



Unidad I - Antecedentes

- La tecnología Java
- Estructura del lenguaje
- Conceptos de la Programación Orientada a Objetos
- Trabajando con clases y objetos en Java
- Paquetes y Modificadores de acceso
- Excepciones
- AWT, creación de applets y aplicaciones
- Flujos de Entrada/Salida (E/S)
- Programación en red con sockets
- JDBC
- Servlets

Sitios recomendados

-  Programación en Java: fundamentos de programación y principios de diseño
<http://elvex.ugr.es/decsai/java>
-  Java en castellano (Tutorial Java Básico)
http://www.programacion.com/java/tutorial/java_basico/

Comentarios

Java ofrece tres tipos de comentarios: dos para comentarios regulares en el código fuente y uno para la documentación especial del sistema javadoc.

- **Comentarios de varias líneas.**

Los comentarios de varias líneas se incluyen entre los símbolos `/*` y `*/`, como en C y C++.

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

- **Comentarios de una sola línea.**

Para comentar una sola línea se utiliza la doble diagonal `//`. El comentario se inicia cuando se encuentra la doble diagonal y continua hasta el final de la línea.

```
// Este es un comentario de una sola línea  
//Este es otro comentario
```

- **Comentarios para documentación.**

Realmente este tipo de comentario es el mismo que el de varias líneas con la diferencia de que la información que contenga será usada para un procesamiento especial que lleva a cabo la herramienta *javadoc*.

Se distingue del comentario de varias líneas porque se agrega un asterisco adicional al inicio del comentario.

```
/**
Este tipo de comentarios
los utiliza la
herramienta javadoc
*/
```

Identificadores

Un identificador es una secuencia de caracteres comenzando por una letra y conteniendo letras y números. Los identificadores no se limitan a los caracteres ASCII, si el editor de texto lo soporta, se pueden escribir identificadores utilizando caracteres Unicode.

Las *letras Java* incluyen los caracteres ASCII A-Z y a-z. Los *dígitos Java* incluyen los dígitos ASCII 0-9. Para propósitos de construir identificadores, los caracteres ASCII \$ y _ son también considerados *letras Java*.

No hay un límite en lo concerniente al número de caracteres que pueden tener los identificadores.

Estos son algunos ejemplos de identificadores válidos:

```
_varx      $var1      MAX_NUM      var2
```

Palabras clave

La siguiente tabla muestra las palabras claves de Java, éstas son reservadas y no pueden ser utilizadas como identificadores.

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

* Son palabras claves de Java que no son usadas actualmente.

** Palabra clave agregada en Java 2

true, **false**, and **null** no son palabras claves pero son palabras reservadas, así que tampoco pueden ser utilizadas como identificadores.

Literales

Una literal es un valor constante formado por una secuencia de caracteres. Cualquier declaración en Java que defina un valor constante -un valor que no pueda ser cambiado durante la ejecución del programa- es una literal.

Son ejemplos de literales los números, los caracteres y las cadenas de caracteres.

- **Literales numéricas**

Se pueden crear literales numéricas a partir de cualquier tipo de dato primitivo.

Ej.

```
123                //literal int
123.456           //literal double
123L              //literal long
123.456F         //literal flota
```

- **Literales booleanas**

Las literales booleanas consisten de las palabras reservadas *true* y *false*.

- **Literales de caracteres**

Las literales de caracteres se expresan por un solo carácter entre comillas sencillas

Ej. 'a', '%', '7'

- **Literales de cadena**

Una cadena es una combinación de caracteres. Las cadenas en Java son instancias de la clase **String**, por ello cuentan con métodos que permiten combinar, probar y modificar cadenas con facilidad.

Las literales de cadena se representan por una secuencia de caracteres entre comillas dobles.

Ej. "hola", "cadena123", "12345"

Expresiones y Operadores

- **Expresión**

Una expresión es una combinación de variables, operadores y llamadas de métodos construida de acuerdo a la sintaxis del lenguaje que devuelve un valor.

El tipo de dato del valor regresado por una expresión depende de los elementos usados en la expresión.

- **Operadores**

Los operadores son símbolos especiales que por lo común se utilizan en expresiones.

La tabla siguiente muestra los distintos tipos de operadores que utiliza Java.

Operador	Significado	Ejemplo
<i>Operadores aritméticos</i>		
+	Suma	a + b
-	Resta	a - b
*	Multiplicación	a * b
/	División	a / b
%	Módulo	a % b
<i>Operadores de asignación</i>		

=	Asignación	a = b
+=	Suma y asignación	a += b (a=a + b)
-=	Resta y asignación	a -= b (a=a - b)
*=	Multiplicación y asignación	a *= b (a=a * b)
/=	División y asignación	a / b (a=a / b)
%=	Módulo y asignación	a % b (a=a % b)

Operadores relacionales

==	Igualdad	a == b
!=	Distinto	a != b
<	Menor que	a < b
>	Mayor que	a > b
<=	Menor o igual que	a <= b
>=	Mayor o igual que	a >= b

Operadores especiales

++	Incremento	a++ (postincremento) ++a (preincremento)
--	Decremento	a-- (postdecremento) --a (predecremento)
(tipo)expr	Cast	a = (int) b
+	Concatenación de cadenas	a = "cad1" + "cad2"
.	Acceso a variables y métodos	a = obj.var1
()	Agrupación de expresiones	a = (a + b) * c

La tabla siguiente muestra la precedencia asignada a los operadores, éstos son listados en orden de precedencia.

Los operadores en la misma fila tienen igual precedencia

Operador	Notas
[] ()	Los corchetes se utilizan para los arreglos
++ -- ! ~	! es el NOT lógico y ~ es el complemento de bits
new (tipo)expr	new se utiliza para crear instancias de clases
* / %	Multiplicativos

+ -	Aditivos
<< >> >>>	Corrimiento de bits
< > <= >=	Relacionales
== !=	Igualdad
&	AND (entre bits)
^	OR exclusivo (entre bits)
	OR inclusivo (entre bits)
&&	AND lógico
	OR lógico
? :	Condicional
= += -= *= /= %= &= ^= = <<= >>= >>>=	Asignación

Todos los operadores binarios que tienen la misma prioridad (excepto los operadores de asignación) son evaluados de izquierda a derecha.

Los operadores de asignación son evaluados de derecha a izquierda.

Variables y tipos de datos

Las variables son localidades de memoria en las que pueden almacenarse datos. Cada una tiene un nombre, un tipo y valor. Java tiene tres tipos de variables: de instancia, de clase y locales.

- **Variables de instancia.**
Se utilizan para definir los atributos de un objeto.
- **Variables de clase.**
Son similares a las variables de instancia, con la excepción de que sus valores son los mismos para todas las instancias de la clase.
- **Variables locales.**
Se declaran y se utilizan dentro de las definiciones de los métodos.

* A diferencia de otros lenguajes, Java no tiene variables globales, es decir, variables que son vistas en cualquier parte del programa.

Variables y tipos de datos

Java es un lenguaje "fuertemente tipado o tipificado" por lo que es necesario especificar el tipo de dato para cada una de las variables que se vayan a utilizar en un programa. El nombre de la variable debe de ser un identificador válido, y se usa para referirse a los datos que contiene una variable.

El tipo de una variable determina los valores que puede almacenar y las operaciones que se pueden hacer sobre ella. Para dar a una variable un tipo y un nombre, se escribe una declaración de una variable, que tiene la siguiente forma:

```
TipoDato nombreVariable;
```

Ej.

```
String nombre; // variable de tipo String
int edad; // variable de tipo int
Punto p; // variable del tipo Punto
Se pueden escribir varios nombres de variables del mismo tipo en una sola línea,
int x, y, z;
String nombre, apellido;
También se puede asignar un valor inicial a las variables al momento de crearlas,
String nombre, apellido="MiApellido";
int edad = 24;
```

Variables y tipos de datos

Las variables en Java pueden ser uno de los siguientes tipos:

- **Tipo primitivo.**

Una variable de tipo primitivo contiene un solo valor del tamaño y formato apropiado de su tipo: un número, un carácter, o un valor booleano.

La tabla siguiente lista los tipos de datos primitivos soportados por Java.

Tipo	Descripción	Tamaño/Formato
<i>Números enteros</i>		
Byte	Entero byte	8-bit 2's
Short	Entero corto	16-bit 2's
Int	Entero	32-bit 2's
Long	Entero largo	64-bit 2's
<i>Números reales</i>		
Flota	Punto flotante	32-bit IEEE 754
Double	Punto flotante de doble precisión	64-bit IEEE 754
<i>Otros tipos</i>		
Char	Un solo carácter	16-bit caracteres Unicode
boolean	Un valor booleano	true o false

En Java, cualquier numérico con punto flotante automáticamente se considera double. Para que sea considerado float se agrega una letra "f" o "F" al final del valor.

```
double d = 10.50;
float f = 10.50F;
```

- **Referencia.**

Los arreglos, las clases y las interfaces son del tipo referencia. El valor de una variable del tipo referencia es una dirección de un conjunto de valores representados por una variable.

Las referencias en Java no son como en C++, éstas son identificadores de instancias de alguna clase en particular.

Ej.

```
String cad; //referencia a un objeto de la clase String
Punto p; //referencia a un objeto de la clase Punto
int[] var_arreglo; //referencia a un arreglo de enteros
```

Bloques y sentencias

- **Sentencia**

Una instrucción o sentencia representa la tarea más sencilla que se puede realizar en un programa.

- **Sentencias de expresión**

Los siguientes tipos de expresiones pueden ser hechas dentro de una sentencia terminando la expresión con punto y coma (;):

- * Expresiones de asignación
- * Cualquier uso de los operadores ++ y --
- * Llamada de métodos
- * Expresiones de creación de objetos

Esta clase de sentencias son llamadas sentencias de expresión.

Ej.

```
valorA = 8933.234; // asignación
valorA++; // incremento
System.out.println(valorA); // llamada a un método
Integer objInt = new Integer(4); // creación de objetos
```

- **Sentencias de declaración de variables**

Las sentencias de declaración de variables se utilizan para declarar variables.

Ej.

```
int bValue;
double aValue = 8933.234;
String varCad;
```

- **Sentencias de control de flujo**

Las sentencias de control de flujo determinan el orden en el cual serán ejecutadas otro grupo de sentencias. Las sentencias *if* y *for* son ejemplos de sentencias de control de flujo.

Un bloque es un grupo de cero o más sentencias encerradas entre llaves ({ y }). Se puede poner un bloque de sentencias en cualquier lugar en donde se pueda poner una sentencia individual.

Las sentencias de control de flujo se pueden utilizar para ejecutar sentencias condicionalmente, para ejecutar de manera repetida un bloque de sentencias y en general para cambiar la secuencia normal de un programa.

- **La sentencia if**

La sentencia **if** permite llevar a cabo la ejecución condicional de sentencias.

```
if ( Expresion ){
    sentencias;
}
```

Se ejecutan las sentencias si al evaluar la expresión se obtiene un valor booleano `true`.

```
if ( Expresion ){
    sentenciasA;
}
else{
    sentenciasB;
}
```

Si al evaluar la expresión se obtiene un valor booleano `true` se ejecutarán las sentenciasA, en caso contrario se ejecutarán las sentenciasB.

- **La sentencia switch**

Cuando se requiere comparar una variable con una serie de valores diferentes, puede utilizarse la sentencia **switch**, en la que se indican los posibles valores que puede tomar la variable y las sentencias que se tienen que ejecutar si es que la variable coincide con alguno de dichos valores.

```
switch( variable ){
    case valor1:
        sentencias;
        break;
    case valor2:
        sentencias;
        break;
    ...
    case valorN:
        sentencias;
        break;
    default:
        sentencias;
}
```

Cada `case` ejecutará las sentencias correspondientes, con base en el valor de la variable, que deberá de evaluarse con valores de tipo **byte**, **char**, **short** o **int**.

Si el valor de la variable no coincide con ningún valor, entonces se ejecutan las sentencias por *default*, si es que las hay.

La sentencia *break* al final de cada `case` transfiere el control al final de la sentencia **switch**; de esta manera, cada vez que se ejecuta un `case` todos los enunciados `case` restantes son ignorados y termina la operación del **switch**.

- **El ciclo for**

El ciclo **for** repite una sentencia, o un bloque de sentencias, mientras una condición se cumpla. Se utiliza la mayoría de las veces cuando se desea repetir una sentencia un determinado número de veces.

La forma general de la sentencia **for** es la siguiente;

```
for(inicialización;condición;incremento){
    sentencias;
}
```

- En su forma más simple, la inicialización es una sentencia de asignación que se utiliza para establecer una variable que controle el ciclo.
- La condición es una expresión que comprueba la variable que controla el ciclo y determinar cuando salir del ciclo.
- El incremento define la manera en como cambia la variable que controla el ciclo.

Los ciclos **while** y **do-while**, al igual que los ciclos *for* repiten la ejecución de un bloque de sentencias mientras se cumpla una condición específica.

- **La sentencia while**

El formato de la sentencia **while** es la siguiente:

```
while (condición){
    sentencias;
}
```

La condición es una condición booleana, que mientras tenga el valor **true** permite que se ejecuten las sentencias correspondientes.

- **La sentencia do-while**

Al contrario de los ciclos *for* y *while* que comprueban una condición en lo alto del ciclo, el ciclo **do-while** la examina en la parte más baja del mismo. Esta característica provoca que un ciclo **do-while** siempre se ejecute por lo menos una vez.

El formato de la sentencia **do-while** es el siguiente:

```
do{
    sentencias;
}while (condición);
```

- **break**

La sentencia **break** tiene dos usos. El primer uso es terminar un *case* en la sentencia *switch*. El segundo es forzar la terminación inmediata de un ciclo, saltando la prueba condicional normal del ciclo.

- **continue**

La sentencia **continue** es similar a la sentencia *break*. Sin embargo, en vez de forzar la terminación del ciclo, **continue** fuerza la siguiente iteración y salta cualquier código entre medias.

- **return**

Se utiliza la sentencia **return** para provocar la salida del método actual; es decir, return provocará que el programa vuelva al código que llamó al método.

La sentencia return puede regresar o no un valor. Para devolver un valor, se pone el valor después de la palabra clave return.

```
return valor;
```

El tipo de dato del valor regresado debe ser el mismo que el que se especifica en la declaración del método.

Cuando un método es declarado void, el método no regresa ningún valor.

```
return;
```

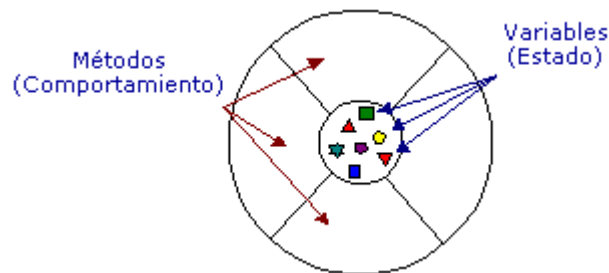
Clases y objetos

- **Objeto**

Un objeto es una encapsulación genérica de datos y de los procedimientos para manipularlos.

Al igual que los objetos del mundo real, los objetos de software tienen un estado y un comportamiento. El estado de los objetos se determina a partir de una o más *variables* y el comportamiento con la implementación de *métodos*.

La siguiente figura muestra la representación común de los objetos de software

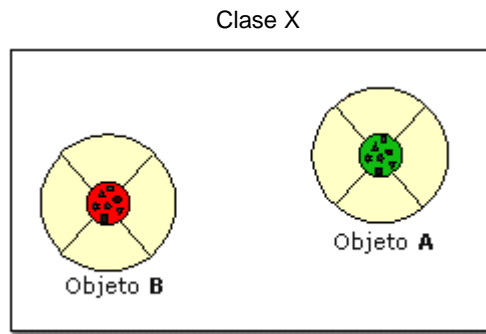


Como se observa en la figura, todos los objetos tienen una parte pública (su comportamiento) y una parte privada (su estado). En este caso, hicimos una vista transversal pero desde el mundo exterior, el objeto se observará como una esfera.

- **Clase**

Una clase está formada por los *métodos* y las *variables* que definen las características comunes a todos los objetos de esa clase. Precisamente la clave de la OOP está en abstraer los métodos y los datos comunes a un conjunto de objetos y almacenarlos en una clase.

Una clase equivale a la generalización de un tipo específico de objetos. Una *instancia* es la concreción de una clase.



En la figura anterior, el objeto A y el objeto B son instancias de la clase X.

Cada uno de los objetos tiene su propia copia de las variables definidas en la clase de la cual son instanciados y comparten la misma implementación de los métodos.

Mensajes y métodos

El modelado de objetos no sólo tiene en consideración los objetos de un sistema, sino también sus interrelaciones.

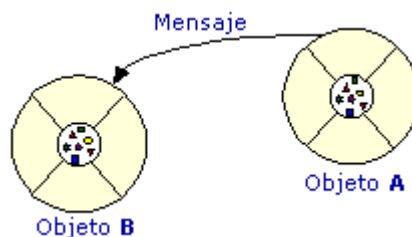
- **Mensaje**

Los objetos interactúan enviándose mensajes unos a otros. Tras la recepción de un mensaje el objeto actuará. La acción puede ser el envío de otros mensajes, el cambio de su estado, o la ejecución de cualquier otra tarea que se requiera que haga el objeto.

- **Método**

Un método se implementa en una clase, y determina cómo tiene que actuar el objeto cuando recibe un mensaje.

Cuando un objeto A necesita que el objeto B ejecute alguno de sus métodos, el objeto A le manda un mensaje al objeto B.



Al recibir el mensaje del objeto A, el objeto B ejecutará el método adecuado para el mensaje recibido.

Encapsulamiento

Como se puede observar de los diagramas, las variables del objeto se localizan en el centro o núcleo del objeto. Los métodos rodean y esconden el núcleo del objeto de otros objetos en el programa. Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama *encapsulamiento*. Típicamente, el encapsulamiento es utilizado para esconder detalles de la puesta en práctica no importantes de otros objetos. Entonces, los detalles de la puesta en práctica pueden cambiar en cualquier tiempo sin afectar otras partes del programa.

El encapsulamiento de variables y métodos en un componente de software ordenado es, todavía, una simple idea poderosa que provee dos principales beneficios a los desarrolladores de software:

- *Modularidad*, esto es, el código fuente de un objeto puede ser escrito, así como darle mantenimiento, independientemente del código fuente de otros objetos. Así mismo, un objeto puede ser transferido alrededor del sistema sin alterar su estado y conducta.
- *Ocultamiento de la información*, es decir, un objeto tiene una "interfaz pública" que otros objetos pueden utilizar para comunicarse con él. Pero el objeto puede mantener información y métodos privados que pueden ser cambiados en cualquier tiempo sin afectar a los otros objetos que dependen de ello.

Los objetos proveen el beneficio de la modularidad y el ocultamiento de la información. Las clases proveen el beneficio de la reutilización. Los programadores de software utilizan la misma clase, y por lo tanto el mismo código, una y otra vez para crear muchos objetos.

En las implantaciones orientadas a objetos se percibe un objeto como un paquete de datos y procedimientos que se pueden llevar a cabo con estos datos. Esto encapsula los datos y los procedimientos. La realidad es diferente: los atributos se relacionan al objeto o instancia y los métodos a la clase. ¿Por qué se hace así? Los atributos son variables comunes en cada objeto de una clase y cada uno de ellos puede tener un valor asociado, para cada variable, diferente al que tienen para esa misma variable los demás objetos. Los métodos, por su parte, pertenecen a la clase y no se almacenan en cada objeto, puesto que sería un desperdicio almacenar el mismo procedimiento varias veces y ello va contra el principio de reutilización de código.

Herencia

- **Herencia**

La herencia es un mecanismo que permite la definición de una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente métodos y datos entre clases, subclases y objetos.

La herencia está fuertemente ligada a la reutilización del código en la OOP. Esto es, el código de cualquiera de las clases puede ser utilizado sin más que crear una clase derivada de ella, o bien una *subclase*.

Hay dos tipos de herencia: *Herencia Simple* y *Herencia Múltiple*. La primera indica que se pueden definir nuevas clases solamente a partir de una clase inicial mientras que la segunda indica que se pueden definir nuevas clases a partir de dos o más clases iniciales. Java sólo permite herencia simple.

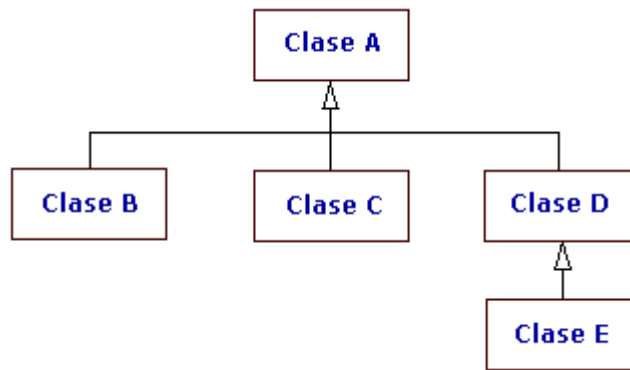
- **Superclase y Subclases**

El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la OOP todas las relaciones entre clases deben ajustarse a dicha estructura.

En esta estructura jerárquica, cada clase tiene sólo una clase padre. La clase padre de cualquier clase es conocida como su *superclase*. La clase hija de una superclase es llamada una *subclase*.

* Una superclase puede tener cualquier número de subclases.

* Una subclase puede tener sólo una superclase.



- A es la superclase de B, C y D.
- D es la superclase de E.
- B, C y D son subclases de A.
- E es una subclase de D.

Polimorfismo

Otro concepto de la OOP es el polimorfismo. Un objeto solamente tiene una forma (la que se le asigna cuando se construye ese objeto) pero la referencia a objeto es *polimórfica* porque puede referirse a objetos de diferentes clases (es decir, la referencia toma *múltiples formas*). Para que esto sea posible **debe haber una relación de herencia entre esas clases**. Por ejemplo, considerando la figura anterior de herencia se tiene que:

- Una referencia a un objeto de la clase B también puede ser una referencia a un objeto de la clase A.
- Una referencia a un objeto de la clase C también puede ser una referencia a un objeto de la clase A.
- Una referencia a un objeto de la clase D también puede ser una referencia a un objeto de la clase A.
- Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase D.
- Una referencia a un objeto de la clase E también puede ser una referencia a un objeto de la clase A.

Abstracción

Volviendo a la figura anterior de la relación de herencia entre clases, se puede pensar en una jerarquía de clase como la definición de conceptos demasiado abstractos en lo alto de la jerarquía y esas ideas se convierten en algo más concreto conforme se desciende por la cadena de la superclase.

Sin embargo, las clases hijas no están limitadas al estado y conducta provistos por sus superclases; pueden agregar variables y métodos además de los que ya heredan de sus clases padres. Las clases hijas pueden, también, sobrescribir los métodos que heredan por implementaciones especializadas para esos métodos. De igual manera, no hay limitación a un sólo nivel de herencia por lo que se tiene un árbol de herencia en el que se puede heredar varios niveles hacia abajo y mientras más niveles descienda una clase, más especializada será su conducta.

La herencia presenta los siguientes beneficios:

- Las subclases proveen conductas especializadas sobre la base de elementos comunes provistos por la superclase. A través del uso de herencia, los programadores pueden reutilizar el código de la superclase muchas veces.
- Los programadores pueden implementar superclases llamadas **clases abstractas** que definen conductas "genéricas". Las superclases abstractas definen, y pueden implementar parcialmente, la conducta pero gran parte de la clase no está definida ni

implementada. Otros programadores concluirán esos detalles con subclases especializadas.

Definición de clases

La definición de una clase especifica cómo serán los objetos de dicha clase, esto es, de qué variables y de qué métodos constarán.

La siguiente es la definición más simple de una clase:

```
class nombreClase    /* Declaración de la clase */
{
    /* Aquí va la definición de variables y métodos */
}
```

Como se puede observar, la definición de una clase consta de dos partes fundamentales:

* La declaración de la clase

Indica el nombre de la clase precedido por la palabra clave **class**.

* El cuerpo de la clase

El cuerpo de la clase sigue a la declaración de la clase y está contenido entre la pareja de llaves ({ y }). El cuerpo de la clase contiene las declaraciones de las variables de la clase, y también la declaración y la implementación de los métodos que operan sobre dichas variables.

- **Declaración de variables de instancia**

El estado de un objeto está representado por sus variables (variables de instancia). Las variables de instancia se declaran dentro del cuerpo de la clase. Típicamente, las variables de instancia se declaran antes de la declaración de los métodos, pero esto no es necesariamente requerido.

- **Implementación de métodos**

Los métodos de una clase determinan los mensajes que un objeto puede recibir.

Las partes fundamentales de un método son el valor de retorno, el nombre, los argumentos (opcionales) y su cuerpo. Además, un método puede llevar otros modificadores opcionales que van al inicio de la declaración del método y que se analizarán más adelante. La sintaxis de un método es la siguiente:

```
<otrosModificadores> valorRetorno nombreMetodo( <lista de argumentos>
)
{
    /* Cuerpo del método */
    sentencias;
}
```

Los signos <> indican que no son obligatorios.

Los métodos en Java pueden ser creados únicamente como parte de una clase. Cuando se llama a un método de un objeto se dice comúnmente que se envía un mensaje al objeto.

Ejemplo

```
/* Usuario.java */
```

```

class Usuario
{
String nombre;
int edad;
String direccion;

void setNombre(String n)
{
    nombre = n;
}

String getNombre()
{
    return nombre;
}

void setEdad(int e)
{
    edad = e;
}

int getEdad()
{
    return edad;
}

void setDireccion(String d)
{
    direccion = d;
}

String getDireccion()
{
    return direccion;
}
}

```

Constructores y creación de objetos

Una vez que se tiene definida la clase a partir de la cual se crearán los objetos se está en la posibilidad de instanciar los objetos requeridos.

Para la clase Usuario del ejemplo anterior podemos crear un objeto de la siguiente manera:

```

Usuario usr1; //usr1 es una variable del tipo Usuario
usr1 = new Usuario();

```

La primera línea corresponde a la declaración del objeto, es decir, se declara una variable del tipo de objeto deseado.

La segunda línea corresponde a la iniciación del objeto.

- **El operador new**

El operador **new** crea una instancia de una clase asignando la cantidad de memoria necesaria de acuerdo al tipo de objeto. El operador **new** se utiliza en conjunto con un *constructor*. El operador **new** regresa una referencia a un nuevo objeto.

- **Constructores**

Un constructor es un tipo específico de método que siempre tiene el mismo nombre que la clase, y que se utiliza cuando se desean crear objetos de dicha clase, es decir, se utiliza al crear e iniciar un objeto de una clase.

- **Constructores múltiples**

Cuando se declara una clase en Java, se pueden declarar uno o más constructores (*constructores múltiples*) opcionales que realizan la iniciación cuando se instancia un objeto de dicha clase.

Para la clase Usuario del ejemplo anterior no se especificó ningún constructor, sin embargo, Java proporciona un constructor por omisión que inicia las variables del objeto a sus valores predeterminados.

Ej.

```
/* ProgUsuario.java */

class ProgUsuario
{
    public static void main(String args[])
    {
        Usuario usr1, usr2;      /* Se declaran dos objetos de la clase Usuario
        */
        boolean si_no;

        usr1 = new Usuario();    /* Se utiliza el constructor por omisión */
        si_no = usr1 instanceof Usuario;

        if(si_no == true)
            System.out.println("\nEl objeto usr1 SI es instancia de
            Usuario.");
        else
            System.out.println("\nEl objeto usr1 NO es instancia de
            Usuario.");

        usr2 = usr1;            /* usr1 y usr2 son el mismo objeto */
        si_no = usr2 instanceof Usuario;

        if(si_no == true)
            System.out.println("\nEl objeto usr2 SI es instancia de
            Usuario.");
        else
            System.out.println("\nEl objeto usr2 NO es instancia de
            Usuario.");
    }
}
```

Acceso a variables y métodos

Una vez que se ha creado un objeto, seguramente se querrá hacer algo con él. Tal vez se requiera obtener información de éste, se quiera cambiar su estado, o se necesite que realice alguna tarea.

Los objetos tienen dos formas de hacer esto:

- **Manipular sus variables directamente..**

Para acceder a las variables de un objeto se utiliza el operador punto (.). La sintaxis es la siguiente:

```
nombreObjeto.nombreVariable;
```

- **Llamar a sus métodos.**

Para llamar a los métodos de un objeto, se utiliza también el operador punto (.). La sintaxis es la siguiente:


```
nombreObjeto.nombreMetodo( <lista de argumentos opcionales> );
```

Ejemplo

```
/* Usuario2.java */

class Usuario2
{
    String nombre;
    int edad;
    String direccion;

    Usuario2( )      /* Equivale al constructor por omisión */
    {
        nombre = null;
        edad = 0;
        direccion = null;
    }

    Usuario2(String nombre, int edad, String direccion)
    {
        this.nombre = nombre;
        this.edad = edad;
        this.direccion = direccion;
    }

    Usuario2(Usuario2 usr)
    {
        nombre = usr.getNombre();
        edad = usr.getEdad();
        direccion = usr.getDireccion();
    }

    void setNombre(String n)
    {
        nombre = n;
    }

    String getNombre()
    {
        return nombre;
    }

    void setEdad(int e)
    {
        edad = e;
    }

    int getEdad()
    {
        return edad;
    }

    void setDireccion(String d)
    {
        direccion = d;
    }

    String getDireccion()
    {
        return direccion;
    }
}
```

Ejemplo

```
/* ProgUsuario2.java */
```

```

class ProgUsuario2
{

void imprimeUsuario(Usuario2 usr)
{
//   usr.nombre equivale en este caso a usr.getNombre()
System.out.println("\nNombre: " + usr.nombre );
System.out.println("Edad: " + usr.getEdad() );
System.out.println("Direccion: " + usr.getDireccion() +"\n");
}

public static void main(String args[])
{
ProgUsuario2 prog = new ProgUsuario2( );
Usuario2 usr1,usr2;      /* Se declaran dos objetos de la clase Usuario2
*/

/* Se utiliza el constructor por omisión */
usr1 = new Usuario2( );
prog.imprimeUsuario(usr1);

/* Se utiliza el segundo constructor de Usuario2 */
usr2 = new Usuario2("Eduardo",24,"Mi direccion");
prog.imprimeUsuario(usr2);

/* Se utiliza el tercer constructor de Usuario2 */
usr1 = new Usuario2(usr2);

/* En este caso usr1.setDireccion("nuevoValor"); equivale
a usr1.direccion = "nuevoValor"; */
usr1.setDireccion("Otra direccion");

prog.imprimeUsuario(usr1);
prog.imprimeUsuario(usr2);
}
}

```

Variables y métodos de clase

- **Variables de clase**

Las variables de clase son variables cuyos valores son los mismos para la clase y para todas sus instancias.

Para indicar que una variable es una variable de clase se utiliza la palabra clave **static** en la declaración de la variables:

```
static tipoVariable nombreVariable;
```

- **Métodos de clase**

Los métodos de clase al igual que las variables de clase, se aplican a la clase como un todo y no a sus instancias.

Se utiliza de igual manera la palabra clave **static** para indicar que un método es un método de clase:

```
static valorRetorno nombreMetodo( <lista argumentos opcionales> )
{
    /* cuerpo del método */
}

```

Para acceder a las variables o métodos de clase se utiliza el mismo operador punto (.).

Aunque se puede acceder a las variables y métodos de clase a través de un objeto, está permitido y se recomienda utilizar mejor el nombre de la clase,

```
/* Utilizar esto */
nombreClase.nombreVarClase;
nombreClase.nombreMetodoClase();

/* en lugar de esto */
nombreObjeto.nombreVarClase;
nombreObjeto.nombreMetodoClase();
```

Ejemplo

```
/* Usuario3.java */

class Usuario3
{
    static char MAS = 'm';
    static char FEM = 'f';

    String nombre;
    int edad;
    String direccion;
    char sexo;

    Usuario3( )
    {
        nombre = null;
        edad = 0;
        direccion = null;
        sexo = '\0';
    }

    Usuario3(String nombre, int edad, String direccion, char sexo)
    {
        this.nombre = nombre;
        this.edad = edad;
        this.direccion = direccion;
        this.sexo = sexo;
    }

    Usuario3(Usuario3 usr)
    {
        nombre = usr.getNombre();
        edad = usr.getEdad();
        direccion = usr.getDireccion();
        sexo = usr.getSexo();
    }

    void setNombre(String n)
    {
        nombre = n;
    }

    String getNombre( )
    {
        return nombre;
    }

    void setEdad(int e)
    {
        edad = e;
    }

    int getEdad()
    {
        return edad;
    }
}
```

```

void setDireccion(String d)
{
    direccion = d;
}

String getDireccion( )
{
    return direccion;
}

void setSexo(char s)
{
    sexo = s;
}

char getSexo( )
{
    return sexo;
}

public String toString()
{
    return nombre;
}

/* ProgUsuario3.java */

class ProgUsuario3
{
    static int NUM_USUARIOS = 0;
    static java.util.Vector usuarios = new java.util.Vector();

    String nombreObj = null;

    ProgUsuario3(String nombre)
    {
        this.nombreObj = nombre;
    }

    static int getNumUsuarios()
    {
        return NUM_USUARIOS;
    }

    static void imprimeUsuario(Usuario3 usr)
    {
        System.out.println("\nNombre: " + usr.nombre );
        System.out.println("Edad: " + usr.getEdad() );
        System.out.println("Sexo: " + usr.getSexo() );
        System.out.println("Direccion: " + usr.getDireccion() );
    }

    void addUsuario(Usuario3 usr)
    {
        usuarios.addElement(usr);
        System.out.print(usr.toString( )+ " agregado por el "+ this.toString(
        +",");
        NUM_USUARIOS ++;
    }

    void delUsuario(Usuario3 usr)
    {
        boolean b = usuarios.removeElement(usr);
        if( b == true )
        {
            NUM_USUARIOS--;
        }
    }
}

```

```

System.out.print(usr.toString() + " eliminado por el " + this.toString()
+ ",");
}
else System.out.println("No se pudo eliminar al usuario.");
}

public String toString()
{
return nombreObj;
}

public static void main(String args[])
{
ProgUsuario3 obj1 = new ProgUsuario3("objeto1");
ProgUsuario3 obj2 = new ProgUsuario3("objeto2");

Usuario3 usr1,usr2,usr3,usr4;

usr1 = new Usuario3( );
usr2 = new Usuario3("Usuario B",24,"La direccion A",Usuario3.FEM);
usr1 = new Usuario3(usr2);
usr1.setNombre("Usuario A");
usr3 = new Usuario3("Usuario C",35,"La direccion C",Usuario3.MAS);
usr4 = new Usuario3("Usuario D",15,"La direccion D",Usuario3.MAS);

obj1.addUsuario(usr1);
System.out.println( "\t Total: " +ProgUsuario3.getNumUsuarios() );
obj2.addUsuario(usr2);
System.out.println( "\t Total: " +obj1.getNumUsuarios() );
obj1.addUsuario(usr3);
System.out.println( "\t Total: " +ProgUsuario3.NUM_USUARIOS );
obj2.addUsuario(usr4);
System.out.println( "\t Total: " +getNumUsuarios() +"\n");

obj2.delUsuario(usr4);
System.out.println( "\t Total: " +ProgUsuario3.getNumUsuarios() );
obj1.delUsuario(usr3);
System.out.println( "\t Total: " +obj1.getNumUsuarios() );
obj2.delUsuario(usr2);
System.out.println( "\t Total: " +ProgUsuario3.NUM_USUARIOS );
obj1.delUsuario(usr1);
System.out.println( "\t Total: " +getNumUsuarios() +"\n");
}
}

```

Heredando clases en Java

El concepto de herencia conduce a una estructura jerárquica de clases o estructura de árbol, lo cual significa que en la OOP todas las relaciones entre clases deben ajustarse a dicha estructura.

En esta estructura jerárquica, cada clase tiene sólo una clase padre. La clase padre de cualquier clase es conocida como su *superclase*. La clase hija de una superclase es llamada una *subclase*.

De manera automática, una subclase hereda las variables y métodos de su superclase (más adelante se explica que pueden existir variables y métodos de la superclase que la subclase no puede heredar. Véase Modificadores de Acceso). Además, una subclase puede agregar nueva funcionalidad (variables y métodos) que la superclase no tenía.

- Los constructores no son heredados por las subclases.
- Para crear una subclase, se incluye la palabra clave **extends** en la declaración de la clase.

```

class nombreSubclase extends nombreSuperclase{
}

```

- En Java, la clase padre de todas las clases es la clase **Object** y cuando una clase no tiene una superclase explícita, su superclase es Object.

Sobrecarga de métodos y de constructores

La firma de un método es la combinación del tipo de dato que regresa, su nombre y su lista de argumentos.

La sobrecarga de métodos es la creación de varios métodos con el mismo nombre pero con diferentes firmas y definiciones. Java utiliza el número y tipo de argumentos para seleccionar cuál definición de método ejecutar.

Java diferencia los métodos sobrecargados con base en el número y tipo de argumentos que tiene el método y no por el tipo que devuelve.

También existe la sobrecarga de constructores: Cuando en una clase existen constructores múltiples, se dice que hay sobrecarga de constructores.

Ejemplo

```
/* Métodos sobrecargados */
int calculaSuma(int x, int y, int z){
    ...
}
int calculaSuma(double x, double y, double z){
    ...
}

/* Error: estos métodos no están sobrecargados */
int calculaSuma(int x, int y, int z){
    ...
}
double calculaSuma(int x, int y, int z){
    ...
}
```

Ejemplo

```
/* Usuario4.java */

class Usuario4
{
    String nombre;
    int edad;
    String direccion;

    /* El constructor de la clase Usuario4 esta sobrecargado */
    Usuario4( )
    {
        nombre = null;
        edad = 0;
        direccion = null;
    }

    Usuario4(String nombre, int edad, String direccion)
    {
        this.nombre = nombre;
        this.edad = edad;
        this.direccion = direccion;
    }

    Usuario4(Usuario4 usr)
    {
        nombre = usr.getNombre();
        edad = usr.getEdad();
        direccion = usr.getDireccion();
    }
}
```

```

}

void setNombre(String n)
{
nombre = n;
}

String getNombre()
{
return nombre;
}

/* El método setEdad() está sobrecargado */
void setEdad(int e)
{
edad = e;
}

void setEdad(float e)
{
edad = (int)e;
}

int getEdad()
{
return edad;
}

void setDireccion(String d)
{
direccion = d;
}

String getDireccion()
{
return direccion;
}

```

Ejemplo

```

/* ProgUsuario4.java */

class ProgUsuario4
{
void imprimeUsuario(Usuario4 usr)
{
//   usr.nombre equivale en este caso a usr.getNombre()
System.out.println("\nNombre: " + usr.nombre );
System.out.println("Edad: " + usr.getEdad() );
System.out.println("Direccion: " + usr.getDireccion() +"\n");
}

public static void main(String args[])
{
ProgUsuario4 prog = new ProgUsuario4( );
/* Se declaran dos objetos de la clase Usuario4 */
Usuario4 usr1,usr2;

/* Se utiliza el constructor por omisión */
usr1 = new Usuario4( );
prog.imprimeUsuario(usr1);

/* Se utiliza el segundo constructor de Usuario4 */
usr2 = new Usuario4("Eduardo",24,"Mi direccion");
prog.imprimeUsuario(usr2);

/* Se utiliza el tercer constructor de Usuario4 */

```

```

usr1 = new Usuario4(usr2);

usr1.setEdad(50);
usr2.setEdad(30.45f);

prog.imprimeUsuario(usr1);
prog.imprimeUsuario(usr2);
}
}

```

Sobre escritura de métodos

Una subclase hereda todos los métodos de su superclase que son accesibles a dicha subclase a menos que la subclase sobrescriba los métodos.

Una subclase sobrescribe un método de su superclase cuando define un método con las mismas características (nombre, número y tipo de argumentos) que el método de la superclase.

Las subclases emplean la sobre escritura de métodos la mayoría de las veces para agregar o modificar la funcionalidad del método heredado de la clase padre.

Ejemplo

```

class ClaseA
{
void miMetodo(int var1, int var2)
{ ... }

String miOtroMetodo( )
{ ... }
}

class ClaseB extends ClaseA
{
/* Estos métodos sobrescriben a los métodos
de la clase padre */

void miMetodo (int var1 ,int var2)
{ ... }

String miOtroMetodo( )
{ ... }
}

```

Clases abstractas

Una clase que declara la existencia de métodos pero no la implementación de dichos métodos (o sea, las llaves {} y las sentencias entre ellas), se considera una clase abstracta.

Una clase abstracta puede contener métodos no-abstractos pero al menos uno de los métodos debe ser declarado abstracto.

Para declarar una clase o un método como abstractos, se utiliza la palabra reservada **abstract**.

```

abstract class Drawing
{
abstract void miMetodo(int var1, int var2);
String miOtroMetodo( ){ ... }
}

```

Una clase abstracta no se puede instanciar pero si se puede heredar y las clases hijas serán las

encargadas de agregar la funcionalidad a los métodos abstractos. Si no lo hacen así, las clases hijas deben ser también abstractas.

Interfaces

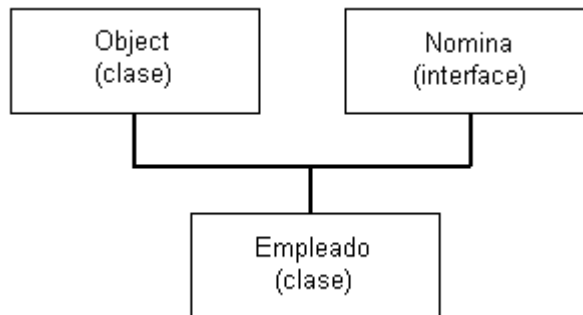
Una interface es una variante de una clase abstracta con la condición de que todos sus métodos deben ser abstractos. Si la interface va a tener atributos, éstos deben llevar las palabras reservadas **static final** y con un valor inicial ya que funcionan como constantes por lo que, por convención, su nombre va en mayúsculas.

```
interface Nomina
{
    public static final String EMPRESA = "Patito, S. A.";
    public void detalleDeEmpleado(Nomina obj);
}
```

Una clase *implementa* una o más interfaces (separadas con comas ",") con la palabra reservada **implements**. Con el uso de interfaces se puede "simular" la herencia múltiple que Java no soporta.

```
class Empleado implements Nomina
{
    ...
}
```

En este ejemplo, la clase Empleado tiene una clase padre llamada Object (implícitamente) e implementa a la interface Nomina, quedando el diagrama de clases de la siguiente manera:



La clase que implementa una interface tiene dos opciones:

- 1) Implementar todos los métodos de la interface.
- 2) Implementar sólo algunos de los métodos de la interface pero esa clase debe ser una clase abstracta (debe declararse con la palabra **abstract**).

Paquetes

Para hacer que una clase sea más fácil de localizar y utilizar así como evitar conflictos de nombres y controlar el acceso a los miembros de una clase, las clases se agrupan en paquetes.

- **Paquete**

Un paquete es un conjunto de clases e interfaces relacionadas.

La forma general de la declaración **package** es la siguiente:

```
package nombrePaquete;
```

donde nombrePaquete puede constar de una sola palabra o de una lista de nombres de paquetes separados por puntos.

Ejemplo

```
package miPaquete;
```

```
class MiClase  
{  
  ...  
}
```

Ejemplo

```
package nombre1.nombre2.miPaquete;
```

```
class TuClase  
{  
  ...  
}
```

Los nombres de los paquetes se corresponden con nombre de directorios en el sistema de archivos.

De esta manera, cuando se requiera hacer uso de estas clases se tendrán que importar de la siguiente manera.

Ejemplo

```
import miPaquete.MiClase;  
import nombre1.nombre2.miPaquete.TuClase;
```

```
class OtraClase  
{  
  /* Aqui se hace uso de la clase 'MiClase' y de la  
  clase 'TuClase' */  
  ...  
}
```

Para importar todas las clases que están en un paquete, se utiliza el asterisco (*).

Ejemplo

```
import miPaquete.*;
```

Si no se utiliza la sentencia package para indicar a que paquete pertenece una clase, ésta terminará en el package por default, el cual es un paquete que no tiene nombre.

Ejemplo

```
/* Usuario5.java */  
  
package paquetel;  
  
class Usuario5  
{  
  static char MAS = 'm';  
  static char FEM = 'f';  
  
  String nombre;  
  int edad;  
  String direccion;  
  char sexo;  
  
  Usuario5( )  
  {  
    nombre = null;  
    edad = 0;  
    direccion = null;  
    sexo = '\\0';  
  }  
}
```

```

Usuario5(String nombre, int edad, String direccion, char sexo)
{
    this.nombre = nombre;
    this.edad = edad;
    this.direccion = direccion;
    this.sexo = sexo;
}

Usuario5(Usuario5 usr)
{
    nombre = usr.getNombre();
    edad = usr.getEdad();
    direccion = usr.getDireccion();
    sexo = usr.getSexo();
}

void setNombre(String n)
{
    nombre = n;
}

String getNombre()
{
    return nombre;
}

void setEdad(int e)
{
    edad = e;
}

int getEdad()
{
    return edad;
}

void setDireccion(String d)
{
    direccion = d;
}

String getDireccion()
{
    return direccion;
}

void setSexo(char s)
{
    sexo = s;
}

char getSexo()
{
    return sexo;
}

public String toString()
{
    return nombre;
}
}

```

Ejemplo

```

/* ProgUsuario5.java */

package paquetel;

```

```

import java.util.Vector;

class ProgUsuario5
{
    static int NUM_USUARIOS = 0;

    static Vector usuarios = new Vector();
    /* La siguiente línea sería obligatoria si
    se omitiera la línea import java.util.Vector; */
    // static java.util.Vector usuarios = new java.util.Vector();

    String nombreObj = null;

    ProgUsuario5(String nombre)
    {
        this.nombreObj = nombre;
    }

    static int getNumUsuarios()
    {
        return NUM_USUARIOS;
    }

    static void imprimeUsuario(Usuario5 usr)
    {
        System.out.println("\nNombre: " + usr.nombre );
        System.out.println("Edad: " + usr.getEdad() );
        System.out.println("Sexo: " + usr.getSexo() );
        System.out.println("Direccion: " + usr.getDireccion() );
    }

    void addUsuario(Usuario5 usr)
    {
        usuarios.addElement(usr);
        System.out.print(usr.toString() + " agregado por el " + this.toString()
        + ",");
        NUM_USUARIOS ++;
    }

    void delUsuario(Usuario5 usr)
    {
        boolean b = usuarios.removeElement(usr);
        if( b == true )
        {
            NUM_USUARIOS--;
            System.out.print(usr.toString() + " eliminado por el " +
            this.toString() + ",");
        }
        else System.out.println("No se pudo eliminar al usuario.");
    }

    public String toString()
    {
        return nombreObj;
    }

    public static void main(String args[])
    {
        ProgUsuario5 obj1 = new ProgUsuario5("objeto1");
        ProgUsuario5 obj2 = new ProgUsuario5("objeto2");

        Usuario5 usr1,usr2,usr3,usr4;

        usr1 = new Usuario5( );
        usr2 = new Usuario5("Usuario B",24,"La direccion A",Usuario5.FEM);
        usr1 = new Usuario5(usr2);
        usr1.setNombre("Usuario A");
    }
}

```

```

usr3 = new Usuario5("Usuario C",35,"La direccion C",Usuario5.MAS);
usr4 = new Usuario5("Usuario D",15,"La direccion D",Usuario5.MAS);

obj1.addUsuario(usr1);
System.out.println( "\t Total: " +ProgUsuario5.getNumUsuarios() );
obj2.addUsuario(usr2);
System.out.println( "\t Total: " +obj1.getNumUsuarios() );
obj1.addUsuario(usr3);
System.out.println( "\t Total: " +ProgUsuario5.NUM_USUARIOS );
obj2.addUsuario(usr4);
System.out.println( "\t Total: " +getNumUsuarios() +"\n");

obj2.delUsuario(usr4);
System.out.println( "\t Total: " +ProgUsuario5.getNumUsuarios() );
obj1.delUsuario(usr3);
System.out.println( "\t Total: " +obj1.getNumUsuarios() );
obj2.delUsuario(usr2);
System.out.println( "\t Total: " +ProgUsuario5.NUM_USUARIOS );
obj1.delUsuario(usr1);
System.out.println( "\t Total: " +getNumUsuarios() +"\n");
}
}

```

Control de acceso a miembros de una clase

Los modificadores más importantes desde el punto de vista del diseño de clases y objetos, son los que permiten controlar la visibilidad y acceso a los métodos y variables que están dentro de una clase.

Uno de los beneficios de las clases, es que pueden proteger a sus variables y métodos (tanto de instancia como de clase) del acceso desde otras clases.

Java soporta cuatro niveles de acceso a variables y métodos. En orden, del más público al menos público son: público (public), protegido (protected), sin modificador (también conocido como *package*) y privado (private).

La siguiente tabla muestra el nivel de acceso permitido por cada modificador:

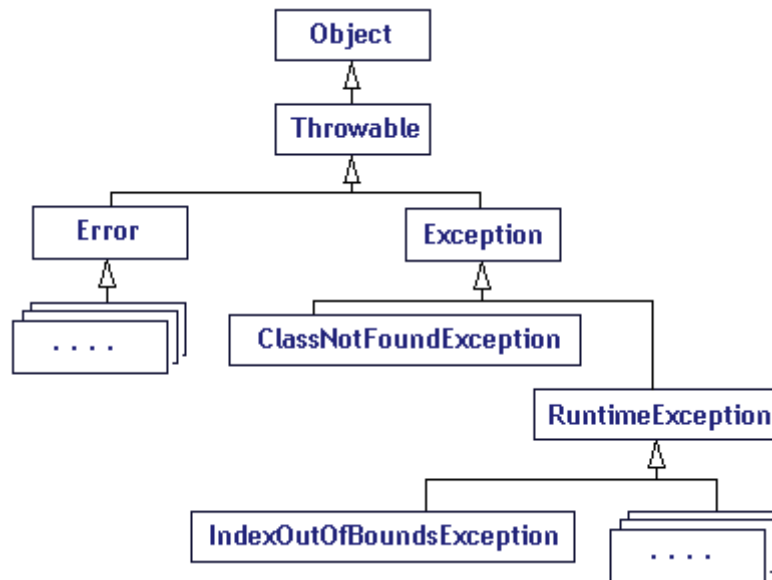
	public	protected	(sin modificador)	private
Clase	SI	SI	SI	SI
Subclase en el mismo paquete	SI	SI	SI	NO
No-Subclase en el mismo paquete	SI	SI	SI	NO
Subclase en diferente paquete	SI	SI/NO (*)	NO	NO
No-Subclase en diferente paquete (Universo)	SI	NO	NO	NO

(*) Los miembros (variables y métodos) de clase (static) si son visibles. Los miembros de instancia no son visibles.

Como se observa de la tabla anterior, una clase se ve a ella misma todo tipo de variables y métodos (desde los public hasta los private); las demás clases del mismo paquete (ya sean subclases o no) tienen acceso a los miembros desde los public hasta los sin-modificador. Las subclases de otros paquetes pueden ver los miembros public y a los miembros protected, éstos últimos siempre que sean static ya de no ser así no serán visibles en la subclase (Esto se explica en la siguiente página). El resto del universo de clases (que no sean ni del mismo paquete ni subclases) pueden ver sólo los miembros public.

Jerarquía de las excepciones

Las excepciones en Java son objetos reales, y tienen su propia jerarquía. La clase raíz de ellas es **Throwable**, una subclase de **Object**. Los métodos que se definen dentro de ella serán los encargados de los mensajes de error que estén relacionados con cada una de las excepciones.

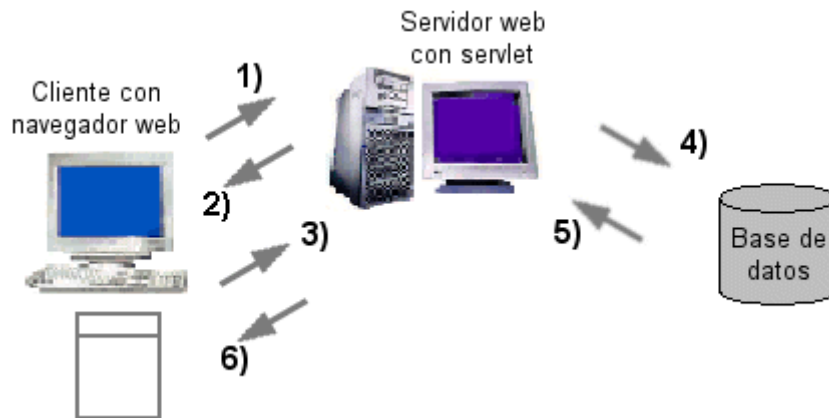


Todos los errores y excepciones son subclases de Throwable, por lo que podrán acceder a sus métodos. Los métodos más utilizados son los siguientes:

- **getMessage()** Se usa para obtener un mensaje de error asociado con una excepción.
- **printStackTrace()** Se utiliza para imprimir el registro del stack donde se ha iniciado la excepción.
- **toString()** Se utiliza para mostrar el nombre de una excepción junto con el mensaje que devuelve getMessage().

Conceptos

Los servlets son módulos que extienden la funcionalidad de los servidores orientados a peticiones y respuestas, tales como los servidores de Web. Por ejemplo, un servlet puede recibir los datos de una forma HTML y guardarlos en una base de datos.



- 1) El cliente hace una petición de un recurso del servidor (en este caso es una forma HTML).
- 2) El servidor responde enviando el archivo HTML y es desplegado en el navegador del cliente.
- 3) El cliente envía sus datos al servidor de Web, donde son recibidos por el servlet.
- 4) El servlet toma los datos y los envía a una base de datos (aquí se realizan sentencias SQL, tales como *select*, *insert*, *update* y *delete*).
- 5) La base de datos responde a la transacción SQL realizada por el servlet.
- 6) El servlet responde al cliente en formato HTML que se despliega en el navegador del cliente.