el modificador `transient`. En ese caso, no se envía, recuperándose en la deserialización como `null`.
 - También se pueden marcar como `transient` atributos serializables, en caso que no se quieran incluir en la serialización.
- Si se implementa el método `writeObject`, y no se utiliza la invocación a `defaultWriteObject()`, pueden haber atributos no serializables, siempre que no se utilicen.

Sabiendo que la clase `Employee` cumple con alguna de las condiciones, a continuación se muestra un ejemplo de cómo serializar un objeto y almacenarlo en un archivo `.bin` dentro de la carpeta `data`.

```
public static void main(String[] args) {
    Employee em = new Employee();
    em.setName("Juan");
    em.setSalary(BigDecimal.valueOf(23456.78));

    serialize(em);
}

private static void serialize(Employee emp) {

    File file = new File("data", "emp2.bin");
    FileOutputStream fos = null;
    ObjectOutputStream oos = null;
    try {
        fos = new FileOutputStream(file);
        oos = new ObjectOutputStream(fos);
        oos.writeObject(emp);

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        IOUtil.close(oos);
    }
}
```

El objeto `ObjectOutputStream` internamente verifica que el objeto cumpla con la interfaz `Externalizable`, y si no, `Serializable`. Si no cumple con ninguna, lanza una `NotSerializableException`.

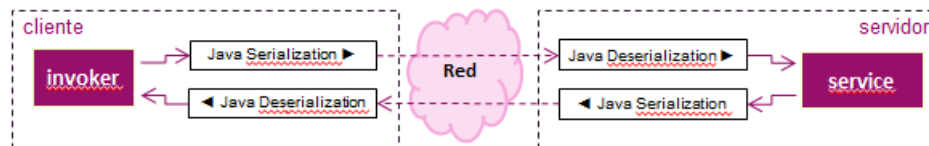
Para poder recuperar el archivo `.bin` y poder recuperar el objeto `Employee`, deberíamos deserializarlo con el siguiente código:

```
private static Employee deserialize() {
    Employee emp = null;
    File file = new File("data", "emp2.bin");
    FileInputStream fis = null;
    ObjectInputStream ois = null;
    try {
        fis = new FileInputStream(file);
        ois = new ObjectInputStream(fis);
        emp = (Employee) ois.readObject();

    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        IOUtil.close(ois);
    }
    return emp;
}
```

aplicación

Aunque los objetos se pueden almacenar serializados, la mayor utilidad de la serialización es la invocación remota en un entorno cliente/servidor:



En la invocación (lado cliente), los parámetros del método son serializados, enviados a través de la red, y deserializados (lado servidor). Después de la ejecución del método, el objeto de retorno es serializado por el servidor, enviado a través de la red, y deserializado por el lado cliente.

Colecciones

Las colecciones son una especie de arrays de tamaño dinámico. Para usarlas haremos uso del Java Collections Framework (JCF), el cual contiene un conjunto de clases e interfaces del paquete `java.util` para gestionar colecciones de objetos.

En Java las principales interfaces que disponemos para trabajar con colecciones son:

Collection, Set, List, Queue y Map:

- `Collection<E>`: Un grupo de elementos individuales, frecuentemente con alguna regla aplicada a ellos.
- `List<E>`: Elementos en una secuencia particular que mantienen un orden y permite duplicados. La lista puede ser recorrida en ambas direcciones. Hay 3 tipos de implementaciones:
 - `ArrayList<E>`: Su ventaja es que el acceso a un elemento en particular es ínfimo. Su desventaja es que para eliminar un elemento, se debe mover toda la lista para eliminar ese “hueco”.
 - `Vector<E>`: Es igual que `ArrayList`, pero sincronizado. Es decir, que si usamos varios hilos, no tendremos de qué preocuparnos hasta cierto punto.
 - `LinkedList<E>`: En esta, los elementos están conectados con el anterior y el posterior. La ventaja es que es fácil mover/eliminar elementos de la lista, simplemente moviendo/eliminando sus referencias hacia otros elementos. La desventaja es que para usar el elemento N de la lista, debemos realizar N movimientos a través de la lista.

- Ejemplo

```
List lista2 = new ArrayList();
// Añadimos nodos y creamos un Iterator
lista2.add("Madrid");
lista2.add("Valencia");
Iterator iterador2 = lista2.iterator();

// Recorremos y mostramos la lista
while (iterador2.hasNext()) {
    String elemento = (String) iterador2.next();
    System.out.print(elemento + " ");
}
```

- `Set<E>`: No puede haber duplicados. Cada elemento debe ser único, por lo que si existe uno duplicado, no se agrega. Por regla general, cuando se redefine `equals()`, se debe redefinir `hashCode()`. Es necesario redefinir `hashCode()` cuando la clase definida será colocada en un `HashSet`. Los métodos `add(o)` y `addAll(o)` devuelven `false` si “o” ya estaba en el conjunto.
 - `HashSet<E>`: Se obtienen tiempos performantes en las operaciones básicas (`add`, `remove`, `contains` y `size`) No garantiza que el orden de los elementos se mantenga en el tiempo. En los objetos deben definir un `hashCode`.
 - `TreeSet<E>`: Un Set ordenado respaldado por un árbol. De esta forma, se puede extraer una secuencia ordenada de objetos. No es muy performante a la hora de iterar. El orden está dado por la implementación del método `compareTo` de la interfaz `Comparable`.

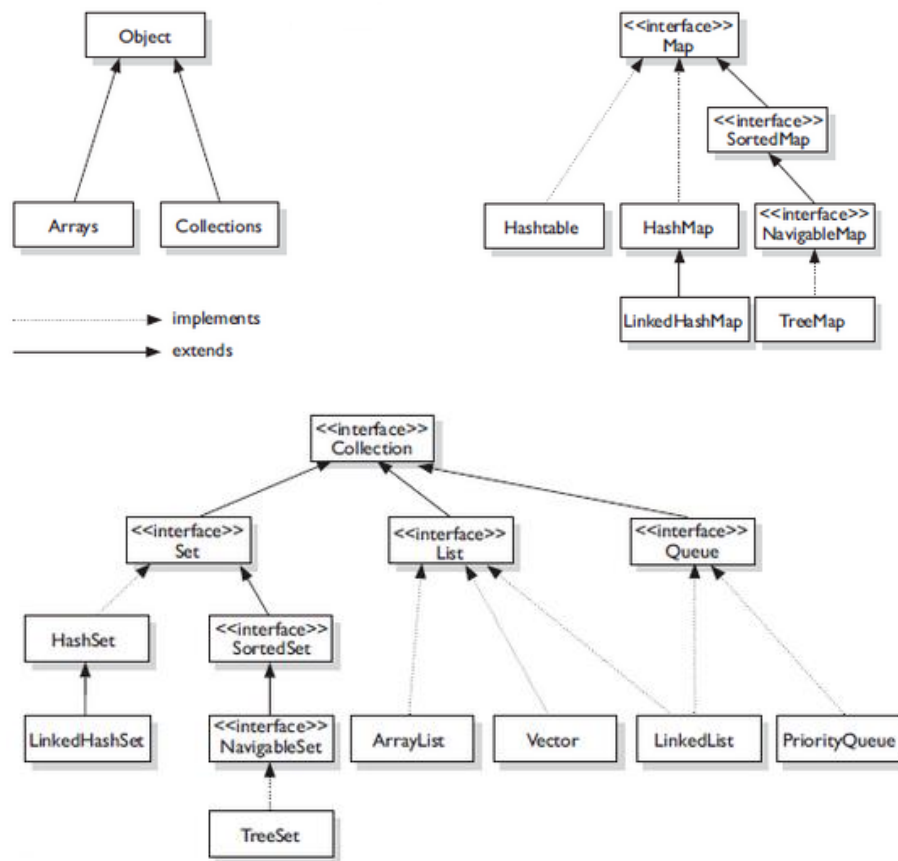
- **Queue<E>**: Colección ordenada con extracción por el principio e inserción por el principio (LIFO – Last Input, First Output) o por el final (FIFO – First Input, First Output). Se permiten elementos duplicados. No da excepciones cuando la cola está vacía/llena, hay métodos para interrogar, que devuelven null. Los métodos put()/take() se bloquean hasta que hay espacio en la cola/haya elementos.
- **Map<K,V>**: Un grupo de pares objeto clave-valor, que no permite duplicados en sus claves. Es quizás el más sencillo, y no utiliza la interfaz Collection. Los principales métodos son: put(), get(), remove().

- **HashMap<K,V>**: Se basa en una tabla hash, pero no es sincronizado.
- **HashTable<K,V>**: Es sincronizado, aunque que no permite null como clave.
- **LinkedHashMap<K,V>**: Extiende de HashMap y utiliza una lista doblemente enlazada para recorrerla en el orden en que se añadieron. Es ligeramente más rápida a la hora de acceder a los elementos que su superclase, pero más lenta a la hora de añadirlos.
- **TreeMap<K,V>**: Se basa en una implementación de árboles en el que ordena los valores según las claves. Es la clase más lenta.
- **Ejemplo:**

```
// Definir un HashMap
Map global = new HashMap();

// Insertar valores "key"- "value" al HashMap
global.put("Laura", "667895789");
global.put("Pepe", "645895756");
global.put("Abelardo", "55895711");

// Definir Iterator para extraer o imprimir valores
for( Iterator it = global.keySet().iterator(); it.hasNext();) {
    String s = (String)it.next();
    String s1 = (String)global.get(s);
    System.out.println("Alumno: "+s + " - " + "Telefono: "+s1);
}
```



La siguiente tabla muestra todo lo que se puede hacer con una Collection (sin incluir los métodos que vienen automáticamente con Object), y por consiguiente, todo lo que se puede hacer con un Set o un List. (List también tiene funcionalidad adicional.) Los objetos Map no se heredan de Collection, y se tratarán de forma separada.

Método	Descripción
boolean add(Object)	Asegura que el contenedor tiene el parámetro. Devuelve falso si no añade el parámetro.
boolean addAll(Collection)	Añade todos los elementos en el parámetro. Devuelve verdadero si se añadió alguno de los elementos.
void clear()	Elimina todos los elementos del contenedor
boolean contains(Object)	Verdadero si el contenedor almacena el parámetro.
boolean containsAll(Collection)	Verdadero si el contenedor guarda todos los elementos del parámetro.
boolean isEmpty()	Verdadero si el contenedor no tiene elementos.
Iterator iterator()	Devuelve un Iterator que se puede usar para recorrer los elementos del contenedor
boolean remove(Object)	Si el parámetro está en el contenedor, se elimina una instancia de ese elemento. Devuelve verdadero si se produce alguna eliminación.
boolean removeAll(Collection)	Elimina todos los elementos contenidos en el parámetro.

<code>removeAll(Collection)</code>	Devuelve verdadero si se da alguna eliminación
<code>boolean retainAll(Collection)</code>	Mantiene sólo los elementos contenidos en el parámetro (una "intersección" en teoría de conjuntos). Devuelve verdadero si se dio algún cambio
<code>int size()</code>	Devuelve el número de elementos del contenedor
<code>Object[] toArray()</code>	Devuelve un array que contenga todos los elementos del contenedor
<code>Object[] toArray(Object[] a)</code>	Devuelve un array que contiene todos los elementos del contenedor, cuyo tipo es el del array a y no un simple Object (hay que convertir el array al tipo correcto).

Nótese que no hay función `get()` para selección de elementos por acceso al azar. Eso es porque `Collection` también incluye `Set`, que mantiene su propia ordenación interna (y por consiguiente convierte en carente de sentido el acceso aleatorio).

Ordenar un ArrayList en Java

Para realizar estas operaciones necesitaremos trabajar con la Clase “Collections” de Java. No vamos a explicar las características de esta clase, sino que simplemente vamos a utilizarla para ordenar `ArrayList`.

Para ordenar un `ArrayList` de menor a mayor, vamos a utilizar el método “sort” de la clase `Collections`, con este método nos devolverá el `ArrayList` ordenado de menor a mayor.

```
public static void main (String[] args) {
    List<Integer> lista = new ArrayList<Integer>();
    lista.add(3); //index 0
    lista.add(1); //index 1
    lista.add(2); //index 2
    System.out.println("Lista sin ordenar:");
    for(Integer i : lista){
        System.out.println(i);
    }
    Collections.sort(lista);
    System.out.println("Lista ordenada:");
    for(Integer i : lista){
        System.out.println(i);
    }
}
```

La salida en la consola es la siguiente:

```
Lista sin ordenar:
3
1
2
Lista ordenada:
1
2
3
```

Para ordenarlo de mayor a menor, tenemos que crearnos un objeto de la clase `Comparator` para que compare los elementos y los ponga en orden inverso. Esto lo hacemos de la siguiente manera:

```
Comparator<Integer> comparador = Collections.reverseOrder();
Collections.sort(lista, comparador);
```

Cuando nosotros creamos una clase de algún tipo, podemos especificar el orden natural de los objetos de esa clase cuando se encuentran en una lista, simplemente implementando la interfaz Comparable y sobrescribiendo el método:

int compareTo(Object o);

Este método compara este objeto con el objeto recibido como parámetro. Este método devolvera un valor negativo si este objeto es menor que el recibido, 0 si son iguales o un valor positivo si es mayor, según el orden natural de los objetos.

Por ejemplo si tenemos una clase Persona y queremos ordenar por el nombre de menor a mayor, deberimos sobrescribir el método de la siguiente manera:

```
@Override
public int compareTo(Object o) {
    Persona p = (Persona) o;
    return this.nombre.compareTo(p.nombre);
}
```

Luego simplemente ejecutando la siguiente línea se va a ordenar:

```
Collections.sort(lista);
```

Tipos genéricos

Los tipos genéricos permiten forzar la seguridad de los tipos, en tiempo de compilación, en las colecciones (u otras clases y métodos que utilicen tipos parametrizados).

El problema que resuelven es que si al crear una colección de un tipo determinado, pongamos String, yo meto un elemento de tipo entero, entonces me dará una excepción. Los genéricos ayudaron a crear una comprobación de tipo en listas y mapas en tiempo de compilación, Antes de Java 5 cuando introducíamos objetos en una colección estos se guardaban como objetos de tipo Object, aprovechando el polimorfismo para poder introducir cualquier tipo de objeto en la colección. Esto nos obligaba a hacer un casting al tipo original al obtener los elementos de la colección. Esta forma de trabajar no solo nos ocasiona tener que escribir más código innecesariamente, sino que es propenso a errores porque carecemos de un sistema de comprobación de tipos. Si introdujéramos un objeto de tipo incorrecto el programa compilaría pero lanzaría una excepción en tiempo de ejecución al intentar convertir el objeto en String. Desde Java 5 contamos con una característica llamada generics que puede solventar esta clase de problemas. Los generics son una mejora al sistema de tipos que nos permite programar abstrayéndonos de los tipos de datos, de forma parecida a los templates o plantillas de C++ (pero mejor).

Gracias a los generics podemos especificar el tipo de objeto que introduciremos en la colección, de forma que el compilador conozca el tipo de objeto que vamos a utilizar, evitándonos así el casting. Además, gracias a esta información, el compilador podrá comprobar el tipo de los objetos que introducimos, y lanzar un error en tiempo de compilación si se intenta introducir un objeto de un tipo incompatible, en lugar de que se produzca una excepción en tiempo de ejecución.

Para utilizar generics con nuestras colecciones solo tenemos que indicar el tipo entre `< >` a la hora de crearla. A estas clases a las que podemos pasar un tipo como “parámetro” se les llama clases parametrizadas, clases genéricas, definiciones genéricas o simplemente genéricas (generics).

Otra cosa que podemos hacer con los tipos parametrizados o genéricos es crear nuestras propias clases.

Ejemplo:

```
// --- Sin tipos genericos --- /
public class Box{
    private Object object;

    public void add(Object object) {
        this.object = object;
    }

    // A la hora de devolver un objeto, necesitamos hacer un casting
    // y si hacemos un casting de varios tipos, nos generara una
    // excepcion en tiempo de ejecucion
    public Object get() {
        return this.object;
    }
}

// --- Con tipos genericos --- /
// Para solucionarlo, usaremos esto:
public class Box<T>{
    private T t;
```

```

    public void add(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}
// Debemos instanciar la clase de la siguiente forma:
Box <String> boxes = new Box<String> ();
// Todos los tipos parametrizados se reemplazaran por la <Clase>

```

En ocasiones también puede interesarnos limitar los tipos con los que se puede parametrizar nuestra clase. Por ejemplo, podríamos querer crear una clase que solo nos permita guardar números Integer, Double, etc. Nuestro código podría tener un aspecto similar al siguiente.

```

// --- Con tipos genericos solo numerico--- /
// Para solucionarlo, usaremos esto:
public class Box<N extends Number>{
    private N numero;

    public void add(N numero ) {
        this.numero = numero;
    }
    public N get() {
        return numero;
    }
}

```

Otra cosa que podemos hacer es utilizar más de un parámetro de tipo, separando los tipos con comas. Supongamos por ejemplo que quisiéramos crear una clase que imprimiera la suma de dos números. Escribiríamos algo como esto:

```

public class Sumador< N1 extends Number, N2 extends Number> {
    private N1 numero1;
    private N2 numero2;

    public Sumador(N1 numero1, N2 numero2) {
        this.numero1 = numero1;
        this.numero2 = numero2;
    }

    public void sumar() {
        System.out.println(numero1.doubleValue() + numero2.doubleValue());
    }
}

```

No solo podemos utilizar datos genéricos a nivel de clases, también los podemos utilizar a nivel de métodos

```

public class Consulta {
    public <T extends Object> T mostrarRetornar(T parametro){
        System.out.println(parametro.toString());
        return parametro;
    }
}

```

En el ejemplo anterior tenemos un método el cual recibe por parámetro un tipo de dato genérico “T” y lo retorna. De esta manera nos olvidamos los casteos.

Es decir, básicamente es un tipo “inventado” al cual le decimos de qué tipo queremos que nos pase los datos y él internamente, hace los diferentes castings que haya que hacerle a los diferentes elementos de la colección, siendo los tipos de éstos transparentes para nosotros.

Las convenciones de declaración utilizan la letra T para tipos y E para elementos de una colección. Se puede utilizar más de un tipo parametrizado en una declaración. Por convención los tipos parametrizados son letras solitarias mayúsculas, sin esta convención sería difícil leer el código y decir la diferencia entre una variable parametrizada y una clase ordinaria o un nombre de una interface.

- E – Elemento (Usado extensivamente en las colecciones en java)
- K – Key
- N – Number
- T – Type
- V – Value

Patrones





Introducción

Diseñar software orientado a objetos es difícil, y diseñar software orientado a objetos reutilizable es todavía más difícil. Un software capaz de evolucionar tiene que ser reutilizable (al menos para las versiones futuras).





Para anticiparse a los cambios en los requisitos hay que diseñar pensando en que aspectos pueden cambiar y tener en cuenta que los patrones de diseño están orientados al cambio.

Introducción a los patrones

¿Cómo puedo yo, llegar a ser un maestro del ajedrez?





-  Debo aprender las reglas del juego (nombre de las piezas, movimientos legales, orientación del tablero, etc)
-  Debo aprender los principios (valores relativos de las piezas, valores estratégicos de las casillas centrales, jaque cruzado, etc.)
-  Sin embargo para llegar a ser un maestro hay que estudiar las partidas de otros maestros (Estas partidas contienen patrones que deben ser entendidos, memorizados y aplicados repetidamente)
-  Hay cientos de estos patrones.

¿Cómo puedo yo, llegar a ser un maestro del software?

-  Debo aprender las reglas (algoritmos, estructuras de datos, lenguajes de programación, etc.)
-  Debo aprender los principios (programación estructurada, programación modular, programación OO, programación genérica, etc.)
-  Sin embargo, para llegar a ser un maestro, hay que estudiar los diseños de otros maestros (Estos diseños contienen patrones que deben ser entendidos, memorizados y aplicados repetidamente)
-  Hay cientos de estos patrones

Christopher Alexander (1977): *Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y describe la esencia de la solución a ese problema, de tal modo que pueda utilizarse esta solución un millón de veces más, sin siquiera hacerlo de la misma manera dos veces*

Un patrón es:

-  una solución a un problema en un contexto particular
-  recurrente (lo que hace la solución relevante a otras situaciones)
-  enseña (permite entender cómo adaptarlo a la variante particular del problema donde se quiere aplicar)
-  tiene un nombre para referirse al patrón

- 🔥 Los patrones facilitan la reutilización de diseños y arquitecturas software que han tenido éxito
- 🔥 conexiones entre componentes de programas
- 🔥 la forma de un diagrama de objeto o de un modelo de objeto.

Clasificación de los patrones:

🔥 Patrones *arquitecturales*

- Expresan un paradigma fundamental para estructurar un sistema software
- Proporcionan un conjunto de subsistemas predefinidos, con reglas y guías para organizar las relaciones entre ellos

🔥 Ejemplos:

- Jerarquía de capas
- MVC
- Cliente/Servidor
- Maestro-Esclavo
- Control centralizado y distribuido

🔥 Patrones *de diseño*

- Compuestos de varias unidades arquitecturales más pequeñas
- Describen el esquema básico para estructurar subsistemas y componentes

🔥 Ejemplos:

- Singleton
- Factory
- Facade
- Composición

🔥 Patrones elementales (*idioms*)

- Específicos de un lenguaje de programación
- Describen cómo implementar componentes particulares de un patrón

🔥 Ejemplos:

- Modularidad
- Interfaces mínimas
- Encapsulación
- Objetos
- Acciones y Eventos
- Concurrencia

Patrones vs. Frameworks

Los patrones de diseño tienen descripciones más abstractas que los frameworks. Las descripciones de patrones suelen ser independientes de los detalles de implementación o del lenguaje de programación (salvo ejemplos usados en su descripción) en cambio los frameworks están implementados en un lenguaje de programación, y pueden ser ejecutados y reutilizados directamente.

Los patrones de diseño son elementos arquitecturales más pequeños que los frameworks. Un framework incorpora varios patrones por eso, en algunos casos, se utilizan los patrones para describir el funcionamiento de un framework.

Model View Controller (MVC): En español: Modelo Vista Controlador. Es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Este patrón plantea la separación del problema en tres capas: la capa model, que representa la realidad; la capa controler, que conoce los métodos y atributos del modelo, recibe y realiza lo que el usuario quiere hacer; y la capa vista, que muestra un aspecto del modelo y es utilizada por la capa anterior para interaccionar con el usuario.

Patrones de diseño

Hay diferentes tipos de patrones de diseño, se dividen en:

Patrones Creacionales: tratan con las formas de crear instancias de objetos. El objetivo de estos patrones es de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados.

Patrones Estructurales: describen como las clases y objetos pueden ser combinados para formar grandes estructuras y proporcionar nuevas funcionalidades. Estos objetos adicionales pueden ser incluso objetos simples u objetos compuestos.

Patrones de Comportamiento: nos ayudan a definir la comunicación e iteración entre los objetos de un sistema. El propósito de este patrón es reducir el acoplamiento entre los objetos.

Creacionales	<p>Creacional de la Clase Los patrones creacionales de Clases usan la herencia como un mecanismo para lograr la instanciación de la Clase. Por ejemplo el método Factory.</p> <p>Creacional del objeto Los patrones creacionales de objetos son más escalables y dinámicos comparados de los patrones creacionales de Clases. Por ejemplo la Factory abstract y el patrón Singleton.</p>
Estructurales	<p>Estructural de la Clase Los patrones estructurales de Clases usan la herencia para proporcionar interfaces más útiles combinando la funcionalidad de múltiples Clases. Por ejemplo el patrón Adaptador (Clase).</p> <p>Estructural de Objetos Los patrones estructurales de objetos crean objetos complejos agregando objetos individuales para construir grandes estructuras. La composición del patrón estructural del objeto puede ser cambiado en tiempo de ejecución, el cual nos da flexibilidad adicional sobre los patrones estructurales de Clases. Por ejemplo el Adaptador (Objeto), Facade, Bridge,</p>

	Composite.
Comportamiento	<p>Comportamiento de Clase Los patrones de comportamiento de Clases usan la herencia para distribuir el comportamiento entre Clases. Por ejemplo Interpreter.</p> <p>Comportamiento de Objeto Los patrones de comportamiento de objetos nos permiten analizar los patrones de comunicación entre objetos interconectados, como objetos incluidos en un objeto complejo. Ejemplo Iterator, Observer, Visitor.</p>

Lista de patrones de diseño:

Patrones creacionales

- **Object Pool** : se obtienen objetos nuevos a través de la clonación. Utilizado cuando el costo de crear una clase es mayor que el de clonarla. Especialmente con objetos muy complejos. Se especifica un tipo de objeto a crear y se utiliza una interfaz del prototipo para crear un nuevo objeto por clonación. El proceso de clonación se inicia instanciando un tipo de objeto de la clase que queremos clonar.
- **Abstract Factory** (fábrica abstracta): permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando. El problema a solucionar por este patrón es el de crear diferentes familias de objetos, como por ejemplo la creación de interfaces gráficas de distintos tipos (ventana, menú, botón, etc.).
- **Builder** (constructor virtual): abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.
- **Factory Method** (método de fabricación): centraliza en una clase constructora la creación de objetos de un subtipo de un tipo determinado, ocultando al usuario la diversidad de casos particulares que se pueden prever, para elegir el subtipo que crear. Parte del principio de que las subclases determinan la clase a implementar. A continuación se muestra un ejemplo de este patrón.
- **Prototype** (prototipo): crea nuevos objetos clonándolos de una instancia ya existente.
- **Singleton** (instancia única): garantiza la existencia de una única instancia para una clase y la creación de un mecanismo de acceso global a dicha instancia. Restringe la instanciación de una clase o valor de un tipo a un solo objeto. A continuación se muestra un ejemplo de este patrón.

Patrones estructurales

- **Adapter o Wrapper** (Adaptador o Envoltorio): Adapta una interfaz para que pueda ser utilizada por una clase que de otro modo no podría utilizarla.
- **Bridge** (Puente): Desacopla una abstracción de su implementación.
- **Composite** (Objeto compuesto): Permite tratar objetos compuestos como si se tratase de uno simple.
- **Decorator** (Decorador): Añade funcionalidad a una clase dinámicamente.
- **Facade** (Fachada): Provee una interfaz unificada simple para acceder a una interfaz o grupo de interfaces de un subsistema. A continuación se muestra un ejemplo de este patrón.

- Flyweight (Peso ligero): Reduce la redundancia cuando gran cantidad de objetos poseen idéntica información.
- Proxy: Mantiene un representante de un objeto.
- Módulo: Agrupa varios elementos relacionados, como clases, singletons, y métodos, utilizados globalmente, en una entidad única.

Patrones de comportamiento

- Chain of Responsibility (Cadena de responsabilidad): Permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.
- Command (Orden): Encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma.
- Iterator (Iterador): Permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos.
- Mediator (Mediador): Define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto.
- Memento (Recuerdo): Permite volver a estados anteriores del sistema.
- Observer (Observador): Define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.
- State (Estado): Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. A continuación se muestra un ejemplo de este patrón.
- Strategy (Estrategia): Permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución.
- Template Method (Método plantilla): Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
- Visitor (Visitante): Permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.

A continuación vamos a presentar algunos ejemplos de patrones de diseño que ya conocen. A cada diseño de proyecto le sigue el problema que trata de resolver, la solución que aporta y las posibles desventajas asociadas. Un desarrollador debe buscar un equilibrio entre las ventajas y las desventajas a la hora de decidir que patrón utilizar. Lo normal es, como observará a lo largo de la materia, buscar el balance entre flexibilidad y rendimiento.

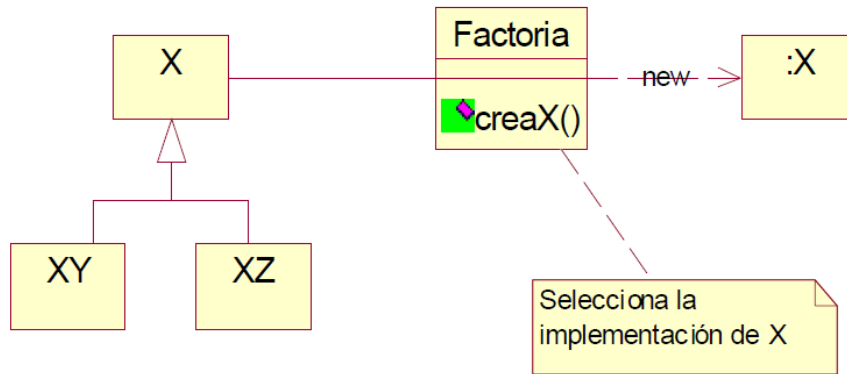
Factory

Propósito: Crear distintos objetos de la misma naturaleza en tiempo de ejecución.

Motivación:

- La fábrica es responsable de crear el objeto que necesitamos en tiempo de ejecución.
- En principio la fabrica no conoce las clases concretas que se van a definir en una aplicación.
- El método factoría es un método abstracto que encapsula el conocimiento de qué subclase se va a crear y pone este conocimiento fuera del almacén.

Diagrama UML:



Aplicación

- Cuando una clase no puede anticipar el tipo de objetos que debe crear
- Cuando una clase quiere que sus clases especifiquen los objetos que deben crear
- Cuando una clase delega la responsabilidad a una subclase de ayuda (entre varias), y se quiere localizar el conocimiento de qué subclase de ayuda es la delegada

Ejemplo JAVA:

Supongamos que tengo un sistema de control de stock, el cual se conecta a una base de datos para guardar la cantidad de stock que tengo de cada producto. Utilizo java para que sea un sistema multiplataforma pero me encuentro con un problema... Necesito que soporte múltiples bases de datos.

//creo una clase abstracata para poder realizar diferentes implementaciones por cada bd que quiera implantar

```
public abstract class AbstractConexion {
    //Creo un metodo abstrcato para sobrescribirlo en cada implementación
    public abstract String descripcion();
}
```

//Creo una implementación de mi conexión que se llama MySql

```
public class MySql extends AbstractConexion{
```

```
    //Sobrescribo mi nombre y digo a que bd me conecte
    @Override
    public String descripcion() {
        return "Conexion a MySql";
    }
}
```

//Creo una implementación de mi conexión que se llama Oracle

```
public class Oracle extends AbstractConexion{
```

```
    //Sobrescribo mi nombre y digo a que bd me conecte
    @Override
    public String descripcion() {
        return "Conexion a Oracle";
    }
}
```

//Creo una clase Factory la cual me va a proveer la implantación que necesito dependiendo de un valor

```
public class Factory {
```

```
    //creo un método abstracto para requerir una instancia.
```

```

public static AbstractConexion obtenerConexion(String tipo){
    //dependiendo del valor de entrada retorno la implementación deseada
    if(tipo.equals("MySQL")){
        return new MySQL();
    }else if(tipo.equals("Oracle")){
        return new Oracle();
    }else{
        return null;
    }
}

//En cualquier parte de mi programa puedo conectarme a MySQL o Oracle
public static void main(String[] args) {

    AbstractConexion conexion = Factory.obtenerConexion("Oracle");
    System.out.println(conexion.descripcion());

}

```

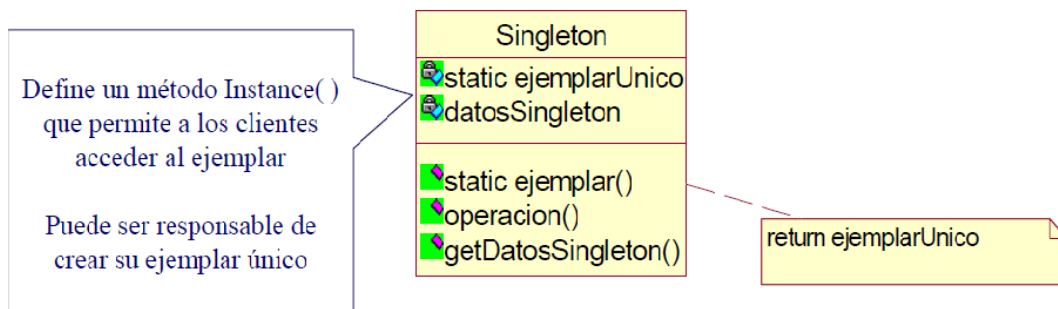
Singleton

Propósito: Asegurar que una clase sólo tiene un ejemplar, y proporcionar un punto de acceso global a éste

Motivación

- Algunas clases sólo necesitan exactamente un ejemplar
- Un spooler de impresión en un sistema, aunque haya varias impresoras
- Un sólo sistema de archivos
- Un sólo gestor de ventanas
- En vez de tener una variable global para acceder a ese ejemplar único, la clase se encarga de proporcionar un método de acceso

Diagrama UML:



Aplicación

- Cuando sólo puede haber un ejemplar de una clase, y debe ser accesible a los clientes desde un punto de acceso bien conocido
- Cuando el único ejemplar pudiera ser extensible por herencia, y los clientes deberían usar el ejemplar de una subclase sin modificar su código

Ejemplo JAVA:

Tengo un sistema al cual acceden muchas personas y no quiero que cada persona que ingrese cree un objeto ya que ocupa espacio en memoria innecesario.

```
//Creo mi clase
public class UnicaService {
    //Creo una variable de clase para almacenar mi instancia
    private static UnicaService instancia;
    //Creo constructores privados para que no puedan instanciar mi clase por fuera.
    private UnicaService(){

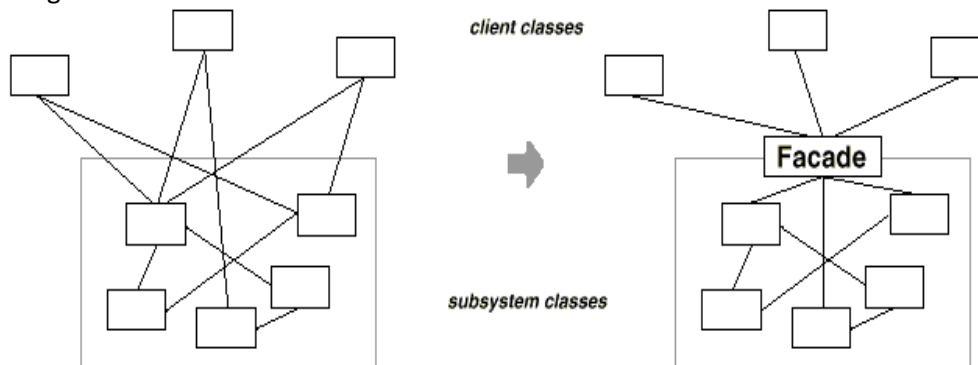
    }
    //Creo un método estático el cual retorna una instancia de la clase.
    public static UnicaService getInstance(){
        //Si mi instancia es nula la creo
        if(instancia==null){
            instancia = new UnicaService();
        }
        //retorno mi única instancia.
        return instancia;
    }
}
```

Facade

Propósito: Simplifica el acceso a un conjunto de objetos proporcionando uno que todos los clientes pueden usar para comunicarse con el conjunto

Motivación: Minimizar las comunicaciones y dependencias entre subsistemas

Diagrama UML:



Aplicación:

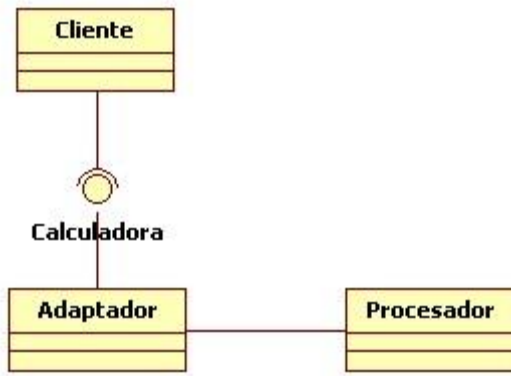
- Para proporcionar una interfaz sencilla a un subsistema complejo
- Cuando hay muchas dependencias entre los clientes y las clases de implementación de una abstracción. Esto mejora la independencia de subsistemas y la portabilidad
- Para estructurar un sistema en capas

Adapter

Propósito: Convierte la interfaz de una clase en otra interfaz que el cliente espera. *Adapter* permite a las clases trabajar juntas, lo que de otra manera no podría hacerlo debido a sus interfaces incompatibles.

Motivación: La motivación del patron es servir de interfaz entre dos clases no compatibles.

Diagrama UML:



Aplicación:

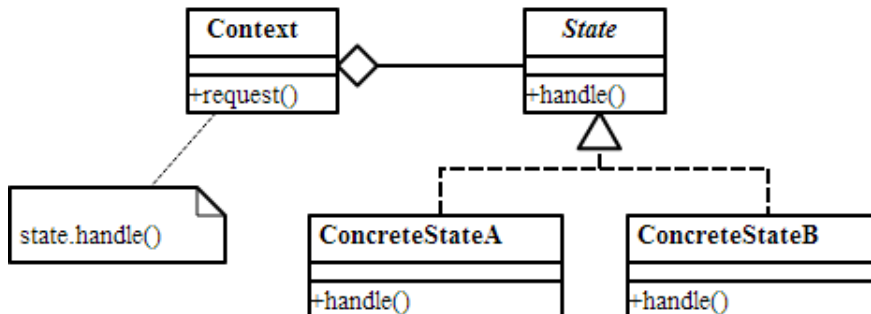
- Se desea usar una clase existente, y su interfaz no se iguala con la necesitada.
- Cuando se desea crear una clase reusable que coopera con clases no relacionadas, es decir, las clases no tienen necesariamente interfaces compatibles.

State

Propósito: Permite a un objeto alterar su comportamiento según el estado interno en que se encuentre.

Motivación: El patrón State está motivado por aquellos objetos en que, según su estado actual, varía su comportamiento ante los diferentes mensajes.

Diagrama UML:



Ejemplo JAVA:

Tengo un sistema para registrar empleados, cada empleado puede pasar del estado “Alta” al estado “Baja” y del estado “Baja” a “Alta” nuevamente. Y quiero diseñar un modelo el cual siga funcionando si agrego nuevos estados.

//Creo una interfaz la cual sea implementada por cualquier estado.

```
public interface IEstado {
```

//Creo un método que va a tener diferentes implementaciones dependiendo el estado

```
public void cambiarEstado(Empleado e);
```

```
}
```

//Creo el estado Alta implementado la interfaz

```
public class Alta implements IEstado{
```

//Sobrescribo el método cambiar de estado y como estoy en “alta” cambio el

//estado del empleado a “baja”

```
@Override
```

```

    public void cambiarEstado(Empleado e) {
        e.setEstado(new Baja());
        System.out.println("Paso a Baja");
    }
}

//Creo el estado Baja implementado la interfaz
public class Baja implements IEstado{
    //Sobrescribo el método cambiar de estado y como estoy en "baja" cambio el
    //estado del empleado a "alta"
    @Override
    public void cambiarEstado(Empleado e) {
        e.setEstado(new Baja());
        System.out.println("Paso a Alta");
    }
}

//La clase context se va a utilizar para cambiar el estado.
public class Context {
    private IEstado estado;
    //En el constructor se recibe el estado en el cual se encuentra
    public Context(IEstado estado) {
        this.estado = estado;
    }
    //se invoca cuando uno quiere cambiar el estado al que corresponda
    public void ejecutarAccion(Empleado emp){
        this.estado.cambiarEstado(emp);
    }
}

//La clase empleado ademas de sus atributos comunes tiene un atributo del tipo IEstado
public class Empleado {
    private String nombre;
    private Integer dni;
    private IEstado estado;

    public IEstado getEstado() {
        return estado;
    }
    public void setEstado(IEstado estado) {
        this.estado = estado;
    }

    public Integer getDni() {
        return dni;
    }

    public void setDni(Integer dni) {
        this.dni = dni;
    }

    public String getNombre() {

```

```

        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
}

```

Cuando (no) utilizar patrones de diseño

La primera regla de los patrones de diseño coincide con la primera regla de la optimización: retrasar. Del mismo modo que no es aconsejable optimizar prematuramente, no se deben utilizar patrones de diseño antes de tiempo. Seguramente sea mejor implementar algo primero y asegurarse de que funciona, para luego utilizar el patrón de diseño para mejorar las flaquezas; esto es cierto, sobre todo, cuando aún no ha identificado todos los detalles del proyecto (si comprende totalmente el dominio y el problema, tal vez sea razonable utilizar patrones desde el principio, de igual modo que tiene sentido utilizar los algoritmos más eficientes desde el comienzo en algunas aplicaciones).

Los patrones de diseño pueden incrementar o disminuir la capacidad de comprensión de un diseño o de una implementación, disminuirla al añadir accesos indirectos o aumentar la cantidad de código, disminuirla al regular la modularidad, separar mejor los conceptos y simplificar la descripción. Una vez que aprenda el vocabulario de los patrones de diseño le será más fácil y más rápido comunicarse con otros individuos que también lo conozcan. Por ejemplo, es más fácil decir “ésta es una instancia del patrón Visitor” que “éste es un código que atraviesa una estructura y realiza llamadas de retorno, en tanto que algunos métodos deben estar presentes y son llamados de este modo y en este orden”.

La mayoría de las personas utiliza patrones de diseño cuando perciben un problema en su proyecto —algo que debería resultar sencillo no lo es— o su implementación— como por ejemplo, el rendimiento. Examine un código o un proyecto de esa naturaleza. ¿Cuáles son sus problemas, cuáles son sus compromisos? ¿Qué le gustaría realizar que, en la actualidad, es muy difícil lograr?

La referencia más utilizada en el tema de los patrones de diseño es el llamado libro de la “banda de los cuatro”, Design Patterns: Elements of Reusable Object-Oriented Software por Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, Addison-Wesley, 1995. Los patrones de diseño son muy populares en la actualidad, por lo que no dejan de aparecer nuevos libros.

Antipatrón

“Es una forma literaria que describe una solución comúnmente dada a un problema que genera consecuencias negativas decididamente” [Browm et al 1998]

Clasificación:

- Antipatrones de desarrollo de software
- Antipatrones de arquitectura de software
- Antipatrones de gestión de proyectos software

Antipatrones de desarrollo de software:

- The Blob (“clases gigantes”)
- Lava Flow (“código muerto”)
- Funcional Decomposition (“Diseño no orientado a objetos”)
- Poltergeists (“No se sabe bien lo que hacen algunas clases”)

- Golden hammer (“Para un martillo todo son clavos”)
- Spaghetti code
- Cut-and-paste programming

Antipatrones de arquitectura de software:

- Stovepipe enterprise (“Aislamiento en la empresa”)
- Stovepipe system (“Aislamiento entre sistemas”)
- Vendor Lock-In (“Arquitectura dependiente de producto”)
- Architecture by implication
- Design by committee (“navaja suiza”)
- Reinvent the Wheel

Antipatrones de gestión de proyectos software:

- Analysis paralysis
- Death by planning
- Corncob (“personas problemáticas”)
- Irrational management
- Project mismanagement

Conexión de base de datos en JAVA

Para empezar debemos de tener presente que cuando se intenta conectar Java, con cualquier motor de base de datos necesitamos conocer si la relación de Conexión de Java con el motor de base de datos es de tipo directa o indirecta.

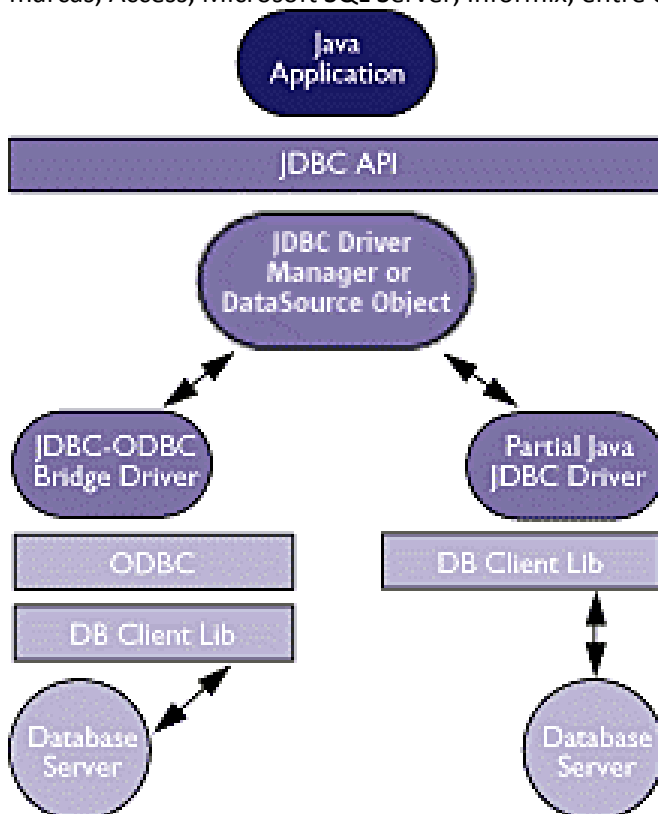
Antes de esto debemos definir que es un controlador JDBC (Java Database Connectivity), el cual es una interfase de comunicación que permite la ejecución entre Java y cualquier motor de Base de Datos.

Conexión Directa

El controlador JDBC accede directamente al controlador del fabricante (DB Client Lib); este tipo de controladores JDBC se denominan de nivel 3 ó 4. Entre los manejadores de base de datos que poseen una conexión directa con Java, tenemos a: My Sql, Sybase DB2, Oracle. Ya que estas no necesitan un puente para comunicarse, el trabajo y la conexión son mucho más rápidos que una conexión indirecta.

Conexión Indirecta

El controlador JDBC hace de "puente" con el controlador ODBC, que es el que accede a la base de datos, este es un esquema de un controlador JDBC de nivel tipo 1. Entre los manejadores de base de datos que necesitan de un puente ODBC para conectarse con Java, tenemos a las marcas, Access, Microsoft SQL Server, Informix, entre otros.



La URL de JDBC

La noción de la URL en JDBC es muy similar al modo en que las URL se utilizan en otras situaciones. Para poder entender la base lógica de las URL de JDBC, consideremos una aplicación que utiliza diversas bases de datos; a cada base de datos se accede mediante diferentes driver, dependiendo del fabricante de base de datos.

Las URL de JDBC proporcionan un modo de identificar un driver de base de datos, en el caso de una conexión directa. La URL de JDBC representa un driver y la información adicional específica del driver para localizar una base de datos y conectarla a él. La sintaxis de la URL de JDBC es como sigue:

`jdbc:<subprotocolo>:<subname>`

Se puede observar que están separadas en tres partes por dos puntos.

- Protocolo: jdbc es el protocolo. Este es el único protocolo permitido en JDBC.
- Sub-protocolo: el sub-protocolo es utilizado para identificar un driver de base de datos o el nombre de un mecanismo de conectividad de una base de datos, elegido por los proveedores del driver de base de datos.
- Subnombre: la sintaxis del subnombre es específica de driver. Un driver puede elegir cualquier sintaxis apropiada para su implementación.

Por ejemplo en una conexión directa con DB2, y una base de datos de nombre libros, sería:

`jdbc:db2:libros`

y para una conexión indirecta con Microsoft SQL Server utilizando un puente de datos JDBC-ODBC de nombre libros, nuestro URL sería:

`jdbc:odbc:libros`

Driver Manager

El propósito de la clase `java.sql.DriverManager` (gestor de driver) en JDBC es proporcionar una capa de acceso común encima de diferentes drivers de base de datos utilizados en una aplicación. En este enfoque las aplicaciones utilizan la clase `DriverManager` para obtener conexiones, a través de su argumento URL.

```
try{
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection ("jdbc:odbc:<ruta_a_la_BD>", "usuario",
"password");
}catch(Exception e){
}
```

Debe tenerse en cuenta que el método estático `forName()` definido por la clase `Class` genera un objeto de la clase especificada. Cualquier controlador JDBC tiene que incluir una parte de iniciación estática que se ejecuta cuando se carga la clase. En cuanto el cargador de clases carga dicha clase, se ejecuta la iniciación estática, que pasa a registrarse como un controlador JDBC en el `DriverManager`.

El bloque `try – catch` es necesario para establecer la conexión, esto para saber de algún problema existente cuando se ejecute la conexión con el driver dentro del bloque `try`, el bloque `catch`, tiene como función capturar el tipo de error generado, al no poder conectarse con la base de datos.

Mediante el método `DriverManager.getConnection`, abrimos una sesión o conexión con la base de datos especificada mediante los parámetros URL, user, pass y con el driver JDBC que tenga registrados el sistema

En muchas ocasiones es necesario especificar el tiempo máximo que debe esperar un driver mientras intenta conectar a una base de datos. Los siguientes dos métodos pueden ser utilizados para fijar/obtener (`set/get`) el tiempo límite de registro:

```
public static void setLoginTimeout(int segundos){ }
public static void getLoginTimeout( ){}
```

Ahora si deseamos conectarnos con MySQL, el proceso es similar que el anterior hecho para Ms SQL Server, solo debemos de cambiar el contenido de Driver, URL, user y pass tal como se muestra:

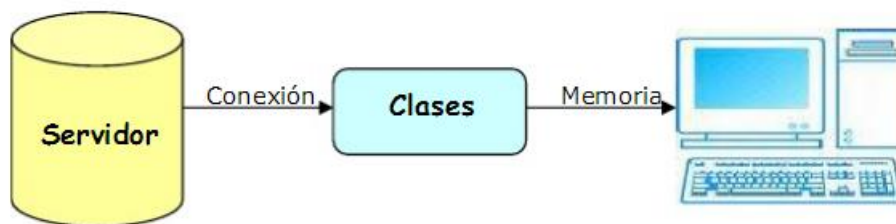
```
String Driver = "com.mysql.jdbc.Driver";
String URL = "jdbc:mysql://localhost:3306/libros";
String user = "root";
String pass = "root"; //la contraseña puede variar según el usuario.
```

Esta parte observemos, la diferencia que hay con los parámetros antes puestos para establecer la conexión con Ms SQL Server. Como la conexión con MySQL con Java es directa, observamos que en la parte de URL, no hay ningún puente ODBC. Debemos de recordar que para cada manejador de base de datos de conexión directa existen drivers distintos, el cual debemos de instalar antes. Pero para las últimas versiones del NetBeans y otros IDE's para java, los driver más usados están contenidos dentro de la instalación del Netbeans.

JDBC utiliza los mismos métodos y clases independientemente del driver usado para conectar al proveedor de base de datos, lo que cambia es el nombre del driver por lo que es bastante sencillo modificar aplicaciones al cambiar de proveedor.

Crear y ejecutar sentencias SQL

Podemos utilizar un objeto Connection para ejecutar instrucciones SQL creando un Statement, un PreparedStatement o un CallableStatement. Estos objetos abstraen instrucciones SQL normales, instrucciones preparadas y procedimientos almacenados respectivamente. Una vez hemos obtenidos unos de esos objetos de instrucciones, podemos ejecutar la instrucción y leer resultados gracias a un objeto ResultSet.



Pasos para la ejecutar una sentencia SQL con Java:

1. getConnection(): primero obtenemos la conexión.
 2. PreparedStatement: prepara un comando SQL
 3. executeQuery(): ejecuta la instrucción SQL que se preparó en el paso anterior.
 4. ResultSet: lleva a memoria los datos de una consulta SQL.
 5. next(): lee fila por fila los datos que están en memoria.
 6. Close(): cerramos la conexión
-
1. Lo primero que debemos realizar es conectarnos a la base de datos y obtener un objeto del tipo Connection.
 2. Como vimos anteriormente hay diferentes maneras de preparar consultas SQL, nosotros vamos a utilizar la interface PreparedStatement, la cual almacena la consulta a la hora de generar el objeto lo que lo hace más segura:

```
PreparedStatement ps = con.prepareStatement(QUERY);
```

A través del objeto del tipo Connection invocamos al método `prepareStatement` el cual recibe por parámetro un String con la query que debe almacenar.

Si necesitamos pasarle parámetros a la query, ya que por ejemplo no queremos concatenarlos o estamos ejecutando un proceso almacenado el cual solicita parámetros, debemos poner un signo de preguntas en el lugar donde iría el valor (?).

Después de almacenar la consulta debemos pasar los parámetros al `PreparedStatement` mediante el método `setObject`, `setInt`, `setString`, etc. el cual solicita un entero (posición del parámetro) y un objeto (valor que queremos pasar a la consulta).

3. Luego de almacenar la query y settear los parámetros (si la consulta los requería), lo que debemos hacer es ejecutarla.

```
ResultSet rs = ps.executeQuery();
```

```
ps.execute();
```

Dependiendo de la consulta que queremos ejecutar debemos llamar al método `executeQuery` o `execute` (select o update, insert o delete respectivamente).

4. Si la consulta realizada fue un select debemos obtener el resultado a través de la interface `ResultSet`, la cual representa un conjunto de datos resultado de una consulta SQL, para acceder a los registros se emplea un cursor que inicialmente apunta antes del primer registro.
5. Para avanzar por los registros del `ResultSet` se emplea el método `next()`. `ResultSet` es de sólo lectura.
6. Una vez finalizado las operaciones en la base de datos es importante cerrar la conexión empleando el metodo `close()` sobre el objeto del tipo `Connection` empleado para generar las consultas. Ya que esta operación libera los recursos empleados sobre el motor de base de datos.

JDBC vs ODBC

JDBC difiere de ODBC (conectividad abierta a bases de datos), API de Microsoft muy utilizada para acceder a BD de diferentes proveedores desde múltiples plataformas:

- ODBC está escrito en C
- el gestor de drivers ODBC y el propio driver han de ser instalados en todos los ordenadores
- JDBC está escrito en Java y permite obtener aplicaciones completas en Java (100% java)
- usando ODBC en base de datos grandes, el rendimiento puede disminuir al convertir llamadas Java a C y viceversa

JDBC puede usar varios drivers, entre ellos los más importantes son:

1 _ Microsoft SQL Server (se descarga desde la web de Microsoft, es un controlador JDBC de SQL Server 2005 y de SQL Server 2000; se trata de un controlador JDBC de Tipo 4 que

proporciona conectividad de base de datos a través de las API JDBC estándar disponibles en J2EE - Java2 Enterprise Edition:

Class.forName: com.microsoft.jdbc.sqlserver.SQLServerDriver

Connection:

jdbc:microsoft:sqlserver:<HOST>:<puerto>;DatabaseName=[nombre_BD];User=[usuario];Password=[password]

2_ JDBC-ODBC Bridge (puente con ODBC, sirve para conectar con BD de Microsoft Access):

Class.forName: sun.jdbc.odbc.JdbcOdbcDriver

Connection: jdbc:odbc:<ruta_a_la_BD>

La conexión con la base de datos de Microsoft Access puede realizarse de 2 maneras:

1. configurando la base de datos como DSN de Sistema en el Panel de control ODBC

/* cadena de conexion: AccessBD es el nombre con el que se ha configurado

* la conexion a la base de datos en el panel de control ODBC */

String url = "jdbc:odbc:AccessBD";

2. usando la ruta física a la base de datos en la cadena de conexión (no hace falta configurarla en el Panel de control ODBC):

//cadena de conexion con la ruta fisica a la base de datos

String db = "D:\\Carpeta\\Subcarpeta\\Subcarpeta\\base_de_datos.mdb";

//si la base de datos esta en la carpeta de la aplicacion Java se escribe

String db = "base_de_datos.mdb";

String url = "jdbc:odbc:MS Access Database;DBQ=" + db;

3_ MySQL (driver MM.MySQL):

Class.forName: org.gjt.mm.mysql.Driver

Connection: jdbc:mysql://<HOST>:<puerto>/<BD>

Carga de clases Java

Sabemos que cuando ejecutamos nuestra aplicación la JRE se encarga de cargar dinámicamente las clases Java en la Java Virtual Machine.

Todas las clases de una aplicación Java son cargadas mediante una subclase de java.lang.ClassLoader. Los ClassLoader en Java están organizados dentro de una jerarquía, por esto para realizar la carga de una clase el ClassLoader debe:

1. Chequea que la clase no este cargada.
2. Si no esta cargada, pregunta al classloader padre si la puede cargar.
3. Si el padre no puede cargar la clase, intenta cargarla en memoria.

Reflexión Informática

En informática, reflexión (o reflexión computacional) es la capacidad que tiene un programa para observar y opcionalmente modificar su estructura de alto nivel.

La reflexión es dinámica o en tiempo de ejecución, aunque algunos lenguajes de programación permiten reflexión estática o en tiempo de compilación. Es más común en lenguajes de programación de alto nivel ejecutándose sobre una máquina virtual, como Smalltalk o Java, y menos común en lenguajes como C.

En un sentido más amplio, la reflexión es una actividad computacional que razona sobre su propia computación.

Cuando el código fuente de un programa se compila, normalmente se pierde la información sobre la estructura del programa conforme se genera el código de bajo nivel (normalmente lenguaje ensamblador). Si un sistema permite reflexión, se preserva la estructura como metadatos en el código generado. Dependiendo de la implementación, el código con reflexión tiende a ser más lento que el que no lo tiene.

En los lenguajes que no distinguen entre tiempo de ejecución y tiempo de compilación, no hay diferencia entre compilación o interpretación de código y reflexión.

Un lenguaje con reflexión proporciona un conjunto de características disponibles en tiempo de ejecución que, de otro modo, serían muy difícilmente realizables en un lenguaje de más bajo nivel. Algunas de estas características son las habilidades para:

- Descubrir y modificar construcciones de código fuente (tales como bloques de código, clases, métodos, protocolos, etc.) como objetos de "categoría superior" en tiempo de ejecución.
- Convertir una cadena que corresponde al nombre simbólico de una clase o función en una referencia o invocación a esa clase o función.
- Evaluar una cadena como si fuera una sentencia de código fuente en tiempo de ejecución.

Reflexión en JAVA

En JAVA tenemos disponible una API la cual nos permite examinar el código JAVA, "reflexionar" sobre los componentes en tiempo de ejecución y para usar miembros reflexionados.

La Reflexión se usa para instanciar clases e invocar métodos usando sus nombres, un concepto que permite la programación dinámica. Clases, interfaces, métodos, campos, y constructores pueden ser todos descubiertos y usados en tiempo de ejecución.

En la API Java de Reflexión tenemos dos clases muy importantes: Field y Method ("Class" no es parte de esta API, pero es de vital importancia igual).

Estas por lo menos son de las que más se suelen utilizar para interactuar con objetos a través de reflexión, aunque si nos ponemos a ver, muchas de las cosas que podemos obtener de un objeto del tipo Class son muy interesantes.

A través de Class podemos acceder a los siguientes métodos:

```

● getAnnotation(Class<A> annotationClass) : A - Class
● getAnnotations() : Annotation[] - Class
● getCanonicalName() : String - Class
● getClass() : Class<?> - Object
● getClasses() : Class<?>[] - Class
● getClassLoader() : ClassLoader - Class
● getComponentType() : Class<?> - Class
● getConstructor(Class<?>... parameterTypes) : Constructor<?> ext
● getConstructors() : Constructor<?>[] - Class
● getDeclaredAnnotations() : Annotation[] - Class
● getDeclaredClasses() : Class<?>[] - Class
● getDeclaredConstructor(Class<?>... parameterTypes) : Construc
● getDeclaredConstructors() : Constructor<?>[] - Class
● getDeclaredField(String name) : Field - Class
● getDeclaredFields() : Field[] - Class
● getDeclaredMethod(String name, Class<?>... parameterTypes)
● getDeclaredMethods() : Method[] - Class
● getDeclaringClass() : Class<?> - Class
● getEnclosingClass() : Class<?> - Class
● getEnclosingConstructor() : Constructor<?> - Class
● getEnclosingMethod() : Method - Class
● getEnumConstants() : LoteBean[] - Class
● getField(String name) : Field - Class
● getFields() : Field[] - Class
● getGenericInterfaces() : Type[] - Class
● getGenericSuperclass() : Type - Class
● getInterfaces() : Class<?>[] - Class
● getMethod(String name, Class<?>... parameterTypes) : Method
● getMethods() : Method[] - Class
● getModifiers() : int - Class

```

A continuación voy a explicar los métodos más importantes o los que más use yo y van a necesitar a lo largo de la cursada:

- ***getDeclaredMethods*** retorna un Array del tipo Method con todos los métodos declarados en la clase.
- ***getDeclaredFields*** retorna un Array del tipo Field con todos los atributos de esa Clase, esta opción incluye los atributos public, protected, default (package), private, pero excluye los campos heredados.
- ***getDeclaredMethod*** retorna un objeto del tipo Method, se le debe pasar el nombre del método y un Array de Class con los parámetros que recibe.
- ***getDeclaredField*** retorna un objeto del tipo Field, se debe pasara el nombre del atributo.
- ***getConstructors*** retorna un Array del tipo Constructor con todos los constructores declarados en la clase.
- ***newInstance*** crea una nueva instancia de la clase, invoca al constructor por defecto. Si se quiere invocar a otro constructor se debe ejecutar este método sobre un objeto del tipo Constructor.
- ***getDeclaredAnnotations*** retorna un Array del tipo Annotation con todas las anotaciones que posea el elemento reflexionado.
- ***getAnnotation*** retorna un objeto del tipo Annotation, se debe pasara el tipo de Annotation que queremos obtener.

Los últimos dos métodos ya los veremos más adelante para que nos sirven, por lo portento con los métodos de Class mencionados nos alcanzan para poder explicar cómo funciona esta API.

Para entender un poco como utilizar estos métodos voy a realzar unos ejemplos a que van a consistir en:

- Imprimir el nombre de los atributos de una Clase
- Leer el valor de los atributos utilizando los métodos Getters.
- Creando una instancia de un Objeto (combinado con lectura de atributo para caso que retorna NULL)
- Escribiendo datos en un objeto utilizando los métodos Setters.
- Como se pueden leer atributos heredados de una Clase Padre.
- Tomemos como ejemplo la siguiente Clase sobre la que se aplicaran los siguientes ejemplos:

```
public class Entidad {

    private int id;
    private String nombre;
    private String otroAtributo;
    private OtraClase otra;

    public Entidad(){}

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getOtroAtributo() {
        return otroAtributo;
    }
    public void setOtroAtributo(String otroAtributo) {
        this.otroAtributo = otroAtributo;
    }
    public OtraClase getOtra() {
        return otra;
    }
    public void setOtra(OtraClase otra) {
        this.otra = otra;
    }
}
```

Como se puede ver es una simple Clase con 4 Atributos (uno de ellos una referencia a otra Clase) y los respectivos metodos Getters y Setters para sus atributos.

Ahora queremos hacer un Ciclo cuyo fragmento de código puede ser parte del Programa que contiene a esta Clase o ajeno a esta mediante el cual podamos recorrer los atributos de la Clase e ir imprimiéndolos por consola.

Esta es una de las características fundamentales de Reflexión, justamente el hecho de no tener la obligación de "conocer" a la Clase o el Objeto...

Los ejemplos del tipo "como setear un valor a un atributo" uno podría preguntarse: "para que quiero usar Reflexión cuando puedo hacer un [objeto.setAtributo(valor)]?"... Es cierto... pero supongamos que estamos haciendo una *Librería* que debe ser capaz de trabajar con todo tipo de objetos, tanto los creados por nosotros, como otras entidades de software que hagan uso de la misma funcionalidad, en ese caso es probable que gran cantidad de los objetos que interactúen con nuestra *Librería* sean ajenos a nuestro conocimiento y podamos necesitar de la ayuda de Reflexión para este trabajo. Cuando entienda estas características y los usos que se le puede dar a esta API van a querer desarrollar todo a través de reflexión.

1. Imprimir el Nombre de los Atributos de una Clase:

```
import java.lang.reflect.Field;
public class App{

    public static void main( String[] args ) {
        //Obtener Array con los Fields(Campos/Atributos)
        //de la Clase
        Field[] fields = Entidad.class.getDeclaredFields();

        //Recorrer cada uno de los Campos en el Array
        //e imprimir su Nombre
        for(Field f : fields){
            System.out.println(f.getName());
        }
    }
}
```

2. Supongamos ahora que queremos leer el Valor de estos atributos utilizando el método "Get" de cada uno... y no sabemos cual es el nombre de estos métodos, pero asumimos que siguen la convención de: "getNombre" donde cada palabra después de "get" se escribe con la primera letra en mayúscula.

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class App {

    public static void main( String[] args ) {

        //Obtener Array con los Fields(Campos/Atributos)
        //de la Clase
        OtraClase otra = new OtraClase();
        Entidad entidad = new Entidad(5, "Diego", "Gato", otra);
        Field[] fields = entidad.getClass().getDeclaredFields();

        //Recorrer cada uno de los Campos en el Array
        //e imprimir el Valor de cada Campo
        for(Field f : fields){
```

```

//Obtener nombre de Atributo
String field = f.getName();

try{
    //Obtener nombre de Método Getter
    Method getter = entidad.getClass().getMethod("get" +
        String.valueOf(field.charAt(0)).toUpperCase() +
        field.substring(1));

    //Llamo al Método especificado de este Objeto
    //con un array con los respectivos Parametros
    //En este caso al ser un Getter no recibe parametros
    Object value = getter.invoke(entidad, new Object[0]);

    System.out.println(value);
}catch(Exception ex){
    ex.printStackTrace();
}
}
}
}

```

Como se puede ver en el código, se crea un Objeto del tipo "Method" a través del nombre del método al que se lo intentara relacionar, y luego sobre ese objeto Method creado se invoca su operación asignada... en este caso obtener el valor del atributo.

3. Ahora supongamos que en lugar de imprimir el Valor de cada Atributo, queremos imprimir el Tipo de dato que es.

Si en el caso de la Clase con la que estamos trabajando (Entidad), no se le setteara un valor para la referencia a la Clase "OtraClase", esta tendria por defecto NULL, y al intentar imprimir tanto su Valor como su Tipo de Dato nos arrojaría una excepción, para este podemos utilizar un metodo de la Clase "Method" que nos permite obtener el Tipo de Dato que devuelve ese Metodo, y llegado el caso de ser necesario para algun fin... en base a ese tipo de dato que nos devuelve, podriamos crear una instancia vacia de esa clase:

```

import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class App {

    public static void main( String[] args ) {
        Entidad entidad = new Entidad(5, "Diego", "Gato", null);
        Field[] fields = entidad.getClass().getDeclaredFields();

        for(Field f : fields){

            try{
                Method getter = entidad.getClass().getMethod("get" +
                    String.valueOf(f.charAt(0)).toUpperCase() +
                    f.substring(1));

                Object value = getter.invoke(entidad, new Object[0]);
                //Si el objeto obtenido es NULL, intenta crear una
                //instancia vacia de ese tipo de objeto.
            }
        }
    }
}

```

```

        if(value == null){
            value = App.emptyInstance(value.getClass().getName());
        }
    }catch(Exception ex){
        ex.printStackTrace();
    }
}

}

public static Object emptyInstance(String type) {
    //Este método crea una instancia de Class con el tipo
    //de dato obtenido por parámetro, y luego itera sobre
    //los Constructores de esta Clase para intentar
    //crear una Instancia en base a un Constructor Vacío
    Object obj = null;
    try {
        Class clazz = Class.forName(type);

        for (java.lang.reflect.Constructor con : clazz.getConstructors()) {
            if (con.getParameterTypes().length == 0) {
                obj = con.newInstance();
                break;
            }
        }
    } catch (Exception e) {
        return null;
    }
    return obj;
}
}

```

4. Traten de deducir el siguiente ejemplo, ya que tiene comentarios y además a esta altura no les va a resultar complicado.

```

import java.lang.reflect.Method;
import java.util.ArrayList;

public class App {
    public static void main( String[] args ) {
        Entidad entidad = new Entidad(5, "Diego", "Gato", null);
        Field[] fields = entidad.getClass().getDeclaredFields();
        String field = fields[1].getName();

        try{
            String setterName = "set" +
                String.valueOf(field.charAt(0)).toUpperCase() +
                field.substring(1);

            //Crea Método Setter en base al Nombre establecido
            //arriba siguiendo la misma convención que para Get
            Method setter = entidad.getClass().
                getMethod(setterName, fields[1].getType());

            //Crea un Array con los Parámetros que se le pasaran

```

```

        //al Método al ser invocado, la función se encarga
        //de los casteos correspondientes.
        ArrayList results = new ArrayList();
        results.add("Otro Nombre!!");
        //se invoca al Metodo del respectivo Objeto
        //y pasando un Array con los Parámetros
        setter.invoke(entidad, new Object[]{results});
    }catch(Exception ex){
        ex.printStackTrace();
    }
}
}

```

5. Ahora para leer Atributos heredados de una Clase es otra historia, ya que al usar el método "getDeclaredFields" para obtener el nombre de los Campos, no nos devuelve aquellos que se estén heredando, pero por cuestiones de herencia, sabemos que están ahí, y sabemos que estarán sus métodos públicos Getters y Setters

```

import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class App {
    public static void main( String[] args ) {
        OtraClase otra = new OtraClase();
        Entidad entidad = new Entidad(5, "Diego", "Gato", null);

        //De esta forma obtenemos un Array con los Campos
        //de la Clase de la que se extiende Directamente, si
        //a su vez quisiéramos los atributos del Padre del Padre
        //tendríamos que repetir este proceso de forma recursiva.
        Field[] fields = entidad.getClass().getSuperclass().getDeclaredFields();

        for(Field f : fields){

            try{
                Method getter = entidad.getClass().getMethod("get" +
                    String.valueOf(field.charAt(0)).toUpperCase() +
                    field.substring(1));

                //Armamos el Método Getter en base al Field obtenido
                //de la Clase Padre, pero lo ejecutamos sobre la Clase
                //Hija, ya que sabemos que heredara este Método, por lo
                //tanto estará disponible aunque no podamos obtener
                //directamente el nombre de su Campo
                Object value = getter.invoke(entidad, new Object[0]);
                System.out.println(value);
            }catch(Exception ex){
                ex.printStackTrace();
            }
        }
    }
}

```

```
}  
}
```

En los ejemplos utilizamos los metodos get y set de los atributos para accede o modificar su valor, pero también tenemos unos métodos de la clase Field que nos ayunda a realizar esto. Los métodos son:

- get: Pasamos por prarametro la instacincia sobre el cual esta el atributo y retorna el valor que posee.
- set: Pasamos por prarametro la instacincia sobre el cual esta el atributo y el valor que queremos que le settee al atributo.

Hay que tener en cuenta que si el atributo es privado nos va a dar una Exception del tipo IllegalAccessException, pero esto lo solucionamos muy fácil, utilizando el método setAccessible y pasándole true. Esto es debido a que el atributo no es accesible directamente, ya que es privado... pero como esta en memorial, le podemos decir que si sea accesible aunque este definido como privado.

Estos son las cosas básicas que se pueden hacer con Reflexión, a continuación vamos a ver Annotations que es el recurso más importante que tiene esta API. Seguramente las conozcan por los Frameworks (Hibernate, Spring, JUnti, etc.). Ahora vamos a ver cómo nos pueden ayudar y que es lo que realmente hacen.

Annotations

Una Annotation Java es una forma de añadir metadatos al código Java que están disponibles para la aplicación en tiempo de ejecución. Muchas veces se usa como una alternativa a la tecnología XML.

Las Annotations Java pueden añadirse a los elementos de programa tales como clases, métodos, constructores, variables locales, y paquetes. Al contrario que las etiquetas de documentación Java, las Anotaciones Java son completamente accesibles al programador mientras que el software se ejecuta usando reflexión.

Cuando se compila el código fuente de Java, el compilador almacena los metadatos de la Annotation en los ficheros/archivos de clases. Posteriormente, la JVM pueden buscar los metadatos para determinar cómo interactuar con los elementos del programa o cambiar su comportamiento.

Crear una Annotation es muy simple y nos permite realizar muchas cosas. Pensar que con cuatro o cinco Annotations, Hibernate puede persistir en cualquier tabla un objeto y no solo eso, recuperarlo a él y a todas sus referencias.

Para ejemplificar el funcionamiento de las Annotations voy a crear una llamada “Test”, la cual la quiero utilizar para que me indique que métodos puedo testear. La declaración sería algo así:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test{
}
```

Las Annotations “Retention” y “Target” indican que la annotation debe estar disponible en tiempo de ejecución y que va a estar disponible para agregarse a nivel de método respectivamente. Luego se indica el nivel de visibilidad deseado, @interface y el nombre que va a llevar.

Retention: Indica por cuanto tiempo la anotación debe ser retenida. Los valores disponibles para la anotación Retention son RUNTIME (no se descarta), CLASS (descartada durante la carga de clases, valor por defecto) y SOURCE (descartada durante la compilación)

Target: Indica el tipo de elemento a la que el tipo de anotación es aplicable. Si no está declarado puede ser utilizada en cualquier elemento del programa. Los valores disponibles para la anotación Target son ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER y TYPE. Dependiendo del valor, vamos a poder incluir nuestra anotación a cualquiera tipo de elemento.

Ya tenemos disponible en todo nuestro sistema la @Test para adicionarle a cualquier método. Esto se puede realizar de la siguiente manera.

```
public class ClasePrueba{
    @Test
    public double sumar(double numeroUno, double numeroDos){
        return numeroUno + numeroDos;
    }
}
```

Así de simple... ya tenemos una primera implementación de nuestra `@Test`, ahora veremos como, utilizando reflexión, podemos hacer que esto tenga sentido.

```
public class App {
    public static void main( String[] args ) {
        Method[] metodos = ClasePrueba.class.getDeclaredMethods();
        for(Method metodo : metodos){
            if(metodo.getAnnotation(Test.class)!=null){
                System.out.println(metodo.getName());
            }
        }
    }
}
```

En el ejemplo anterior, recorro todos los metodos y pregunto cuales son los que tienen declarada una `@Test`. Es algo muy simple pero todavía no hicimos nada.

Nosotros podemos declarar a las Annotations atributos, por ejemplo voy a declararle un atributo que se va a llamar “intentos” y va a ser entero... Lo voy a utilizar para indicar cuantas veces voy a testear los métodos declarados con `@Test` y también voy a declarar un atributo alfanumerico llamado “nombre” que va a tener un valor por defecto “Sin Nombre”.

```
public @interface Test{
    int intentos();
    String nombre() default "Sin Nombre";
}

//ClasePrueba...
@Test(intentos = 3)
public double sumar(double numeroUno, double numeroDos) {
    ...
}

// Método main
Method[] metodos = ClasePrueba.class.getDeclaredMethods();
for(Method metodo : metodos){
    if(metodo.getAnnotation(Test.class)!=null){
        Test miAnotacion = metodo.getAnnotation(Test.class);
        for(int i=0;i<miAnotacion.intentos();i++){
            System.out.println(miAnotacion.nombre());
        }
    }
}
```

Como podrán observar, al método `sumar` solo le asigne un valor al atributo “intentos” ya que es obligatorio porque no tiene un valor por defecto. Creo que con el ejemplo queda bastante claro, que por la consola se verá tres veces las palabras Sin Nombre.

No tengo mucho más para agregar, reflexión es lo más potente que tiene Java para mi pensar, ya que podemos inspeccionar atributos y Annotations en tiempo de ejecución. Y no solo eso, podemos ejecutar métodos públicos o privados sin la necesidad de saber el nombre, crear instancias de objetos invocando cualquier constructor, etc.

Este apunte les da todas las herramientas necesarias para realizar el TP de reflexión.

Hilos en java

Para hablar de hilos en Java primeramente se necesita entender lo que es un hilo. Un hilo es un proceso que se está ejecutando en un momento determinado en nuestro sistema operativo, como cualquier otra tarea, esto se realiza directamente en el procesador. Existen los llamados “demonios” que son los procesos que define el sistema en sí para poder funcionar y otros que llamaremos los hilos definidos por el usuario o por el programador, estos últimos son procesos a los que el programador define un comportamiento e inicia en un momento específico.

En Java, el proceso que siempre se ejecuta es el llamado main que es a partir del cual se inicia prácticamente todo el comportamiento de nuestra aplicación, y en ocasiones a la

existen algunas aplicaciones que requieren más de un proceso (o hilo) ejecutándose al mismo tiempo (multithreading), por ejemplo, se tiene una aplicación de un shopping en la cual se actualizan el precio y el stock varias veces al día a través de la red, se verifican los nuevos descuentos y demás pero que a su vez es la encargada de registrar las compras y todos movimientos que se realice con la mercancía dentro de la tienda, si se decide que dicha aplicación trabajará de la manera simple y con un solo proceso (o hilo), el trabajo de la actualización de precios y stock debe finalizar antes de que alguien pueda hacer algún movimiento con un producto dentro del local, o viceversa, ya que la aplicación no es capaz de mantener el proceso de actualización en segundo plano mientras se registra un movimiento. Si se toma este modelo mono-hilo el tiempo y dinero que se perderá dentro del local será muchísimo mayor comparando con un modelo multi-hilo. En un modelo multi-hilo se pueden realizar todas las actualizaciones en segundo plano mientras se registra una o más ventas o movimientos, cada proceso independiente del otro viviendo o ejecutándose al mismo tiempo dentro de la misma aplicación.

Al hablar de multi-hilo pudiera parecer que necesitamos más de un procesador para realizar dichas tareas pero no es así, el procesador mismo junto con la máquina virtual de Java gestionan el flujo de trabajo y dan la impresión de que se puede ejecutar más de algún proceso al mismo tiempo (aunque en términos estrictos eso no es posible), de cualquier manera no ahondaré en el funcionamiento del procesador, basta con entender que en Java, 2 o más procesos pueden ejecutarse al mismo tiempo dentro de una misma aplicación y para ello son necesarios los Threads o hilos.

Ahora que ya entendemos lo que son los hilos pasaremos a una definición un poco más específica de Java. En Java un hilo o Thread puede ser 2 cosas:

- Una instancia de la clase `java.lang.Thread`
- Un proceso en ejecución

Una instancia de la clase `java.lang.Thread`, no es más que cualquier otro objeto, con variables y métodos predefinidos. Un proceso en ejecución es un proceso individual que realiza una tarea o trabajo, tiene su propia pila de información independiente a la de la aplicación principal.

Es necesario entender que el comportamiento de los hilos o threads varía de acuerdo a la máquina virtual, incluso el concepto más importante a entender con los hilos en Java es que "Cuando se trata de hilos, muy pocas cosas están garantizadas" por ello se debe ser cautelosos al momento de interpretar el comportamiento de un hilo. Pasemos al código.

Crear un hilo (Thread):

Un hilo o proceso en Java comienza con una instancia de la clase `java.lang.Thread`, si analizamos la estructura de dicha clase podremos encontrar bastantes métodos que nos ayudan a controlar el comportamiento de los hilos, desde crear un hilo, iniciarlo, pausar su ejecución, etc. Aquellos métodos que siempre tenemos que tener presentes con respecto a los hilos son:

- `start()`
- `yield()`
- `join()`
- `currentThread()`
- `sleep()`
- `run()`
- `interrupted()`
- `isAlive()`

La acción sucede dentro del método `run()`, digamos que el código que se encuentra dentro de dicho método es el trabajo por hacer, por lo tanto, si queremos realizar diversas operaciones cada una simultánea pero de manera independiente, tendremos varias clases, cada una con su respectivo método `run()`. Dentro del método `run()` puede haber llamados a otros métodos como en cualquier otro método común, pero la pila de ejecución del nuevo proceso siempre comenzará a partir de la llamada al método `run()`.

Para crear un hilo tenemos dos posibilidades:

- Heredar de `Thread` redefiniendo el método `run()`.
- Crear una clase que implemente la interfaz `Runnable` que nos obliga a definir el método `run()`.

En ambos casos debemos definir un método `run()` que será el que contenga el código del hilo. Desde dentro de este método podremos llamar a cualquier otro método de cualquier objeto, pero este método `run()` será el método que se invoque cuando iniciemos la ejecución de un hilo. El hilo terminará su ejecución cuando termine de ejecutarse este método `run()`. Para crear nuestro hilo mediante herencia haremos lo siguiente:

```
public class EjemploHilo extends Thread{
    public void run(){
        // Código del hilo
    }
}
```

Una vez definida la clase de nuestro hilo deberemos instanciarlo y ejecutarlo de la siguiente forma:

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

Al llamar al método `start` del hilo, comenzará ejecutarse su método `run`. Crear un hilo heredando de `Thread` tiene el problema de que al no haber herencia múltiple en Java, si heredamos de `Thread` no podremos heredar de ninguna otra clase, y por lo tanto un hilo no podría heredar de ninguna otra clase.

Este problema desaparece si utilizamos la interfaz `Runnable` para crear el hilo, ya que una clase puede implementar varios interfaces. Definiremos la clase que contenga el hilo como se muestra a continuación:

```
public class EjemploHilo implements Runnable{
    public void run(){
        // Código del hilo
    }
}
```

Para instanciar y ejecutar un hilo de este tipo deberemos hacer lo siguiente:

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

Esto es así debido a que en este caso `EjemploHilo` no deriva de una clase `Thread`, por lo que no se puede considerar un hilo, lo único que estamos haciendo implementando la interfaz es asegurar que vamos a tener definido el método `run()`. Con esto lo que haremos será proporcionar esta clase al constructor de la clase `Thread`, para que el objeto `Thread` que creamos llame al método `run()` de la clase que hemos definido al iniciarse la ejecución del hilo, ya que implementando la interfaz le aseguramos que esta función existe.

Toma en cuenta que puedes sobrecargar el método `run()` sin ningún problema como se muestra a continuación...

```
class MiHilo extends Thread{
    public void run(){
        System.out.println("Trabajo por hacer dentro de MiHilo");
    }
    public void run(String s){
        System.out.println("La cadena ingresada es " + s);
    }
}
```

...sin embargo, al realizar esto, no estarás utilizando tu nuevo método `public void run (String s)` en un proceso separado, es un simple método común y corriente como cualquier otro que tienes que mandar llamar de manera independiente ya que los hilos trabajan con un método `run()` sin argumentos.

Existen diversos tipos de constructores a partir de los cuales se puede crear un hilo, algunos de los más importantes son:

- `Thread()`
- `Thread(objetoRunnable)`
- `Thread(objetoRunnable, String nombre)`
- `Thread(String nombre)`

Ciclo de vida

Un hilo pasará por varios estados durante su ciclo de vida.

```
Thread t = new Thread(this);
```

Una vez se ha instanciado el objeto del hilo, diremos que está en estado de Nuevo hilo (`new Thread`).

```
t.start();
```

Cuando invoquemos su método `start()` el hilo pasará a ser un hilo vivo, comenzándose a ejecutar su método `run()`. Una vez haya salido de este método pasará a ser un hilo muerto (`dead`). Debes de saber que una vez que se ha llamado al método `start()` de un hilo, no puedes volver a realizar otra llamada al mismo método, independientemente de si el hilo ha terminado de realizar lo que está definido dentro de su método `run()` o no, si vuelves a llamar al método `start()` en un hilo que ya lo llamó previamente, obtendrás una excepción `java.lang.IllegalThreadStateException`.

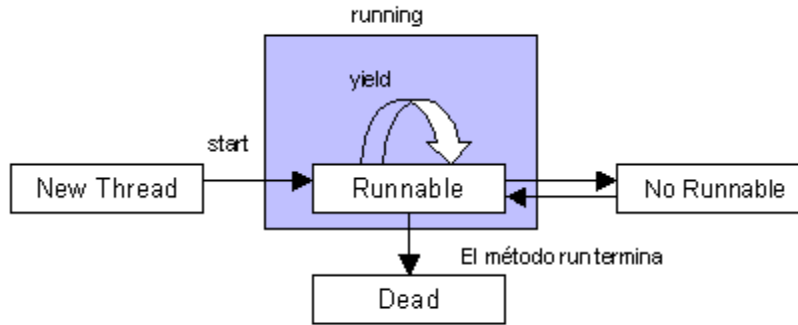
La única forma de parar un hilo es hacer que salga del método `run()` de forma natural. Podremos conseguir esto haciendo que se cumpla una condición de salida de `run()` (lógicamente, la condición que se nos ocurra dependerá del tipo de programa que estemos haciendo). Las funciones para parar, pausar y reanudar hilos están deprecadas en las versiones actuales de Java.

Mientras el hilo esté vivo, podrá encontrarse en dos estados: Ejecutable (`Runnable`) y No ejecutable (`No Runnable`).

El hilo pasará de Ejecutable a No ejecutable en los siguientes casos:

- Cuando se encuentre dormido por haberse llamado al método `sleep()`, permanecerá No ejecutable hasta haber transcurrido el número de milisegundos especificados.

- Cuando se encuentre bloqueado en una llamada al método `wait()` esperando que otro hilo lo desbloquee llamando a `notify()` o `notifyAll()`. Veremos cómo utilizar estos métodos más adelante.
- Cuando se encuentre bloqueado en una petición de E/S, hasta que se complete la operación de E/S.



Lo único que podremos saber es si un hilo se encuentra vivo o no, llamando a su método `isAlive()`.

En la máquina virtual de Java existe algo que llamamos el programador de hilos que es el encargado de decidir qué hilo es el que se va a ejecutar por el procesador en un momento determinado y cuándo es que debe de parar o pausar su ejecución. Asumiendo que se tiene un equipo de un solo procesador, solo un proceso puede estar corriendo en un momento dado, como lo hemos mencionado anteriormente, la idea de multi-proceso o multithreading es meramente aparente. Para que un hilo sea elegible para ser ejecutado, este debe de estar en estado Ejecutable (runnable), si se encuentra en cualquier otro estado no podrá ser elegible por el programador de hilos. Es importante tomar en cuenta que...

"El orden en que un hilo en estado de ejecución es escogido por el programador de hilos no es garantizado".

...aunque no podemos controlar al programador de hilos, en algunas ocasiones podemos influenciarlo mandando a llamar algunos métodos contenidos en la clase `java.lang.Thread`.

Prioridades

Además, una propiedad importante de los hilos será su prioridad. Mientras el hilo se encuentre vivo, el “scheduler” de la máquina virtual Java le asignará o lo sacará de la CPU, coordinando así el uso de la CPU por parte de todos los hilos activos basándose en su prioridad. Se puede forzar la salida de un hilo de la CPU llamando a su método `yield()`. También se sacará un hilo de la CPU cuando un hilo de mayor prioridad se haga Ejecutable, o cuando el tiempo que se le haya asignado expire.

Para cambiar la prioridad de un hilo se utiliza el método `setPriority()`, al que deberemos proporcionar un valor de prioridad entre `MIN_PRIORITY` y `MAX_PRIORITY` (tienes constantes de prioridad disponibles dentro de la clase `Thread`, consulta la API de Java para ver qué valores de constantes hay).

Interrupción

Los objetos de clase Thread cuentan con un método `.interrupt()` que permite al hilo ser interrumpido. En realidad la interrupción simplemente cambia un flag del hilo para marcar que debe ser interrumpido, pero cada hilo debe estar programado para soportar su propia interrupción.

Si el hilo invoca un método que lance la excepción `InterruptedException`, tal como el método `sleep()`, entonces en este punto del código terminaría la ejecución del método `run()` del hilo. Podemos manejar esta circunstancia con:

```
for (int i = 0; i < maxI; i++) {
    // Pausar 4 segundos
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // El hilo ha sido interrumpido. Vamos a salir de run()
        return;
    }
}
```

De esta manera si queda algo que terminar se puede terminar, a pesar de que la ejecución del `sleep()` ha sido interrumpida.

Si nuestro hilo no llama a métodos que lancen `InterruptedException`, entonces debemos ocuparnos de comprobarla periódicamente:

```
for (int i = 0; i < maxI; i++) {
    trabajoCon(i);
    if (Thread.interrupted()) {
        // El flag de interrupción ha sido activado
        return;
    }
}
```

Metodos

Ya vimos mas o menos como funcionan y para que sirven los métodos `start()`, `run()`, `isAlive()`, `sleep()` e `interrupted()`. Ahora vamos a ver algunos métodos más de la clase Thread.

El método `join()`

Si un Thread necesita esperar a que otro termine (por ejemplo el Thread padre espera a que termine el hijo) puede usar el método `join()`. ¿Por qué se llama así? Crear un proceso es como una bifurcación, se abren 2 caminos, que uno espere a otro es lo contrario, una unificación.

A continuación se presenta un ejemplo más complejo: una reunión de alumnos. El siguiente ejemplo usa Threads para activar simultáneamente tres objetos de la misma clase, que comparten los recursos del procesador peleándose para escribir a la pantalla.


```

public static void main(String args[]) throws InterruptedException{
    Thread juan = new Thread (new Alumno("Juan"));
    Thread luis = new Thread (new Alumno("Luis"));
    Thread nora = new Thread (new Alumno("Nora"));
    juan.start();
    juan.join();
    luis.start();
    luis.join();
    nora.start();
    nora.join();
}

```

El metodo join() que llamamos al final hace que el programa principal espere hasta que este Thread este “muerto”(finalize su ejecucion). Este método puede disparar la excepción InterruptedException, por lo que lo hemos tenido en cuenta en el encabezamiento del método.

En nuestro ejemplo, simplemente a cada instancia de Alumno(...) que creamos la hemos ligado a un Thread y puesto a andar. Corren todas en paralelo hasta que mueren de muerte natural, y también el programa principal termina.

```

public class Alumno implements Runnable{
    String mensaje;
    public Alumno(String nombre){
        mensaje = "Hola, soy " + nombre + " y este es mi mensaje numero: ";
    }
    public void run(){
        for (int i=1; i<6;i++){
            try {
                String msj = mensaje + i;
                System.out.println(msj);
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}

```

La salida seria así

```

Hola, soy Juan y este es mi mensaje numero: 1
Hola, soy Juan y este es mi mensaje numero: 2
Hola, soy Juan y este es mi mensaje numero: 3
Hola, soy Juan y este es mi mensaje numero: 4
Hola, soy Juan y este es mi mensaje numero: 5
Hola, soy Luis y este es mi mensaje numero: 1
Hola, soy Luis y este es mi mensaje numero: 2

```

Existe una versión sobrecargada de join() que puede incluir el tiempo en milisegundos, por ejemplo si llamamos a t.join(5000), significa que el hilo que queremos ejecutar debe esperar a que el hilo que se está ejecutando en este momento termine, pero si tarda más de 5 segundos entonces debe dejar de esperar y entrar en estado de Ejecutable(runnable).

El método yield()

El método yield() tiene la función de hacer que un hilo que se está ejecutando vuelva al estado en “Ejecutable” para permitir que otros hilos de la misma prioridad puedan ejecutarse. Sin embargo, el funcionamiento de este método (al igual que de los hilos en general) no está garantizado, puede que después de que se establezca un hilo por medio del método yield() a su estado “Ejecutable”, éste vuelva a ser elegido para ejecutarse. El método yield() nunca

causará que un hilo pase a estado de No ejecutable, simplemente pasa de ejecutándose(running) a “Ejecutable”.

A continuación tomando el ejemplo anterior solo haremos una modificación en el método run, vemos como se implementa dicho metodo:

```
public static void main(String args[]){
    Thread juan = new Thread (new Alumno("Juan"));
    Thread luis = new Thread (new Alumno("Luis"));
    Thread nora = new Thread (new Alumno("Nora"));
    juan.start();
    luis.start();
    nora.start();
}

public void run(){
    for (int i=1; i<6;i++){
        String msj = mensaje + i;
        System.out.println(msj);
        Thread.yield();
    }
}

Hola, soy Juan y este es mi mensaje numero: 1
Hola, soy Nora y este es mi mensaje numero: 1
Hola, soy Juan y este es mi mensaje numero: 2
Hola, soy Luis y este es mi mensaje numero: 1
Hola, soy Nora y este es mi mensaje numero: 2
```

Sincronización de hilos

Muchas veces los hilos deberán trabajar de forma coordinada, por lo que es necesario un mecanismo de sincronización entre ellos.

Un primer mecanismo de sincronización es la variable cerrojo incluida en todo objeto Object, que permitirá evitar que más de un hilo entre en la sección crítica para un objeto determinado. Los métodos declarados como synchronized utilizan el cerrojo del objeto al que pertenecen evitando que más de un hilo entre en ellos al mismo tiempo.

```
public synchronized void metodoSeccionCritica(){
    // Código sección crítica
}
```

Todos los métodos synchronized de un mismo objeto (no clase, sino objeto de esa clase), comparten el mismo cerrojo, y es distinto al cerrojo de otros objetos (de la misma clase, o de otras).

También podemos utilizar cualquier otro objeto para la sincronización dentro de nuestro método de la siguiente forma:

```
synchronized(objetoConCerrojo){
    // Código sección crítica
}
```

de esta forma sincronizaríamos el código que escribiésemos dentro, con el código synchronized del objeto objetoConCerrojo.

Además podemos hacer que un hilo quede bloqueado a la espera de que otro hilo lo desbloquee cuando suceda un determinado evento. Para bloquear un hilo usaremos la función `wait()`, para lo cual el hilo que llama a esta función debe estar en posesión del monitor, cosa que ocurre dentro de un método `synchronized`, por lo que sólo podremos bloquear a un proceso dentro de estos métodos.

Para desbloquear a los hilos que haya bloqueados se utilizará `notifyAll()`, o bien `notify()` para desbloquear sólo uno de ellos aleatoriamente. Para invocar estos métodos ocurrirá lo mismo, el hilo deberá estar en posesión del monitor.

Cuando un hilo queda bloqueado liberará el cerrojo para que otro hilo pueda entrar en la sección crítica del objeto y desbloquearlo.

Algunos puntos clave con la sincronización y el seguro de los objetos son:

- Solo métodos (o bloques) pueden ser sincronizados, nunca una variable o clase.
- Cada objeto tiene solamente un seguro.
- No todos los métodos de una clase deben ser sincronizados, una misma clase puede tener métodos sincronizados y no sincronizados.
- Si una clase tiene ambos tipos de métodos, múltiples hilos pueden acceder a sus métodos no sincronizados, el único código protegido es aquel dentro de un método sincronizado.
- Si un hilo pasa a estado dormido (`sleep`) no libera el o los seguros que pudiera llegar a tener, los mantiene hasta que se completa.
- Se puede sincronizar un bloque de código en lugar de un método.

Un ejemplo de un bloque sincronizado puede ser así...

```
public class Ejemplo{

    public void hacerAlgo(){
        System.out.println("No sincronizado");

        synchronized(this){
            System.out.println("Sincronizado");
        }
    }
}
```

Nota: cuando sincronizas un bloque de código debes de especificar el objeto del cual quieres obtener su seguro, si deseas actuar sobre la misma instancia sobre la que se está trabajando se utiliza la palabra `this`. P. ej.:

```
public synchronized void hacerAlgo()
System.out.println("Sincronizado");
}

es equivalente a:
public void hacerAlgo(){
    synchronized(this){
        System.out.println("Sincronizado");
    }
}
```

```
}
```

Pools de hilos

Una Thread Pool es básicamente un contenedor dentro del cual se crean y ejecutan hilos. Esto nos permite por ejemplo especificar el número máximo de hilos que se pueden ejecutar en un momento dado, lo cual es muy útil si nos interesa obtener resultados de un hilo rápidamente y no que todos coexistan a la vez.

Otra de los beneficios que obtenemos es que el consumo de memoria será mucho menor ya que al solor permitir la ejecución de un número limitado de hilos, los demás no bloquearán recursos.

Java proporciona dos clases para trabajar con Thread Pools. La primera de ellas es Executors, una clase abstracta con métodos estáticos para crear contenedores de capacidad limitada, contenedores con cache, etc.

La segunda clase es ExecutorService, una interface que hereda de Executor y que nos permitirá trabajar de una manera más fácil y eficaz.

```
public static void main(String a[]) {
    ExecutorService exec = Executors.newFixedThreadPool(5);
    for (int i = 0; i < 100; i++) {
        exec.execute(new Runnable() {
            public void run() {

                System.out.println("Running" + Thread.currentThread());

            }
        });
    }
    exec.shutdown();
    try {
        boolean b = exec.awaitTermination(50, TimeUnit.SECONDS);

        System.out.println("All: " + b);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

La parte más importante es Executors.newFixedThreadPool(5). Con esto estamos definiendo un contenedor de un tamaño máximo de 5 espacios. En estos espacios se insertarán hilos y se ejecutarán. Cuando uno de ellos termine su ejecución, se sacará el hilo de ese espacio y se insertará otro de los que este en la cola.

Más abajo, después de crear 100 hilos y ordenar que se ejecuten, invocamos al método shutdown() de la Thread Pool. Con esta llamada indicamos que, una vez se hayan ejecutado todos los hilos, habremos acabado con nuestro trabajo.

La última parte del código, y para mí la más interesante, es “boolean b = exec.awaitTermination(50, TimeUnit.SECONDS);”. El método awaitTermination detiene la ejecución del programa hasta que haya acabado todo el trabajo o hasta que se sobrepase el número de segundos de espera que se pasan como argumento. En el caso de que el trabajo se finalice en el tiempo especificado, el método devolverá true, indicando que todo ha ido bien. Si se sobrepasa el tiempo, devolverá false, lo que nos indicará que posiblemente tengamos algún problema en la ejecución de algún hilo, ya que la ThreadPool no ha terminado su ejecución en el tiempo esperado.

Guía de Ejercicios Java

1. Crear un POJO llamado auto, con los siguientes atributos:

- patente: cadena
- marca: EMarca
- modelo: cadena
- precio: numérico

En el main crear tres objetos y mostrarlos por pantalla. Sobrescriba el método toString, equals y hashCode.

EMarca es un enumerado.

2. Crea las siguientes clases:

- Motor: con métodos para arrancar el motor y apagarlo.
- Rueda: con métodos para inflar la rueda y desinflarla.
- Ventana: con métodos para abrirla y cerrarla.
- Puerta: con una ventana y métodos para abrir la puerta y cerrar la puerta.
- Coche: con un motor, cuatro ruedas y dos puertas;
con los métodos: Arrancar (chequea que las puertas estén cerradas, que las ruedas estén infladas y arranca el motor. Si falla algo, se deberá informar lo ocurrido por consola), Parar (chequea que el coche este andando, apaga el motor, abre las puertas y cierra las ventanas. Si falla algo, se deberá informar lo ocurrido por consola) y control (debe informar el estado de las ruedas).
- Agregue los atributos que crea necesarios

3. Cargar un String por teclado e implementar los siguientes métodos:

- Imprimir la cantidad de caracteres que posee.
- Imprimir la primera mitad de los caracteres de la cadena.
- Imprimir el último caracter.
- Imprimirlo en forma inversa.
- Imprimir cada caracter del String separado con un guión.
- Implementar un método que verifique si la cadena posee la palabra “hola” (indistintamente si posee letras mayúsculas o minúsculas)

4. Cree un algoritmo para comprimir cadenas, dada una cadena “aAaBbccccaab” el resultado debe ser “a3b2c4a2b1”. Nótese que cuenta la cantidad de letras consecutivas si distinguir entre mayúsculas y minúsculas.

5. Cree una interface llamada ICalcular, que tenga los siguientes métodos:

- sumar
- restar
- multiplicar
- dividir throw MiExcepcion

Todos los métodos reciben dos parámetros del tipo Numbre y retornan un Double. El método dividir debe lanzar una excepción si se intenta dividir por 0.

Cree una clase llamada calculadora la cual implemente la interface anterior.

En el método main se deben mostrar por consola todos los métodos disponibles y un número. El usuario debe ingresar la operación deseada y los valores requeridos por el método. El sistema debe mostrar por consola el resultado. Utilice la estructura switch para la ejecución de los métodos.

6. Desarrolla una clase llamada Cafetera con atributos:

- capacidadMaxima (la cantidad máxima de café que puede contener la cafetera)
- cantidadActual (la cantidad actual de café que hay en la cafetera).

Implementa, al menos, los siguientes métodos:

- Constructor predeterminado: establece la capacidad máxima en 1000 (c.c.) y la actual en cero (cafetera vacía).
- Constructor con la capacidad máxima de la cafetera; inicializa la cantidad actual de café igual a la capacidad máxima.
- Constructor con la capacidad máxima y la cantidad actual. Si la cantidad actual es mayor que la capacidad máxima de la cafetera, la ajustará al máximo.
- Getters y Setters
- llenarCafetera(): hace que la cantidad actual sea igual a la capacidad.
- servirTaza(int): simula la acción de servir una taza con la capacidad indicada. Si la cantidad actual de café “no alcanza” para llenar la taza, se sirve lo que quede.
- vaciarCafetera(): pone la cantidad de café actual en cero.
- agregarCafe(int): añade a la cafetera la cantidad de café indicada. Si supera el máximo llenarla.

7. Creo un POJO llamado persona, con los siguientes atributos:

- nombre: cadena
- apellido: cadena
- documento: entero

Cree una clase derivada llamada alumno:

- legajo: cadena.

Cree otra clase derivada llamada profesor:

- materia: EMateria
- sueldo: numérico

En el main cree dos objetos de cada una de las clases, agregue los cuatro objetos a un Array de tipo Persona y muéstrellos por pantalla. Sobrecargue el método toString.

8. En un modelo de una empresa hay definida una clase empleado que tienen los siguientes atributos: nombre, edad, departamento. Se necesita extender el concepto empleado para abarcar nuevos tipos de empleados, a saber:

- Empleado temporal, del que nos interesa saber la fecha de alta, de baja en la empresa y el sueldo mensual
- Empleado por horas. Nos interesa el precio de la hora trabajada, y el número de horas que ha trabajado este mes. El primero es un dato fijo, mientras el segundo varía todos los meses.
- Empleado fijo. Debemos añadir el sueldo mensual

Además debemos añadir a todos los empleados la funcionalidad de cálculo del sueldo con las siguientes consideraciones:

- En los empleados temporales y fijos el sueldo mensual es fijo.
- En los empleados por horas el sueldo se calcula multiplicando su sueldo por hora por el número de horas de este mes.

Diseñe las clases necesarias y sus relaciones.

9. La empresa informática “IPM Tech” necesita llevar un registro de todos sus empleados que se encuentran en la oficina central, para eso ha creado un diagrama de clases que debe incluir lo siguiente:

Empleado

Atributos:

- nombre: tipo cadena (Debe ser nombre y apellido)
- cedula: tipo cadena
- edad: entero (Rango entre 18 y 45 años)
- casado: boolean
- salario: tipo numérico doble

Métodos:

+ Constructor con y sin parámetros

- Método que permita mostrar la clasificación según la edad de acuerdo al siguiente algoritmo:

Si edad es menor o igual a 21, Principiante

Si edad es ≥ 22 y ≤ 35 , Intermedio

Si edad es > 35 , Senior.

- Imprimir los datos del empleado por pantalla (utilizar salto de línea \n para separar los atributos.

- Un método que permita aumentar el salario en un porcentaje que sería pasado como parámetro al método.

Programador (Especialización de Empleado). Clase que hereda de Empleado todos los atributos y métodos.

- Atributos:

- lineasDeCodigoPorHora: tipo entero
- lenguajeDominante: tipo cadena

- Métodos:

+Constructor con y sin parámetros

10. Crear un clase abstracta llamado telefono

- numero: cadena

Crear una clase derivada llamada celular:

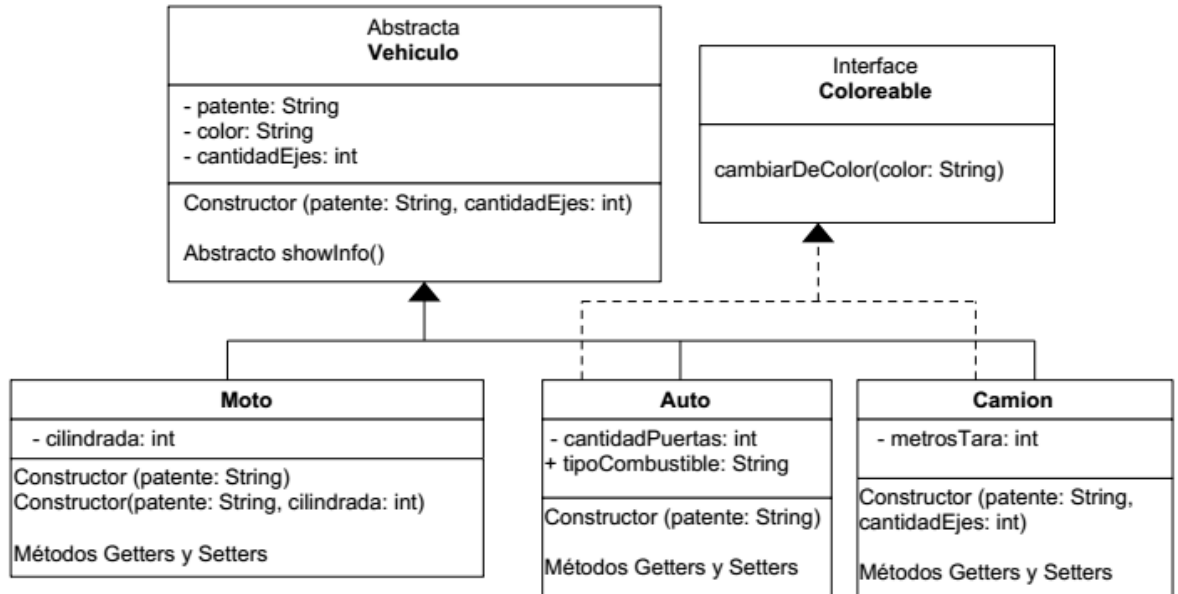
- documentoPropietario: cadena
- nombrePropietario: cadena

Crear una clase derivada llamada fijo:

- calleDomicilio: cadena
- numeroDomicilio: entero

En el main crear dos objetos de cada una de las clases y agregarlas a una lista. Recorre la lista y mostrarlos por pantalla. Sobrecargue el método toString.

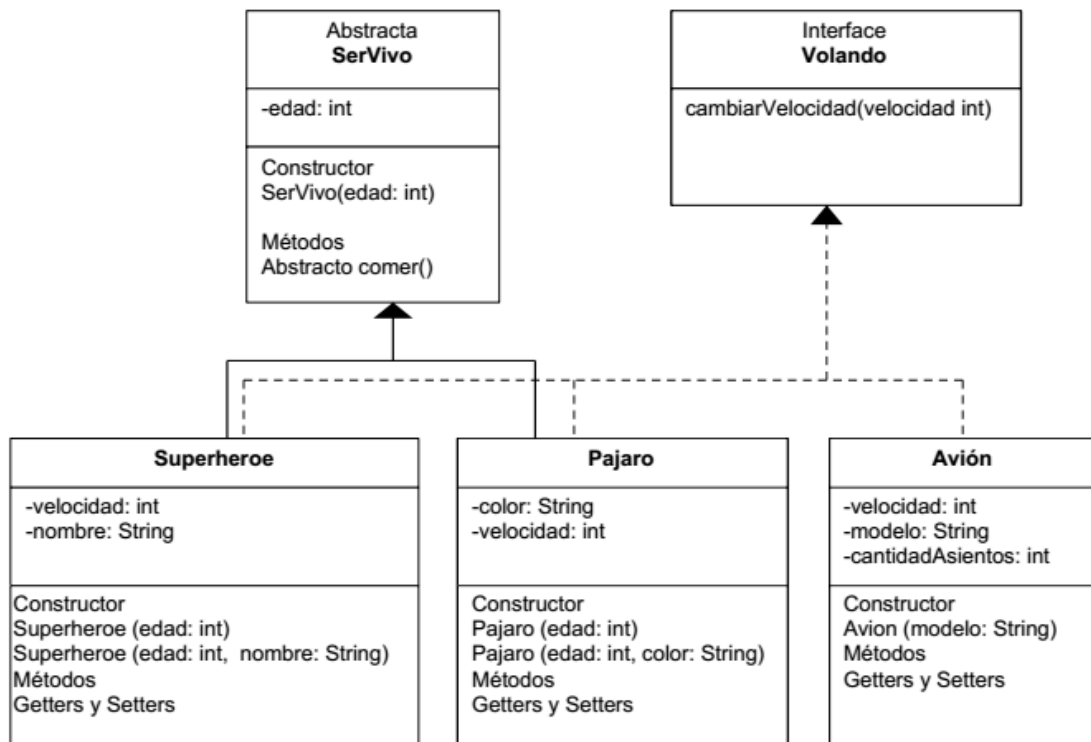
11. Desarrolle el siguiente diagrama.



Instancia un objeto de cada clase en las que sea posible hacerlo. Si alguna clase tiene constructores sobrecargados, realizar una instancia por cada constructor. A excepción de los métodos constructores y los getters y setters, el resto de los métodos contendrá sólo un mensaje de salida por consola que indique la clase donde se está ejecutando. Asignar valores a los atributos que no los reciban por medio del constructor.

Mostrar los atributos mediante el método `showInfo`. En las clases que utilicen el método `CambiarDeColor()` mostrar los atributos antes de la llamada al método y después de dicha llamada mostrar el atributo que ha sido modificado.

12. Desarrolle el siguiente diagrama.



Instancia un objeto de cada clase en las que sea posible hacerlo. Si alguna clase tiene constructores sobrecargados, realizar una instancia por cada constructor. A excepción de

los métodos constructores y los getters y setters, el resto de los métodos contendrá sólo un mensaje de salida por consola que indique la clase donde se está ejecutando. Asignar valores a los atributos que no los reciban por medio del constructor.

13. Cree el siguiente POJO

Persona

-apellido: tipo cadena
-nombre: tipo cadena
-dni: entero

+ Constructor con y sin parámetros.

Genere un test con tres métodos, en cada uno de ellos agregue una colección distinta: List, Set y Map (Utilice el dni como "Key"). Agregue a cada una de ellas tres objetos de persona. Juan Perez 29942000, Victor Lopez 24540789 y Juan Perez 29942000. Luego recorra y muestre los resultados en la consola.

Sobrescriba el método equals y el toString en la clase persona y vuelva a recorrer las listas.

14. Crear una clase llamada empleado, que tenga nombre, apellido, cuil, estado, sueldo, cantidadHorasTrabajadas y tipoCobro.

Los tipos de cobro tiene que estar dados por un Enum: mensual o porHora.

Crear una interface llamada ICalculoSuelo, la cual debe poseer un método que se llame calcular que retornara el monto que debe cobrar el empleado.

Cree dos implementaciones de la interfaz ICalculoSuelo: SueldoMensual y

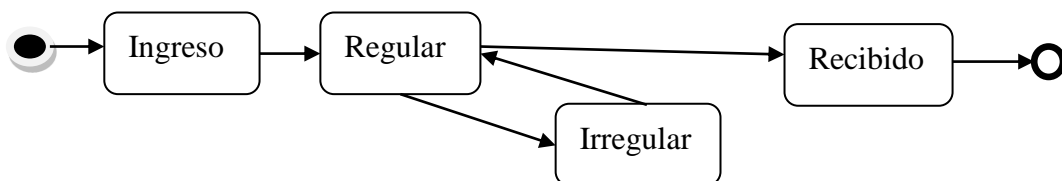
SueldoPorHora, en la primera debe retornar el atributo sueldo y en la segunda debe retornar el atributo sueldo por cantidad de horas.

Genere en un test una lista de 5 empleados y liquídele el sueldo. Utilice el patrón Factory para identificar que implementación utilizar para cada empleado.

15. Crear un clase llamada alumno que tenga nombre, apellido, legajo, carrear y estado Carrera y estado deben estar dados por un enumerado.

Un alumno puede tener los siguientes estados: Ingreso, Regular, irregular, Recibido.

Aplique el patrón State para proporcionar la siguiente máquina de estados



16. Crear un POJO llamado ciudad, con los siguientes atributos:

- nombre
- cantidadHabitantes
- presupuesto

Crear un test que tenga cinco métodos:

- mostrarAtributos: por medio de reflexión mostrar los nombres de los atributos.
- motrarMetodos: por medio de reflexión mostrar los nombres de los metodos.

- ejecutarGetter: por medio de reflexión ejecutar los métodos get y mostrar el valor en la pantalla.
 - ejecutarSetter: por medio de reflexión setearle valores a todos los atributos.
 - crearObjeto: por medio de reflexión crear un nuevo objeto.
17. Crear una anotación propia que se llame AutoGenerar. Se utilizara a nivel de atributos. Crear una clase Auto con una referencia a Patente y otra a Modelo, los atributos quedan a criterio del alumno. Ambas referencias deben tener la anotación creada previamente. Sobrecargue el método toString en la clase Auto y cree constructores por defecto en la clase Patente y Modelo poniendo valores por defecto a sus atributos. Genere un test que inspeccione las anotaciones de la clase Auto, en las referencias que las encuentre la anotación cr eele un objeto mediante el constructor por defecto y muestre por consola el objeto de la clase Auto.
18. Crear una clase llamada mi hilo que implemente la runnable. En el m etodo run creo un for de dos mil iteraciones que muestren por consola el n umero de iteraci on y el nombre del hilo. Generar un test que cree 3 hilos, los ejecute y muestre un mensaje cuando todos los hilos finalicen.
19. Cree un hilo que se llame ejecuci on, que posea tres m etodos
- Detener : debe matar el hilo
 - Frenar: debe frenar el proceso del hilo
 - Reanudar: debe retornar a la ejecuci on.
- Cree un test adecuado para probar la funcionalidad
20. Cree un hilo que se llame tomar palabra implementado Runnable, la cual contendr a una lista de palabras. El hilo debe tomar una palabra de la lista y mostrar su nombre y la palabra por consola (cada hilo debe tomar una palabra distinta y removerla, no pueden repetir las palabras). Genere un test con tres hilos y ejec utelos para que obtengan las palabras y las muestren por consola. Cree los m etodos que considere necesarios. Ayuda: Cree dentro de la clase tomar palabra un atributo que contenga la lista de palabras y un m etodo que se encargue de tomar una palabra, removerla y devolverla.
21. Retomar el ejercicio 23 y agregue un nuevo hilo llamado “agregar palabra” que se encargue de agregar una palabra a la lista cada 5 seg. y modifique el hilo “tomar palabra” para que se detenga si no hay palabras en la lista y vuelva a tomar palabras cuando el hilo “agregar palabra” le aviese que agrego una palabra. El hilo “tomar palabra” debe indicar que se detuvo esperando palabras y el hilo “agregar palabra” debe informar que agrego una nueva palabra. Ayuda: Utilice el m etodo wait en el hilo agregar palabra, sincronizando la lista y en utilice el m etodo notify en el hilo agregar palabra.

Trabajo practico Reflexión.

Se debe realizar un “Framework” de persistencia por medio de reflexión (inspeccionar un objeto en memoria). Al finalizar el “Framework” se deberá poder hacer cualquier operación DML automáticamente en cualquier proyecto JAVA adjuntando el .jar generado.

La estructura constara de 3 paquetes: Anotaciones, Servicios y Utilidades.

1. Dentro del paquete “Utilidades” se debe crear una clase llamada UBean, la cual constara de 3 métodos públicos y estáticos:
 - a. obtenerAtributos(Object): Devuelve un ArrayList<Field> con todos los atributos que posee el parámetro Object.
 - b. ejecutarSet(Object o, String att, Object valor): Se debe ejecutar el método Setter del String dentro del Object.
 - c. ejecutarGet(Object o, String att): devolverá el valor del atributo pasado por parámetro, ejecutando el getter dentro del objeto.
2. Dentro del paquete “Anotaciones” se deben crear como mínimo 3 anotaciones: **Id**, **Columna** y **Tabla**. Las primeras dos anotaciones debe declararse a nivel de atributos y la última a nivel de clase. La anotación **Columna** y **Tabla** deben poder ingresarse una propiedad String llamada nombre.
3. Dentro del paquete Servicios crear una clase que se llame Consultas, la cual contara con 4 métodos estáticos:
 - a. guardar(Object o): el cual debe guardar en la base de datos el objeto. Debe armarse la query por medio de reflexión utilizando las anotaciones creadas en el punto 2 y utilizando los métodos creados en UBean.
 - b. modificar(Object o): el cual debe modificar todas las columnas, excepto la columna Id, la cual se va a utilizar para la restricción(*where*). Debe armarse la query por medio de reflexión utilizando las anotaciones creadas en el punto 2 y utilizando los métodos creados en UBean.
 - c. eliminar(Object o): el cual debe eliminar el registro de la base de datos. Debe armarse la query por medio de reflexión utilizando las anotaciones creadas en el punto 2 y utilizando los métodos creados en UBean.
 - d. obtenerPorId(Class c, Object id): el cual debe devolver un objeto del tipo definido en el parámetro Class, con todos sus datos cargados.). Debe armarse la query por medio de reflexión utilizando las anotaciones creadas en el punto 2 y utilizando los métodos creados en UBean.
4. En el paquete “Utilidades” debe crearse la clase UConexion, la cual va a tener implementada el patrón de diseño “Singleton”. Va a ser la encargada de armar una conexión a base de datos. El Driver, la ubicación de la base de datos, el usuario y la contraseña debe obtenerse por medio de un archivo llamado “framework.properties” y debe poder configurarse externamente.
5. Modificar el método guardar(Object o) del punto 3.a. para que, además de guardar, retorne el objeto con el atributo “Id” generado por las base de datos.
6. Debe agregarse un método en la clase Consultas llamado guardarModificar(Object) el cual va a guardar el objeto, si no existe en la base de datos o va a modificar si ya se encuentra persistido en la base de datos.

7. Debe agregarse un método en la clase Consultas llamado obtenerTodos(Class) debe obtener todos los registros de la base de datos.
8. Generar una nueva anotación a nivel de atributo, la cual va a permitir guardar en cascada dos objetos. Ej, si la clase Persona tiene un atributo Domicilio. Al guardar persona se debe guardar también el domicilio.
9. Generar un método en la clase Consultas la cual realice la funcionalidad el punto 8.

Para aprobar debe tener como mínimo realizado hasta el punto 4 inclusive sin ningún error. Si no se realiza correctamente la reutilización de código o se encuentran validaciones mediante el “hardcode” de datos (Ej: *if(id.nombre.equals("id_persona"))*) se considera como error.

Trabajo practico con conexión a base de datos JAVA

Desarrollar una agenda virtual que permita:

1. Agregar personas con domicilio y uno o más teléfonos.
2. Modificar todos los datos ingresados.
3. Eliminar una persona con todos sus datos.
4. Pedido de confirmación al usuario antes de realizar cualquiera de las acciones anteriores.
5. Listar todas las personas ingresadas al sistema.
6. Exportar el listado a XML.

Al momento de la programación tener en cuenta:

1. Sobrecargar métodos toString, equals y hashCode en las entidades.
2. Manejo de excepciones para informar al usuario posibles errores.
3. Realizar test unitario de todas las clases del paquete Service.
4. Documentar todos los métodos y clases del proyecto.
5. Utilización del patrón de arquitectura “Jerarquía de clases”.
6. Utilización del patrón de diseño “Facade”.
7. Utilización del patrón de diseño “Singleton”.
8. Utilización del patrón de diseño “Factory” para poder realizar 2 tipos de conexión a base de datos.

Diagrama de entidades relacionadas (DER).

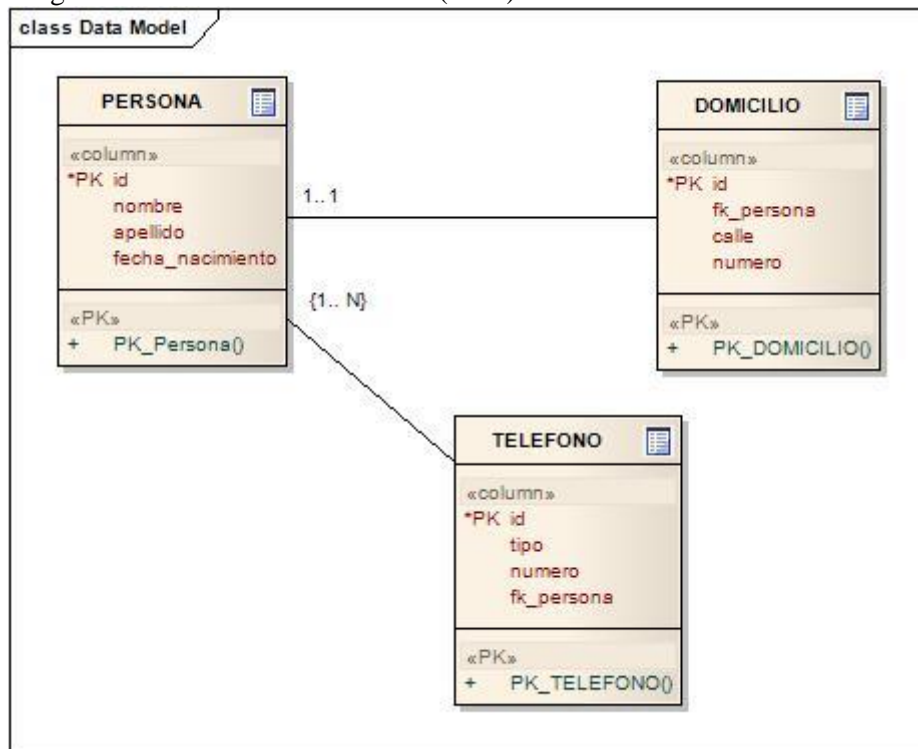
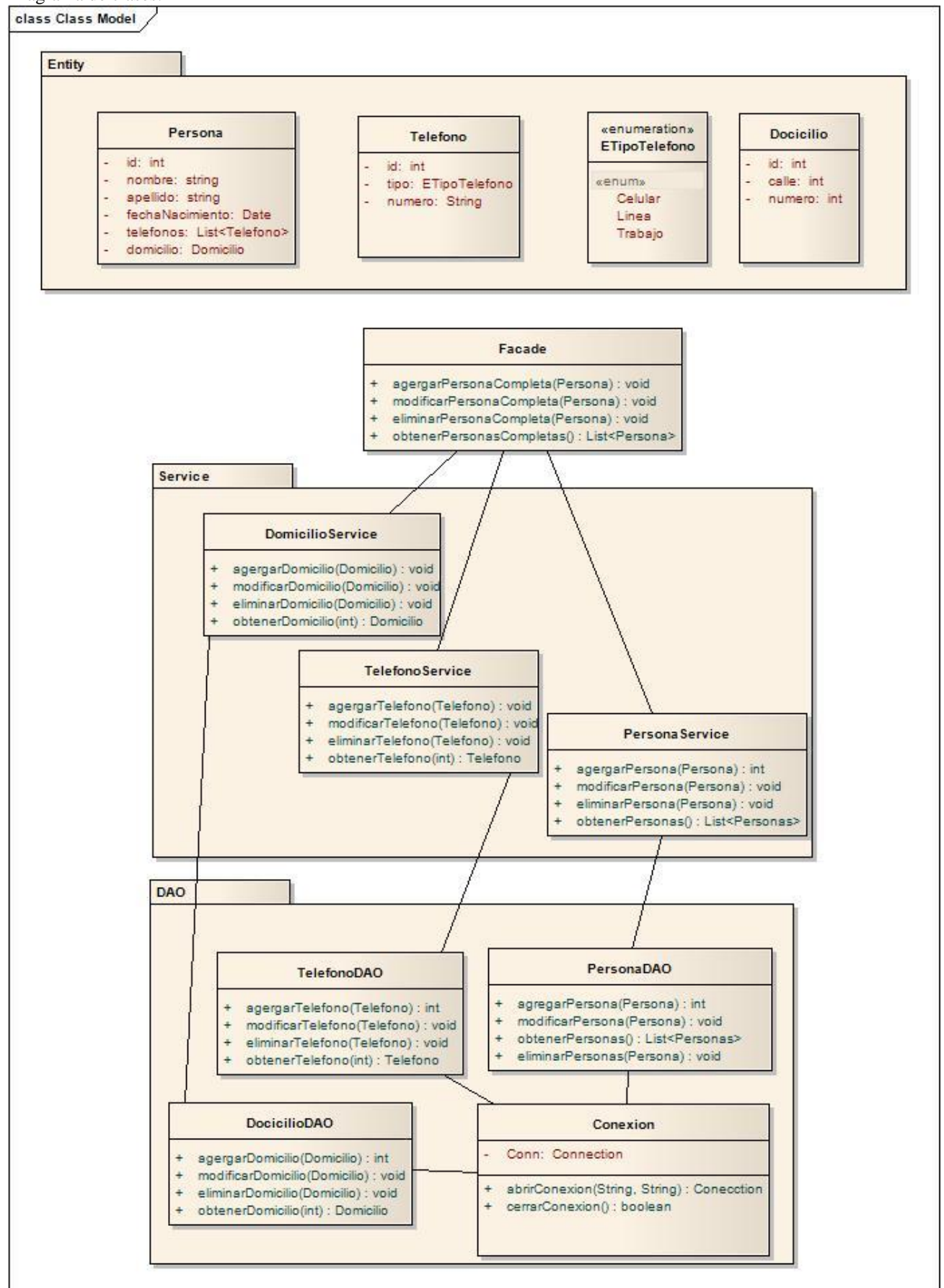


Diagrama de clases:



Agenda

Buscar

Salir

Por nombre
Ctrl-N

Por apellido
Ctrl-A

Nombre	Apellido	Fecha de nacimiento
Matias	Ramos	16/05/1988

Eliminar

Modificar

Agregar

Datos

Persona

Contacto

Datos basicos

Nombre:

Apellido:

Fecha de nacimiento:

Cancelar

Guardar

The image shows a Java Swing window titled "Datos" with a blue title bar and standard window controls. It contains two tabs: "Persona" and "Contacto". The "Contacto" tab is selected and active. Inside the "Contacto" tab, there are two main sections: "Domicilio" and "Telefono".

The "Domicilio" section contains two text input fields: "Calle:" with the value "Urquiza" and "Nro.:" with the value "2437".

The "Telefono" section contains a "Tipo:" dropdown menu set to "Linea" and a "Numero:" text input field with the value "4689-5544". Below the "Numero:" field is a green button with a "+" sign. To the right of these fields is a list box containing two items: "Linea - 4444-4444" and "Celular - 15-5455-4444".

At the bottom of the window, there are two buttons: "Cancelar" (with a red 'X' icon) and "Guardar" (with a person icon).