

ISIL TECH

Java Enterprise Developer



Java Fundamentos

&

Programación Orientada a Objetos

Contenido

CAPÍTULO 1 FUNDAMENTOS DE JAVA	6
INTRODUCCIÓN	6
BREVE HISTORIA	6
¿QUÉ ES JAVA?	7
¿QUÉ APLICACIONES HAY EN EL SDK?	8
COMPILACIÓN: javac	9
EJECUCIÓN DE APLICACIONES: java	10
CONCEPTOS GENERALES	10
Comentarios	10
Identificadores	10
Palabras reservadas	11
Tipos de datos primitivos	11
Enteros	11
Tipos en coma flotante	11
Caracteres: char	11
Booleanos: boolean	12
DECLARACIONES DE VARIABLES	12
CONSTANTES	12
ASIGNACIONES	13
STRINGS	14
OPERADORES	15
ESTRUCTURAS DE CONTROL DE FLUJO	15
Condicionales	16
Bucles	17
EL MÉTODO <i>main</i>	17
TRABAJANDO CON ARREGLOS	18
Definición	18
Arreglos multidimensionales	20
CADENAS	21
Construcción de cadenas.	21
Concatenación	21
Operaciones con cadenas	22
Longitud de una cadena.	22
Ubicar un carácter mediante un índice	22
Extraer una subcadena	22
Convertir a mayúsculas o minúsculas.	22
Eliminar espacios del inicio y el final de la cadena.	22
Ubicar una subcadena desde una ubicación.	22

Comparando dos cadenas.	22
Comparando regiones de una cadena.	23
Obtener cadenas desde las primitivas.	23
Obtener primitivas desde las cadenas.	23
<i>Arreglos de cadenas</i>	23
<i>Método main</i>	24
CAPÍTULO 2 PROGRAMACIÓN ORIENTADA A OBJETOS	25
UN NUEVO PARADIGMA	25
¿QUE ES UN OBJETO?	25
<i>Los objetos realizan operaciones</i>	25
<i>Los objetos tiene valores.</i>	25
<i>Los objetos son una abstracción</i>	26
<i>Encapsulamiento</i>	26
<i>Relaciones entre objetos</i>	26
<i>Asociación entre objetos</i>	27
<i>Composición de objetos.</i>	27
CLASES	27
HERENCIA	28
POLIMORFISMO	28
CLASES EN JAVA	28
<i>Paquetes</i>	28
<i>Modificadores de acceso</i>	29
<i>Creación de objetos</i>	29
<i>La referencia null</i>	30
<i>Asignando referencias</i>	31
MÉTODOS	31
<i>Argumentos</i>	31
<i>Valores de retorno</i>	32
<i>Invocando métodos</i>	32
ENCAPSULAMIENTO	33
<i>Código de una clase.</i>	33
<i>Paso de variables a métodos</i>	34
SOBRECARGA DE MÉTODOS	35
INICIACIÓN DE VARIABLES DE INSTANCIA	35
CONSTRUCTORES	36
<i>La referencia: this</i>	37
VARIABLES DE CLASE	38
MÉTODOS DE CLASE	39

HERENCIA Y POLIMORFISMO	
39 Herencia	
39	
La herencia en Java	
40	
La referencia super	40
Métodos	41
La referencia super	43
Polimorfismo	44
El operador instanceof y cast	46
ATRIBUTOS, MÉTODOS Y CLASES FINAL	47
Variables finales	47
Métodos finales	47
Clases finales	47
EL MÉTODO finalize()	48
CAPÍTULO 3 CLASES ABSTRACTAS E INTERFASES	49
CLASES ABSTRACTAS.....	49
Clases abstractas	49
Métodos abstractos	49
INTERFACES	50
CAPÍTULO 4 UTILIDADES	53
LA CLASE Object.....	53
CONVERTIR DATOS PRIMITIVOS EN REFERENCIAS	53
COLECCIONES	54
Arquitectura	54
Interfaces de colecciones	55
La interface Collection	55
La interface List	56
La interface Map	57
Clases implementadas	57
Definiendo una clase	58
Mostrar los elementos de una colección: ArrayList	58
Evitar objetos duplicados: HashSet	60
Manejar colecciones ordenadas: TreeSet	62
Ordenar y buscar en Colecciones: Collections	65

<i>Ejemplo de clases implementadas: Map</i>	68
Ejemplo de HashMap	68
Ejemplo de TreeMap	69
CAPÍTULO 5 MANEJO DE EXCEPCIONES.	72
¿QUÉ ES UNA EXCEPCIÓN?	72
¿QUÉ ES UN ERROR?	72
¿CUÁL ES LA DIFERENCIA?	72
CARACTERÍSTICAS DEL JAVA	72
SEPARANDO EL MANEJO DE ERRORES	72
EXCEPCIONES	73
<i>Las excepciones no pueden ignorarse</i>	74
Throwable	74
Errores	75
Excepciones no controladas	75
Excepciones controladas	75
¿Qué se puede hacer con una Excepción?	75
EXCEPCIONES NO CONTROLADAS	75
<i>Como capturar y manejar una excepción</i>	76
Capturando una excepción	76
Capturando múltiples excepciones	77
Ejecución del bloque finally	77
<i>Como pasar la excepción al método invocado</i>	78
<i>Como lanzar una excepción</i>	79
COMO CREAR UNA EXCEPCIÓN	79
COMO CAPTURAR UNA EXCEPCIÓN Y LANZAR OTRA DIFERENTE.	80
CAPÍTULO 6 STREAM Y ARCHIVOS.	81
INTRODUCCIÓN	81
<i>java.io.InputStream y java.io.OutputStream</i>	81
<i>java.io.Writer</i>	y
<i>java.io.Reader</i>	82
ENTRADA Y SALIDA ESTANDAR	83
System.out	83
System.in	83
LECTURA DE ARCHIVOS	84
ESCRITURA DE ARCHIVOS	85

LA CLASE File	87
DIRECTORIOS	88
CASO DE ESTUDIO	89
<i>Parte 1</i>	89
<i>Parte 2</i>	91

Capítulo 1

FUNDAMENTOS DE JAVA

INTRODUCCIÓN

Java es un lenguaje de programación orientado a objetos desarrollado por SUN cuya sintaxis está basada en C++, por lo que todos aquellos que estén acostumbrados a trabajar en este lenguaje encontrarán que la migración a Java se produce de forma sencilla y podrán verlo como su evolución natural: un lenguaje con toda la potencia de C++ que elimina todas las estructuras que inducían a confusión y que aumentaban la complejidad del código y que cumple todos los requerimientos actualmente del paradigma de programación orientada a objetos.

BREVE HISTORIA

Java no surgió inicialmente como un lenguaje de programación orientado a la web. Los orígenes se remontan al año 1991 cuando Mosaic (uno de los primeros browsers) o la World Wide Web no eran más que meras ideas interesantes. Los ingenieros de Sun Microsystems estaban desarrollando un lenguaje capaz de ejecutarse sobre productos electrónicos de consumo tales como electrodomésticos.

Simultáneamente James Gosling, el que podría considerarse el padre de Java, estaba trabajando en el desarrollo de una plataforma software de bajo costo e independiente del hardware mediante C++. Por una serie de razones técnicas se decidió crear un nuevo lenguaje, al que se llamó Oak, que debía superar algunas de las deficiencias de C++ tales como problemas relacionados con la herencia múltiple, la conversión automática de tipos, el uso de punteros y la gestión de memoria.

El lenguaje Oak se utilizó en ciertos prototipos de electrónica de consumo pero en un principio no tuvo el éxito esperado dado que la tecnología quizás era demasiado adelantada a su tiempo. No obstante lo positivo de estos primeros intentos fue que se desarrollaron algunos de los elementos precursores de los actuales componentes Java; componentes tales como el sistema de tiempo de ejecución y la API.

En 1994 eclosionó el fenómeno web y Oak fue rebautizado como Java. En un momento de inspiración, sus creadores decidieron utilizar el lenguaje para desarrollar un browser al que se llamó WebRunner, que fue ensayado con éxito, arrancando en ese momento el proyecto Java/HotJava. HotJava fue un browser totalmente programado en Java y capaz así mismo de ejecutar código Java.

A lo largo de 1995 tanto Java, su documentación y su código fuente como HotJava pudieron obtenerse para múltiples plataformas al tiempo que se introducía soporte para Java en la versión 2.0 del navegador Netscape.

La versión beta 1 de Java despertó un inusitado interés y se empezó a trabajar para que Java fuera portable a todos los sistemas operativos existentes. En diciembre de 1995 cuando se dio a conocer la versión beta 2 de Java y Microsoft e IBM dieron a conocer su intención de solicitar licencia para aplicar la tecnología Java, su éxito fue ya inevitable.

El 23 de enero 1996 se publicó oficialmente la versión Java 1.0 que ya se podía obtener descargándola de la web. A principios de 1997 aparece la versión 1.1 mejorando mucho la primera versión. Java 1.2 (Java 2) apareció a finales de 1998 incorporando nuevos elementos. Según Sun esta era la primera versión realmente profesional. En mayo del 2000 se lanza la versión 1.3 del J2SE (Java 2 Standar Edition), luego tenemos la versión 1.4, 1.5 (Java 5.0) y 1.6 (Java 6).

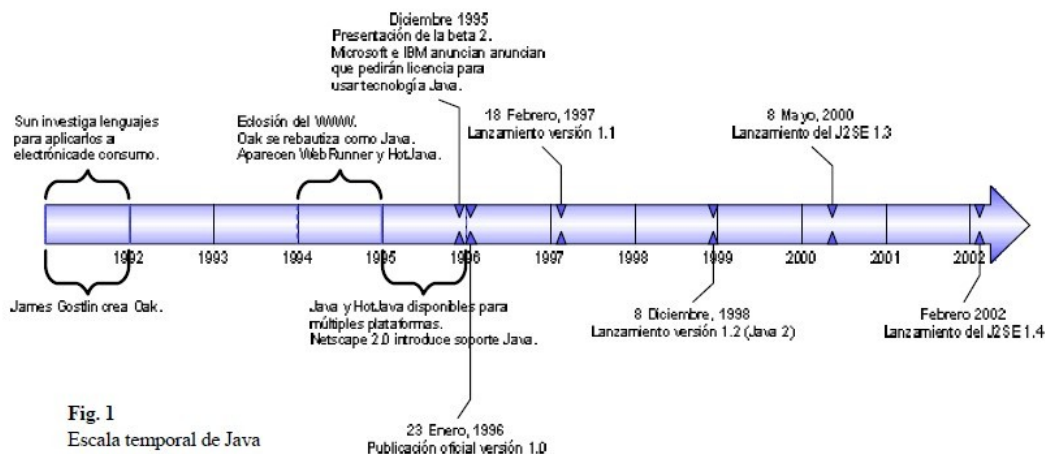


Fig. 1
Escala temporal de Java

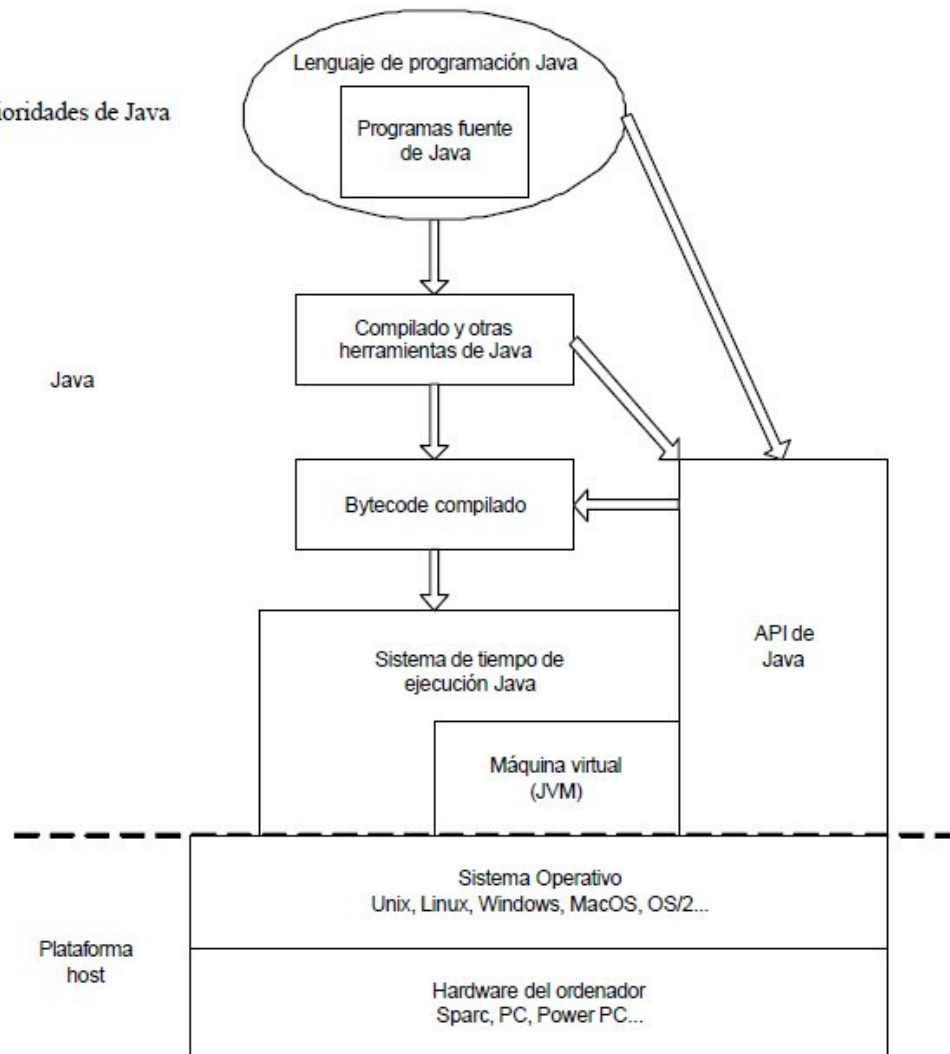
¿QUÉ ES JAVA?

Java no es sólo un lenguaje de programación, Java es además un sistema de tiempo de ejecución, un juego de herramientas de desarrollo y una interfaz de programación de aplicaciones (API). Todos estos elementos así como las relaciones establecidas entre ellos se esquematizan en la figura 2.

El desarrollador de software escribe programas en el lenguaje Java que emplean paquetes de software predefinidos en la API. Luego compila sus programas mediante el compilador Java y el

resultado de todo ello es lo que se denomina bytecode compilado. Este bytecode es un archivo independiente de la plataforma que puede ser ejecutado por máquina virtual Java. La máquina virtual puede considerarse como un microprocesador que se apoya encima de la arquitectura concreta en la que se ejecuta, interactuando con el sistema operativo y el hardware, la máquina virtual es por tanto dependiente de la plataforma del host pero no así el bytecode. Necesitaremos tantas máquinas virtuales como plataformas posibles pero el mismo bytecode podrá ejecutarse sin modificación alguna sobre todas ellas.

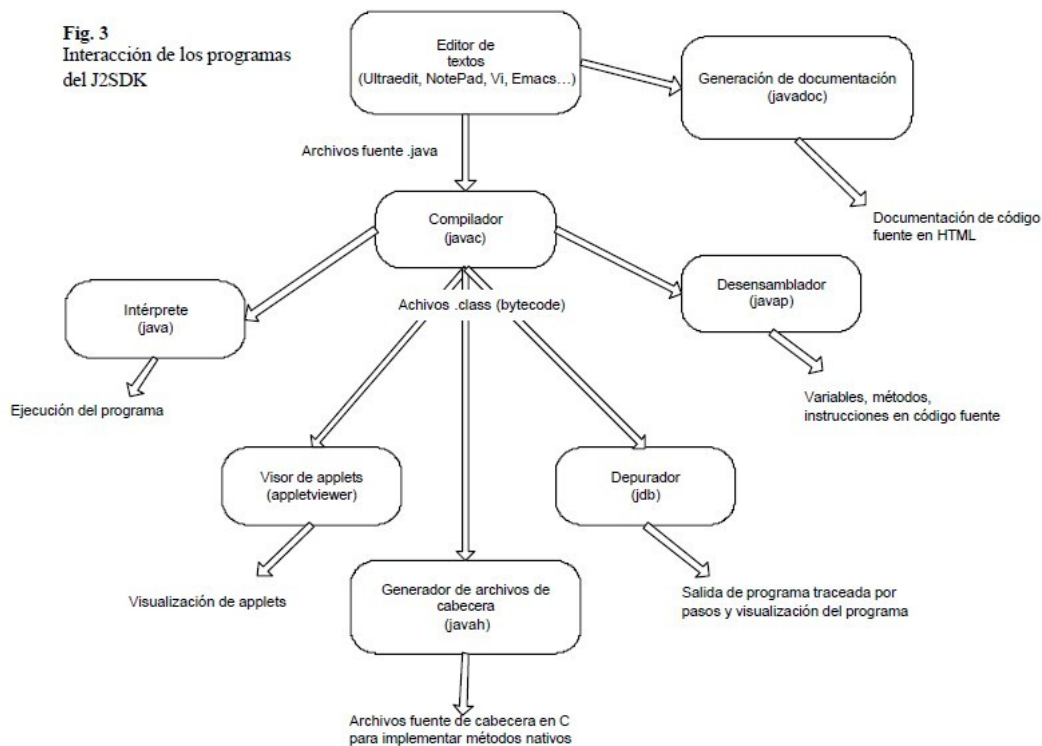
Fig. 2
Las interioridades de Java



¿QUÉ APLICACIONES HAY EN EL SDK?

El entorno de SDK es de tipo línea de comando. Constituye todas las herramientas necesarias para desarrollar aplicaciones Java, así pues consta del compilador/linkador, del intérprete de aplicaciones, de un debugger, de un visor de applets y de un programa de generación automática de documentación, entre otros. Existen entornos de desarrollo Java ajenos a Sun como pueden ser JCreator, NetBeans, Eclipse, JDeveloper entre otros.

Fig. 3
Interacción de los programas
del J2SDK



COMPILACIÓN: javac

Para generar el archivo .class, que es el bytecode que recibe el entorno de ejecución, se usa el compilador javac que recibe el código fuente en un archivo con extensión .java. Para compilar una aplicación basta con compilar la clase principal, aquella donde está el método **main**, y después de forma automática se va llamando para compilar todas las clases que necesite. Estas clases deben ser accesibles a través de la variable de entorno CLASSPATH.

Vamos a hacer un pequeño ejemplo para probar el compilador. Abrir el NotePad y copiar las siguientes líneas y a continuación guardar el archivo como *HolaMundo.java*

```
public class HolaMundo {

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

Después en la línea de comando ejecuta el compilador de la siguiente manera:

```
> javac HolaMundo.java [Enter]
```

EJECUCIÓN DE APLICACIONES: java

Ahora que ya hemos compilado nuestro flamante *HolaMundo.java* y estamos ansiosos por ver el resultado de nuestro primer programa, usaremos el intérprete Java para ejecutarlo.

Nada más sencillo que teclear el comando:

```
> java HolaMundo
```

CONCEPTOS GENERALES

En Java todos los programas se construyen a partir de clases, dentro de esas clases encontramos declaraciones de variables (instrucciones atómicas) y procedimientos o funciones (conjuntos de instrucciones).

Comentarios

Los comentarios son cadenas de texto que el programa usa para entender y hacer inteligible su código a otros. Los comentarios son ignorados por el compilador. Java tiene tres tipos de comentarios como los que ilustraremos en el siguiente programa:

```
/* Clasico HolaMundo.java */
public class HolaMundo {

    /*
     ** Clásico Programa "Hola Mundo!"
     */
    public static void main(String[] args)
    { // Escribe por la salida estándar
        System.out.println("Hola mundo!");
    }

}
```

Identificadores

Los identificadores se usan para nombrar y referirnos a las entidades del lenguaje Java, entidades tales como clases, variables o métodos. Java es sensible a la diferenciación de mayúsculas y minúsculas, así pues una variable *contador* no es la misma que otra *Contador* y pueden, aunque no sea aconsejable, coexistir sin problemas. Un identificador tampoco puede ser igual a una palabra reservada.

Palabras reservadas

Las palabras reservadas por el lenguaje Java son las que se muestran a continuación. Al final de este curso conoceremos el significado de la mayoría de ellas:

abstract, default, if, this, implements, package, throw, boolean, double, import, private, break, else, byte, extends, instanceof, public, try, case, final, int, cast, finally, return, void

Tipos de datos primitivos

Java es un lenguaje fuertemente tipificado, lo que quiere decir toda variable debe ser declarada de un tipo. De los ocho tipos primitivos, hay seis numéricos (cuatro enteros y dos en coma flotante), otro es el carácter y el último es el booleano. La portabilidad de Java garantiza todos los tipos tendrán el mismo tamaño independientemente de la plataforma.

Enteros

Tabla 1. Tipos enteros

Tipo	Tamaño (en bytes)
int	4
short	2
long	8
byte	1

Tipos en coma flotante

Tabla 2. Tipos en coma flotante

Tipo	Tamaño (en bytes)	Cifras significativas
float	4	7
double	8	15

Caracteres: char

Los caracteres se almacenan en el tipo *char*. Java utiliza el código Unicode de 16 bits (diferenciándose de la mayor parte de lenguajes clásicos, como C/C++, que utilizan ASCII de 8 bits). Unicode es un superconjunto de ASCII, es decir ASCII está contenido en Unicode, pero este último proporciona muchísimos caracteres más (los dos bytes permiten tener $2^{16}=65.536$ caracteres diferentes frente a los $2^8=256$ caracteres del ASCII extendido). Los caracteres se encierran entre comillas sencillas (') y no dobles ("). Los caracteres de escape, al igual que en C/C++, se preceden de la barra invertida (\). Tenemos los siguientes códigos de escape:

Tabla 4. Códigos de escape

Código de escape	Carácter	Cifra hexadecimal
\b	retroceso	\u0008
\t	tabulación	\u0009
\n	avance de línea	\u000a
\f	avance de papel	\u0006
\r	retroceso de carro	\u000d
\"	comillas normales	\u0022
\'	comillas sencillas	\u0027
\\	barra invertida	\u005c

Booleanos: boolean

A diferencia de C/C++ los valores cierto y falso que se utilizan en expresiones lógicas, no se representan con un entero que toma los valores 1 y 0, sino que existe un tipo destinado a tal efecto, el tipo *boolean* que toma valores *true* y *false*.

DECLARACIONES DE VARIABLES

Una variable es una estructura que se referencia mediante un identificador que nos sirve para almacenar los valores que usamos en nuestra aplicación. Para usar una variable debemos declararla previamente de un tipo. Veamos algunos ejemplos:

```
boolean b; int numero  
float  
decimal=43.32e3f int  
contador=0; char  
c='a';
```

Como vemos la forma de declarar variables tiene la siguiente sintaxis:

```
tipoVariable identificadorVariable [= valorInicial];
```

Donde primero indicamos el tipo al que pertenece la variable, a continuación el identificador o nombre de la variable, opcionalmente un valor inicial y finalmente acabamos con un punto y coma (;). A diferencia de otros lenguajes, Java permite declarar las variables en cualquier parte de un bloque de instrucciones (no tiene porqué ser al principio).

CONSTANTES

El concepto de constante no puede entenderse estrictamente en el paradigma de orientación a objetos y por ello en Java no existen las constantes propiamente dichas. Esto se explicará con más detalle cuando pasemos al capítulo de orientación a objetos, por el momento diremos que lo más parecido que tenemos a una constante es una *variable de clase no modificable* que se declara de la siguiente manera:

```
static final tipo identificador = valor;
```

Por ejemplo:

```
static final float pi = 3.141592F;
```

ASIGNACIONES

El operador de asignación, como ya hemos visto en las inicializaciones de variables, es el "=", por ejemplo:

```
bool
javaEsGenial;
javaEsGenial=true
; int edad;
edad=22;
```

Es posible hacer asignaciones entre tipos numéricos diferentes. Esto es posible de una manera implícita en el sentido de las flechas (no puede haber pérdida de información):

```
byte -> short -> int -> long -> float -> double
```

También puede hacerse en forma explícita cuando tiene ser al revés mediante un *casting*, siendo responsabilidad del programador la posible pérdida de información ya que Java no producirá ningún error. Por ejemplo:

```
float radio; radio = 5; // no es necesaria la conversión explícita
de int a float int perimetro;
perimetro = radio * 2 * PI; // error! no se puede guardar en un int
// el resultado de una operación con float perimetro = (int)
(radio * 2 * PI) // Ahora es correcto porque el
// casting fuerza la conversión
```

El *casting* de carácter a un tipo numérico también es posible, aunque no demasiado recomendable. En cambio no es posible la conversión entre booleanos y tipos numéricos.

STRINGS

Ante todo dejar claro que los strings no son tipos primitivos en Java. Son una clase implementada en la biblioteca estándar aunque con ciertas funcionalidades que hacen que su uso sea comparable al de los tipos primitivos en ciertos aspectos. Podemos declarar e inicializar *strings* como:

```
String st = ""; // String vacío
String st1 = "Hola";
String st2 = "cómo estás?";
String st3 = st1 + ", " + st2; // st3 vale "Hola, cómo estás?"
```

La cadena `st3` contiene una concatenación de las cadenas `st1`, un espacio en blanco y `st2`. Para concatenar Strings basta con operarlos mediante el símbolo "+". Cuando concatenamos un String con algo que no lo es, este valor es automáticamente convertido a String.

```
String st = "numero: " + 3; // st vale "numero: 3"
```

Además de la concatenación, también podemos usar otras operaciones de la clase `String` para trabajar con ellos. Algunos ejemplos pueden ser (veremos la forma general de llamar operaciones que actúan sobre objetos en el próximo capítulo):

```
String st = "String de ejemplo";  
String st2 = "String de ejemplo";
```

- ③ **st.length()** Devuelve la longitud de la cadena st.
- ③ **st.substring(a, b)** Devuelve la subcadena a partir de la posición a (incluida) hasta la b (no incluida).
- ③ **st.charAt(n)** Devuelve el carácter en la posición n.
- ③ **st.equals(st2)** Devuelve un booleano que evalúa a cierto si los dos String st y st2 son iguales, en este caso valdrá true.

Cabe destacar que cuando consideramos posiciones dentro de un String, si la longitud es n, las posiciones válidas van de la 0 a la n-1. Otro aspecto a tener en cuenta es que una variable de tipo String es un puntero a una zona de memoria (ya veremos que ocurre en general con todos los objetos). Así pues, (si no hay ciertas optimizaciones por parte del compilador) la comparación `st == st2` evalúa a falso.

OPERADORES

Java define operadores aritméticos, relacionales, lógicos de manipulación de bits, de conversión de tipo, de clase, de selección y de asignación. No debe preocuparse si ahora no queda claro el significado de alguno de ellos

Tipo de operador	Operador	Descripción	Ejemplo
Aritmético	+	Suma	a+b
	-	Resta	a-b
	*	Multiplicación	a*b
	/	División	a/b
	%	Módulo	a%b
Relacional	>	Mayor que	a>b
	<	Menor que	a=	Mayor o igual que	a>=b
	<=	Menor o igual que	a<=b
	!=	Diferente	a!=b
	==	Igual	a==b
Lógico	!	No	!a
	&&	Y	a&&b
		O	a o
Asignación	~	Complemento	~a
	&	Y bit a bit	a&b
		O bit a bit	a b
	=	Asignación	a=b
	++	Incremento y asignación	a++
	--	Decremento y asignación	a--
	+=	Suma y asignación	a+=b
	-=	Resta y asignación	a-=b
	=	Multiplicación y asignación	a=b
	/=	División y asignación	a/=b
	%=	Módulo y asignación	a%=b
	=	O y asignación	a =b
	&=	Y y asignación	a&=b
	^=	O exclusiva y asignación	a^=b
	<<=	Desplazamiento a la izda. y asignación	a<<=b
	>>=	Desplazamiento a la dcha. y asignación	a>>=b
	>>>=	Desplazamiento a la dcha. rellenando ceros y asignación	a>>>=b
Conversión de tipo (casting)	(tipo)	Convertir tipo	(char)b
Instancia	instanceof	¿Es instancia de clase	a instanceof b

ESTRUCTURAS DE CONTROL DE FLUJO

Antes de ingresar a fondo con las estructuras de control de flujo debemos tener claro qué es un *bloque*. Los bloques consisten en secuencias de declaraciones e instrucciones de variables locales. Se escribe como sigue:

```
{
    bloqueCuerpo }

```

Donde `bloqueCuerpo` es una secuencia de declaraciones e instrucciones de variables locales. Un bloque también puede consistir en una instrucción única sin tener que estar entre llaves. La sentencia vacía consiste en un solo punto y coma (;) y no realiza proceso alguno.

Condicionales

Permiten desviar el flujo de ejecución por una rama u otra dependiendo de la evaluación de una condición. Existen dos estructuras condicionales, el *if* :

```
if (condición1)
{bloque1}
[else if (condición2)
{bloque2}
...]
[else
{bloqueN}]
```

Que puede tener una o más ramas que evalúen condiciones y opcionalmente un *else* cuyo bloque se ejecutará sólo si no se ha cumplido ninguna de las condiciones anteriores. Toda condición (en estas estructura y en adelante) debe evaluarse a un valor booleano no pudiéndose utilizar valores enteros como en C/C++. La otra estructura condicional es el *switch* que permite elegir un bloque en función del valor de una variable de referencia:

```
switch (variableReferencia)
{
case valor1:
{bloque1}
[case valor2:
{bloque2} ...]
[default:
{bloqueN}
]
}
```

variableReferencia sólo puede ser una expresión que evalúe a los tipos primitivos enteros o char. Se evalúa la expresión y sucesivamente se va comprobando que coincida con alguno de los valores, en caso de que así sea se ejecuta el bloque correspondiente. Hay que tener en cuenta que la última instrucción de cada bloque debe ser un *break* porque en caso contrario el flujo del programa seguiría por la primera instrucción del siguiente bloque y así sucesivamente (esto puede ser útil en alguna circunstancia pero no es deseable generalmente). Si *variableReferencia* no coincide con ninguna de los valores del case, se ejecutará, caso de existir, el bloque correspondiente al *default*; en caso contrario, simplemente se seguiría con la próxima instrucción después del switch.

Bucles

Los bucles son estructuras iterativas que ejecutan un cierto bucle mientras se da una cierta condición. Java dispone de tres tipos de bucles. Tenemos en primer lugar el *while*:

```
while (condicion){
    bloque
}
```

Que ejecutará el código del bloque mientras condición se evalúe a cierto. La sentencia *do ..while*:


```
do{    bl
oque

}while (condicion)
```

Es muy similar sólo que garantiza que el bloque se ejecutará al menos una vez, ya que la evaluación de la condición se produce después de la ejecución del bloque. Finalmente tenemos el clásico *for*:

```
for(inicializacion;condicion;incremento)
{   bloqu
e }
```

Que de forma análoga permite ejecutar un bloque mientras se da una cierta condición. El *for* tiene de particular que la inicialización (generalmente un contador) y el incremento queda encapsulado en la misma estructura. Es útil para recorrer estructuras de datos secuenciales. La palabra reservada *break* en cualquiera de los tres bucles fuerza la salida del bucle pasándose a ejecutar la siguiente instrucción después del mismo. La sentencia *continue* fuerza el abandono de la iteración actual haciendo que se ejecute la siguiente; también para todos los tipos de bucle.

EL MÉTODO *main*

El método *main* es la primera operación que se ejecuta en un programa Java, el que se encarga de poner todo en marcha, y sólo puede haber uno. Su declaración es como sigue: `public static void main(String[]args)` y siempre debe ser exactamente así, ya que si modificamos (u olvidamos) alguno de los modificadores Java no lo interpreta como el método *main* y cuando intentáramos ejecutar la aplicación obtendríamos un error que precisamente nos diría esto más o menos: “no puedo ejecutar el programa porque no existe un método *main*”. Como puede observar, el método *main* siempre recibe un parámetro que es un array de *String*.

Este array contiene los parámetros que opcionalmente le hemos podido pasar por la línea de comandos. ¿Qué significa esto? Habitualmente cuando ejecutamos un programa lo hacemos de la siguiente manera:

```
> java HolaMundo [Enter]
```

Pero también habríamos podido ejecutarlo pasándole al programa información adicional:

```
> java HolaMundo p1 p2 p3 [Enter]
```

En este caso hemos llamado a nuestra aplicación y le hemos pasado tres parámetros, *p1*, *p2* y *p3*, que se almacenan precisamente en el parámetro *args* del método *main*. Para acceder a ellos nada más fácil que hacer:

```
public static void main(String[]args)
{   ...
    String s = args[0]; ...
}
```

Tener en cuenta que *args* es un array de *String*'s por lo que aunque nosotros pasemos desde la línea de parámetros números, estos son interpretados como *String* y si queremos el *int*, por ejemplo, correspondiente a esa representación tendremos que aplicar el método de conversión adecuado.

El número de parámetros que se han pasado por la línea de comandos puede consultarse mirando la longitud del array:

```
int numParam = args.length;
```

TRABAJANDO CON ARREGLOS

Se estudiara el uso de arreglos y cadenas en la solución de problemas en Java.

Definición

Un arreglo es una colección de variables del mismo tipo. La longitud de un arreglo es fija cuando se crea.

Elemento 0	1
Elemento 1	2
Elemento 2	4
Elemento 3	8

Pasos de la creación de arreglos de primitivas

<p>Se declara el arreglo</p> <p>Inicialmente la variable referencia un valor nulo</p> <pre>int[] potencias; //forma más usada int potencias[];</pre>	<p>potencias-> null</p>				
<p>Se crea el arreglo</p> <p>Se requiere la longitud del arreglo</p> <pre>potencias = new int[4];</pre> <p>En caso de variables primitivas se inician en 0 o false. Las primitivas no se pueden operar con el valor null.</p> <p>En caso de variables referencia se inician en null. No referencian a ningún objeto.</p>	<p>potencias-></p> <table><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr></table>	0	0	0	0
0					
0					
0					
0					

<p>Se inicia los valores del arreglo.</p> <p>Se asignan valores elemento por elemento</p> <pre>potencias[0] = 1; potencias[1] = 2; potencias[2] = 4; potencias[3] = 8;</pre>	<table><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>4</td></tr><tr><td>8</td></tr></table> <p>potencias-></p>	1	2	4	8
1					
2					
4					
8					
<p>Los arreglos pueden ser creados e iniciados al mismo tiempo</p> <pre>int[] potencias = {1,2,4,8};</pre>					

Los arreglos son muy usados para buscar datos, especialmente si se conocen sus valores cuando se crean.

```
int[] diasMesesAnioBisiesto = {31,29,31,30,31,30,31,31,30,31,30,31};
```

Arreglos multidimensionales

Se trata de arreglos de arreglos y se declara de la siguiente forma:

```
tipo[][] nombreArreglo = new tipo[cantidadFilas][cantidadColumnas];
```

Ejemplo:

```
int[][] tabla = new int[4][2];
tabla[0][0] = 1;
tabla[0][1] = 7;
tabla[1][0] = 3;
tabla[1][1] = 5;
tabla[2][0] = 4;
tabla[2][1] = 8;
```

tabla->	[0]	tabla[0]	->	1	7
	[1]	tabla[1]	->	3	5
	[2]	tabla[2]	->	4	8
	[3]	tabla[3]	->	0	0

Definir una matriz de enteros colocando como valores la suma de su número de fila y columna en la matriz

```
{ int n=
10; int p=
20;
```

```
double[][] m = new double[n][p];
for (int i= 0; i<n; i++)    for
(int j= 0; j<p; j++)
    m[i][j]= i+j;
}
```

CADENAS

Una cadena es una secuencia de caracteres. La librería String (o clase String) se usa para definir todas las cadenas en Java. Las cadenas se delimitan con comillas dobles.

```
System.out.println("Hola Mundo.");
String camara = "Camara";
String luces = camara + " Accion";
String vacio = "";
```

Construcción de cadenas.

También se puede usar la siguiente sintaxis para construir cadenas.

```
// Con una constante
String nombreEmpleado = new String("Jennifer Lopez");

// Con una cadena vacía
String inicio = new String();

// Copiando una cadena
String copiaEmpleado = new String(nombreEmpleado);

// Con un arreglo de char
char[] vocales = {'a','e','i','o','u'};
String cadenaVocales = new String(vocales);
```

Concatenación

Para concatenar cadenas puede usar lo siguiente:

```
// Usando el operador +
System.out.println(" Nombre = " + nombreEmpleado );

// Puede concatenar primitivas y cadenas.
int edad = 22;
System.out.println(" Edad = " + edad );

// Mediante la función concat()
String nombre = " Gianella ";
String apellidos = " Neyra";
String nombreCompleto = nombre.concat(apellidos);
```

Operaciones con cadenas

Longitud de una cadena.

```
String nombre = " Angie Jibaja";  
int longitud = nombre.length();
```

Ubicar un carácter mediante un índice

```
String nombre = " Adrea Montenegro";  
char c = nombre.charAt(0);
```

Extraer una subcadena

```
//          01234567890123  
String nombre = "Stephanie Cayo";  
//          12345678901234 String subCadena =  
nombre.substring(2,6);
```

Convertir a mayúsculas o minúsculas.

```
String titulo = "Segunda Fundacion";  
String mayusculas = titulo.toUpperCase();  
String minusculas = titulo.toLowerCase();
```

Eliminar espacios del inicio y el final de la cadena.

```
String autor = " Isaac Asimov      ";  
String resaltar = "*" + autor.trim() + "*";
```

Ubicar una subcadena desde una ubicación.

```
//          0123456789012 String  
alcalde = "Salvor Hardin"; int  
ubicacion1 = alcalde.indexOf("vor");  
int ubicacion2 =  
alcalde.indexOf("r",7); int ubicacion3  
= alcalde.indexOf("h");
```

Comparando dos cadenas.

```
String password = "FGHPUW"; if  
password.equals("fgHPUw")  
System.out.println("Correcto!");  
else  
    System.out.println("Error!");
```

```
String password = "FGHPUW"; if password.equalsIgnoreCase("fgHPUw")
System.out.println("Correcto!"); else
    System.out.println("Error!");
```

Comparando regiones de una cadena.

```
String url = "http://www.isil.pe";
if (url.endsWith(".pe"))
    System.out.println("Pagina Nacional"); else
    System.out.println("Pagina Extranjera");
String parametro = "ip = 192.100.51.2";
if (parametro.startsWith("ip"))
    System.out.println("La direccion " + parametro); else
    System.out.println("El parámetro no es una ip");
```

Obtener cadenas desde las primitivas.

Se utilizan funciones de la librería String.

```
String seven = String.valueOf(7); String unoPuntoUno =
String.valueOf(1.1); float pi = 3.141592;
String piString = String.valueOf(pi);
```

Obtener primitivas desde las cadenas.

Para esto se utilizan funciones de las librerías Integer y Float.

```
String alfa = "1977";
int alfaInteger = Integer.parseInt(alfa);

String beta = "19.77"; float betaFloat = Float.parseFloat(beta);
```

Arreglos de cadenas

Un arreglo de cadenas también sigue los pasos de creación de arreglos.

Declaracion String [] categorías;

Creación categorias = new String[3];

Iniciación. categorias[0] = "Drama";

```
// Creando una arreglo de cadenas vacias.
String [] arreglo = new String [4]; for
( int i = 0; i < arreglo.length; i++ )
{ arreglo[i] = new String();
```

```
}

// Creando e iniciando un arreglo.
String [] categorias = {"Drama", "Comedia", "Accion"};

// Accesando los elementos del arreglo
String [] categorias = {"Drama", "Comedia", "Accion"};
System.out.println(" Comedia = " + categorias[1].length() );
```

Método main

main() cuenta con un único parámetro que es un arreglo de cadenas. Este arreglo de cadenas son los parámetros de la línea de comandos.

```
C:\> java Eco Hola Mundo
```

El siguiente es el programa Java:

```
// Eco.java

package ejercicios;

public class Eco {

    public static void main (String [] args) {
        if (args.length != 2)
            System.out.println("Uso: java Eco Arg1 Arg2");
        else
            System.out.println(args[0] + " " + args[1]);
    }
}
```

Capítulo 2

PROGRAMACIÓN ORIENTADA A OBJETOS

UN NUEVO PARADIGMA

OO es un nuevo paradigma de diseño y programación.

OO se basa en modelar objetos del mundo real.

OO crea componentes de software reusables.

Los objetos contienen en si mismo información (atributos y datos) y comportamiento (métodos).

¿QUE ES UN OBJETO?

Definición filosófica: Es una entidad que se puede reconocer.

Para la tecnología de objetos: Es una abstracción de un objeto del mundo real.

En términos de negocios: Es una entidad relevante al dominio del negocio.

En términos de software: Es una estructura que asocia datos y funciones.

Algunos ejemplos de objetos en POO son: Cliente, Factura, Contrato, Película. Un Cliente tiene un nombre, dirección, crédito (atributo). Un Cliente podría alquilar una película, pagar una factura, devolver una película (comportamiento).

Los objetos realizan operaciones

Un objeto es útil si tiene alguna función o comportamiento en el sistema.

Cada comportamiento se denomina operación.

Objeto: Mi lapicero azul	Objeto: El cajero automático P5
Operación: Escribir	Operación: Entregar dinero.

Los objetos tiene valores.

Los objetos conocen cual es su estado actual. Cada conocimiento del objeto se denomina atributo.

Objeto: Mi lapicero azul	Objeto: El cajero automático P5
Atributo: Cantidad de tinta	Atributo: Disponible en soles.

Los objetos son una abstracción

Todo depende del contexto. Cuando se modela un objeto, solo se requiere modelar las operaciones y atributos que son relevantes para el problema

Objeto: Mi lapicero azul

Operación: Apuntar una pizarra.

Atributos: Longitud, Marca.

Encapsulamiento

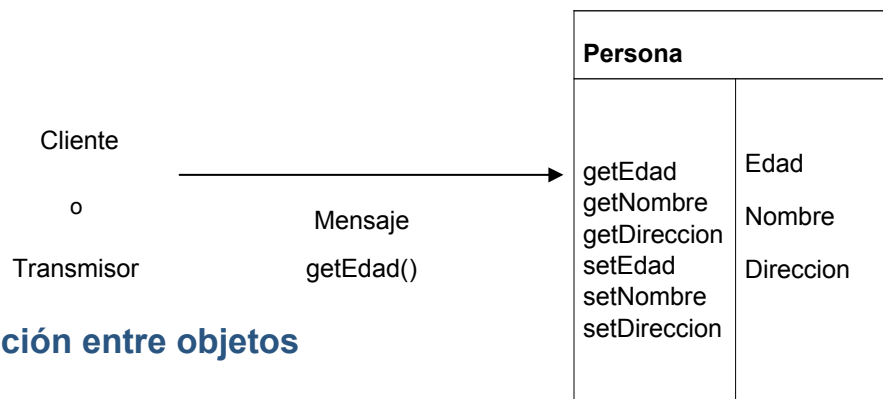
El encapsulamiento oculta como las cosas funcionan dentro de un objeto. Solo nos comunicamos con el objeto a través sus métodos. Los métodos son una interfaz que permite ignorar como están

implementados. No es posible evadir el encapsulamiento en OO El cajero automático P5 es un objeto que entrega dinero.

El cajero encapsula todas las operaciones a los tarjetahabientes.

Relaciones entre objetos

Los objetos se comunican unos con otros enviando mensajes. El trasmisor del mensaje pide que el receptor realice una operación. El receptor ejecuta el método correspondiente. En programación estructurada se invocan funciones o procedimientos. En OO se envía un mensaje a un objeto antes que invocarse un procedimiento. Los métodos presentan polimorfismo (varias formas).



Asociación entre objetos

Para que un objeto envíe un mensaje a otro, el receptor debe ser visible para el transmisor. Esta visibilidad se da a través de enlaces (métodos públicos). Un objeto envía mensajes a otro invocando sus métodos.



Composición de objetos.

Los objetos están compuestos de otros objetos. Los objetos son partes de otros objetos. Esta relación entre objetos se conoce como Agregación

El Banco de la Nación es un objeto	El cajero automático P5 es un objeto. El cajero es del Banco de la Nación.	El cajero automático P5 esta compuesto por objetos como: El teclado El dispensador de billetes El lector de la tarjeta
------------------------------------	---	---

CLASES





Una clase es un molde para crear objetos. Para definir una clase se tiene que indicar las operaciones y atributos. Los objetos son instancias de la clase. Cuando se cree un cajero automático P3 no se requiere indicar sus operaciones y atributos.

Solo se requiere indicar a que clase pertenece.

Clase	Cliente	Película
Atributos	int edad	String titulo
Métodos	String nombre String dirección cambiarDireccion()	double precio String categoria cambiarPrecio()

HERENCIA

Entre diferentes clases puede haber similitudes. La herencia es una relación entre clases donde una es padre de otra. Las propiedades comunes definen la superclase. Clase padre. Las subclases heredan estas propiedades. Clase hija. Un objeto de una clase hija es un-tipo-de una superclase. Un Helicóptero es un tipo de Nave Aérea.

Nave Aérea (padre)			
			
Jumbo	Helicóptero	Globo	Dirigible

POLIMORFISMO

Significa que la misma operación se realiza en las clases de diferente forma. Estas operaciones tienen el mismo significado, comportamiento. Internamente cada operación se realiza de diferente forma.

Abordar pasajeros

 <p>Barco</p>	 <p>Tren</p>	 <p>Helicóptero</p>
--	---	--

CLASES EN JAVA

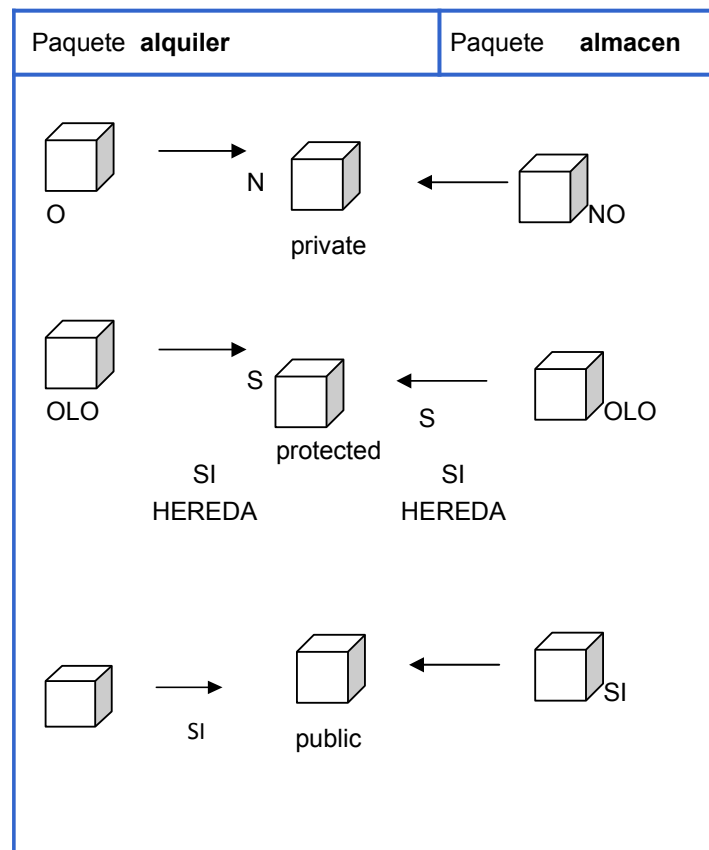
Una clase es un molde para crear múltiples objetos que encapsulan datos y comportamiento.

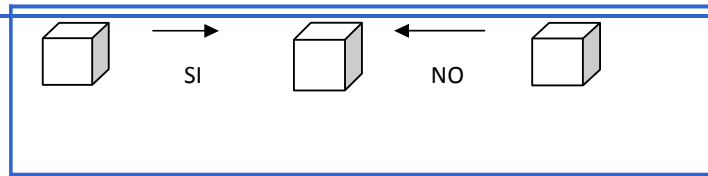
Paquetes

Un paquete es un contenedor (agrupador) de clases que están relacionadas lógicamente. Un paquete tiene sus clases en un mismo directorio. Varias clases pueden tener el mismo nombre pero en diferente paquete.

Modificadores de acceso

Java controla el acceso a las variables y métodos a través de modificadores de acceso como: **private**, **public** y **protected**. Un elemento público puede invocarse en cualquier clase. Un elemento sin modificador solo se puede invocar desde la misma clase. Un elemento protegido solo se invocar en clases heredadas. Un elemento privado no puede ser invocado por otra clase.





Creación de objetos

Cada objeto es una instancia de alguna clase.

Película
private String titulo; private String tipo;
Public void mostrarDetalles() public void obtenerTitulo()



Los objetos se crean con el operador new.

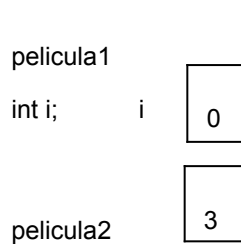
```
Película pelicula1 = new Película();
Película pelicula2 = new Película();
```

El operador new realiza las siguientes acciones:

- Separa memoria para el nuevo objeto
- Invoca el método de inicio de la clase llamado constructor.
- Retorna la referencia a un nuevo objeto.

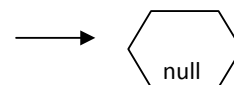


Las variables primitivas almacenan valores.



Los objetos almacenan referencias

Película pelicula1;



Película pelicula2 = new



```
Pelicula(); int j = 3; j
```

La referencia null

- Los objetos inician en null `Pelicula pelicula1 = null;`
- Se puede comparar un objeto con null `if (pelicula1 == null)`
- Se puede liberar la referencia con null.
`pelicula1 = new Pelicula(); pelicula1 = null;`

Asignando referencias

Se puede asignar una variable referencia a otra resultado en dos referencias al mismo objeto.

```
Pelicula pelicula1 = new Pelicula("Betty Blue");  
  
Pelicula pelicula2 = pelicula1;
```

Las variables de instancia se declaran en la clase. Estos son atributos de la clase.

```
public class Pelicula {  
  
    public String titulo; public String tipo;  
  
}
```

Las variables públicas de instancia se acceden con el operador punto.

```
Pelicula pelicula1 = new Pelicula(); pelicula1.titulo  
= "Kramer vs Kramer"; if (pelicula1.titulo.equals("El  
planeta de los simios") pelicula1.tipo = "Ciencia  
Ficcion";
```

MÉTODOS

Un método es equivalente a una función o subrutina de otros lenguajes. Los métodos solo se definen dentro de una clase.

```
modificador tipoDeRetorno nombreMetodo (ListaDeArgumentos) {  
  
    // desarrollo del método;
```

```
}  
}
```

Argumentos

En la definición del método se indica el tipo y el nombre de los argumentos del método.

```
public void setTipo (String nuevoTipo) {  
    tipo = nuevoTipo; }  
}
```

Si el método tiene varios argumentos, estos se separan por comas.

```
public void setDatos (String nuevoTitulo, String nuevoTipo)  
{    tipo = nuevoTipo;    titulo = nuevoTitulo; }  
}
```

Si el método no tiene argumentos, se deja solo los paréntesis

```
public void mostrarDetalles() {  
    System.out.println("El titulo es " + titulo);  
    System.out.println("El tipo es " + tipo); }  
}
```

Valores de retorno

Se usa la sentencia `return` para salir del método retornando un valor. No se requiere `return` si el tipo de retorno es `void`.

```
public class pelicula  
{    private String  
    tipo;    //...  
    public String obtenerTipo () {  
        return tipo;  
    }  
}
```

Invocando métodos

Se utiliza el operador punto para invocar el método de una clase, si el método es de la misma clase, no se requiere el calificador de la clase. **Pelicula.java**

```
public class Pelicula {    private String titulo,tipo;    //...  
    public String getTipo () {  
        return tipo;  
    }  
    public void setTipo (String nuevoTipo) {  
        tipo = nuevoTipo;  
    }  
}
```

PruebaPeliculas.java

```
public class PruebaPeliculas {    public static
```

```
void main (String[] args) {      Pelicula
pelicula1 = new Pelicula();
pelicula1.setTipo("Comedia");      if
(pelicula1.getTipo().equals("Drama")) {
    System.out.println("La película es un drama");
}
else {
    System.out.println("La película no es un drama");
}
}
}
```

ENCAPSULAMIENTO

Las variables de instancia de una clase deberían ser declaradas como privadas. Solo un método debería acceder a las variables privadas. No se debe acceder a las variables de instancia directamente, sino a través de un método.

```
Pelicula pelicula1 = new Pelicula();
if( pelicula1.titulo.equals("Los doce del patibulo") )
{  pelicula1.setTipo("Acción"); }
```

¿Cuál de estas líneas no cumple con el encapsulamiento?

Código de una clase.

Pelicula.java

```
public class Pelicula {

    private String titulo;
private String tipo;

    public void setTitulo (String nuevoTitulo) {
titulo = nuevoTitulo;
}
```

```
    }  
    public void setTipo (String nuevoTipo) {  
tipo = nuevoTipo;  
    }  
    public String getTitulo () {  
return titulo;  
    }  
    public String getTipo () {  
return tipo;  
    }  
    public String toString () {      return  
"titulo "+titulo+" tipo "+tipo;  
    }  
}
```

Paso de variables a métodos

Cuando el argumento es de tipo primitivo, se genera una copia de la variable para el método.

TestPrimitivas.java

```
public class TestPrimitivas {  
  
    public static void main (String[] args) {  
        int numero = 150;    unMetodo(numero);  
        System.out.println(numero);  
    }  
  
    private static void unMetodo(int argumento)  
{    if (argumento < 0 || argumento > 100) {  
        argumento = 0;  
        System.out.println(argumento);  
    }  
}  
}
```

Cuando se pasa como argumento un objeto referencia, no se genera copia. El argumento referencia al objeto original.

TestReferencias.java

```
public class TestReferencias {  
  
    public static void main (String[] args)  
{    Pelicula pelicula1 = new Pelicula();
```



```
pelicula1.setTitulo("El Resplandor");
pelicula1.setTipo("Drama");    unMetodo(pelicula1);
    System.out.println(pelicula1.getTipo());
}
public static void unMetodo(Pelicula referencia)
{
    referencia.setTipo("Terror");
}
}
```

SOBRECARGA DE MÉTODOS

Algunos métodos en una clase pueden tener el mismo nombre. Estos métodos deben contar con diferentes argumentos. El compilador decide que método invocar comparando los argumentos. Se genera un error si los métodos solo varían en el tipo de retorno.

```
public class Pelicula {

    private float precio;

    public void setPrecio() {
        precio = 3.50;
    }
    public void setPrecio(float nuevoPrecio) {
        precio = nuevoPrecio;
    }
}
```

INICIACIÓN DE VARIABLES DE INSTANCIA

Las variables de instancia se pueden iniciar en la declaración. Esta iniciación ocurre cuando se crea un objeto.

```
public class Pelicula {
    private String titulo;           // implícito private
    String tipo = "Drama";          // explícito private
    private int numeroDeOscars;      // implícito }
}
```

Las variables de tipos primitivos se inician implícitamente como:

char	'0'
byte, short, int,	0
long boolean float,	false
double	0.0
	null

Referencia a Objeto

En forma explícita se puede indicar un valor inicial.

```
private String tipo = "Drama";
```

Un constructor provee una iniciación de variables de instancia más compleja.

CONSTRUCTORES

Para una adecuada iniciación de variables de instancia, la clase debe tener un constructor. Un constructor se invoca automáticamente cuando se crea un objeto. Se declaran de forma pública. Tiene el mismo nombre que la clase. No retorna ningún valor. Si no se codifica un constructor, el compilador crea uno por defecto sin argumentos que solo inicia las variables de instancia.

Pelicula.java

```
public class Pelicula {  
  
    private String titulo;  
    private String tipo = "Drama";  
  
    public Pelicula() {        titulo =  
"Pelicula sin definir.";    }  
    public Pelicula(String nuevoTitulo) {  
titulo = nuevoTitulo;  
    }  
}
```

```
    public Pelicula(String nuevoTitulo, String nuevoTipo)
{
    titulo = nuevoTitulo;        tipo = nuevoTipo;
}

    public String getTitulo() {
return titulo;
}

    public String getTipo() {
return tipo;
}

}
```

TestConstructores.java

```
public class TestConstructores {    public
static void main (String[] args) {
    Pelicula pelicula1 = new Pelicula();
    Pelicula pelicula2 = new Pelicula("La lista de Schindler.");
    Pelicula pelicula3 = new Pelicula("El dormilon.,"Comedia");
    System.out.println(pelicula1.getTitulo()+pelicula1.getTipo());
    System.out.println(pelicula2.getTitulo()+pelicula2.getTipo());
    System.out.println(pelicula3.getTitulo()+pelicula3.getTipo());
}
}
```

La referencia: this

Los métodos de instancia reciben el argumento this implícitamente que se refiere al mismo objeto.

```
public class Pelicula {    private String titulo;    public void
setTitulo(String titulo) {        this.titulo = titulo;
    }
}
```

Se puede compartir código entre constructores usando la referencia this. Un constructor invoca a otro pasándole los argumentos que requiere.

Pelicula.java

```
public class Pelicula
{
    private String titulo;
    private String tipo;

    public Pelicula()
    {
        this("Pelicula sin definir");
    }
}
```

```
}  
}
```

TestThis.java

```
public class TestThis {    public static  
void main  (String[] args) {  
    Pelicula pelicula1 = new Pelicula();  
    Pelicula pelicula2 = new Pelicula("Todo sobre mi madre");  
    System.out.println(pelicula1.getTitulo());  
    System.out.println(pelicula2.getTitulo());  
}  
}
```

VARIABLES DE CLASE

Las variables de clase comparten un único valor entre todas las instancias de la clase. Se declaran con el calificador **static**.

```
public Class Pelicula {  
  
    // Iniciación por defecto    private static double precioMinimo;  
private String titulo, tipo;  
  
}
```

Las variables de clase se pueden iniciar en la declaración. La iniciación ocurre cuando se carga la clase en memoria. Para una iniciación compleja se usara un bloque `static`

```
public Class Pelicula {  
  
    // Iniciación explícita  
private static double precioMinimo = 3.29;  
  
}
```

MÉTODOS DE CLASE

Estos métodos son compartidos por todas las instancias. Se usan estos métodos principalmente para manipular variables de instancia. Se les declara con el calificador `static`. Se invoca a este método de clase con el nombre de la clase o con el nombre de una instancia.

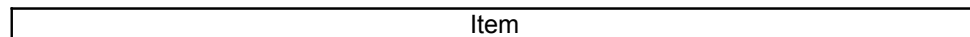
HERENCIA Y POLIMORFISMO

Se estudiara el uso de la herencia y el polimorfismo en el reúso de clases.

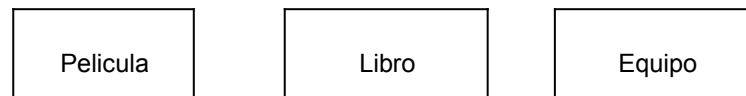
Herencia

Permite a una clase compartir la misma estructura de datos y comportamiento de otra clase. La herencia minimiza la necesidad de duplicar código. El Polimorfismo permite utilizar el método de acuerdo al objeto heredado.

Superclase



Subclase



¿Qué atributos tienen en común las subclases?

¿Qué atributos no tienen en común las subclases?

¿Qué método no tienen en común las subclases?

La herencia en Java

Una subclase se define indicando a que superclase extiende.

```
public class Item {  
    // Definición de la superclase Item.  
}  
  
public class Pelicula extends Item {  
    // Atributos y métodos adicionales para distinguir una  
    // pelicula de otros tipos de  
    item }
```

Una subclase hereda todas las variables instancia de la superclase. Las variables de instancia deben ser private para que instancias de la subclase hereden sus valores.

```
public class Item
{   protected float precio =
0;
    protected String estado = "Excelente";
}
public class Pelicula extends Item
{   private String titulo = "";
private int duracion = 0; }
```

Una subclase no hereda ningún constructor de la superclase, debe declararse explícitamente. Solo en caso no se declare explícitamente, se ejecutaran los constructores por defecto de las superclases y finalmente de la subclase.

```
Pelicula pelicula = new Pelicula ();
// Inicia variables de la clase Item. Constructor por defecto.
// Inicia variables de la clase Pelicula. Constructor por defecto.
```

La referencia super

Se refiere a la clase padre. Se usa para invocar constructores de la clase padre. Debe ser la primera sentencia del constructor de la clase hijo. Esta referencia también se usa para invocar cualquier método del padre.

```
public class Item
{   protected float precio =
0;   Item (float precio)
{   this.precio = precio;
}
}
public class Pelicula extends Item {

    private String titulo = "";

    Pelicula (float precio, String titulo) {
        super(precio);
        this.titulo = titulo;
    }
}
```

Métodos

La superclase define los métodos para todas las subclases. La subclase puede especificar métodos propios.

Item0.java

```
public class Item0
{   protected float precio =
```

```
0;

    Item0 (float precio)
{
    this.precio = precio;
}

    public float getPrecio() {
return precio;
    }
}
```

Pelicula0.java

```
public class Pelicula0 extends Item0 {
    private String titulo = "";

    Pelicula0 (float precio, String titulo) {
        super(precio);    this.titulo = titulo;
    }
    public String getTitulo()
    {
        return titulo;
    } }
}
```

TestSuper.java

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

    Item0 item = new Item0(1.1f);
    System.out.println( item.getPrecio() );

    Pelicula0 pelicula = new Pelicula0(2.2f,"Zelig");
    System.out.println( pelicula.getPrecio() );
    System.out.println( pelicula.getTitulo() );

    }
}
```

¿Qué diferencia existe entre `this` y `super`? ¿Se puede reemplazar `super(precio);` por `this.precio = precio;`? ¿Que métodos puede invocar una subclase?. La subclase hereda todos los métodos del padre. La subclase puede re-escribir un método del padre. **Item1.java**

```
public class Item1
{
    public float
    calcularImporte(int
    cliente) {        return
    50;
    } }
}
```

Pelicula1.java

```
public class Pelicula1 extends Item1 {    public float
calcularImporte(int cliente) {          if (cliente < 500)
return 10;          else          return 30;
    }
}
```

TestSobrescribir.java

```
public class TestSobrescribir {    public
static void main (String[] args)
{    Item1 item1 = new Item1();

    System.out.println( item1.calcularImporte(599) );

    Pelicula1 pelicula1 = new Pelicula1();
    System.out.println( pelicula1.calcularImporte(399) );
    System.out.println( pelicula1.calcularImporte(599) );

} }
```

¿Cuál es la diferencia entre sobre-carga de métodos y sobre-escritura de métodos?

La referencia super

Si una subclase sobrescribe un método de la superclase; el método de la superclase se puede invocar con la referencia `super`. **Item2.java**

```
public class Item2
{    public float
calcularImporte(int
cliente) {    return
50;
    } }
```

Equipo2.java

```
public class Equipo2 extends Item2 {    public float
calcularImporte(int cliente) {    float
seguroEquipo = 25;    float alquiler =
super.calcularImporte(cliente);    return
seguroEquipo + alquiler;
    } }
```

TestSuper2.java

```
public class TestSuper2 {    public static
```



```
void main (String[] args) {  
  
    Item2 articulo = new Item2();  
    System.out.println( articulo.calcularImporte(599) );  
  
    Equipo2 vhs = new Equipo2();  
    System.out.println( vhs.calcularImporte(599) );  
  
} }
```

Polimorfismo

Permite efectuar una misma operación dependiendo del tipo de objeto.

TacoraFilms inicia sus operaciones alquilando únicamente películas. Tres meses después amplía el alquiler a equipos, juegos y libros. El alquiler de una película es 2 soles por día de alquiler. El alquiler de un equipo consta de un seguro de 50 soles además de 5 soles por día. El alquiler de juegos depende del fabricante. PlayStation 2 soles/día, Nintendo 1 sol/día. Los libros no se alquilan, se prestan uno a la vez, mientras sean clientes de la tienda.

Explique por qué se obtienen los resultados

Alquiler3.java

```
public class Alquiler3  
{  
    private int dias;  
    public Alquiler3(int dias) {  
        this.dias = dias;  
    }  
    public int getDias () {  
        return dias;  
    }  
}
```

Item3.java

```
public class Item3  
{  
    protected float  
    calcularImporte(Alquiler3  
    contrato) {  
        return 0;  
    }  
}
```

Pelicula3.java

```
public class Pelicula3 extends Item3 {  
    protected  
    float calcularImporte(Alquiler3 contrato) {  
        int  
        importe = 2*contrato.getDias();  
        return importe;  
    }  
}
```

Equipo3.java

```
public class Equipo3 extends
Item3 {    protected float
calcularImporte(Alquiler3
contrato) {        int
seguroEquipo = 50;        int
importe = seguroEquipo +
5*contrato.getDias();
return seguroEquipo + importe;
    } }
```

Juego3.java

```
public class Juego3 extends Item3
{    String fabricante;
    public Juego3(String fabricante)
{        this.fabricante = fabricante;
    }
    public String getFabricante() {
return fabricante;
    }
    protected float calcularImporte(Alquiler3 contrato)
{        String fabricante = this.fabricante;        int
tasa = 0;        if (fabricante.equals("PlayStation"))
tasa = 2;        if (fabricante.equals("Nintendo"))
tasa = 1;        int importe = tasa*contrato.getDias();
return importe;
    }
}
```

Libro3.java

```
public class Libro3 extends Item3 {    protected float
calcularImporte(Alquiler3 contrato) {        return 0;
    }
}
```

TestPolimorfismo.java

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

```
Alquiler3 contrato = new Alquiler3(10);

Película3 oscar = new Película3();
System.out.println( oscar.calcularImporte(contrato) );
Equipo3 vhs = new Equipo3();
System.out.println( vhs.calcularImporte(contrato) );

Juego3 mu = new Juego3("Nintendo");
System.out.println( mu.calcularImporte(contrato) );
Libro3 agua = new Libro3();
System.out.println( agua.calcularImporte(contrato) );
}
}
```

El operador instanceof y cast

El operador instanceof permite determinar la clase de un objeto en tiempo de ejecución. La operación cast permite modificar la clase de un objeto.

```
public class TestOperador {    public
static void main (String[] args) {
    Película3 oscar = new Película3();
    Equipo3    vhs    = new Equipo3();
    Juego3     mu     = new Juego3("Nintendo");
    Libro3     agua   = new Libro3();

    testOperador(oscar);
testOperador(vhs);
testOperador(mu);
testOperador(agua);
}
    public static void testOperador (Item3 articulo) {
        if (articulo instanceof Juego3) {
            Juego3 juego = (Juego3) articulo;
            System.out.println(juego.getFabricante());
        } else {
            System.out.println("No tiene Fabricante");
        }
    }
}
```

ATRIBUTOS, MÉTODOS Y CLASES FINAL

Variables finales

Una variable final es una constante. Una variable final no puede ser modificada. Una variable final debe ser iniciada. Una variable final por lo general es pública para que pueda ser accesada externamente.

```
public final static String NEGRO = "FFFFFF";  
public final static float PI = 3.141592f;  
public final static int MAXIMO_ITEMS = 10;
```

Métodos finales

Un método puede ser definida como final para evitar la sobre-escritura en una subclase. Un método final no se puede redefinir en una clase hijo.

```
public final static String getBlanco() {  
    return "000000";  
}  
  
public final boolean verificarPassword(String password) {  
    if (password.equals(...  
    }  
}
```

Clases finales

Una clase final no puede ser padre de otra clase. Una clase puede ser definida como final para evitar la herencia. El compilador es más eficiente con definiciones final por qué no buscara estas clases o métodos al tratar clases heredadas.

```
public final class Color {  
    public final static String NEGRO =  
    "FFFFFF";  
    public final static String getBlanco() {  
        return  
        "000000";  
    }  
}
```

EL MÉTODO finalize()

Cuando todas las referencias de un objeto se pierden, se marcan para que el Garbage Collector los recoja y libere ese espacio en memoria.

```
Pelicula pelicula = new Pelicula("Zelig");  
pelicula = null;
```

El objeto "Zelig" que estaba referenciado por `pelicula` ha perdido todas sus referencias. Luego el Garbage Collector liberara el espacio ocupado por "Zelig". El método `finalize` es llamado justo antes que el Garbage Collector libere la memoria. En este instante se puede aprovechar para realizar otras operaciones.

```
public class Pelicula4 {    private
String titulo;    public
Pelicula4(String titulo)
{
    this.titulo = titulo;
}
    public void finalize() {
        System.out.println("Se acabo "+titulo);
    }
}
public class TestFinalize {    public static
void main (String[] args) {        Pelicula4
globo = new Pelicula4("Zelig");        globo
= null;
    }
}
```

Capítulo 3

CLASES ABSTRACTAS E INTERFASES

Se estudiara el uso de clases abstractas, métodos abstractos, interfaces, implementación de interfaces.

CLASES ABSTRACTAS

Clases abstractas

Sirven para modelar objetos de alto nivel, no contienen código, sino solo declaraciones. Todos sus métodos deben existir en sus clases hijas. Una clase abstracta no puede ser instanciada (Crear un objeto a partir de ella).

Métodos abstractos

Estos métodos son parte de clases abstractas. Un método abstracto debe ser redefinido en las subclases. Cada subclase puede definir el método de manera diferente. Las clases abstractas pueden contener métodos que no son abstractos. **Item.java**

```
public abstract class
Item {    protected
String titulo;
protected float precio
= 5.0f;    public
abstract boolean
esAlquilable();
public float
```

```
getPrecio()  
{  
    return precio;  
} }
```

Pelicula.java

```
public class Pelicula  
extends Item  
{  
    public boolean  
esAlquilable()  
{  
    return true;  
} }
```

Libro.java






```
public class Libro  
extends Item  
{  
    public float  
getPrecio()  
{  
    return 0.0f;  
} }
```

Abstracto.java

```
public class Abstracto {  
  
    public static void main (String[] args) {  
  
        Pelicula pelicula = new Pelicula();  
        Libro libro      = new Libro();  
  
        System.out.println(pelicula.esAlquilable());  
        System.out.println(pelicula.getPrecio());  
        System.out.println(libro.esAlquilable());  
        System.out.println(libro.getPrecio());  
  
    }  
}
```

INTERFACES

Una interface es totalmente abstracta; todos sus métodos son abstractos, sus atributos son públicos estáticos y final. Una interface define los métodos que otras clases pueden implementar pero no provee ninguna línea de código. Una clase solo puede heredar de una superclase. Una clase puede implementar muchas interfaces; por lo que las interfaces permiten herencia múltiple.

Conducible			
No Conducible			

Las interfaces describen la conducta que requiere muchas clases. El nombre de una interface por lo general es un adjetivo como `Conducible`, `Ordenable`, `Ubicable`.

Aquí se diferencia de una clase que usualmente es un sustantivo como `Pelicula`, `Cliente`, `Alquiler`. Las clases implementadas por una interface pueden no tener ninguna relación unas con otras. A diferencia de las clases heredadas de una superclase tiene similitudes. Las clases que implementan una interface deben definir todos los métodos de la interface. **Conducible.java**

```
public interface Conducible {

    public static final int MAXIMO_GIRO = 45; public abstract
void girarIzquierda(int grados); public abstract void
girarDerecha(int grados);

}
```

También se puede definir la interface sin los calificadores `public static final abstract`, puesto que son implícitos. Para declarar que una clase que implementa una interface se usa `implements` **Conducible.java**

```
public interface Conducible
{    int MAXIMO_GIRO = 90;    void
girarIzquierda(int grados);
void girarDerecha(int grados); }
```

NaveArea.java

```
public class NaveAerea {    protected char
direccion;    protected int altura;
public void setDireccion(char direccion) {
this.direccion= direccion;
}
    public char getDireccion() {
```

```
return this.direccion;
    }
}
```

Globo.java

```
public class Globo extends NaveAerea implements Conducible
{
    private int volumenGas;
    public void
    setVolumenGas(int volumenGas) {
        this.volumenGas=
        volumenGas;
    }
    public int getVolumenGas() {
return this.volumenGas;
    }
    public void girarDerecha(int grados) {
        if
        (getDireccion()=='N' && grados==90) setDireccion('E');
    }
}
```

Patin.java

```
public class Patin implements Conducible
{
    public void girarDerecha(int grados) {
        System.out.println("Giro de "+grados+" grados a la derecha");
    }
    public void girarIzquierda(int grados) {
        System.out.println("Giro de "+grados+" grados a la izquierda");
    }
}
```

TestInterface.java

```
public class TestInterface {

    public static void main (String[] args) {

        Globo zepelin = new Globo();
        zepelin.setDireccion('N');
        zepelin.girarDerecha(90);
        System.out.println(zepelin.getDireccion());

        Patin patin = new patin();
        patin.girarDerecha(90);

    }
}
```

Capítulo 4**UTILIDADES**

Se estudiara el uso de colecciones de objetos y una técnica de ordenación mediante interfaces.

LA CLASE Object

Todas las clases en java heredan de la clase `java.lang.Object`. Los siguientes métodos se heredan de la clase `Object`

`boolean equals(Object obj)` Indica si otro objeto es igual a otro.

`int hashCode()` Proporciona un valor hashcode para el objeto (id)

`String toString()` Proporciona una representación en cadena del objeto

Otros métodos se encuentran en: <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Object.html>

CONVERTIR DATOS PRIMITIVOS EN REFERENCIAS

Los datos primitivos no se comparan con `null`. Los datos primitivos no pueden integrar una Colección.

Para cada dato primitivo java provee una clase para convertirlo en referencia y tratarlo como un objeto y resolver los problemas previos.

```
int precio ;
System.out.println(precio);
precio = 2;
System.out.println(precio + 5);

Integer objetoPrecio;
objetoPrecio = new Integer(precio);
System.out.println(objetoPrecio.toString());
System.out.println(objetoPrecio);
System.out.println(objetoPrecio+5);
System.out.println(objetoPrecio.intValue());
System.out.println(objetoPrecio.intValue()+5);
```

COLECCIONES

Es un conjunto librerías para manipular y almacenar datos. Estas librerías se llaman colecciones.

Las colecciones se organizan en:

- Interfaces: Manipulan los datos independientemente de los detalles de implementación.
- Clases: Implementan las interfaces.

Para programar con colecciones se debe:

- Elegir una interface adecuada a la funcionalidad requerida.
- Elegir una clase que implemente la interfaz ③ Extender la clase si fuera necesario.

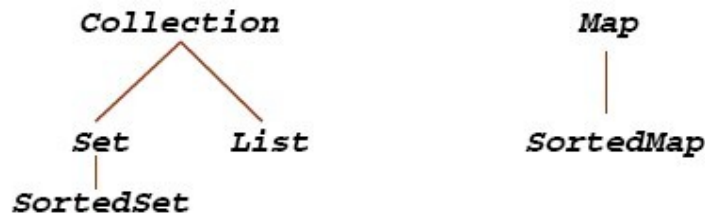
¿En qué se diferencia una clase de una interface?

Arquitectura

Referencia: <http://java.sun.com/j2se/1.4.2/docs/api/java/util/package-tree.html>

Las interfaces y las clases están relacionadas en un armazón (framework) de colecciones para facilitar su uso.

- Interfaces de colecciones que son definiciones abstractas de los tipos de colecciones.
- Clases que implementan las interfaces.
- Clases abstractas que implementan parcialmente las interfaces.
- Métodos estáticos que son algoritmos (por ejemplo ordenar una lista).
- Interfaces de soporte para las colecciones. (una infraestructura).



Interfaces de colecciones

Referencia: <http://java.sun.com/j2se/1.4.2/docs/api/java/util/package-tree.html>

Interface

Collection Representa un grupo de objetos. Sin implementaciones directas, agrupa la funcionalidad general que todas las colecciones ofrecen.

Set Colección que no puede tener objetos duplicados.

SortedSet Set que mantiene los elementos ordenados

List Colección ordenada que puede tener objetos duplicados

Map Colección que enlaza claves y valores; no puede tener claves duplicadas y cada clave debe tener al menos un valor.

SortedMap Map que mantiene las claves ordenadas.

La interface Collection

Referencia: <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Collection.html>

Métodos de la interface

`add(Object)` Añade el objeto en la colección `addAll(Collection)`

Añade la colección.

`clear()` Quita todos los elementos de la colección.

`contains(Object)` ¿El objeto se encuentra en la colección? `containsAll(Collection)`

¿Todos esos elementos están en la colección?

`equals(Object)` ¿Es igual esta colección al argumento? `isEmpty()`

¿La colección está vacía?

`Iterator iterator()` Devuelve un iterador para la colección.

`remove(Object)` Elimina una aparición del objeto

`removeAll(Collection)` Elimina todos esos objetos `retainAll(Collection)`

Se queda sólo con los objetos del argumento

`size()` Número de elementos en la Colección

`toArray()` Devuelve un arreglo con los objetos de la colección.

La interface List

Referencia: <http://java.sun.com/j2se/1.4.2/docs/api/java/util/List.html>

Colecciones ordenadas (secuencias) en las que cada elemento ocupa una posición identificada por un índice. El primer índice es el 0. Las listas admiten duplicados.

Métodos de la interface

`add(int, Object)` Añade el objeto en la posición indicada `add(Object)`

Añade el objeto al final de la lista

`addAll(int, Collection)` Añade la colección en la posición `addAll(Collection)`

Añade la colección al final

`clear()` Quita todos los elementos de la lista.

`contains(Object)` ¿El objeto se encuentra en la lista? `containsAll(Collection)`

¿Todos esos elementos están en la lista?

`equals(Object)` ¿Es igual la lista con el argumento?

`get(int)` Devuelve el objeto en la posición.

`indexOf(Object)` Devuelve la 1ra posición en la que está el objeto `isEmpty()`

¿La lista está vacía?

`Iterator iterator()` Devuelve un iterador para la colección. `lastIndexOf(Object)`

Devuelve la última posición del objeto

`ListIterator listIterator()` Devuelve un iterador de lista

`ListIterator listIterator(int)` Devuelve un iterador de lista para la sublista que inicia en `int`

`remove(int)` Quita de la lista el objeto en esa posición

`remove(Object)` Elimina una aparición del objeto en la lista

`removeAll(Collection)` Elimina todos esos objetos `retainAll(Collection)`

Se queda sólo con los objetos del argumento `set(int, Object)`

Reemplaza el objeto en esa posición por el objeto que se

proporciona `size()` Número de elementos en la Colección

`List subList(int, int)` Devuelve la sublista que comienza en el índice del primer argumento hasta el índice del segundo argumento.

`toArray()` Devuelve un arreglo con los objetos de la colección.

La interface Map

Referencia: <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Map.html>

Son pares de datos (clave, valor). No puede haber claves duplicadas y cada clave se corresponde con al menos un valor.

Métodos de la interface

`clear()` Elimina todos los pares del mapa

`containsKey(Object)` ¿La clave proporcionada se encuentra en el mapa?

`containsValue(Object)` ¿Hay algún par con el valor proporcionado?

`equals(Object)` ¿Es igual este mapa y el proporcionado?

`get(Object)` Devuelve el objeto que se corresponde con la clave dada.

`isEmpty()` ¿La lista está vacía?

`put(Object clave, Object valor)` Asocia la clave proporcionada con el valor proporcionado

`putAll(Map)` Agrega los pares de ese mapa `remove(Object)` Quita la clave del mapa junto con su correspondencia.

`size()` Devuelve el número de pares que hay en el mapa.

Clases implementadas

Las interfaces **List**, **Set** y **SortedSet** son descendientes de la interface **Collection**

El concepto de Polimorfismo aplica para todas las clases que implementan estas interfaces.

Las clases que implementan la interface **List** son: **ArrayList** y **LinkedList**

Las clases que implementan la interface **Set** son: **HashSet** y **LinkedHashSet**

La clase que implementa la sub-interface **SortedSet** es: **TreeSet**.

Definiendo una clase

Para manipular las colecciones usaremos la clase `Producto` compuesta por dos atributos, un constructor y un método `get`.

Producto.java

```
public class Producto {  
  
    private String nombre; private  
    int cantidad;  
  
    public Producto(String s, int i) {  
        nombre = s; cantidad  
    = i;  
    }  
  
    public String toString(){ return ("Nombre: " + nombre + "  
Cantidad: " + cantidad);  
    }  
}
```

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

Mostrar los elementos de una colección: ArrayList

En el ejemplo crearemos una lista del mercado y mostraremos sus elementos.

- Primero se importan las librerías de java.util.* donde se concentran la gran mayoría de las Clases del "Collection Framework".
- Se inicia la declaración de la clase seguido de su método principal main.
- Se definen 5 instancias con el constructor de la clase Producto.
- Agregamos estas instancias al ArrayList con el método add
- Mostramos el número de objetos de la colección mediante el método size.
- Se declara una instancia Iterator la cual facilita la extracción de objetos de la colección.
- Se extrae los objetos del ArrayList y se muestran en pantalla.
- Se elimina el objeto con índice 2. Se muestra la lista nuevamente.
- Se eliminan todos los objetos mediante el método clear.

```
import java.util.*;

public class MercadoLista{

    public static void main(String args[]){

        // Definir 5 instancias de la Clase Producto
        Producto pan = new Producto("Pan", 6);
        Producto leche = new Producto("Leche", 2);
        Producto manzanas = new Producto("Manzanas", 5);
        Producto brocoli = new Producto("Brocoli", 2);
        Producto carne = new Producto("Carne", 2);

        // Definir un ArrayList
        ArrayList lista = new ArrayList();

        // Colocar Instancias de Producto en ArrayList
        lista.add(pan);    lista.add(leche);
        lista.add(manzanas);    lista.add(brocoli);

        // Indica el indice de insercion
        lista.add(1, carne);    lista.add(carne);

        // Imprimir contenido de ArrayLists
        System.out.println(" Lista del mercado con " +
        lista.size() + " productos");

        // Definir Iterator para extraer e imprimir sus valores
        Iterator it = lista.iterator();    while ( it.hasNext() ) {
            Object objeto = it.next();
            Producto producto = (Producto)objeto;
            System.out.println(producto);
        }
    }
}
```

```
}

// Eliminar elemento de ArrayList
lista.remove(2);
System.out.println(" Lista del mercado con " + lista.size() + "
productos");

// Definir Iterator para extraer e imprimir valores      Iterator it2
= lista.iterator();   while ( it2.hasNext() ) {
    Producto producto = (Producto)it2.next();
    System.out.println(producto);
}

// Eliminar todos los valores del ArrayList
lista.clear();
System.out.println(" Lista del mercado con " + lista.size() + "
productos");
}
}
```

Evitar objetos duplicados: HashSet

- Primero modificamos la clase Producto agregando dos métodos equals y hashCode
- El método equals desarrolla como se comparan dos objetos
- El método hashCode devuelve un identificador único.

Producto.java

```
public class Producto {

    private String nombre; private
    int cantidad;

    public Producto(String s, int i) {
        nombre = s; cantidad
    = i;
    }

    public String toString(){ return ("Nombre:
"+nombre+" Cantidad: "+cantidad);
    }
}
```



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

public boolean equals( Object objeto )
{   if (objeto == null) return false;
    Producto producto = (Producto)objeto;
    if (this.getNombre() == producto.getNombre() ) return true;    return
false;
}
public int hashCode() {
    return this.getNombre().hashCode();
}
}
```

- Aun cuando se agregaron 6 elementos, la lista solo cuenta con 5. Set no permite duplicados.
- La evaluación de duplicidad de objetos se realiza mediante los métodos equals y hashCode.
- Un Set no cuenta con índice, por lo que para eliminar un elemento se indica el objeto.

MercadoHashSet.java

```
import java.util.*;

public class MercadoHashSet{

    public static void main(String args[]){

        // Definir 5 instancias de la Clase Producto
        Producto pan = new Producto("Pan", 6);
        Producto leche = new Producto("Leche", 2);
        Producto manzanas = new Producto("Manzanas", 5);
        Producto brocoli = new Producto("Brocoli", 2);
        Producto carne = new Producto("Carne", 2);
        Producto res    = new Producto("Carne", 3);

        // Definir un HashSet
        HashSet lista = new HashSet();    lista.add(pan);

        lista.add(leche);
        lista.add(manzanas);
        lista.add(brocoli);
```

```
lista.add(carne);    lista.add(res);

    // Imprimir contenido de HashSet
    // Aunque son insertados 6 elementos, el HashSet solo contiene 5
    // Se debe a que un Set no permite elementos duplicados.
    System.out.println("Lista del mercado con " +    lista.size() + "
productos");

    // Definir Iterator para extraer e imprimir valores    for(
Iterator it = lista.iterator(); it.hasNext(); ) {
    Object objeto = it.next();
    Producto producto = (Producto)objeto;
    System.out.println(producto);
}

    // No es posible eliminar elementos por indice    // En
un HashSet solo se elimina por valor de Objeto
lista.remove(manzanas);
    System.out.println(" Lista del mercado con " +
    lista.size() + " productos");
    for( Iterator it2 = lista.iterator(); it2.hasNext();) {
    Producto producto = (Producto)it2.next();
    System.out.println(producto);
}

    // Eliminar todos los valores del ArrayList    lista.clear();
    System.out.println("Lista del mercado con " +
lista.size() + " productos");

}

}
```

Manejar colecciones ordenadas: TreeSet

- Primero modificamos la clase Producto implementando un comparador para el ordenamiento.
- La clase implementa la interface Comparable.
- El método `compareTo` de la interfase `Comparable` indica que atributos se usaran para comparar.

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

```

public class Producto implements Comparable {

    private String nombre; private
    int cantidad;

    public Producto(String s, int i) {
        nombre    = s;    cantidad
    = i;
    }
    public String toString(){    return ("Nombre:
"+nombre+" Cantidad: "+cantidad);
    }

    public String getNombre() {
        return this.nombre;
    }

    public boolean equals( Object objeto ) {
        // Indica en base a que atributos se iguala el objeto
        if (objeto == null) return false;    Producto
producto = (Producto)objeto;
        if (this.getNombre() == producto.getNombre() ) return true;    return
false;
    }
    public int hashCode() {
        // retorna un identificador único del objeto.
        return this.getNombre().hashCode();
    }
    public int compareTo( Object objeto ) {
        // Indica en base a que atributos se compara el objeto
        // Devuelve +1 si this es > que objeto
        // Devuelve -1 si this es < que objeto
        // Devuelve 0 si son iguales
        Producto producto = (Producto)objeto;
        String nombreObjeto = producto.getNombre().toLowerCase();

        String nombreThis    = this.getNombre().toLowerCase();
        return( nombreThis.compareTo( nombreObjeto ) );
    }
}

```

- Un TreeSet no permite elementos duplicados.
- Un TreeSet mantiene la lista ordenada.
- El elemento a comparar debe contar con métodos equals, hashCode y compareTo.

MercadoTreeSet.java

```
import java.util.*;

public class MercadoTreeSet {

    public static void main(String args[]) {

        // Definir 5 instancias de la Clase Producto
        Producto pan = new Producto("Pan", 6);
        Producto leche = new Producto("Leche", 2);
        Producto manzanas = new Producto("Manzanas", 5);
        Producto brocoli = new Producto("Brocoli", 2);
        Producto carne = new Producto("Carne", 2);
        Producto res = new Producto("Carne", 3);

        // Definir un TreeSet
        TreeSet lista = new TreeSet();
        lista.add(pan);    lista.add(leche);
        lista.add(manzanas);
        lista.add(brocoli);
        lista.add(carne);    lista.add(res);

        // Imprimir contenido de TreeSet
        // Aunque se agregan 6 elementos, el TreeSet solo contiene 5
        // TreeSet no permite elementos duplicados,
        // TreeSet detecta que el elemento "Carne" esta duplicado //
        // Notese que el orden del TreeSet refleja un orden ascendente
        mostrarLista(lista);

        // No es posible eliminar elementos por indice
        // Un TreeSet solo elimina por valor de Objeto
        lista.remove(manzanas);    mostrarLista(lista);

        // Eliminar todos los valores del TreeSet
    }
}
```

```
lista.clear();
mostrarLista(lista);
}

public static void mostrarLista(Collection lista)
{
    System.out.println("Lista del mercado con " +
        lista.size() + " productos");
    for( Iterator it = lista.iterator(); it.hasNext();) {
        Producto producto = (Producto)it.next();
        System.out.println(producto);
    }
}
}
```

Ordenar y buscar en Colecciones: Collections

- ❑ La clase Collections (que no es la interface Collection) nos permite ordenar y buscar elementos en listas.
- ❑ Se usaran los métodos sort y binarySearch
- ❑ Los objetos de la lista deben tener métodos equals, hashCode y compareTo adecuados.

MercadoCollections.java

```
import java.util.*;

public class MercadoCollections {

    public static void main(String args[]) {
        // Definir 5 instancias de la Clase Producto
        Producto pan = new Producto("Pan", 6);
        Producto leche = new Producto("Leche", 2);
        Producto manzanas = new Producto("Manzanas", 5);
        Producto brocoli = new Producto("Brocoli", 2);
        Producto carne = new Producto("Carne", 2);
```

```
// Definir un ArrayList
ArrayList lista = new ArrayList();

// Colocar Instancias de Producto en ArrayList
lista.add(pan);    lista.add(leche);
lista.add(manzanas);    lista.add(brocoli);
lista.add(1,carne);

// Imprimir contenido de ArrayList    mostrarLista(lista);

// Ordenar elemntos de ArrayList
Collections.sort(lista);

// Imprimir contenido de ArrayList    mostrarLista(lista);

// Buscar un elemento que se compare con Pan de tipo String
String buscar = "Pan";
int indice = Collections.binarySearch(lista,buscar);
System.out.println(buscar+" es el elemento "+indice);
}
public static void mostrarLista(Collection lista)
{
    System.out.println("Lista del mercado con " +
        lista.size() + "productos");
    int i=0;
    for( Iterator it = lista.iterator(); it.hasNext(); i++) {
        Producto producto = (Producto)it.next();
        System.out.println(i+"-"+producto);
    }
}
}
```

Producto.java

```
import java.util.*;

public class Producto implements Comparable {
```

```

private String nombre; private
int cantidad;

public Producto(String s, int i) {
    nombre    = s;    cantidad
= i;
}

public String toString(){    return ("Nombre:
"+nombre+" Cantidad: "+cantidad);
}

public String getNombre() {
    return this.nombre;
}

public boolean equals( Object objeto ) {
    // Indica en base a que atributos se iguala el objeto
    if (objeto == null) return false;    Producto
producto = (Producto)objeto;
    if (this.getNombre() == producto.getNombre() ) return true;    return
false;
}
public int hashCode() {
    // retorna un identificador unico del objeto.
    return this.getNombre().hashCode();
}

public int compareTo( Object objeto ) {
    // Indica en base a que atributos se compara el objeto
    // Devuelve +1 si thises > que objeto
    // Devuelve -1 si this es < que objeto
    // Devuelve 0 si son iguales

    // Dependera del argumento como comparar los atributos.
String nombreObjeto;
    if (objeto instanceof Producto )
{    Producto producto = (Producto)objeto;
    nombreObjeto = producto.getNombre().toLowerCase();
    } else if (objeto instanceof String)
{    String producto = (String)objeto;
nombreObjeto = producto.toLowerCase();

```

```
} else {    nombreObjeto = "";
}

String nombreThis    = this.getNombre().toLowerCase();
return( nombreThis.compareTo( nombreObjeto ) );
}
}
```

Ejemplo de clases implementadas: Map

- La clase que implementan la interfase **Map** es **HashMap**.
- La clase que implementa la sub-interfase **SortedMap** es **TreeMap**.

Ejemplo de HashMap

- Se define una instancia de la clase HashMap
- Se colocan 9 pares clave-valor con el método put
- Se muestra el contenido mediante un iterador que extrae los valores del HashMap
- Se define un arreglo de String con tres claves para eliminar de la agenda
- Se elimina las claves de la agenda
- Verifique si permite duplicados **AgendaHashMap.java**

```
import java.util.*;

public class AgendaHashMap {

    public static void main(String args[]) {    // Definir un HashMap
        HashMap agenda = new HashMap();

        // Agregar pares "clave"- "valor" al HashMap    agenda.put("Doctor",
        "(+52)-4000-5000");    agenda.put("Casa", "(888)-4500-3400");
        agenda.put("Hermano", "(575)-2042-3233");    agenda.put("Tio", "(421)-
        1010-0020");

        agenda.put("Suegros", "(334)-6105-4334");
        agenda.put("Oficina", "(304)-5205-8454");
        agenda.put("Abogado", "(756)-1205-3454");    agenda.put("Papa",
        "(55)-9555-3270");    agenda.put("Tienda", "(874)-2400-8600");
    }
}
```



```
// Definir Iterator para extraer/imprimir valores
mostrarMapa(agenda);

// Definir un arreglo con valores determinados
String personas[] = {"Tio", "Suegros", "Abogado"};
// Eliminar los valores contenidos en el arreglo    for(int
i = 0; i < personas.length; i++) {
    agenda.remove(personas[i]);
}
mostrarMapa(agenda);
}

public static void mostrarMapa(Map agenda) {
    System.out.println(" Agenda con " + agenda.size() + " telefonos");
for( Iterator it = agenda.keySet().iterator(); it.hasNext();) {
    String clave = (String)it.next();
    String valor = (String)agenda.get(clave);
    System.out.println(clave + " : " + valor);
}
}
}
```

Ejemplo de TreeMap

- En un TreeMap los elementos estan ordenados por la clave
- Luego se definen dos referencias de la interfase SortedMap
- En la primera se colocan las claves que se encuentran entre A y O. Metodo submap("A", "O")
- La segunda almacena las claves desde la P hacia el final.
- Estas comparaciones se han hecho con la clase String.
- Otras clases deberán definir sus propios métodos compareTo, equals y hashCode.

AgendaTreeMap.java

```
import java.util.*;

public class AgendaTreeMap {

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

```
// Definir un TreeMap
TreeMap agenda = new TreeMap();

// Agregar pares "clave"-valor al HashMap
agenda.put("Doctor", "(+52)-4000-5000"); agenda.put("Casa",
"(888)-4500-3400"); agenda.put("Hermano", "(575)-2042-
3233"); agenda.put("Tio", "(421)-1010-0020");
agenda.put("Suegros", "(334)-6105-4334");
agenda.put("Oficina", "(304)-5205-8454");
agenda.put("Abogado", "(756)-1205-3454"); agenda.put("Papa",
"(55)-9555-3270"); agenda.put("Tienda", "(874)-2400-8600");

// Notese que el orden del TreeMap refleja un orden
ascendente // en sus elementos independientemente del orden de
insercion. // Debido al uso de String se refleja un orden
alfabetico mostrarMapa(agenda);

// Definir dos TreeMap nuevos
SortedMap agendaAO = agenda.subMap("A", "O");
SortedMap agendaPZ = agenda.tailMap("P");
System.out.println("---- Agenda A-O ----"); mostrarMapa(agendaAO);

System.out.println("---- Agenda P-Z ----"); mostrarMapa(agendaPZ);
}

public static void mostrarMapa(Map agenda) {
    System.out.println(" Agenda con " + agenda.size() + " telefonos");
    for( Iterator it = agenda.keySet().iterator(); it.hasNext();) {
        String clave = (String)it.next();
        String valor = (String)agenda.get(clave);

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

Capítulo 5

MANEJO DE EXCEPCIONES.

¿QUÉ ES UNA EXCEPCIÓN?

Es un evento que interrumpe la ejecución de un programa, por ejemplo:

- ▢ Usar un índice fuera de los límites de un arreglo
- ▢ Dividir entre cero
- ▢ ejecutar métodos de objetos nulos.

¿QUÉ ES UN ERROR?

En Java es cuando la situación es irrecuperable y termina el programa.

- ▢ No hay memoria para correr JVM
- ▢ Errores internos en JVM

¿CUÁL ES LA DIFERENCIA?

Una excepción se puede controlar en el programa. Un error no.

CARACTERÍSTICAS DEL JAVA

Cuando ocurre una excepción en un método, Java lanza (throw) una excepción (Exception).

El objeto Exception generado contiene el tipo de excepción, y el estado del programa cuando ocurrió el error.

SEPARANDO EL MANEJO DE ERRORES

El manejo de excepciones en Java permite separarlos del algoritmo principal.

El resultado es un código más legible y menos propenso a errores de programación.

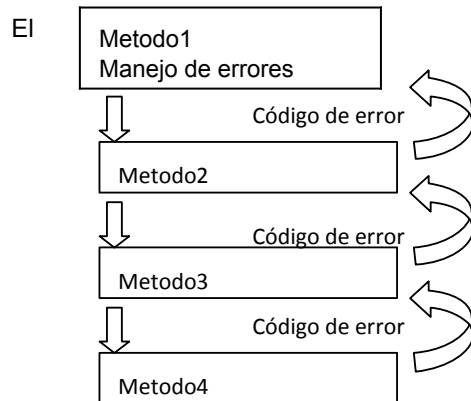
MANEJO TRADICIONAL DE ERRORES	MANEJO DE EXCEPCIONES EN JAVA
<pre> int leerRegistroArchivo() { int errorCode = 0; abrirArchivo(); if (errorAbrirArchivo) { errorCode = OPEN_ERROR; } else { leerArchivo(); if (errorLeerArchivo) { errorCode = READ_ERROR; } cerrarArchivo(); if (errorCerrarArchivo) { errorCode = CLOSE_ERROR; } } } return errorCode; </pre>	<pre> leerRegistroArchivo() { try { abrirArchivo(); leerArchivo(); cerrarArchivo(); } catch (errorAbrirArchivo) { manejarErrorAbrirArchivo; } catch (errorLeerArchivo) { manejarErrorLeerArchivo; } catch (errorCerrarArchivo) { manejarErrorCerrarArchivo; } } </pre>

Java separa los detalles del manejo de errores del código principal, obteniéndose un código mas legible y menos propenso a errores de codificación.

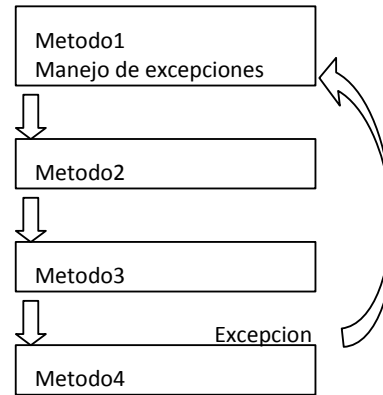
EXCEPCIONES

Devolviendo la excepción hasta el manejador de excepciones. No se requiere que cada método invocado maneje la excepción, sino únicamente lo hará el primer manejador de excepciones de la lista de métodos invocados.

Manejo Tradicional de errores



Excepciones en Java



método4 retorna el código de error al método3. El metodo4 lanza una excepción

El método3 retorna el código de error al método2 El metodo1 captura la excepción y la maneja

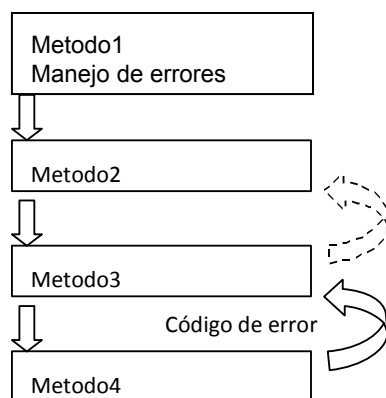
El método2 retorna el código de error al metodo1

El metodo1 maneja el error.

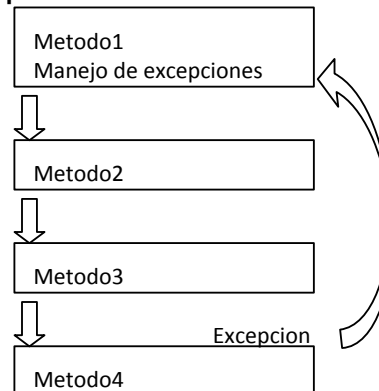
Las excepciones no pueden ignorarse

Una vez que un método lanza un error no puede ignorarse a diferencia de la programación tradicional donde se debe controlar en cada punto de invocación.

Manejo Tradicional de errores

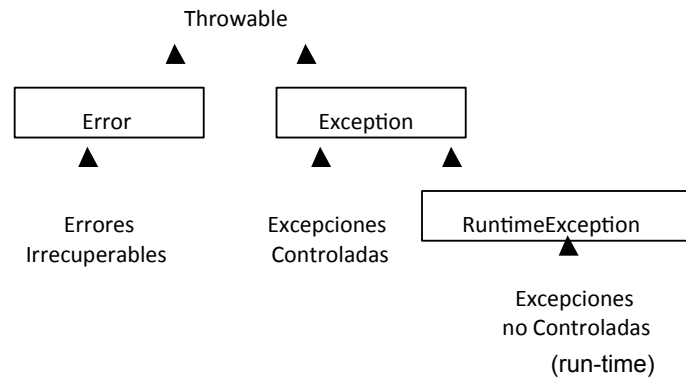


Excepciones en Java



Throwable

Todos los errores y excepciones heredan de la clase `Throwable`



Errores

Heredan de la clase `Error`; estos se generan cuando ocurren errores fatales para el programa como por ejemplo: cuando la memoria está llena o cuando es imposible encontrar una clase requerida.

Excepciones no controladas

Heredan de la clase `RuntimeException`. Estas ocurren cuando por ejemplo se divide entre cero o se intenta acceder a un elemento del arreglo fuera de sus límites. Mediante código se puede capturar, manipular o ignorar estas excepciones. Si no maneja el error, Java terminará el programa indicando el estado del error.

Excepciones controladas

Heredan de la clase `Exception`. Estas deben ser capturadas y manejadas en algún lugar de la aplicación. Las excepciones creadas por el programador serán heredadas de la clase `Exception`.

¿Qué se puede hacer con una Excepción?

- Capturar la excepción y manipularla
- Pasar la excepción al método invocador.
- Capturar la excepción y lanzar una excepción diferente.

EXCEPCIONES NO CONTROLADAS

No necesitan ser controladas en el código.

El JVM terminara el programa cuando las encuentra.

Si no desea que el programa termine tendrá que manejarlas.

Como capturar y manejar una excepción

```
try {  
    Encierre el código del método en un bloque try  
    // código del método  
    Maneje cada exception en un bloque catch.  
}  
  
Cualquier proceso final realícelo en un bloque catch (exception1) { finally. Este bloque  
siempre se ejecutara,  
    // manejar la excepción1  
ocurra o no una excepción.  
}  
  
catch (exception2) {  
    // manejar la excepción2  
} ..  
.  
finally {  
    // cualquier otro proceso final  
}
```

La documentación del java indicara que excepciones lanza los métodos del java.

Clase : `java.io.FileInputStream`

Método: `public FileInputStream(String name) throws FileNotFoundException`

Método: `public int read() throws IOException`

Referencia: <http://java.sun.com/j2se/1.4.2/docs/api/java/io/FileInputStream.html>

Capturando una excepción

Este ejemplo convierte una cadena en un entero

```
int cantidad;  
String cadena =  
"5"; try {  
    cantidad = Integer.parseInt(cadena);  
}
```

```
catch ( NumberFormatException e) {  
    System.err.println(cadena + " no es un entero"); }  
}
```

Capturando múltiples excepciones

Este ejemplo convierte una cadena en un entero y realiza una división.

```
public class TestMultiException {  
  
    public static void main (String[] args) {  
        int cantidad= 0;    int  
        divisor = 0;    String  
        cadena = "5";  
        try {  
            cantidad = Integer.parseInt(cadena);  
            System.out.println(cantidad);    int  
            resultado = cantidad / divisor;  
            System.out.println(resultado);  
        }  
        catch ( NumberFormatException e) {  
            System.err.println(cadena + " no es un entero");  
        }  
        catch ( ArithmeticException e) {  
            System.err.println("Error en "+cantidad+"/"+divisor);  
        }  
    }  
}
```

Ejecución del bloque finally

El bloque finally siempre se ejecuta dependiendo como se termina el bloque try.

- Hasta terminar la ultima sentencia del bloque try.
- Debido a sentencias return o break en el bloque try.
- Debido a una excepción.

```
public class TestFinally {  
  
    public static void main (String[] args) {  
        int cantidad= 0;    int  
        divisor = 0;    String  
        cadena = "5";  
        try {  
            if (cadena.equals("5"))
```



```
        return;
        cantidad = Integer.parseInt(cadena);
System.out.println(cantidad);    int
resultado = cantidad / divisor;
        System.out.println(resultado);
    }
    catch ( NumberFormatException e) {
        System.err.println(cadena + " no es un entero");
    }
    catch ( ArithmeticException e) {
        System.err.println("Error en "+cantidad+"/"+divisor);
    }
    finally {
        System.err.println("Se trabajo con "+cadena+" y "+divisor);
    }
}
}
```

Como pasar la excepción al método invocado

Para pasar el método al invocador, se declara con la declaración `throws`. La excepción se propaga al método que lo invoca. En el ejemplo, las excepciones de los métodos se manejan en el método invocador.

```
public void metodoInvocador() {
    try
    {
        miMetodo();
        getResultado();
    }
    catch {
    }
    finally {
    }
}
public int miMetodo() throws Exception {
    // Código que podría lanzar la Exception
}
public int getResultado() throws NumberFormatException {
    // Código que podría lanzar la exception NumberFormatException
}
```

TestThrows.java

```
public class TestThrows {
```

```
public static void main (String[] args) {
String cadena = "abcde";    int posicion =
6;    char letra = ' ';
    try {
        letra = getLetra(cadena,posicion);
        System.out.println(letra );
    }
    catch (IndexOutOfBoundsException e) {
        System.err.println("Error en "+cadena+" "+posicion);
    }
}
public static char getLetra(String cadena, int posicion)
throws IndexOutOfBoundsException{    char c =
cadena.charAt(posicion);
    return c;
}
}
```

Como lanzar una excepción

Para lanzar una excepción use las declaraciones `throws` y `throw`. Puede crear excepciones para manejar posibles problemas en el código.

```
public String getValor(int indice) throws IndexOutOfBoundsException
{    if (indice < 0 || indice > 100) {
        throw IndexOutOfBoundsException();
    }
}
```

COMO CREAR UNA EXCEPCIÓN

Para crear su propia excepción tiene que heredarla de la clase `Exception`. Puede ser usado en caso quiera tratar cada problema en el código en forma diferente. Por ejemplo el manejo del archivo1 puede tratarlo con una excepción1. El manejo de un archivo2 puede tratarlo con una excepción2. Cada archivo tendría su propia excepción.

UserFileException.java

```
public class UserFileException extends Exception {

    public UserFileException (String mensaje) {
```

```
super(mensaje);  
}  
}
```

COMO CAPTURAR UNA EXCEPCIÓN Y LANZAR OTRA DIFERENTE

En el ejemplo si ocurre un problema de IO (entrada/salida) se captura la excepción y se lanza una excepción propia.

```
public void ReadUserFile throws UserFileException { try {  
    // código que manipula un archivo  
}  
catch (IOException e) {  
    throw new UserFileException(e.toString());  
}  
}
```

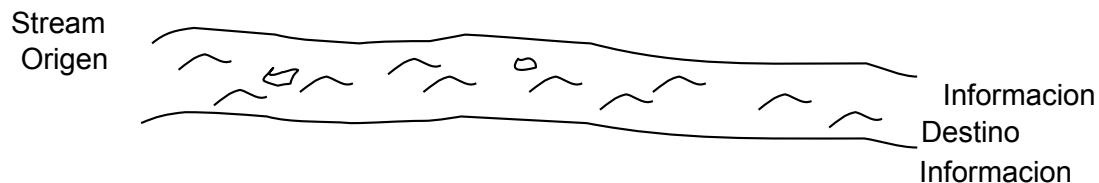
Capítulo 6

STREAM Y ARCHIVOS.

INTRODUCCIÓN

Cualquier cosa que sea leída o escrita hacia un dispositivo es un stream.

- Ejemplos de dispositivos: Consola, File, Pipes, Red, Memoria.
- Ejemplos de cosas a leerse o escribirse: Caracteres, Objetos java, sonido, imágenes...
- Para la lectura siempre se ha usado este modelo de programación ③ El modelo se asemeja al flujo de agua en un río.



```
abrir stream mientras  
existan datos leer o  
escribir datos cerrar  
stream
```

Un stream es un flujo de datos y puede estar compuesto por archivos de texto, datos ingresados por teclado, datos descargados desde Internet.

java.io.InputStream y java.io.OutputStream

- Las clases java están agrupadas según su funcionalidad
- Las que tratan de la lectura heredan de InputStream.java
- Las que tratan de la escritura heredan de OutputStream.java
- Un flujo de datos binarios pueden provenir de un file, un pipe o de un objeto java.
- FileInputStream.java por ejemplo es para el manejo de archivos.

```
java.io.InputStream
FileInputStream  AudioInputStream
ObjectInputStream
FilterInputStream
BufferedInputStream

java.io.OutputStream
FileOutputStream  ByteArrayOutputStream  ObjectOutputStream ...
FilterOutputStream
BufferedOutputStream
```

Una lista completa se encuentra en:

<http://java.sun.com/j2se/1.4.2/docs/api/java/io/InputStream.html>

<http://java.sun.com/j2se/1.4.2/docs/api/java/io/OutputStream.html>

Tanto java.io.InputStream y java.io.OutputStream transmiten datos en forma de bytes (8 bits)

java.io.Writer y java.io.Reader

- Con una variación las clases InputStream y OutputStream.
- Transmiten datos en forma de caracteres (16 bits)
- Las clases Writer y Reader fueron desarrolladas posteriormente y son más veloces que InputStream y OutputStream.
- Se cuenta con clases que permiten actualizar las clases antiguas.

□ `InputStreamReader` convierte un `InputStream` a `Reader` ③ `OutputStreamWriter`

convierte un `OutputStream` a `Writer`

□ Revise otras clases del `Reader` y `Writer` en:

<http://java.sun.com/j2se/1.4.2/docs/api/java/io/Writer.html>

<http://java.sun.com/j2se/1.4.2/docs/api/java/io/Reader.html>

```
java.io.Reader  BufferedReader
InputStreamReader
    FileReader
StringReader
java.io.Writer  BufferedWriter
PrintWriter
OutputStreamWriter  FileWriter
```

ENTRADA Y SALIDA ESTANDAR

System.out

- El metodo `System.out.println` tambien es un stream que muestra datos en consola.
- `System` es una clase
- `out` es un atributo de `System`
- `out` es una clase es del tipo `java.io.PrintStream`
- `out` es una variable de instancia, pública y estática.
- `java.io.PrintStream` hereda de `java.io.OutputStream`.
- `println` es un método
- El atributo `out` de la clase `System` cuenta con el metodo `println` que conforman el conocido `System.out.println`

System.in

- La entrada estandar de la consola se representa por `System.in`

□ in es un objeto del tipo InputStream con limitaciones en la lectura por teclado.

□ Su método read no permite leer una línea completa, ni un número.

□ Se usa el método read en caso requiera realizar una pausa en el programa.

```
try { System.in.read(); } catch ...
```

□ Cuando requiera leer el teclado debe enmascarar System.in

□ Requiere de las clases InputStreamReader y BufferedReader para realizar la máscara.

LECTURA DE ARCHIVOS

Este programa pide el nombre del archivo para leer su contenido y finalmente indicar cuántas líneas tiene. Ingrese por teclado un archivo java.

```
import java.io.*;

public class TestStreamReader {

    public static void main(String[] args) throws IOException {
        String lineaJava;    int
lineas = 1;

        try {

            System.out.print("Que archivo desea analizar?");    //
Se define una variable del tipo BufferedReader que    //
permite almacenar datos en memoria Buffer.
            BufferedReader teclado;

            // System.in es la entrada de la consola, el teclado
            // InputStreamReader es un stream de lectura (de teclado)
            // El constructor BufferedReader direcciona    // la lectura
del teclado al buffer en memoria.
            teclado = new BufferedReader(new InputStreamReader(System.in));
            // Se lee el stream hasta un ENTER.
            // El nombre del archivo se encuentra la variable String.    String
archivoJava = teclado.readLine();

            // Se define una variable del tipo BufferedReader que    //
permite almacenar datos en memoria Buffer.
            BufferedReader archivo;
```

```
// FileReader es un stream de lectura (de archivos)
// El constructor BufferedReader direcciona // la
lectura del archivo al buffer en memoria.
    archivo = new BufferedReader( new FileReader(archivoJava));
    // Este ciclo se realizara hasta que //
ya no existan lineas por leer.
    while (( lineaJava = archivo.readLine()) != null)

        lineas++;
    System.out.println("El archivo " + archivoJava +
        " contiene " + lineas + " lineas.");

} catch (IOException ex) {
    System.err.println(ex);
    System.out.println("Asegurese de haber proporcionado " +
        "la extension del archivo (\".java\")");
} finally {
    System.out.println("");
    System.out.println("Asi funcionan los Streams!");
}
}
```

ESCRITURA DE ARCHIVOS

Este programa lee un archivo java y lo modifica para escribirlo en otro archivo.

```
import java.io.*;

public class TestWriter {

    public static void main(String[] args) {

        try {
            // Se define el stream para leer codigo fuente
            // de un archivo java
            BufferedReader archivoEntrada;
            archivoEntrada = new BufferedReader(        new
            FileReader("C:\\TestWriter.java"));
```

```

String lineaArchivo;
String fuenteArchivo = new String();

// El archivo se lee linea a linea y se agrega en una Cadena //
readLine() retorna nulo cuando no haya mas lineas.
// \n significa fin de linea, retorno de carro.
while ((lineaArchivo = archivoEntrada.readLine()) != null)
fuenteArchivo += lineaArchivo + "\n";

// Se cierra el stream de lectura.
archivoEntrada.close();

// Se define el stream en memoria (BufferedReader)
// para leer datos del teclado (InputStreamReader)
// que se digitan en la consola (System.in)
BufferedReader teclado = new BufferedReader(      new
InputStreamReader(System.in));

// Se lee el teclado y se agrega a la Cadena
System.out.print("Introduzca algun dato: ");      fuenteArchivo +=
"Agrego  \" \" + teclado.readLine()
+ \" \" en la consola";

// Para guardar datos en un archivo
// Se define el stream en memoria (BufferedReader)
// para leer datos de un String (StringReader)
// que se encuentra en una variable (fuenteArchivo)      BufferedReader
fuenteSalida;
fuenteSalida = new BufferedReader(new StringReader(fuenteArchivo));
// Se define un stream de salida (PrintWriter)
// que tomara los datos de memoria (BufferedWriter)
// y los escribira en un archivo (FileWriter)
PrintWriter archivoSalida;      archivoSalida = new
PrintWriter(      new BufferedWriter(
new FileWriter("C:\\TestWriterNew.java")));
// Se lee linea a linea la Cadena (fuenteSalida)
// si no se tiene mas lineas, la lectura devuelve NULL
// Se escribe en el archivo el numero de linea
// y el texto de la Cadena      int
lineaNo = 1;
while ((lineaArchivo = fuenteSalida.readLine()) != null)
archivoSalida.println(lineaNo++ + ": \" + lineaArchivo);
// Se cierra el stream de salida      archivoSalida.close();

```



```
} catch (EOFException e) {  
    System.out.println("Final de Stream");  
}  
} catch (IOException e) {  
    System.out.println("Excepcion Entrada/Salida");  
  
}  
  
} finally {  
    System.out.println("Revise C:\\\\TestWriterNew.java");  
}  
  
}  
  
}
```

LA CLASE File

- La clase `File` se usa para representar un archivo o un conjunto de archivos (directorio)

```
File f1 = new File ("/");  
File f1 = new File ("/", "etc/passwd"); File f1 = new  
File ("config.sys");
```

- Puede obtener atributos del archivo mediante la clase `File`.

Si existe, si se puede leer, si se puede escribir.

Cuando fue la última vez modificado, cual es su tamaño.

- Con `File` puede crear y borrar archivos.

- Otros métodos se encuentran en:

<http://java.sun.com/j2se/1.4.2/docs/api/java/io/File.html>

TestFile.java

```
import java.io.*;  
  
public class TestFile {  
  
    public static void main( String args[] ) throws IOException
```

```
{
    if( args.length > 0 ){
        for( int i=0; i < args.length; i++ ){
            File f = new File( args[i] );
            System.out.println( "Nombre: "+f.getName() );      System.out.println(
"Ruta  : "+f.getPath() );
            if( f.exists() ){
                System.out.print( "El archivo existe." );
                System.out.print( (f.canRead()?" y se puede Leer":""));
                System.out.print( (f.canWrite()?" y se puede Escribir":""));
                System.out.println( "." );
                System.out.println( "La longitud del archivo es " +      f.length()
+" bytes" );
            } else
                System.out.println( "El archivo no existe." );
        }
    }else
        System.out.println( "Debe indicar un archivo." );
    }
}
```

DIRECTORIOS

- El método `list()` de la clase `File` muestra los archivos de un directorio.
- La interfase `FileNameFilter` se usa para obtener un subconjunto de archivos

TestFilenameFilter.java

```
import java.io.*;

public class TestFilenameFilter{

    public static void main( String args[] ) {

        // Buscar en el raiz del disco C
        // Se coloca doble slash puesto que es un caracter de escape.
        // en literales cadenas en java se realiza:
        // \n -> retorno de carro
        // \t -> tabulador
        // \\ -> slash
        // \" -> comillas
        String directorio = "c:\\\\";
    }
}
```

```
// El directorio es un archivo del sistema operativo
// fileDirectorio es este archivo en el programa
File fileDirectorio = new File(directorio);

// Se crea un filtro de archivos que contengan java.
FilenameFilter filtro = new FilterJava("java");

// Se aplica el filtro al directorio.
String [] contenido = fileDirectorio.list(filtro);  for (int i=0; i
< contenido.length; i++) {
    System.out.println(contenido[i]);
}
}
```

FilterJava.java

```
import java.io.*;

public class FilterJava implements FilenameFilter {

    // Esta clase implementa un filtro de archivos
    // usando la interface FilenameFilter
    // mediante el método accept
    String mascara;
    FilterJava(String mascara) {
        this.mascara = mascara;
    }
    public boolean accept(File directorio, String nombre) {    return
nombre.indexOf(mascara) != -1;
    }
}
```

CASO DE ESTUDIO

Parte 1

Se cuenta con el archivo texto Pelicula.def que define una tabla para películas, indicando el nombre del campo, tipo, longitud y en caso sea clave primaria el campo.

Pelicula.def

codigo	int	4	PK
titulo	String	20	
director	String	20	
categoría	String	3	
duracion	int	3	

Lea el archivo y almacene su contenido en un `ArrayList` de objetos `Tabla`. La clase `Tabla.java` contara con los siguientes atributos **Tabla.java**

```
public class Tabla
{
    private int campo;
    private String tipo;
    private int longitud;
    private String clave;
}
```

MicroCase.java

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

public class MicroCase {
    public static void main(String[] args)
    {
        String tabla = "Pelicula";
        ...
    }
}
```

El siguiente es un ejemplo para “Tokenizar” una cadena. El ejemplo divide la cadena en sus componentes separados por espacios. **TestTokenizer.java**

```
import java.util.*;

public class TestTokenizer {
    public static void main(String args[]) {
        String s = "9 23 45.4 56. 7";
        StringTokenizer st = new StringTokenizer(s); while
        (st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }
    }
}
```

Parte 2

Con el `ArrayList` generado anteriormente cree el archivo `PeliculaDispatcher.java` El programa debe funcionar para cualquier otro archivo `*.def`.

PeliculaDispatcher.java

```
public class PeliculaDispatcher {

    public PeliculaDispatcher() {}

    public ArrayList listarPeliculas ()
    {   String sql = " select codigo      "+
        "          ,titulo      "+
        "          ,director    "+
        "          ,categoria   "+
        "          ,duracion    "+
        " from pelicula      ";
        ArrayList lista = new ArrayList();
        ResultSet rs = executeQuery(sql);   while (
        rs.next() ) {
            Pelicula pelicula = new Pelicula();    pelicula.setCodigo(
            rs.getInt("codigo") );    pelicula.setTitulo
            ( rs.getString("titulo") );    pelicula.setDirector
            ( rs.getString("director") );    pelicula.setCategoria
            ( rs.getString("categoria") );    pelicula.setDuracion
            ( rs.getInt("duracion") );    lista.add(pelicula);
        }
        return lista;
    }

}
```