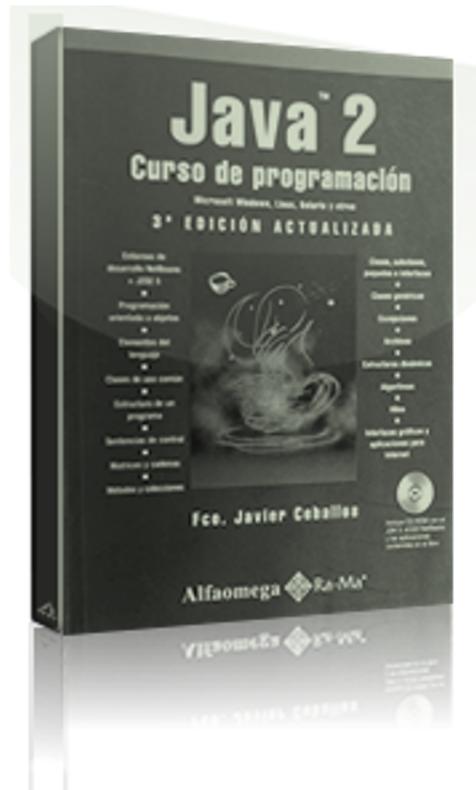


4 Programación orientada a objetos

Objetivo: Aprender y aplicar las técnicas y herramientas de programación orientada a objetos para la solución de problemas.





**JAVA 2 Curso de programación. Francisco Javier Ceballos,
2da edición Alfaomega, 2003.**

4.1 Teoría del diseño de jerarquía de clases

El concepto de jerarquía designa una forma de organización de diversos elementos de un sistema determinado, en el que cada elemento es subordinado del elemento posicionado encima de él (a excepción del primero).

Se conoce como jerarquía de clases al orden o clasificación de abstracciones en una estructura de árbol, es decir, un conjunto de clases que describen, ampliamente, un concepto, desde un caso muy general hasta los casos más particulares.

En la estructura de árbol, las clases que están en los niveles más especializados, heredan sus atributos y métodos de las clases menos especializadas.

Entre más arriba se esté en la jerarquía de clases, más alto es el nivel de abstracción.

NOTA. En java, todas las clases (también las declaradas por el usuario) son implícitamente subclases de la clase Object.

Ejemplo 4.1

0

Polígono

1

Triángulo

Cuadrilátero

2

Triángulo
equiláteroTriángulo
isóscelesTriángulo
rectángulo

Cuadrado

Rombo

Trapecio

3

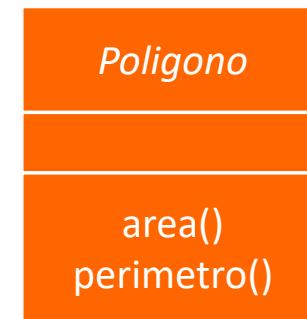
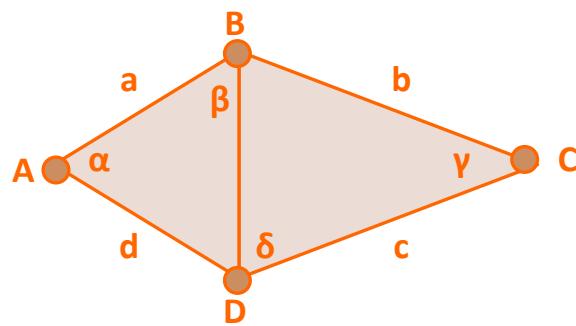
Rectángulo

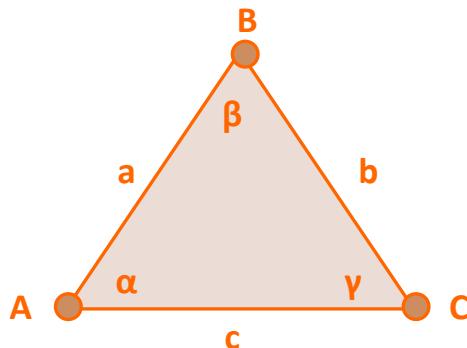
Romboide

Ejemplo 4.1

Polígono

0



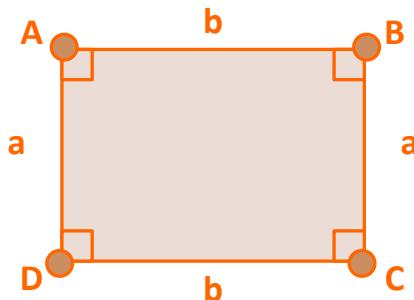
Ejemplo 4.1**Triángulo**

Triangulo

α, β, γ : entero
 a, b, c = flotante
 base, alt = flotante

flotante area()
 flotante perímetro()

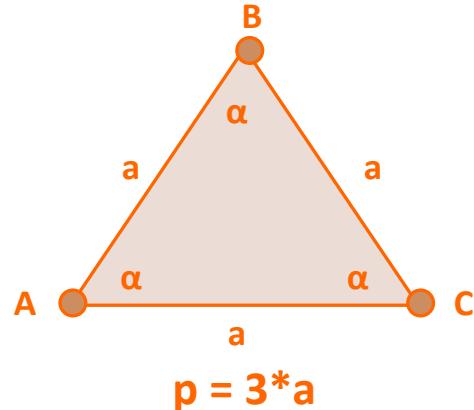
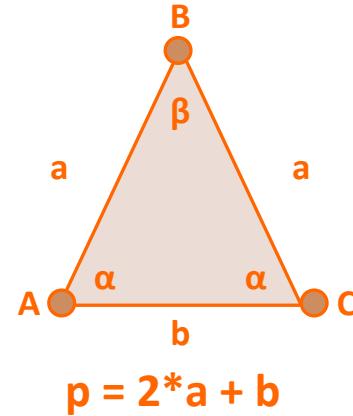
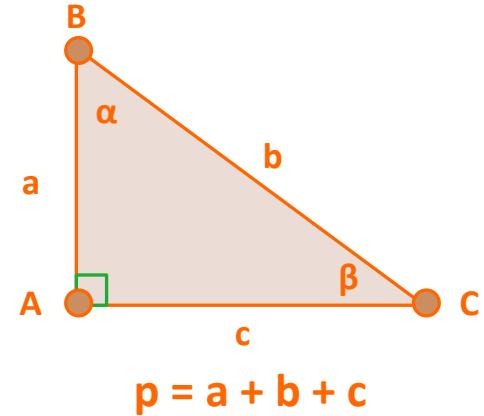
1

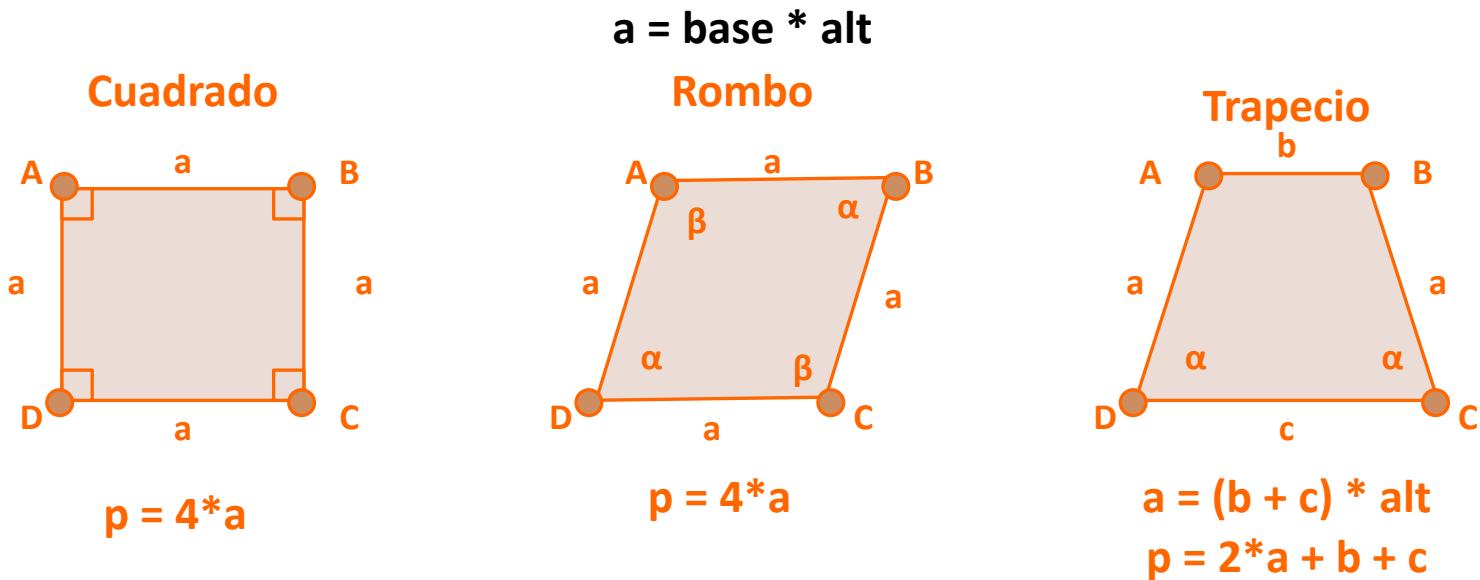
Cuadrilátero

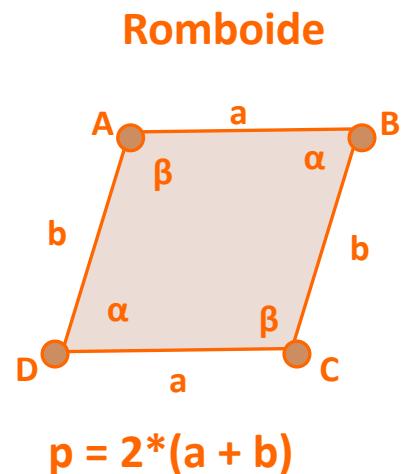
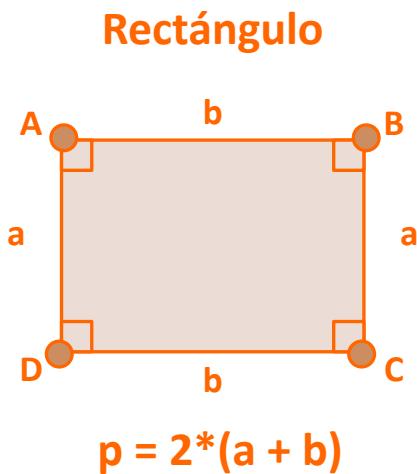
Cuadrilatero

α, β : entero
 a, b = flotante
 base, alt = flotante

flotante area()
 flotante perímetro()

Ejemplo 4.1**Triángulo equilátero****Triángulo isósceles****Triángulo rectángulo**

Ejemplo 4.1

Ejemplo 4.1

¿Qué es java?

Java es un lenguaje de programación y la primera plataforma informática creada por Sun Microsystems en 1995. Es la tecnología subyacente que permite el uso de programas, como herramientas, juegos y aplicaciones de negocios.

Java se ejecuta millones de computadoras personales de todo el mundo y en miles de millones de dispositivos, como dispositivos móviles y aparatos de televisión.



java.com

Máquina virtual

La máquina virtual (JVM) es el entorno en el que se ejecutan los programas Java, permite que el código sea portable, es decir, que se pueda ejecutar en diferentes plataformas.



Archivo.java



Archivo.class

Interprete

Mac

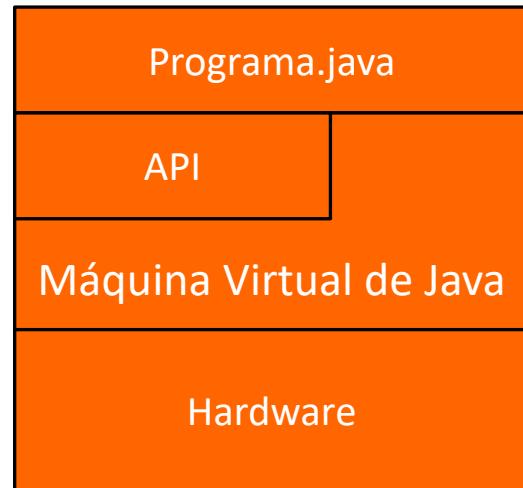
100011101

Windows

100011101

Linux

100011101



La JVM se encarga de:

- **Reservar espacio en memoria para los objetos creados.**
- **Asignar variables a registros y pilas.**
- **Liberar la memoria no utilizada (garbage collector).**
- **Llamar al sistema huésped para ciertas funciones, como los accesos a los dispositivos.**
- **Vigilar que se cumplan las normas de seguridad de las aplicaciones Java.**
- **Gestiona automáticamente el uso de la memoria, de modo que no queden huecos.**

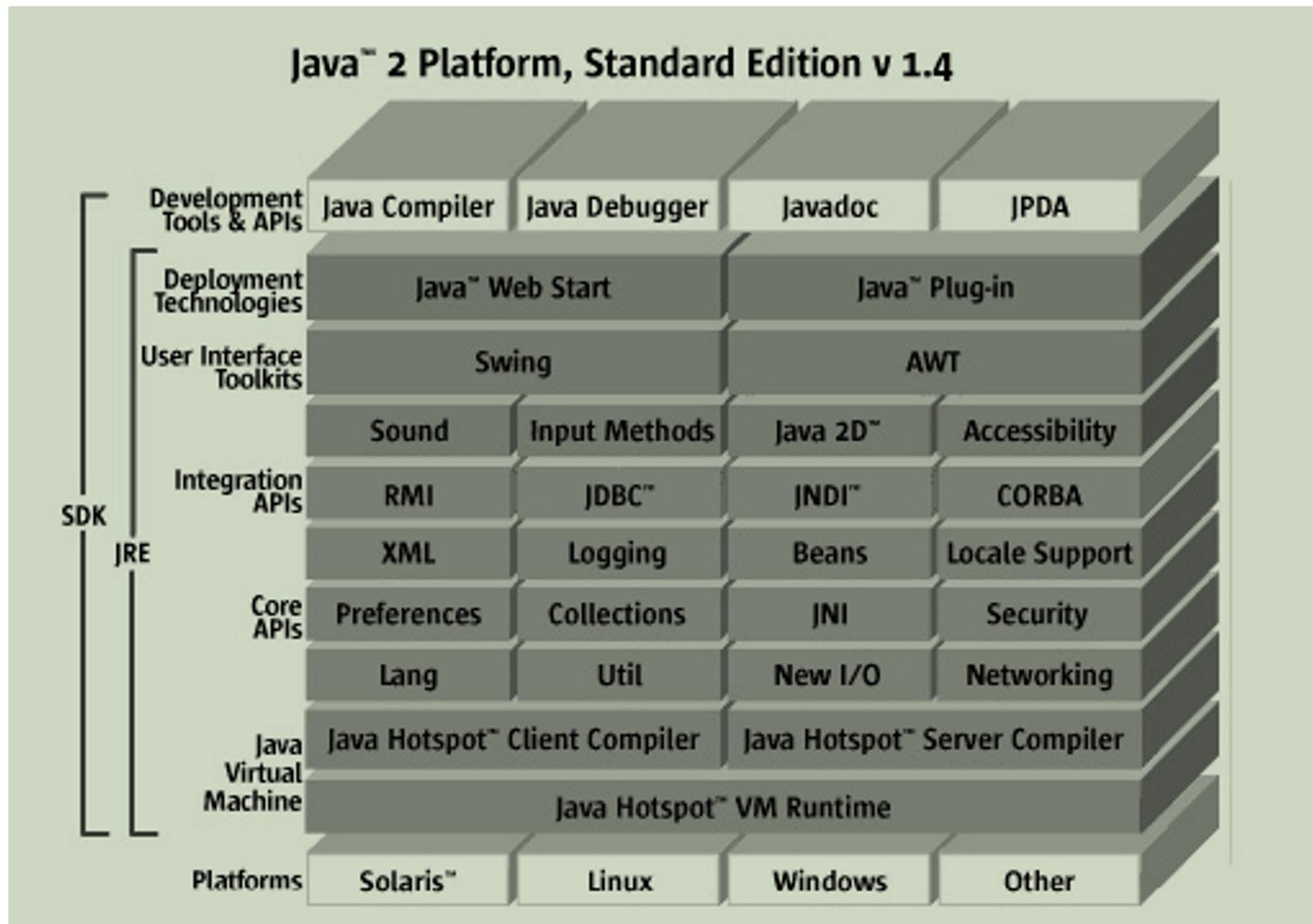
Versiones de java

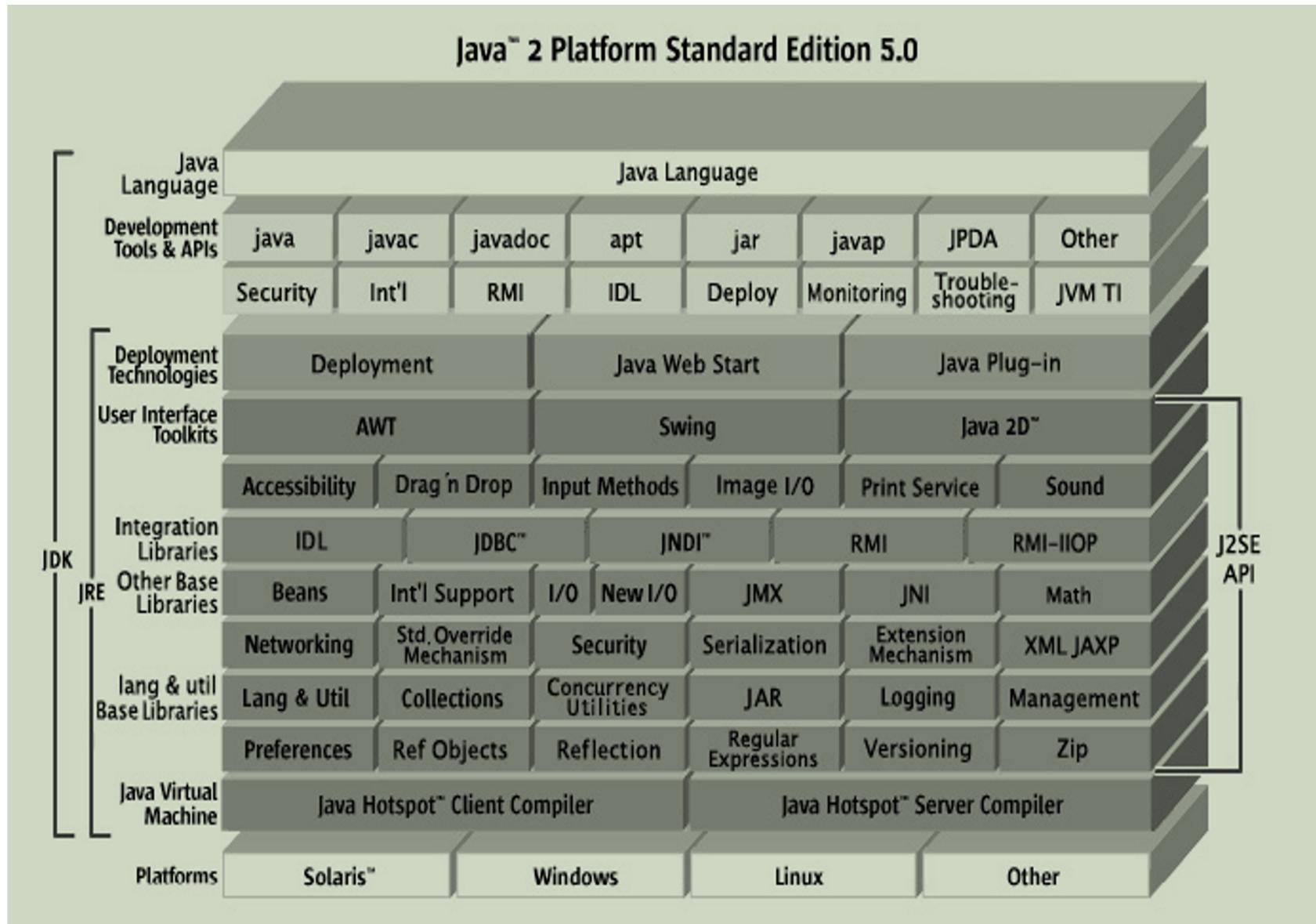
- Java 2 Enterprise Edition (J2EE)
- Java 2 Standard Edition (J2SE)
- Java 2 Micro Edition (J2ME)

J2SE es una herramienta que permite generar programas estándar. Contiene diferentes herramientas como el compilador, el depurador, el documentador, el intérprete, etc.

Para instalar Java 2 Standard Edition se debe descargar el JDK (Java Development Kit) que a su vez incluye el JRE.

El JRE (Java Runtime Environment) es el entorno mínimo requerido para ejecutar programas java, ya que incluye la JVM y el API.





		Java Language								
		java	javac	javadoc	apt	jar	javap	JPDA	jconsole	
		Security	Int'l	RMI	IDL	Deploy	Monitoring	Troubleshoot	Scripting	JVM TI
JDK		Deployment				Java Web Start			Java Plug-in	
JRE		AWT			Swing			Java 2D		
Base Libraries		Accessibility	Drag n Drop		Input Methods		Image I/O	Print Service	Sound	
Integration Libraries		IDL	JDBC™		JNDI™		RMI	RMI-IIOP		Scripting
Other Base Libraries		Bea CORBA Interface Definition Language API	I/O			JMX		JNI		Math
lang and util Base Libraries		Networking	Override Mechanism		Security	Serialization		Extension Mechanism		XML JAXP
Java Virtual Machine		lang and util	Collections	Concurrency Utilities			JAR		Logging	Management
Platforms		Preferences API	Ref Objects	Reflection			Regular Expressions	Versioning	Zip	Instrument
Java SE API		Java Hotspot™ Client VM					Java Hotspot™ Server VM			
		Solaris™			Linux		Windows		Other	



Instalación y configuración de JDK

Una vez instalado el JDK se tiene que actualizar la variable de ambiente PATH y crear las variables CLASSPATH y JAVAPATH. Para ello se necesita saber la ruta donde se instaló el JDK (carpeta java) y de ahí se pueden obtener las rutas del PATH (carpeta bin) y del CLASSPATH (carpeta lib).

Se tienen tres rutas básicas:

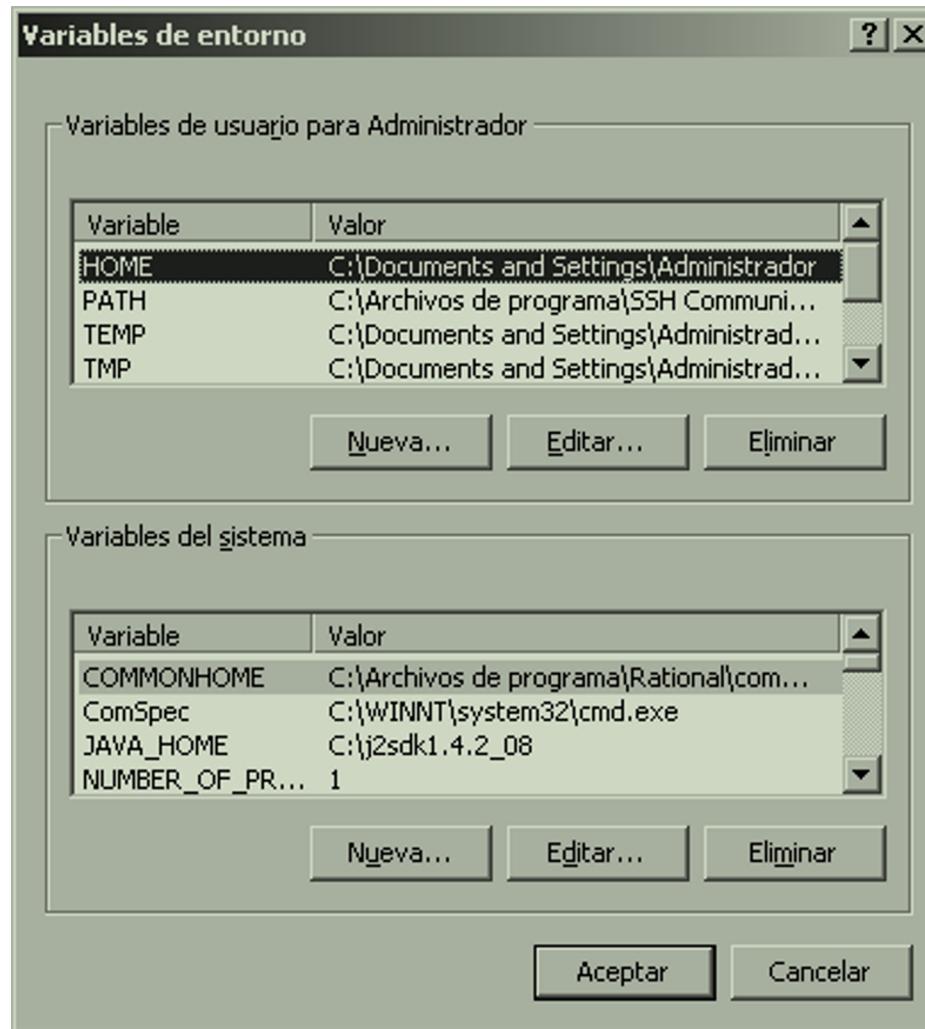
- **JAVAPATH:** Es la ruta del directorio donde está instalado JDK.
- **CLASSPATH:** Ruta que contiene la ubicación de las librerías de JDK.
- **PATH:** Ruta que contiene la ubicación de los binarios de JDK.

Una vez instalado JDK se realiza lo siguiente:

- **set PATH=%PATH%;/ruta/jdk1.X.Y/bin**
- **set CLASSPATH=/ruta/jdk1.X.Y/jre/lib;.**

La variable de ambiente CLASSPATH proporciona a la Máquina Virtual y otras aplicaciones el lugar donde debe buscar las clases que se necesitan para ejecutar un programa.

Nota : Para el sistema operativo Windows hay que abrir el Panel de Control → Sistema → Avanzado. Una vez ahí se da clic en el botón Variables de entorno, hay que localizar la variable PATH en la lista superior, y crear la variable de CLASSPATH.



Un programa en java se puede desarrollar desde un editor de texto, tales como vi (Linux) o block de notas (Windows).

De igual manera, existen diferentes entornos de desarrollo integrados (IDE) para este fin, algunos de ellos son:

- **Netbeans**
- **Eclipse**
- **JCreator**
- **JBuilder**
- **Context**

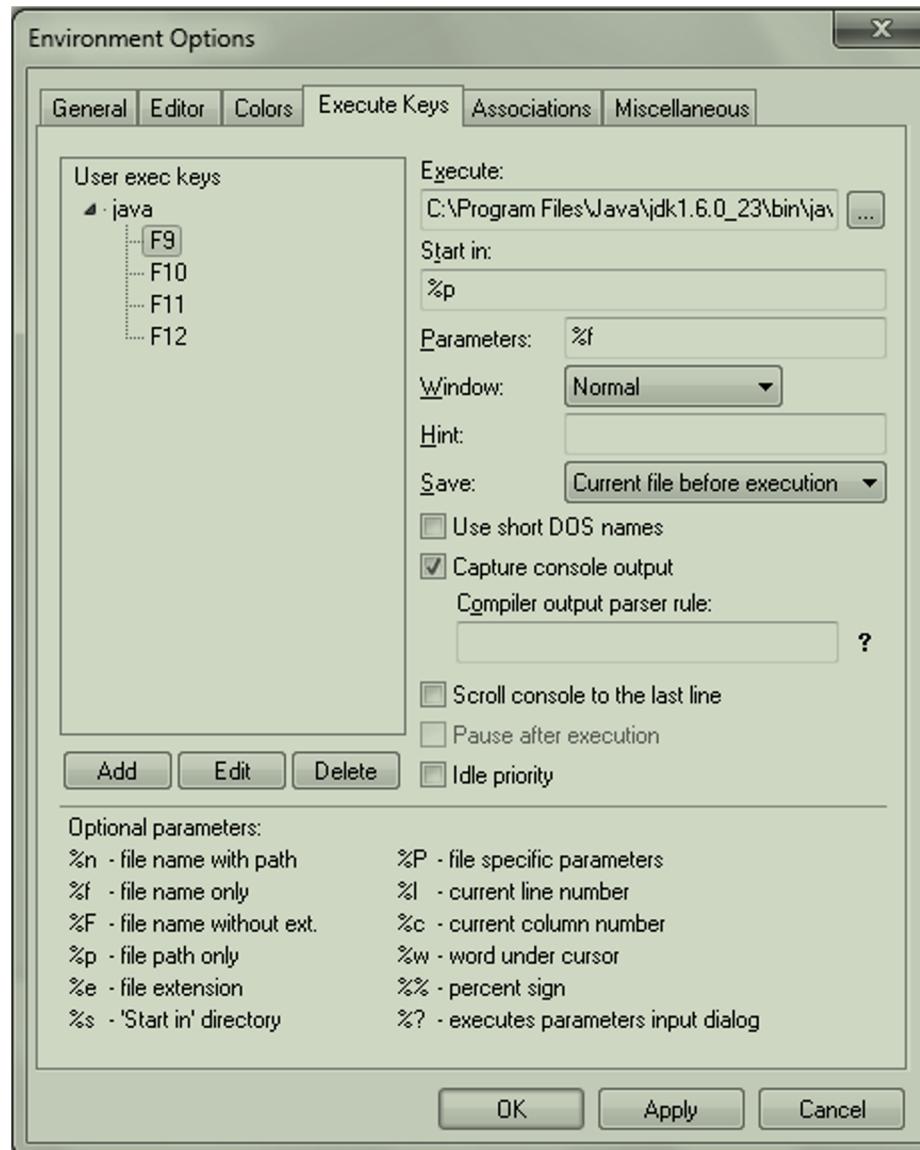


ConTEXT es un editor de texto gratuito, ligero, rápido y poderoso creada como una herramienta secundaria para desarrolladores de software. Existe la versión de escritorio y la versión portable.

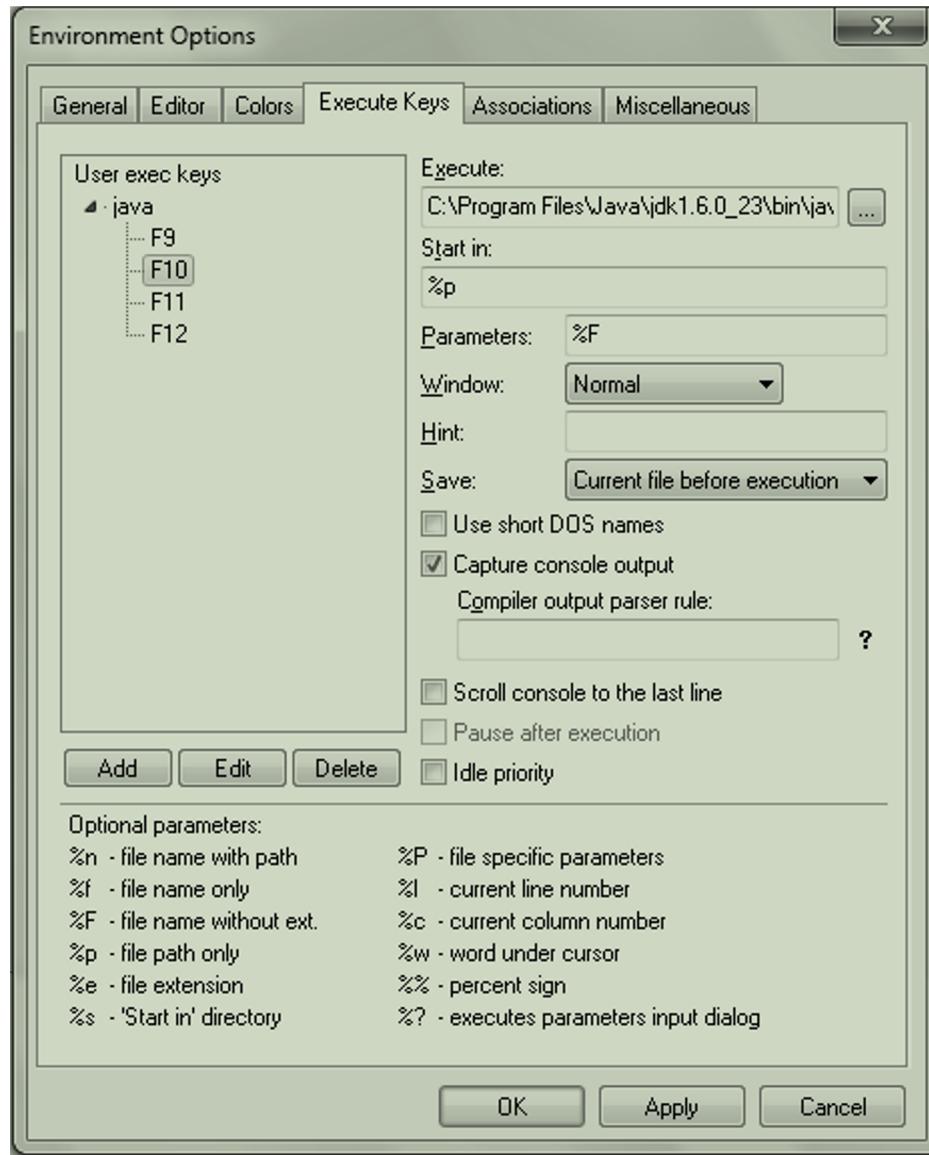
Soporta diferentes lenguajes de programación: C#, C/C++, Delphi/Pascal, Java, JavaScript, Visual Basic, Perl/CGI, HTML, CSS, SQL, FoxPro, 80x86 assembler, Python, PHP, Tcl/Tk, XML, Fortran, Foxpro, InnoSetup scripts.

<http://www.contexteditor.org/>

Compilar (javac.exe)



Ejecutar (java.exe)



Método principal

Un programa en Java se inicia proporcionando al intérprete del lenguaje el nombre de la clase.

La JVM carga en memoria la clase indicada e inicia su ejecución buscando dentro de ella el método estático principal.

El nombre de este método es main (método pivote) y debe declararse de la siguiente forma:

```
public static void main (String [] argumentos) {}
```

Identificadores

Un identificador es un nombre que permite llamar a alguna de las entidades (objetos, atributos o métodos) propias del lenguaje.

Los nombres siguen las siguientes reglas:

- **Comienzan con una letra (A-Z, a-z), el carácter sub-guion (_) o el carácter pesos (\$).**
- **Puede incluir (pero no comenzar) un número.**
- **No puede incluir el carácter espacio en blanco.**
- **No se pueden utilizar palabras reservadas.**

Java es sensible a mayúsculas y minúsculas. Si se quiere crear un identificador con un nombre compuesto (dos o más palabras), la primera palabra va en minúscula y las primeras letras de las demás palabras van en mayúscula (aulaClase o aulaDeClase).

Convenciones

Tipo de identificador	Convención	Ejemplo
Clase	Cada palabra del identificador comienza con letra mayúscula.	Rectangulo, MetodoNewtonRaphson, InterpolacionNewton
Método	La primera palabra va en minúsculas y las primeras letras de las palabras restantes van en mayúscula.	calcularArea, getValor, setValor, interpolar
Atributo / variable	La primera palabra va en minúsculas y las primeras letras de las palabras restantes van en mayúscula.	area, perimetro, objPoligono, triangulo1,
Constantes	Con letras mayúsculas.	PI, MAX_TAM, RADIO

Palabras reservadas

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

null, true y false son literales reservadas.

Clases

Las clases pueden contener:

- **Atributos (se denominan Datos Miembro). Estos pueden ser de tipos primitivos o referencias.**
- **Métodos (se denominan Métodos Miembro).**

La sintaxis general para declarar una clase es la siguiente:

```
[modificadores] class NombreClase {  
    [declaración de datos miembro]  
    [declaración de métodos miembro]  
    [Comentarios]  
}
```

Por ejemplo:

```
class Punto {  
    int x, y;  
}
```

Objetos

Un objeto es una instancia (ejemplar) de una clase, es decir, un objeto es la materialización concreta (en memoria) de una clase.

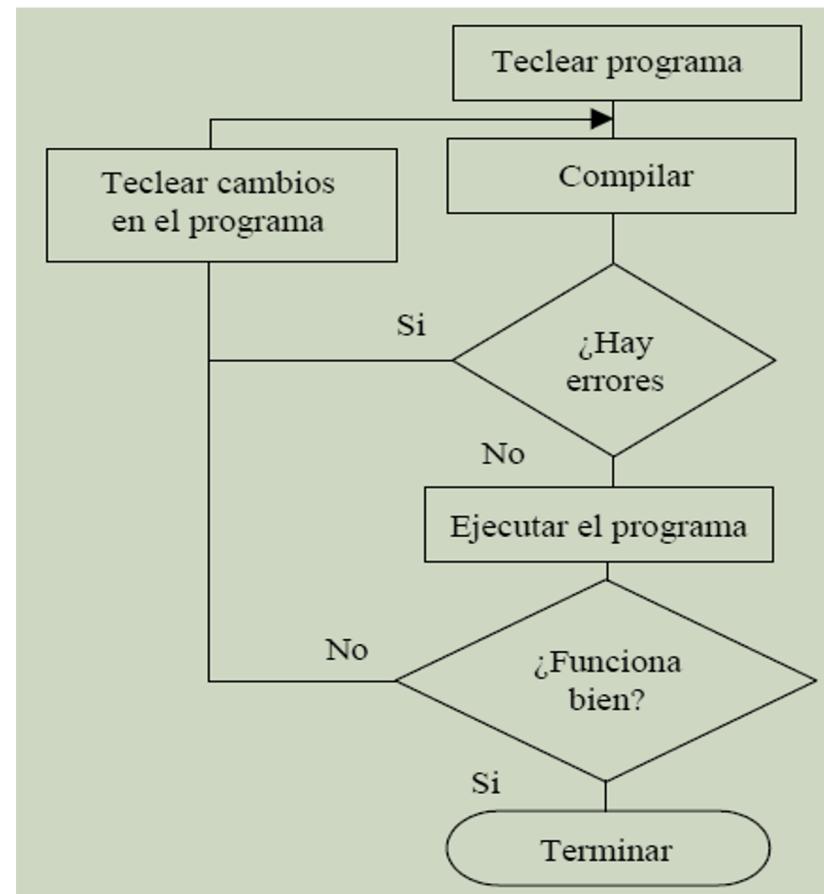
Los objetos se declaran igual que las variables primitivas:

```
NombreClase nombreObjeto = new NombreClase();
```

A los miembros de un objeto se accede a través de su referencia:

```
nombreObjeto.variable;  
nombreObjeto.metodo();
```

Proceso de creación



Mi primer programa

Hola Mundo!!

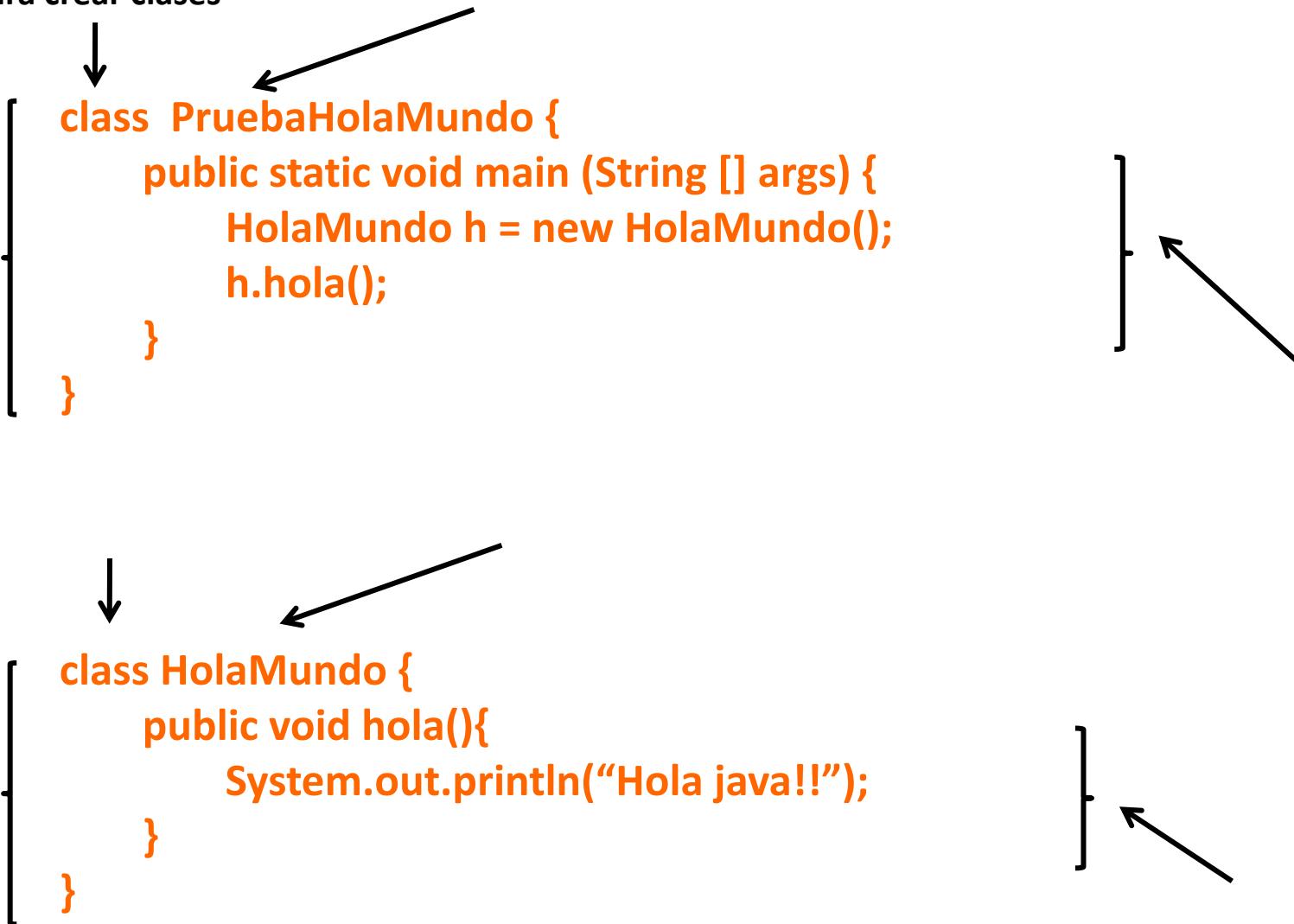


Ejemplo 4.2

Palabra reservada
para crear clases

```
class PruebaHolaMundo {  
    public static void main (String [] args) {  
        HolaMundo h = new HolaMundo();  
        h.hola();  
    }  
}
```

```
class HolaMundo {  
    public void hola(){  
        System.out.println("Hola java!!");  
    }  
}
```



Ejemplo 4.2

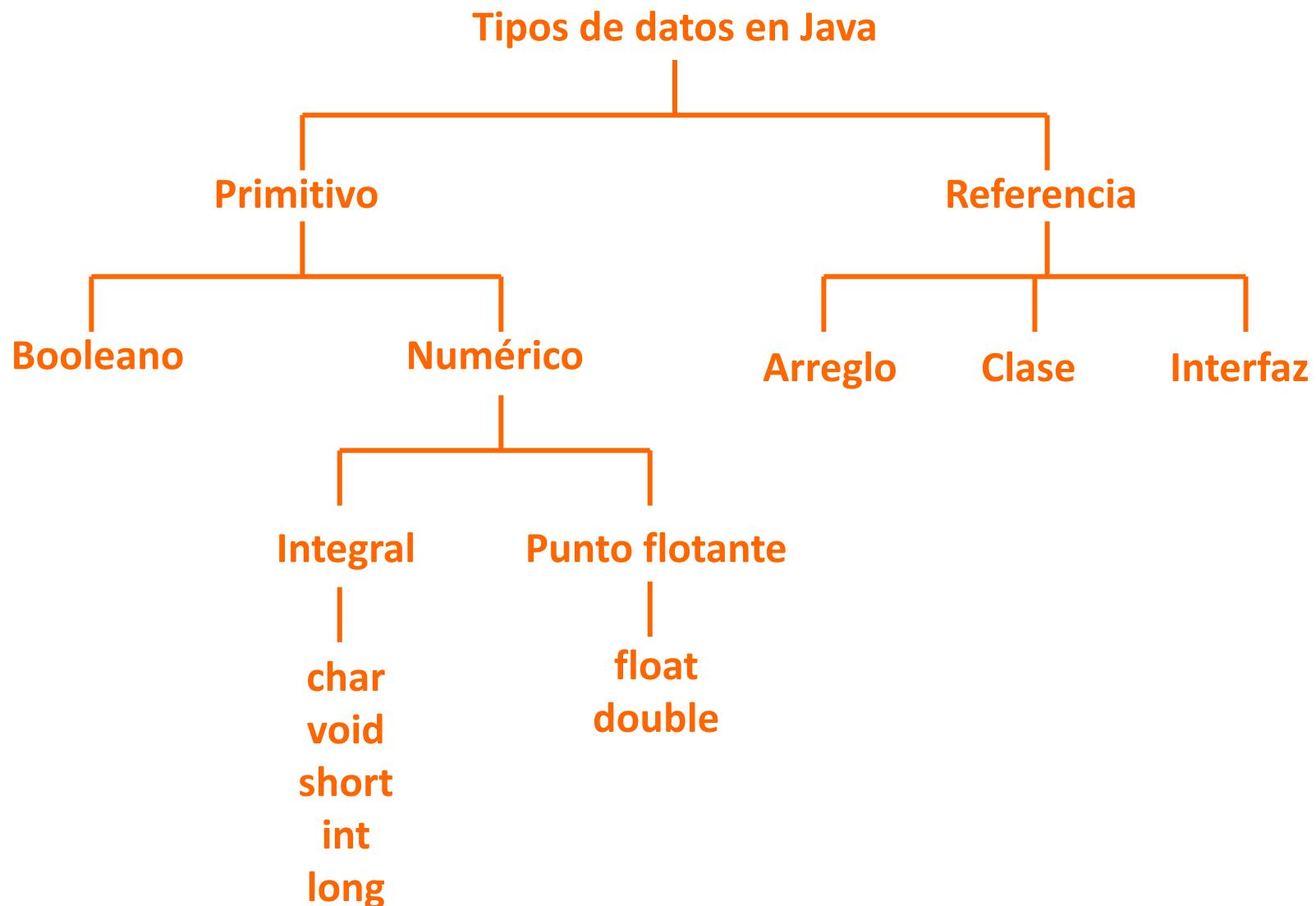
Para compilar las clases se utiliza el comando `javac`. Para ejecutar una clase se utiliza el comando `java`.

Para el ejemplo anterior, es necesario compilar ambas clases:

```
javac HolaMundo.java  
javac PruebaHolaMundo.java
```

Se debe ejecutar la clase que posee el método `main`, en este caso, `PruebaHolaMundo`:

```
java PruebaHolaMundo
```



Tipos Simples	Tamaño
---------------	--------

boolean	1-bit
char	16-bit
byte	8-bit
short	16-bit
int	32-bit
long	64-bit
float	32-bit
double	64-bit

Ejemplos:

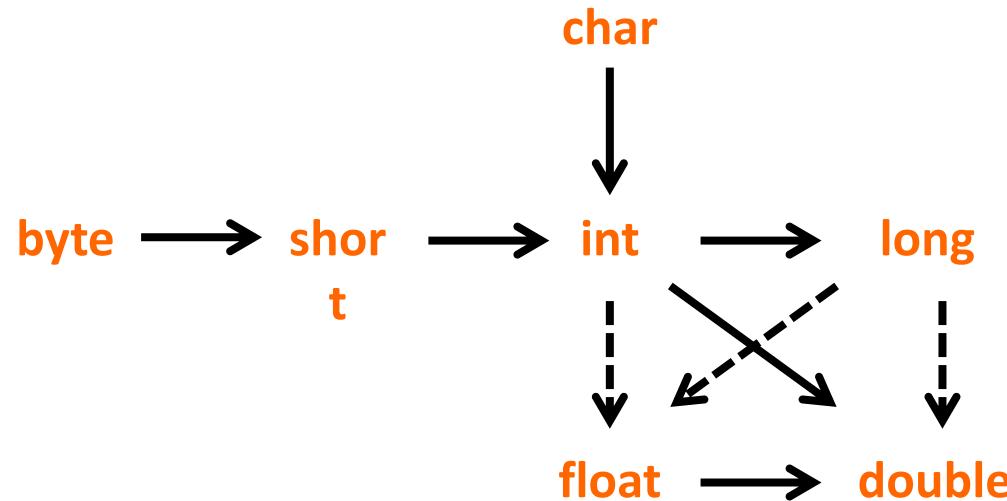
```
double salario;
int a = 10;
float x, y, z;
final double PI = 3.1416;
boolean exito;
```

Caracteres:

```
\n // Salto de línea
\t // Tabulador
\b // Backspace
\r // Retorno de carro
```

Conversiones entre tipos

Las flechas sólidas muestran conversiones sin pérdida de información, en las conversiones con flechas punteadas puede haber pérdida de información.



Ejemplos de cast:

```
double x = 9.997;  
int castX = (int)x          // castX = 9  
String y = "5";  
int castY = Integer.parseInt(y); // castY = 5
```

Operadores sobre tipos primitivos

Aritméticos		
+	Suma	$a + b$
-	Resta	$a - b$
*	Multiplicación	$a * b$
/	División	a / b
%	Módulo	$a \% b$

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

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

Operadores sobre tipos primitivos

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

Operadores sobre tipos primitivos

A nivel de bits		
&	AND	a & b
	OR	a b
^	XOR	a ^ b
<<	Desplazamiento a la izquierda	8 << 1 (8 mult 2)
>>	Desplazamiento a la derecha rellenando con 1	8 >> 1 (8 div 2)
>>>	Desplazamiento a la derecha rellenando con 0	-8 >>> 1

Lógicos		
&&	AND	a && b
	OR	a b
!	NOT	! a

Funciones y constantes matemáticas

Funciones trigonometrías:

- **Math.sqrt**
- **Math.sin**
- **Math.cos**
- **Math.tan**

Función exponencial y logaritmo natural:

- **Math.exp**
- **Math.log**

Constantes:

- **Math.PI**
- **Math.E**

Por ejemplo, para obtener la raíz cuadrada de 4:

```
double x = 4;  
double y = Math.sqrt(x);
```

Clase String

El manejo de cadenas de caracteres está definida en el API de Java mediante la clase String.

La sintaxis o declaración de los objetos tipo String es la siguiente:

```
String cad = new String("Aquí va la cadena de caracteres");  
String cad2 = "Otra manera de declarar una cadena";
```

El operador + permite concatenar (unir) dos cadenas de caracteres:

```
String cad3 = cad + cad2;
```

También es posible concatenar datos primitivos a una cadena de caracteres, por ejemplo:

- `int a = 4;`
- `String cadena = "El valor de a es: " + a`

Cuando se utiliza el operador `+` y una de las variables de la expresión es un `String`, Java transforma la otra variable (si es de tipo primitivo) en un `String` y las concatena.

Si la otra variable es una referencia a un objeto entonces invoca el método `toString()` que existe en todas las clases (es un método de la clase `Object`).

Métodos de la clase String

Método	Descripción
<code>char charAt(int index)</code>	Devuelve el carácter en la posición indicada por index. El rango de index va de 0 a length() - 1.
<code>boolean equals(Object obj)</code>	Compara el String con el objeto especificado. El resultado es true si y solo si el argumento es no nulo y es un objeto String que contiene la misma secuencia de caracteres.
<code>boolean equalsIgnoreCase(String s)</code>	Compara el String con otro, ignorando consideraciones de mayúsculas y minúsculas. Los dos Strings se consideran iguales si tienen la misma longitud y, los caracteres correspondientes en ambos Strings son iguales sin tener en cuenta mayúsculas y minúsculas.
<code>int indexOf(char c)</code>	Devuelve el indice donde se produce la primera aparición de c. Devuelve -1 si c no está en el string.
<code>int indexOf(String s)</code>	Igual que el anterior pero buscando la subcadena representada por s.
<code>int length()</code>	Devuelve la longitud del String (número de caracteres)
<code>String substring(int begin, int end)</code>	Devuelve un substring desde el índice begin hasta el end
<code>static String valueOf(int i)</code>	Devuelve un string que es la representación del entero i. Observese que este método es estático. Hay métodos equivalentes donde el argumento es un float, double, etc.
<code>char[] toCharArray()</code> <code>String toLowerCase()</code> <code>String toUpperCase()</code>	Transforman el string en un array de caracteres, o a mayúsculas o a minúsculas.

Arreglos

Un arreglo es una colección ordenada de elementos del mismo tipo, que son accesibles a través de un índice.

Un arreglo puede contener tanto datos primitivos como referencias a objetos.

La sintaxis para declarar un arreglo es la siguiente:

[modificadores] tipoDeVariable [] nombre;

Declaración e inicialización de arreglos:

int [] a;

a = new int [5];

Punto [] pto;

pto = new Punto [3];

int [] b = {1, 5, 7};

int s={3,4};

char[] s,t,r; -> Todas las variables son arreglos.

char s[],t,r; -> Solo s es un arreglo.

int [] s = new int[]{3,4}; ->Se declara y se inicializa.

int [][] t = new t[4][]; -> Declaración legal

int [][] t = new t[][][4]; -> Declaración ilegal

La longitud de un arreglo se puede conocer utilizando la variable `length`:

```
int arr = {44, 26, 34, 55};  
int tam = arr.length; // tam = 4
```

Un método puede recibir como parámetro y/o devolver como valor de retorno un arreglo.

`String [] metodo (Punto []){}`

También se pueden declarar arreglos de diferentes dimensiones:

```
int [ ][ ] a = { { 1 , 2 } , { 3 , 4 } , { 5 , 6 } };  
int y = a[2][1];
```

```
public class Caracteres {  
    public static void main (String [] a) {  
        char [] x = new char[4];  
        x[0] = 'a';  
        x[1] = 'b';  
        x[2] = 'c';  
        x[3] = 'd';  
        System.out.println(x);  
    }  
}
```

Comentarios

Java maneja dos tipos de comentarios: simples y por bloques:

// Comentario de una línea

/* comienzo de comentario
continua comentario
fin de comentario */

/** comienzo de comentario de documentación
continua comentario de documentación
fin de comentario de documentación */

Inicialización de variables

Las variables se clasifican en dos grupos con base en el lugar donde se declaran: de instancia y locales.

Las variables de instancia son aquellas variables que se declaran dentro de una clase, fuera de cualquier método, es decir, los atributos de la clase.

Las variables locales (automática, temporal o de pila) son aquellas variables que se declaran y usan, exclusivamente, en un bloque de código dentro de algún método

Las variables de instancia se inicializan automáticamente de la siguiente manera:

- Las numéricas con 0.
- Las booleanas con false.
- Las char con el carácter nulo (0h).
- Las referencias con null (literal que indica referencia nula).

Las variables locales no se inicializan automáticamente, es decir, es necesario asignarles un valor antes de ser usadas. Si el compilador detecta que una variable local se utiliza antes de que se le asigne un valor, genera un error.

Por ejemplo, se tiene el siguiente bloque de código:

```
public static void main(String[] args) {  
    int p;  
    int q = p; // error  
}
```

Si se ejecuta el código anterior, el compilador emitirá la siguiente salida:

variable p might not have been initialized

De igual manera, si se tiene una variable local que podría no haberse inicializado, como en el siguiente bloque de código:

```
int p;
if ( . . . ) { p = 5;
}
int q = p; // error
```

el compilador emitirá un error del siguiente tipo:

variable p might not have been initialized

Esto es debido a que, si la condición no se cumple, la variable p no tiene ningún valor y, por tanto, no existe valor alguno que se le pueda asignar a la variable q.

Ámbito de las variables

El ámbito de una variable es el área del programa donde la variable existe (es reconocida) y, por tanto, puede ser utilizada.

El ámbito de una variable de instancia esta dado por el tiempo de vida de un objeto, es decir, una variable de instancia existe y puede ser usada mientras el objeto al que pertenece sea referenciado.

Un objeto es referenciado desde el momento en que se instancia y mientras existe una referencia que apunte a él. Cuando la variable que lo instancia deja de hacerlo, el objeto queda sin referencia y el espacio de memoria ocupado por éste, puede ser recuperado por la JVM en cualquier momento.

El proceso de recuperación de espacio en memoria lo realiza el recolector de basura (garbage collector).

El ámbito de las variables locales (automáticas, temporales o de stack) abarca, exclusivamente, el bloque de código donde se declaran; fuera de ese bloque la variable es irreconocible.

```
metodoEquis() {  
    int x; // Inicia el ámbito de x.  
    if (condicion) {  
        int q; // Inicia el ámbito de q.  
    } // finaliza el ámbito de q.  
} // finaliza el ámbito de x
```

Modificadores de acceso

Los modificadores son elementos del lenguaje que se colocan antes de definir un elemento (atributos, métodos o, incluso, clases) y que alteran o condicionan el acceso o significado de dicho elemento.

Por lo tanto, los modificadores de acceso permiten determinar el acceso a los datos y métodos miembros de una clase. Los modificadores de clases se van a tratar más adelante.

Los modificadores de acceso preceden a la declaración de un elemento de la clase (ya sea dato o método), de la siguiente forma:

- [modificadores] tipo_variable nombre;
- [modificadores] tipo_devuelto metodo(lista_Argumentos);

Los modificadores de acceso que posee java son:

- **public:** Se puede acceder al elemento en cualquier momento. Si es un dato miembro, todo el mundo puede ver el elemento, es decir, usarlo y asignarlo. Si es un método todo el mundo puede invocarlo.
- **private:** Sólo se puede acceder al atributo desde métodos de la clase. Sólo puede invocarse un método privado desde otro método de la clase.
- **protected:** Proporciona acceso público para las clases derivadas y acceso privado (prohibido) para el resto de clases.
- **Default (sin modificador):** Se puede acceder al elemento desde cualquier clase del mismo paquete donde se define la clase.

	public	protect ed	sin modifica dor	private
Clase	Sí	Sí	Sí	Sí
Subclase en el mismo paquete	Sí	Sí	Sí	No
No-Subclase en el mismo paquete	Sí	Sí	Sí	No
Subclase en diferente paquete	Sí	Sí	No	No
No-Subclase en diferente paquete (Universo)	Sí	No	No	No

Herencia

La herencia permite crear una clase a partir de la definición de otra ya existente. La herencia permite compartir automáticamente atributos y métodos entre clases, subclases y objetos.

La herencia se utiliza para reutilizar código, es decir, el código de cualquiera de las clases puede ser utilizado con el simple hecho de crear una clase derivada de ella, es decir, heredar de ella.

Por ejemplo, se tiene la clase Empleado:

```
class Empleado {  
    String nombre;  
    int numEmpleado , sueldo;  
  
    public void aumentarSueldo(int porcentaje) {  
        sueldo += (int)(sueldo * porcentaje / 100);  
    }  
  
    public String toString() {  
        return "Num. Empleado " + numEmpleado + " Nombre: "  
            + nombre + " Sueldo: " + sueldo;  
    }  
}
```

Para realizar herencia en Java se utiliza la palabra reservada `extends` al momento de definir la clase, es decir:

```
class Ejecutivo extends Empleado {  
    int presupuesto;  
    void asignarPresupuesto(int p) {  
        presupuesto = p;  
    }  
}
```

Los objetos de las clases derivadas (subclases) se crean (instancian) igual que los de la clase base y pueden acceder tanto sus atributos y métodos como a los de la clase base.

Como lo dicta la programación orientada a objetos, es posible redefinir algunos métodos de la clase base (sobre-escritura), es decir, métodos con el mismo nombre pero con comportamientos distintos.

```
class Ejecutivo extends Empleado {  
    int presupuesto;  
    void asignarPresupuesto(int p) {  
        presupuesto = p;  
    }  
    public String toString() {  
        String s = super.toString();  
        s = s + " Presupuesto: " + presupuesto;  
        return s;  
    }  
}
```

En el ejemplo anterior se sobre-escribe el método `toString` de la clase base. La palabra reservada `super` representa una referencia interna implícita a la clase base (superclase).

Mediante la sentencia `super.toString()` se invoca el método `toString` de la clase `Empleado`.

Cast

Si se desea acceder a los métodos de la clase derivada teniendo una referencia de la clase base, es posible convertir explícitamente la referencia de un tipo a otro. Esto se hace con el realizando un casting de la siguiente forma:

```
Empleado emp = new Ejecutivo();
// se convierte la referencia creada
Ejecutivo ej = (Ejecutivo)emp;
// Se ejecuta un método de la subclase Ejecutivo
ej.asignarPresupuesto(1500);
```

La referencia this

La palabra reservada **this** es una referencia implícita que tienen todos los objetos y que apunta a si mismo. Por ejemplo:

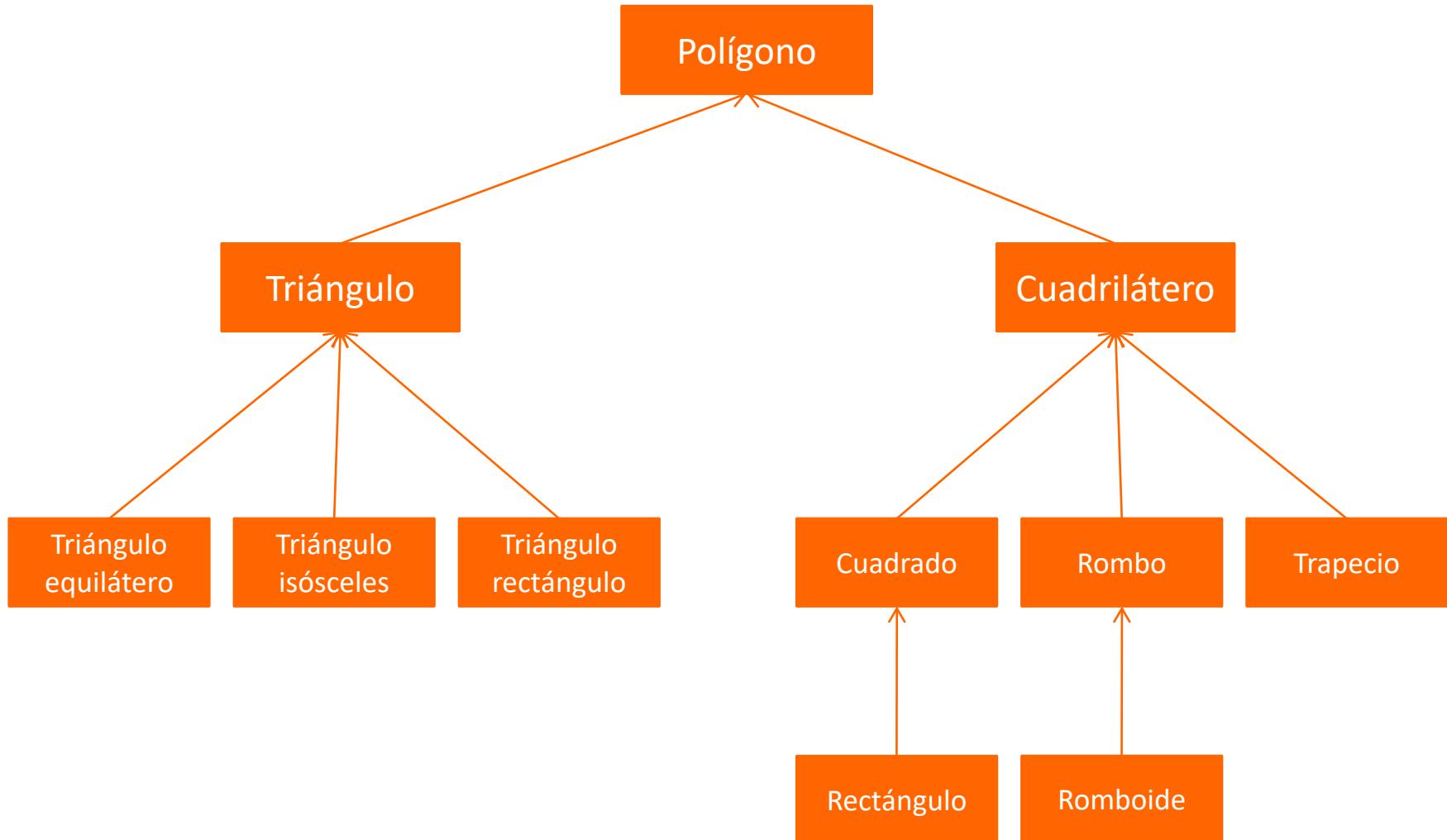
```
class Circulo {  
    Punto centro;  
    int radio;  
    Circulo elMayor(Circulo c) {  
        if (this.radio > c.radio)  
            return this;  
        else  
            return c;  
    }  
}
```

El método `elMayor` devuelve una referencia al círculo que tiene mayor radio, comparando el radio del Círculo `c` que se recibe como argumento y el radio de la propia clase. En caso de que el propio radio resulte mayor, el método debe devolver una referencia a si mismo. Esto se realiza con la expresión `this`.

`this` evita lo que se llama shadowing (el nombre de una variable le hace sombra a otra variable porque poseen el mismo nombre).

Ejercicio 4.1

Programar la siguiente jerarquía de clases:



4.2 Control de flujo

El control de flujo determina el orden en el que se ejecutarán las instrucciones en los programas.

Si no existieran sentencias (o instrucciones) para el control de flujo, los programas se ejecutarían de forma secuencial, es decir, instrucción por instrucción.

El panorama anterior sería muy restringido, debido a que no se podrían evaluar diferentes soluciones o secuencias de código en función de ciertas condiciones (sentencias alternativas).

Además, no podrían ejecutar un bloque de código en repetidas ocasiones (sentencias repetitivas), y, por lo tanto, el bloque de código se haría muy grande.

Empero, existen dos tipos de instrucciones de control de flujo: las sentencias de control alternativas y repetitivas.

Las sentencias alternativas, también conocidas como sentencias selectivas, permiten seleccionar de entre varios caminos uno por donde seguirá la ejecución del programa, con base en una condición lógica.

Las sentencias repetitivas, también conocidas como sentencias iterativas, permiten ejecutar un bloque de código en repetidas ocasiones hasta que se cumpla la condición lógica deseada.

Java soporta varias sentencias o instrucciones de control de flujo, como lo son:

- **Toma de decisiones: if-else, switch-case**
- **Bucles o ciclos: for, while, do-while**
- **Excepciones: try-catch-finally, throw**

4.2.1 Sentencia if-else

La sentencia if-else proporciona la posibilidad de ejecutar selectivamente dos bloques diferentes de código, con base en una condición lógica.

La versión más sencilla de esta sentencia es la instrucción if: la sentencia gobernada por if se ejecuta si la codición evaluada es verdadera.

```
If (condicion_logica) {  
    // Si la condición lógica se cumple, es decir  
    // si condicion_logica == true, se ejecuta  
    // este bloque de código  
}
```

La sentencia completa, permite ejecutar otro bloque de código, si la condición lógica no se cumple, es decir, si condicion_logica es igual a false.

```
If (condicion_logica) {  
    // Si la condición lógica se cumple, es decir  
    // si condicion_logica == true, se ejecuta  
    // este bloque de código  
} else {  
    // Si la condición lógica NO se cumple, es decir  
    // si condicion_logica == false, se ejecuta  
    // este bloque de código  
}
```

Cada parte de la sentencia (el bloque if o el bloque else) pueden contener otra sentencia if o if-else.

```
If (condicion_logica) {  
    If (condicion_logica) {  
        // Bloque de código  
    } else {  
        // Bloque de código  
    }  
} else {  
    If (condicion_logica) {  
        // Bloque de código  
    } else {  
        // Bloque de código  
    }  
}
```

Cuando el bloque de código está constituido por una sola instrucción, es más elegante utilizar el operador ternario. La sintaxis del operador ternario es la siguiente:

`condicion_logica ? expresion1: expresion2`

El operador ternario evalúa la condición lógica y si es verdadera, se ejecuta la `expresion1`, si es falsa se ejecuta la `expresion2`.

Por ejemplo:

`x < y ? x : y`

Devolverá el valor más pequeño entre x y y.

Ejemplo 4.3.1

```
public class IfElse {  
  
    public static void main (String [] args) {  
        int a=45;  
        int res = a<0 ? a*(-1) : a;  
        if ( a%2 == 0 ){  
            System.out.println(a + " es un nUmero par\n");  
        } else {  
            System.out.println(a + " es un nUmero  
impar\n");  
        }  
    }  
}
```

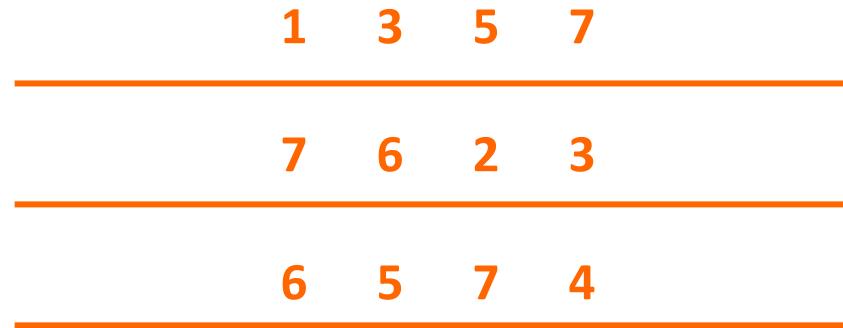
```
class IfElse2 {  
    public static void main (String [] args) {  
        int x=0;  
        int y=0;  
        if (( ++x > 2 ) && (++y > 2))  
            x++;  
        System.out.println("x = " + x + " y = " + y);  
        x=3;  
        y=1;  
        if (( ++x > 2 ) && (++y > 2))  
            x++;  
        System.out.println("x = " + x + " y = " + y);  
        x=3;  
        y=3;  
        if (( ++x > 2 ) && (++y > 2))  
            x++;  
        System.out.println("x = " + x + " y = " + y);  
    }  
}
```

Ejemplo 4.3.2

Ejercicio 4.2

Número mágico

Piensa en un número entre 0 y 7



Programar el juego del número mágico para 16 números
(del 0 al 15).

4.2.2 Sentencia switch

La sentencia switch (o de evaluación múltiple) se utiliza para realizar diferentes acciones con base en una expresión, es decir, buscando la expresión dentro de los casos definidos para el switch.

La sentencia switch evalúa la expresión dada (generalmente un entero, aunque es posible evaluar un carácter) y, si la expresión se encuentra definida en el cuerpo del switch (en algún case), se ejecuta el caso apropiado

En caso de que la expresión no esté contemplada en ningún caso se ejecuta la opción default.

La sintaxis de la sentencia switch es

```
switch ( expresion ) {  
    case valor1:  
        // bloque de código  
        break;  
    case valor2:  
        // bloque de código  
        break;  
    ...  
    default:  
        // bloque de código en caso de que el  
        // valor evaluado en la expresión no exista  
        // en el switch  
}
```

Switch

Ejemplo 4.4.1

```
public class Switch {  
    public static void main (String [] args) {  
        int a = 2;  
        switch(a){  
            case 1:  
                System.out.println("Se eligió la opción 1");  
                break;  
            case 2:  
                System.out.println("Se eligió la opción 2");  
                break;  
            case 3:  
                System.out.println("Se eligió la opción 3");  
                break;  
            default:  
                System.out.println("No se eligió nada");  
        }  
    }  
}
```

Switch

Ejemplo 4.4.2

```
class SwitchChar {  
    public static void main (String [] args) {  
        char caracter = 'a';  
        switch(caracter){  
            case 'a':  
                System.out.println("Se eligió A");  
                break;  
            case 'b':  
                System.out.println("Se eligió B");  
                break;  
            case 'c':  
                System.out.println("Se eligió C");  
                break;  
            default:  
                System.out.println("No se eligió nada");  
        }  
    }  
}
```

Ejemplo 4.4.3

Switch

```
class SwitchSinBreak {  
    public static void main(String [] args) {  
        int z=2;  
        final short x=2;  
        switch (z) {  
            case x: System.out.print("0 ");  
            case x-1: System.out.print("1 ");  
            case x-2: System.out.print("2 ");  
        }  
        System.out.println("\n");  
    }  
}
```

Ejercicio 4.3

Calculadora

Realizar una calculadora que sea capaz de realizar las operaciones básicas y además pueda obtener el módulo, elevar al cubo, elevar al cuadrado, obtener el seno, el coseno, la tangente y el logaritmo natural.

Las opciones se seleccionan de un menú. El usuario decide cuando salir de la aplicación. Las operaciones se realizan desde otra clase diferente a donde se implemente el método pivote.

4.2.3 Ciclo for

El ciclo for es una sentencia o bucle repetitivo que nos permite realizar de manera ciclica un conjunto de instrucciones hasta que se cumpla la condición establecida.

La sintaxis de este ciclo es la siguiente:

```
for ( inicializar ; condicion_logica ; paso ){
    // Código a ejecutarse hasta que la condición
    // lógica se cumpla
}
```

De donde:

- **Iniciar:** es una sentencia que se ejecuta la primera vez que se entra en el ciclo `for`. Normalmente es una asignación. Es un campo opcional.
- **expresion_logica:** es una expresión que se evalúa antes del bloque de código, para cada iteración. La sentencia o bloque de sentencias se ejecutan hasta que la `expresion_logica` se cumpla. Es un campo opcional.
- **paso:** es una sentencia que se ejecuta cada vez que se llega al final del bloque de código (la llave final). Es un campo opcional.

Por lo anterior, si se omiten las tres cláusulas del ciclo for se obtiene un ciclo infinito:

```
for ( ; ; ){
    // Ciclo infinito
}
```

Ejemplo 4.5.1

Parámetros o argumentos de entrada

```
public class ForArgs{  
    public static void main(String args[]){  
        if ( args.length == 0 ){  
            System.out.println("ERROR!, se deben proporcionar argumentos  
                al ejecutar el programa!");  
            System.out.println("Uso: java ForArgs ARG1 ARG2 ... ARGn");  
        }  
  
        for ( int i = 0 ; i < args.length ; i++ ){  
            System.out.println("Parametro " + (i+1) + ":" + args[i]);  
        }  
    }  
}
```

Ejemplo 4.5.2

Tipos de datos

```
public class ForDouble{  
    public static void main (String [] fors){  
        double a;  
        for (a = 0.1; a < 1 ; a+=0.1){  
            System.out.println("a = " + a);  
        }  
    }  
}
```

Ejemplo 4.5.3**Arreglos**

```
public class ForArreglo{  
    public static void main (String [] fors){  
        int [][] x = new int [3][];  
        int i, j;  
        x[0] = new int [4];  
        x[1] = new int [2];  
        x[2] = new int [5];  
        for (i = 0 ; i < x.length ; i++){  
            for (j = 0 ; j < x[i].length ; j++){  
                x[i][j] = i+j+1;  
                System.out.print("x["+i+"]"+"["+j+"] = " + x[i][j]+\t");  
            }  
            System.out.println();  
        }  
    }  
}
```

Foreach

Existe una variante del ciclo for conocida como foreach. Foreach es una sentencia que permite recorrer de forma rápida un arreglo o un enumerador.

La sintaxis del ciclo foreach es la siguiente:

```
for ( tipoDatos x ; arreglo/enumerador ){
    // Código a ejecutarse hasta se termine de
    // recorrer el arreglo o el enumerador
}
```

Ejemplo 4.5.4**foreach**

```
public class Foreach {  
    public static void main (String [] a){  
        int [] nums = {2,3,4,5};  
        for(int x : nums){  
            System.out.println(x);  
        }  
    }  
}
```

Argumentos variables (varargs)

Los argumentos variables se utilizan únicamente como parámetro, es decir, en métodos. Su sintaxis es la siguiente:

```
[modifi] tipoRetorno nombre (tipoDatos ... nombre) {  
    // Código método  
}
```

Indica que el método puede recibir de 0 a n variables de tipoDatos y se va a hacer referencia a esas variables con el identificador nombre.

Solo es posible crear un tipo de argumento variable por método y, si el método recibe más de un parámetro, los argumentos variables debe ir hasta el final de la declaración de argumentos.

Ejemplo 4.5.5

```
public class Varargs {  
    void metodo(int i, String...ok){  
        System.out.println("ITERACIoN " + i);  
        for (int a = 0 ; a < ok.length ; a++ ) {  
            System.out.println(ok[a]);  
        }  
    }  
    public static void main (String ... a){  
        String [] args = {"hola", "adios", "ok"};  
        new Varargs().metodo(1);  
        new Varargs().metodo(2,"un argumento");  
        new Varargs().metodo(3, args);  
    }  
}
```

Palíndromo

Ejercicio 4.4

Frase que puede ser leída de izquierda a derecha o de manera inversa sin sufrir cambios, por ejemplo:

Ave		y		Eva
La	ruta		no	natural
Se es o no se es				

Realizar un programa que determine si una frase es o no un palíndromo. La frase se debe ingresar al momento de ejecutar el programa:

`java Palindromo "Ave y Eva"`

Si se utilizan arreglos, los recorridos se deben realizar mediante la sentencia `foreach`.

4.2.4 Ciclo while y do-while

El ciclo while es un bucle de control que se utiliza para ejecutar un conjunto de instrucciones varias veces hasta que la condición lógica se cumpla. La sintaxis del ciclo while es la siguiente:

```
while (condicion_logica) {  
    // bloque de código a ejecutar  
    // hasta que se cumpla la condición  
    // lógica  
}
```

Este ciclo primero evalúa la condición lógica y, si no se cumple (`condicion_logica == false`), ejecuta el bloque de código.

El ciclo do-while es un bucle de control que se utiliza para ejecutar un conjunto de instrucciones varias veces hasta que la condición lógica se cumpla. La sintaxis del ciclo do-while es la siguiente:

```
do{  
    // bloque de código a ejecutar  
} while (condicion_logica)
```

Al igual que el ciclo while, es un ciclo repetitivo, sin embargo, este ciclo ejecuta por lo menos una vez el bloque de código debido a que la condición lógica se evalúa al final del bloque de código.

Ejemplo 4.6.1

```
public class DoWhile {  
  
    public static void main (String [] args) {  
        int x = 0;  
        int y = 10;  
        System.out.println("Ciclo while");  
        while(x < y) {  
            System.out.println("x = " + x++);  
        }  
        System.out.println("Segundo iterador");  
        do {  
            System.out.println("x = " + x--);  
        } while ( x > 0 );  
    }  
}
```

Ejemplo 4.6.2

```
public class DoWhile2 {  
    public static void main(String [] args) {  
        int cont = 1;  
        do  
            while ( --cont<1 )  
                System.out.print("Contador = " + cont);  
            while ( ++cont>1 );  
    }  
}
```

Ejercicio 4.5

Conjetura de ULAM

A partir de un número entero positivo realizar lo siguiente: Si es par dividirlo entre 2; si es impar multiplicarlo por 3 y sumar 1. Obtener enteros positivos repitiendo el proceso hasta llegar a 1.

Realizar un programa que permita realizar la conjetura de ULAM para cualquier número entero dado.

Break

La sentencia break finaliza el flujo de código dependiendo de la estructura de control donde se encuentre.

Dentro de una caso (case) en una sentencia switch, break finaliza el flujo de código establecido para cada caso, es decir, termina el caso en la sentencia switch.

Dentro de un ciclo, la sentencia break forza la terminación inmediata de un ciclo, saltando la prueba condicional normal del ciclo, evitando así que este siga iterando.

Ejemplo 4.6.3

```
class Break {  
    public static void main (String [] a){  
        int x = 0, y = 0;  
        do {  
            System.out.println("Inicia do-while interno, x = " + x);  
            do {  
                x++;  
                System.out.println("Inicia if");  
                if (x>2) {  
                    break;  
                }  
                System.out.println("Termina if, x = " + x);  
            } while (x<3);  
            y++;  
            System.out.println("Termina do-while interno, x = " + x + ", y = "+ y);  
        } while (y<5);  
        System.out.println("Termina do-while externo");  
    }  
}
```

Ejemplo 4.6.4

```
class BreakEtiqueta {  
    public static void main (String [] a){  
        int x = 0;  
        fuera:  
        do {  
            do {  
                x++;  
                System.out.println("Inicia if");  
                if (x>2) {  
                    break fuera;  
                }  
                System.out.println("Termina if, x = " + x);  
            } while (x<3);  
            x++;  
            System.out.println("Termina do-while interno, x = " + x);  
        } while (x<5);  
        System.out.println("Termina do while externo");  
    }  
}
```

Continue

La sentencia continue hace lo opuesto a la sentencia break, es decir, en vez de forzar la finalización de un ciclo, continue fuerza la siguiente iteración y salta cualquier código después de éste.

Ejemplo 4.6.5

```
class Continue {  
    public static void main (String [] a){  
        int x = 0, y = 0;  
        do {  
            System.out.println("Inicia do-while interno, x = " + x);  
            do {  
                x++;  
                System.out.println("Inicia if");  
                if (x>2) {  
                    continue;  
                }  
                System.out.println("Termina if, x = " + x);  
            } while (x<3);  
            y++;  
            System.out.println("Termina do-while interno, x = " + x + ", y = "+ y);  
        } while (y<5);  
        System.out.println("Termina do-while externo");  
    }  
}
```

Ejemplo 4.6.6

```
class ContinueEtiqueta {  
    public static void main (String [] a){  
        int x = 0;  
        fuera:  
        do {  
            do {  
                x++;  
                System.out.println("Inicia if");  
                if (x>2) {  
                    continue fuera;  
                }  
                System.out.println("Termina if, x = " + x);  
            } while (x<3);  
            x++;  
            System.out.println("Termina do-while interno, x = " + x);  
        } while (x<5);  
        System.out.println("Termina do while externo");  
    }  
}
```

Try-catch (excepciones)

Las excepciones son el mecanismo mediante el cual se pueden controlar los errores producidos en tiempo de ejecución.

Estos errores pueden ser generados por la lógica del programa, como un índice de un arreglo fuera de su rango, una división entre cero, etc., o errores generados por los propios objetos que denuncian algún tipo de estado no previsto o condición que no pueden manejar.

Las excepciones son objetos que contienen información del error que se ha producido. Heredan de la clase `Throwable` o de la clase `Exception`.

Si no se captura la excepción interviene un manejador por defecto que normalmente imprime información que ayuda a encontrar como se produjo el error.

Para capturar/manejar errores en tiempo de ejecución se cuenta con la estructura de control try-catch.

La estructura try-catch está compuesta por dos partes principales, el bloque try y el bloque catch.

En el bloque try se coloca el código que se quiere ejecutar y que podría contener un error (división entre cero, índice fuera del arreglo, comparación entre tipos diferentes, etc.).

El bloque catch permite capturar excepciones, es decir, permite manejar los errores que genere el código del bloque try en tiempo de ejecución impidiendo así que el programa deje de ejecutarse y posibilitando al desarrollador enviar el error generado.

Se pueden tener más de un bloque catch, pero siempre debe haber uno, por lo menos.

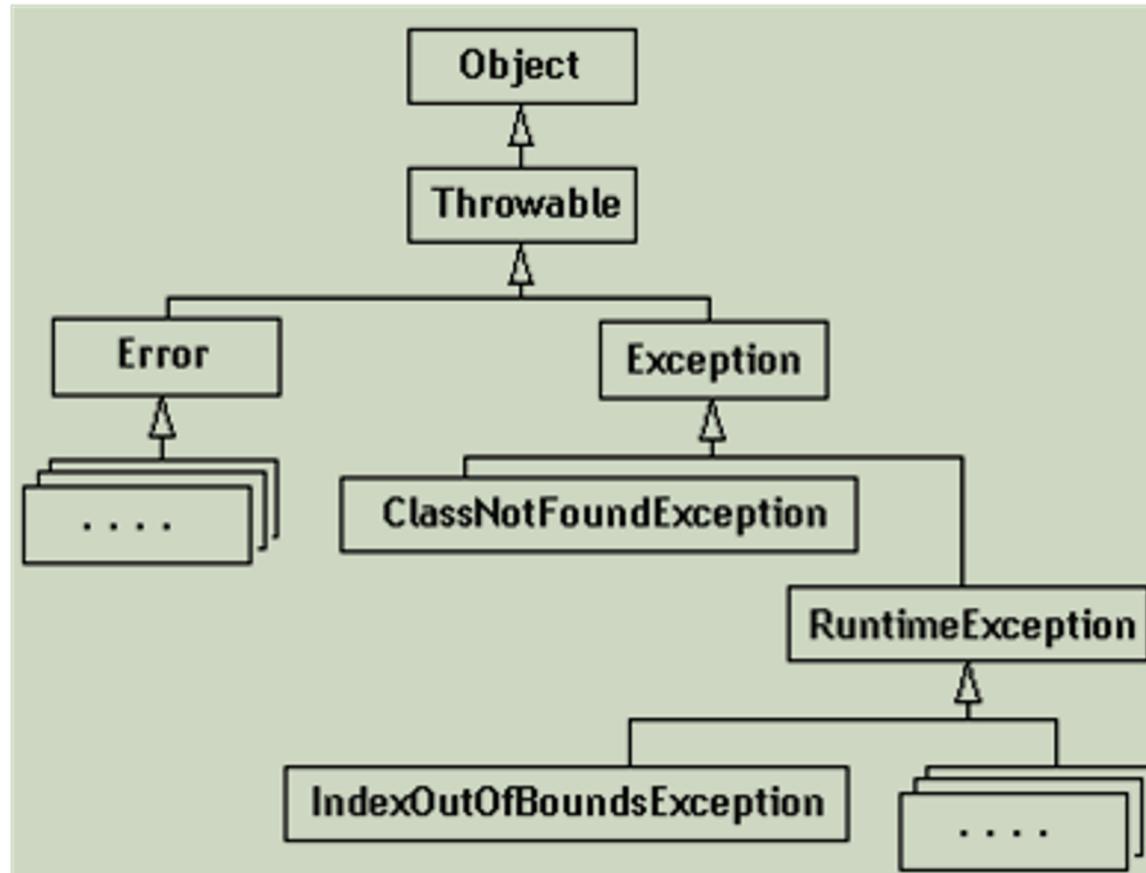
```
catch (ClassNotFoundException e) {...}  
catch (IOException e) {...}  
catch (Exception e) {...}
```

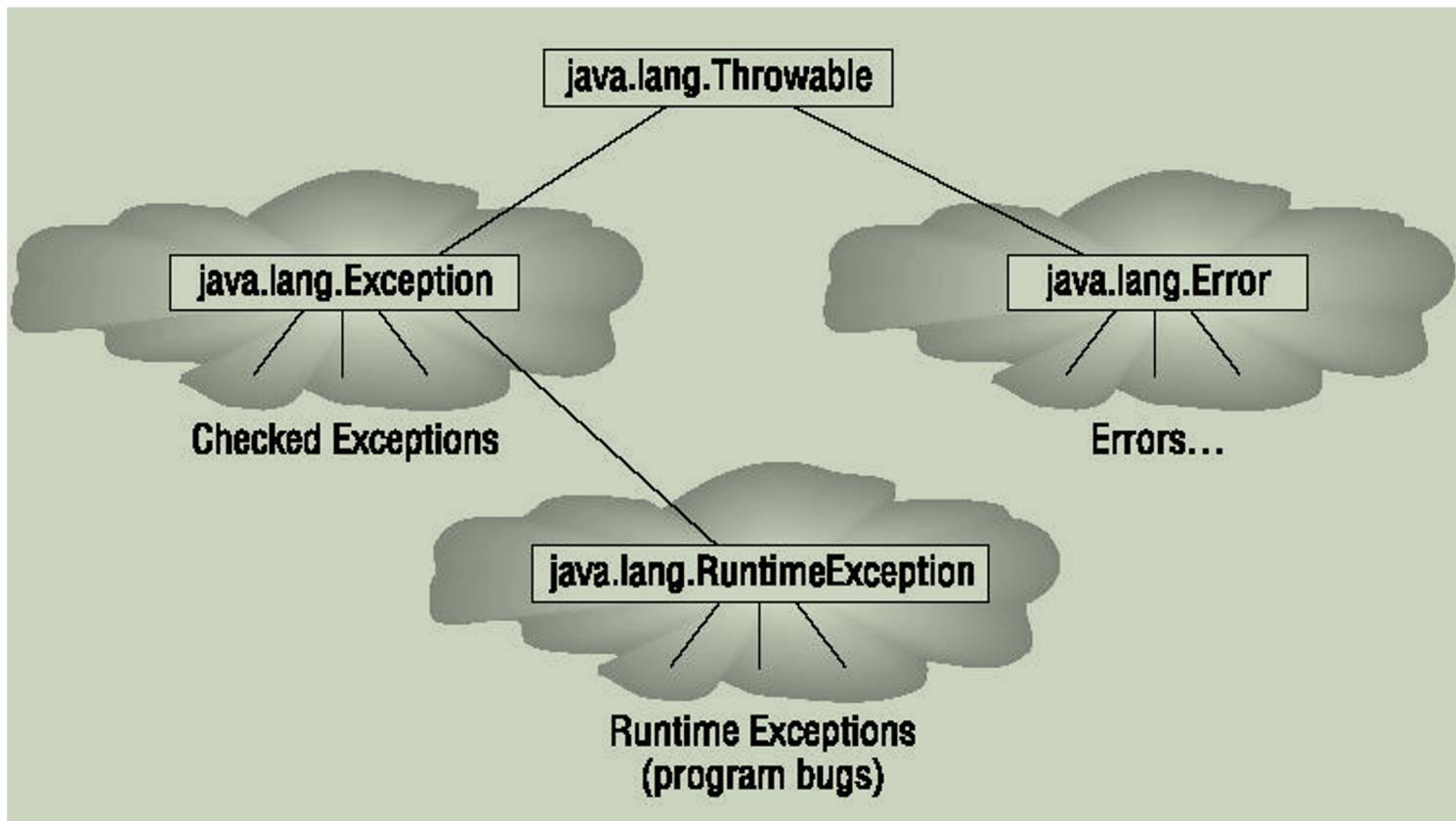
Existe un tercer bloque de código, opcional, llamado **finally**. Este bloque le indica al programa las instrucciones a ejecutar, de manera independiente de los bloques try-catch, es decir, si el código del bloque try se ejecuta de manera correcta, entra al bloque finally, si se genera un error, después de ejecutar el código del bloque catch ejecuta el código del bloque finally.

La sintaxis de esta estructura de control es:

```
try {  
    // bloque de código a ejecutarse en el flujo del programa  
}  
catch (Excepcion e) {  
    // bloque de código a ejecutarse si ocurre un error  
}  
[ finally {  
    // bloque de código a ejecutarse al final  
}]
```

Como ya se mencionó, las excepciones son objetos y, por tanto, poseen su propia jerarquía.





Ejemplo 4.7

```
public class TryCatchFinally {  
  
    public static void main(String[] args) {  
  
        try {  
            String mensajes[] = {"Primero", "Segundo", "Tercero" };  
            for (int i=0; i<=3; i++)  
                System.out.println(mensajes[i]);  
  
        } catch ( ArrayIndexOutOfBoundsException e ) {  
            System.out.println("Error: apuntador fuera del rango del  
arreglo");  
  
        } finally {  
            System.out.println("A pesar de todo, se ejecuta el bloque  
finally");  
        }  
    }  
}
```

Elementos estáticos

Un elemento estático es una variable o método que no se asocia a un objeto (instancia) de una clase, sino que se asocia a la clase misma, es decir, no hay una copia del dato para cada objeto sino una sola copia que es compartida por todos los objetos de la clase.

La sintaxis para declarar una variable estática es la siguiente:

```
static TipoVariable nombreVariable;
```

Por ejemplo:

```
public class Punto {  
    int x , y ;  
    static int numPuntos = 0;  
  
    void iniciarVars ( int x , int y ) {  
        this.x = x ;  
        this.y = y;  
        numPuntos ++ ;  
    }  
}
```

El acceso a las variables estáticas desde fuera de la clase donde se definen se realiza a través del nombre de la clase y no del nombre del objeto como sucede con las variables no estáticas.

Por lo tanto, si se desea acceder al número de puntos que se han creado para el ejemplo anterior, la sentencia sería la siguiente:

```
int x = Punto.numPuntos;
```

No obstante, también es posible acceder a las variables estáticas a través de una referencia a un objeto de la clase:

```
Punto p = new Punto();
int x = p.numPuntos;
```

Sin embargo, las variables estáticas de una clase existen, se inicializan y pueden usarse antes de que se cree ningún objeto de la clase.

Los métodos estáticos, igual que las variables estáticas, se asocian a una clase, no a una instancia. Su sintaxis es la siguiente:

```
static valorRetorno nombreMetodo([Argumentos]) {  
    // bloque de código a ejecutarse  
}
```

Por ejemplo:

```
class Punto {  
    int x , y ;  
    static int numPuntos = 0;  
  
    static int cuantosPuntos() {  
        return numPuntos;  
    }  
}
```

El acceso a los métodos estáticos se realiza de manera análoga a los atributos estáticos, es decir, utilizando el nombre de la clase:

```
int totalPuntos = Punto.cuantosPuntos();
```

Dado que los métodos estáticos tienen sentido a nivel de clase y no a nivel de instancia, los métodos estáticos no pueden acceder a datos que no sean estáticos.

También se puede poseer un bloque estático. Este bloque sirve para inicializar las variables estáticas dentro de la clase.

```
public class Punto {  
    int x, y;  
    static int numPuntos = 0;  
    static int valor;  
    static {  
        if ((numPuntos%2)==0) {  
            valor = 56;  
        } else {  
            valor = -56;  
        }  
    }  
}
```

4.3 Tipos de clase

Java permite definir distintos tipos de clase dependiendo del comportamiento esperado u objetivo final.

Por su nivel de acceso las clases pueden ser públicas o no poseer ningún modificador (default).

Por su tipo, las clases se pueden clasificar en concretas y abstractas.

Por jerarquía de clases pueden ser abstractas (más general) o finales (más específico).

Una clase no puede ser declarada como privada o protegida o estática al menos que sea miembro de otra clase. Estas clases se clasifican como clases internas.

Dependiendo de las necesidades del desarrollador se puede declarar la clase con alguno de los tipos descritos.

Por lo tanto, es importante conocer las características de cada tipo de clase para saber cuando utilizarlos, con el fin de desarrollar software de calidad.

Clase pública

Una clase pública proporciona acceso a todas las clases de cualquier paquete, es decir, todas las clases en el Universo de clases tienen acceso a ella.

Para poder utilizar una clase de tipo publica sólo es necesario importar el paquete al que pertenecen. Una vez hecho lo anterior, cualquier otra clase que puede instanciarla.

La sintaxis de una clase pública es la siguiente:

```
public class MiClase {  
    // Comentarios  
    // Atributos de MiClase  
    // Métodos de MiClase  
}
```

Ejemplo 4.9

**Tipo de clase
publica**

```
public class Triangulo {  
    float ladoA, ladoB, ladoC, vertA, vertB, vertC;  
    float base, altura;  
    int alfa, beta, gama;  
  
    public float area(){  
        return (base * altura)/2;  
    }  
}
```

Clase sin modificador

Una clase sin modificador proporciona acceso a todas las clases que pertenecen a su mismo paquete, hereden o no de ella.

Por tanto, para poder utilizar una clase sin modificador de acceso sólo es necesario pertenecer al paquete de dicha clase.

La sintaxis de una clase sin modificador es la siguiente:

```
class MiClase {  
    // Comentarios  
    // Atributos de MiClase  
    // Métodos de MiClase  
}
```

Ejemplo 4.10

**Tipo de clase sin
modificador**

```
class Triangulo {  
    float ladoA, ladoB, ladoC, vertA, vertB, vertC;  
    float base, altura;  
    int alfa, beta, gama;  
  
    public float area(){  
        return (base * altura)/2;  
    }  
}
```

Clase abstracta

Una clase abstracta define la existencia de métodos pero no su implementación (bloque de código a ejecutar).

Las clases abstractas sirven como modelo base para la creación de clases derivadas.

Algunas características de las clases abstractas son:

- **Pueden contener métodos abstractos y métodos concretos.**
- **Pueden contener atributos.**
- **Pueden heredar de otras clases.**

La sintaxis de una clase abstracta es la siguiente:

```
abstract class Poligono {  
    abstract float perimetro();  
}
```

La clase que hereda de una clase abstracta debe implementar los métodos abstractos definidos en la clase base abstracta:

```
class Triangulo extends Poligono {  
    float perimetro() {  
        // bloque de código para obtener el perímetro  
    }  
}
```

La clase abstracta se declara simplemente con el modificador *abstract*. Los métodos abstractos se declaran de la misma manera (con el modificador *abstract*), pero no se implementan.

La clase derivada implementa a la clase abstracta y debe declarar e implementar el comportamiento de los métodos abstractos de la clase base.

De no declararse (implícitamente), el compilador genera un error indicando que no se han implementado todos los métodos abstractos y que, o bien, se implementan, o bien se declara la clase abstracta.

Es posible crear referencias a una clase abstracta:

Polígono figura;

Sin embargo, una clase abstracta no se puede instanciar, es decir, no se pueden crear objetos de una clase abstracta.

Polígono figura = new Polígono();

El que una clase abstracta no se pueda instanciar es coherente dado que este tipo de clases no tiene completa su implementación y encaja bien con la idea de que un ente abstracto no puede materializarse. Sin embargo, se puede realizar el llamado up-casting:

```
Poligono figura = new Triangulo();
figura.perimetro()
```

Tipo de clase
publica y abstracta

```
public abstract class Poligono {  
    abstract float area();  
    abstract float perimetro();  
}
```

Ejemplo 4.11

```
public class Triangulo extends Poligono {  
    float ladoA, ladoB, ladoC;  
    float puntoA, puntoB, puntoC;  
    float base, altura;  
    int alfa, beta, gama;  
    public float area(){  
        return (base * altura)/2;  
    }  
    public float perimetro(){  
        return 0;  
    }  
}
```

Clase final

Cuando una clase representa el final de la jerarquía de clases y, por ende, no pueda ser heredada o cuando se desea que el método de una clase no sea redefinido en una clase derivada, se utiliza la cláusula *final*, que tiene significados distintos según se aplique a un dato miembro, a un método o a una clase.

En una clase, final significa que la clase no puede heredarse y es, por tanto, el punto final de la cadena de clases derivadas. Por ejemplo si se quisiera impedir la extensión de la clase Ejecutivo, se pondría:

```
final class Ejecutivo {  
    // código de la clase  
}
```

En un método, la palabra reservada *final* impide que el método pueda redefinirse en una clase derivada. Por ejemplo, si se declara:

```
class Empleado {  
    // atributos, métodos y/o comentarios  
  
    public final void aumentarSueldo(int porcentaje) {  
        // Código del método  
    }  
}
```

entonces la clase Ejecutivo, clase derivada de Empleado, no podrá reescribir el método *aumentarSueldo* y, por tanto, cambiar su comportamiento.

En un atributo (dato miembro), la palabra reservada *final* impide que éste se pueda redefinir en una clase derivada, pero también significa que su valor no puede ser cambiado en ningún sitio. Por lo tanto, para los atributos, el modificador final se utiliza para definir valores constantes. Por ejemplo:

```
class Circulo {  
    ...  
    public final static float PI = 3.141592;  
    ...  
}
```

donde se define el valor de PI como de tipo flotante, estático constante (modificador final) y de acceso público.

Tipo de clase
publica y final

Ejemplo 4.12

```
final public class ClaseFinal {  
    void metodo () {  
        System.out.println("Método algo");  
    }  
}
```

```
public class Hereda extends ClaseFinal {  
    public void prueba (){  
        super.metodo();  
        System.out.println("Esta clase intenta heredar de  
una clase final");  
    }  
}
```

Ejemplo 4.12

Al intentar compilar la clase Hereda se obtiene la siguiente salida:

```
# javac Hereda.java
```

```
Hereda.java:1: cannot inherit from final ClaseFinal
```

```
public class Hereda extends ClaseFinal {
```

 ^

1 error

Con esto se comprueba que una clase declarada como *final* no puede ser heredada.

Clases internas

Una clase puede ser etiquetada como privada, protegida o estática únicamente si es un miembro de otra clase. A estas clases se les conoce como **clases internas**.

Una clase interna es un miembro más de la clase que la declara (clase externa) y, por tanto, puede tener el modificador de acceso privado.

Una clase externa solo se puede crear si posee un modificador de acceso público (public) o si no posee modificador (default).

Una clase interna (inner o nested class) es un clase definida dentro de otra clase es decir, una clase que es, a su vez, miembro de otra clase. La sintaxis de este tipo de clases es la siguiente:

```
class Externa {  
    // Comentarios  
    // Atributos  
    // Métodos  
    class Interna {  
        // Atributos  
        // Métodos  
    }  
}
```

Este estilo de programación se utiliza cuando las relaciones entre clases es muy estrecha.

En este tipo de clases, un objeto de la clase interna está siempre relacionado con un objeto de la clase externa, es decir, las instancias de la clase interna deben ser creadas a partir de instancias de la clase externa.

Una instancia de la clase interna tiene acceso a todos los datos miembro de la clase que lo contiene (clase externa) sin utilizar un calificador de acceso especial, es decir, como si le pertenecieran.

Las clases internas se clasifican en:

- **Clases anidadas de alto nivel (clases internas estáticas)**
- **Clases internas miembro**
- **Clases internas locales**
- **Clases internas anónimas**

Clase interna estática

Las clases internas estáticas, obviamente, debe declararse como clases estáticas.

Para acceder a las clases estáticas internas no es necesario crear una referencia (instancia) de la clase externa, sólo hay que especificar donde se encuentra la clase interna estática, es decir, especificar la clase que la contiene.

Desde la clase estática interna se puede acceder a los miembros estáticos de la clase externa. Para acceder a los miembros no estáticos de la clase externa es necesario crear una referencia dentro de la clase interna a la clase externa.

Ejemplo 4.13

```
public class ClaseExterna {  
  
    static int i = 5;  
  
    void metodoClaseExterna() {  
        System.out.println("Dentro de metodo de la clase externa");  
    }  
  
    // Inicia la clase interna estatica  
    public static class InternaEstatica {  
        public static void main(String args[]) {  
            System.out.println("Dentro de la clase interna " + i);  
            ClaseExterna cie = new ClaseExterna();  
            cie.metodoClaseExterna();  
        }  
    } // Fin de la clase interna estatica  
  
} // Fin de la clase Externa
```

Clase interna miembro

Estas clases se definen como miembro (no estático) de la clase contenedora. Se pueden declarar, incluso, como privadas o protegidas.

A cada instancia de la clase externa (contenedora) se asocia una instancia de la clase interna miembro. La clase interna miembro tiene acceso a todos los atributos y métodos de la clase externa, incluyendo los privados.

Una clase interna miembro no puede tener miembros estáticos. Tampoco puede tener nombres comunes con la clase externa.

La única forma para acceder a la clase interna es creando una referencia a la clase externa.

Ejemplo 4.14

```
public class ConjuntoObjetos {  
    Object arreglo[];  
    int cima = 0;  
  
    ConjuntoObjetos(int tam){  
        arreglo = new Object[tam];  
    }  
  
    public void setObject(Object ob){  
        arreglo[cima++] = ob;  
    }  
  
    public Object getObject(){  
        return arreglo[--cima];  
    }  
  
    public boolean vaciar(){  
        return cima == 0;  
    }  
}
```

Ejemplo 4.14

```
// clase interna miembro
class Enumerador implements java.util.Enumeration {
    int cont = cima;
    public boolean hasMoreElements(){
        return cont > 0;
    }
    public Object nextElement(){
        if (cont == 0) {
            throw new java.util.NoSuchElementException("Error");
        }
        return arreglo[--cont];
    }
} // Fin de la clase Enumerador

public java.util Enumeration elementos(){
    return new Enumerador();
}
```

Ejemplo 4.14

```
public class ConjuntoObjetosPrueba {  
    public static void main (String [] args){  
        ConjuntoObjetos pila = new ConjuntoObjetos(5);  
  
        for (int i = 0 ; i < pila.arreglo.length ; i++){  
            String txt = "Objeto " + i;  
            pila.setObject(txt);  
        }  
  
        java.util.Enumeration enumerador;  
        enumerador = pila.elementos();  
  
        while(enumerador.hasMoreElements()){  
            String txt = (String) enumerador.nextElement();  
            System.out.println(txt);  
        }  
    }  
}
```

Para las clases internas, la palabra reservada `this` hace referencia a la clase contenedora: `ClaseExterna.this`.

Para crear una instancia de una clase interna se utiliza la clase externa: `InstanciaClaseExterna.new ClaseInterna()`.

Si se hereda de una clase interna miembro y se quiere acceder a la clase base, es necesario hacer referencia a la instancia de la clase externa: `InstanciaClaseExterna.super()`.

Ejemplo 4.14(2)

```
public class ConjuntoObjetosPrueba2 {  
  
    public static void main (String [] args) {  
        ConjuntoObjetos pila = new ConjuntoObjetos(5);  
        for (int i = 0 ; i < pila.arreglo.length ; i++){  
            String txt = "Objeto " + i;  
            pila.setObject(txt);  
        }  
        ConjuntoObjetos.Enumerador enumerador = pila.new Enumerador();  
        while(enumerador.hasMoreElements()){  
            String txt = (String) enumerador.nextElement();  
            System.out.println(txt);  
        }  
    }  
}
```

Clase interna local

Estas clases también llamadas clases internas de métodos locales, se definen dentro del bloque de código de un método, por lo tanto, solo se pueden utilizar dentro del código donde están declaradas.

Pueden hacer uso de las variables locales y los parámetros del método declarados como final. No utilizan ningún modificador de acceso. No pueden ser estáticas.

Estas clases no están disponibles de manera pública, por lo tanto son inaccesibles.

```
public class ClaseInternaLocal {  
    Object arreglo[];  
    int cima = 0;  
  
    ClaseInternaLocal(int tam){  
        arreglo = new Object[tam];  
    }  
  
    public void setObject(Object ob){  
        arreglo[cima++] = ob;  
    }  
  
    public Object getObject(){  
        return arreglo[--cima];  
    }  
  
    public boolean vaciar(){  
        return cima == 0;  
    }  
}
```

Ejemplo 4.15

Ejemplo 4.15

```
// metodo con clase interna local
java.util.Enumeration enumeradorPropio(final Object objetos[]) {
    class ClaseLocal implements java.util.Enumeration {
        int cont = 0;
        public boolean hasMoreElements(){
            return cont < objetos.length;
        }

        public Object nextElement(){
            if (cont == objetos.length) {
                throw new java.util.NoSuchElementException("Error");
            }
            return objetos[cont++];
        }
    } // Fin de la clase Enumerador
    return new ClaseLocal();
} // Fin del metodo que implementa clase interna local
```

Ejemplo 4.15

```
public java.util.Enumeration elementos(){
    return enumeradorPropio(arreglo);
}
}
```

```
public class ClaseInternalLocalPrueba {
    public static void main (String [] args){
        // Implementar clase principal para comprobar
        // el funcionamiento de la ClaseInternalLocal
    }
}
```

Clase interna anónima

En esencia, las clases anónimas son clases internas locales sin nombre, de ahí su nombre.

Se define al mismo tiempo que se instancia y, por tanto, sólo puede existir una instancia de una clase anónima.

Los constructores de estas clases deben ser sencillos (sin argumentos), para evitar anidar demasiado código en una sola línea.

```
public class ClaseInternaAnonima {  
    Object arreglo[];  
    int cima = 0;  
  
    ClaseInternaAnonima(int tam){  
        arreglo = new Object[tam];  
    }  
  
    public void setObject(Object ob){  
        arreglo[cima++] = ob;  
    }  
  
    public Object getObject(){  
        return arreglo[--cima];  
    }  
  
    public boolean vaciar(){  
        return cima == 0;  
    }  
}
```

Ejemplo 4.16

Ejemplo 4.16

```
// metodo con clase interna anonima
public java.util.Enumeration elementos() {
    return new java.util.Enumeration(){
        int cont = cima;

        public boolean hasMoreElements(){
            return cont > 0;
        }

        public Object nextElement(){
            if (cont == 0) {
                throw new java.util.
                    NoSuchElementException("Error");
            }
            return arreglo[--cont];
        }
    }; // Fin de la clase (una sola linea de codigo)
} // Fin del metodo que implementa clase interna anonima
}
```

En resumen, existen diferentes tipos de clases:

Publicas o sin modificador: Son las más comunes, accesibles desde cualquier otra clase en el mismo paquete (de otro modo hay que importarlas).

Abstractas: Son clases bases que dan las bases para que rigen el comportamiento de las clases que las heredan.

Finales: Son las que terminan la cadena de herencia. Útiles por motivos de seguridad y eficiencia de un programa, ya que no permiten crear más sub-divisiones por debajo de ellas.

Internas: Estas clases sólo se pueden crear dentro de otra clase, es decir, una clase interna es una clase miembro de otra clase.

4.3.2 Métodos constructores

El constructor de una clase es un método estándar para inicializar los objetos de esa clase. Es una función que se ejecuta siempre al crear un objeto.

Si no se declara un constructor explícitamente, el sistema crea un constructor por defecto sin argumentos. Cuando se define un constructor, el constructor del sistema se elimina y es sustituido por éste o los constructores definidos (como el constructor es un método se puede sobrecargar).

Cuando se crea un objeto (se instancia una clase) es posible definir un proceso de inicialización que prepare el objeto para ser usado.

Esta inicialización se lleva a cabo invocando el método constructor.

La invocación al método constructor es implícita y se realiza automáticamente cuando se utiliza el operador new.

Las características principales de los métodos constructores son:

- El nombre del constructor tiene que ser igual al de la clase.
- Puede recibir cualquier número de argumentos de cualquier tipo.
- No devuelve ningún valor (en su declaración no se declara ni siquiera void).

El constructor no es un miembro más de una clase, por lo tanto, solo es invocado cuando se crea el objeto (con el operador new) y no puede ser invocado explícitamente en ningún otro momento.

Por ejemplo, la clase **ConstructorPunto** se puede definir como:

```
public class ConstructorPunto {  
    int x , y ;  
    ConstructorPunto(int a ,int b) { //Constructor  
        x = a ; y = b ;  
    }  
}
```

Así, al momento de instanciar la clase se pueden asignar los valores a sus atributos.

ConstructorPunto p = new ConstructorPunto(2, 3);

Si no se declara constructor alguno, java genera un constructor por defecto (llamado constructor no-args).

```
public class ConstructorPunto {  
    int x, y;  
  
    // Ejemplo constructor no-args  
    ConstructorPunto () { }  
}
```

Los constructores soportan polimorfismo y, por lo tanto, es posible sobre-cargarlos (definir varios constructores). Esto permite que un objeto pueda inicializarse de varias formas.

```
public class ConstructorPunto {  
    int x , y ;  
    ConstructorPunto ( int a , int b ) {  
        x = a ; y = b ;  
    }  
    ConstructorPunto () {  
        x = 0 ; y = 0;  
    }  
}
```

Desde un constructor puede invocarse explícitamente a otro constructor utilizando la palabra reservada `this`. Cuando `this` es seguido por paréntesis se entiende que se está invocando al constructor del objeto en cuestión.

```
public class ConstructorPunto {  
    int x , y ;  
    ConstructorPunto ( int a , int b ) {  
        x = a ; y = b ;  
    }  
  
    ConstructorPunto () {  
        this(5,10);  
    }  
}
```

Inicialización de clases derivadas

Cuando se crea un objeto de una clase derivada se crea, implícitamente, un objeto de la clase base que se inicializa con su constructor correspondiente.

Si en la creación del objeto se usa el constructor no-args, entonces se produce una llamada implícita al constructor no-args para la clase base.

Sin embargo, si se requiere utilizar constructores sobrecargados es necesario invocarlos explícitamente.

Ejemplo 4.17

```
class Empleado {  
    String nombre;  
    int numEmpleado , sueldo;  
  
    Empleado (String nombre, int sueldo) {  
        this.nombre = nombre;  
        this.sueldo = sueldo;  
    }  
  
    public void aumentarSueldo(int porcentaje) {  
        sueldo += (int)(sueldo * porcentaje / 100);  
    }  
  
    public String toString() {  
        return "Num. Empleado " + numEmpleado + " ,  
               nombre: " + nombre + " Sueldo: " + sueldo;  
    }  
}
```

Ejemplo 4.17

Se crea la clase Ejecutivo que hereda de Empleado

```
class Ejecutivo extends Empleado {  
    int presupuesto;  
  
    Ejecutivo (String n, int s) {  
        super(n,s);  
    }  
    void asignarPresupuesto(int p) {  
        presupuesto = p;  
    }  
    public String toString() {  
        String s = super.toString();  
        s = s + " Presupuesto: " + presupuesto;  
        return s;  
    }  
}
```

Ejemplo 4.17

```
public class PruebaEjecutivo {  
    public static void main (String [] args){  
        Ejecutivo jefe = new Ejecutivo();  
        if (jefe instanceof Ejecutivo) {  
            System.out.println("\njefe si es una instancia de  
                    Ejecutivo\n\t* aplausos *\n");  
            jefe.asignarPresupuesto(200);  
            jefe.aumentarSueldo(300);  
            System.out.println("InformaciOn del usuario:\n"  
                + jefe.toString());  
        }  
    }  
}
```

Ejemplo 4.17

Observar que el constructor de Ejecutivo invoca directamente al constructor de Empleado mediante la palabra reservada super:

```
Ejecutivo (String n, int s) {  
    super(n,s);  
}
```

En caso de resultar necesaria la invocación al constructor de la superclase, esta llamada debe ser la primera sentencia del constructor de la subclase.

```
class Super{  
    public int i = 0;  
  
    public Super(String text){  
        i = 1;  
    }  
}  
  
class Sub extends Super{  
    public Sub(String text){  
        i = 2;  
    }  
  
    public static void main(String args[]){  
        Sub sub = new Sub("Hello");  
        System.out.println(sub.i);  
    }  
}
```

Ejemplo 4.18

4.3.3 Interfaces

El concepto de Interfaz lleva un paso más adelante la idea de las clases abstractas. Una interfaz es una clase abstracta pura, es decir una clase donde todos los métodos son abstractos (no se implementa ninguno).

Este tipo de diseños permiten establecer la forma de una clase (nombres de métodos, listas de argumentos y tipos de retorno), pero no bloques de código.

Una interfaz puede contener atributos, pero estos son siempre públicos, estáticos y finales.

Para crear una interfaz, se utiliza la palabra reservada *interface* en lugar de *class*. La interfaz puede definirse pública o sin modificador de acceso, y tiene el mismo significado que para las clases.

Todos los métodos que declara una interfaz son siempre públicos y abstractos.

Para indicar que una clase implementa los métodos de una interfaz se utiliza la palabra reservada *implements*. El compilador se encargará de verificar que la clase que implementa una interfaz efectivamente declare e implemente todos los métodos de la interfaz. Una clase puede implementar más de una interfaz.

La sintaxis general para declarar una interfaz es la siguiente:

```
interface nombreInterfaz {  
    tipoRetorno nombreMetodo([Parametros]);  
}
```

Ejemplo 4.19

Por ejemplo, a continuación se declara la interfaz Instrumento Musical:

```
Interface InstMusical {  
    void tocar();  
    void afinar();  
    String tipolinstrumento();  
}
```

Ejemplo 4.19

Por otro lado, la clase que implementa la interfaz debe declarar los métodos de la interfaz y describir su funcionamiento.

```
public class InstViento extends Object implements InstMusical {  
    void tocar() { // código }  
    void afinar() { // código }  
    String tipoInstrumento() { // código }  
}  
  
class Flauta extends InstViento {  
    String tipoInstrumento() {  
        return "Flauta";  
    }  
}
```

Una clase derivada puede también redefinir, de ser necesario, alguno de los métodos de la interfaz.

Ejemplo 4.19

Se pueden crear referencias a las interfaces, pero no es posible instanciar una interfaz.

```
// La instanciación directa provoca un error  
InstMusical instrumento = new InstMusical();
```

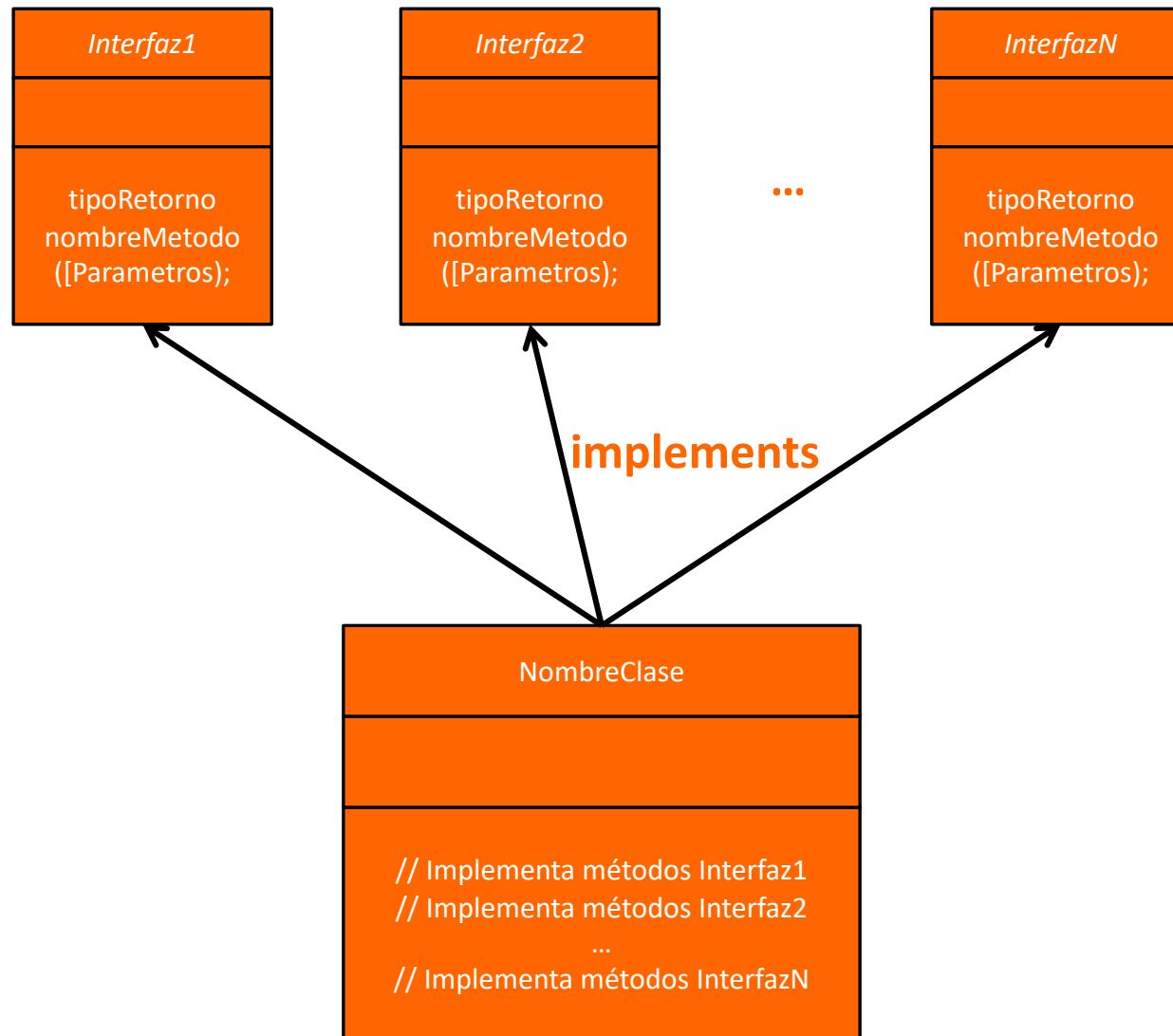
Sin embargo, una referencia a interfaz puede ser asignada cualquier objeto que la implemente

```
InstMusical instrumento = new Flauta();  
instrumento.tocar();  
System.out.println(instrumento.tipoInstrumento());
```

Una clase puede implementar cualquier cantidad de interfaces, la única restricción es que, dentro del cuerpo de la clase, se deben implementar todos y cada uno de los métodos de las interfaces que se implementen. La sintaxis es la siguiente:

```
class NombreClase implements Interfaz1, Interfaz2, ..., InterfazN {  
    // Métodos de la Interfaz1  
    // Métodos de la interfaz 2  
    // ...  
    // Métodos de la interfaz N  
}
```

El orden de la implementación de las interfaces y el orden en el que se implementan los métodos de las interfaces dentro de la clase no es importante.

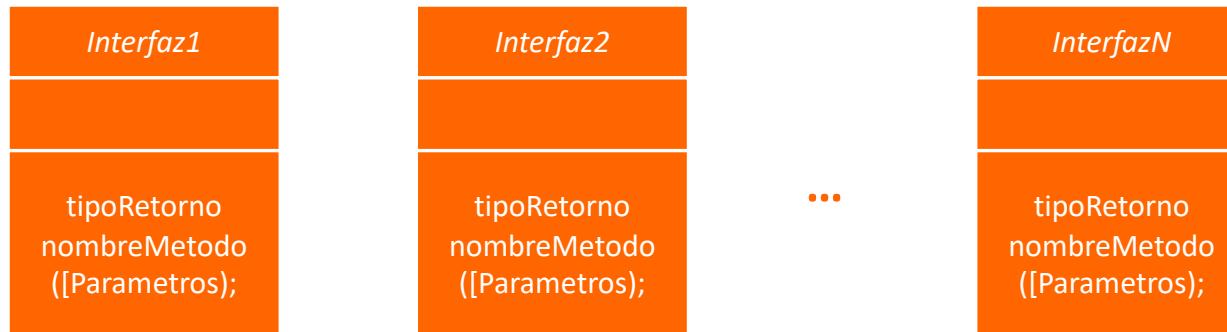


Las interfaces pueden heredar de otras interfaces y, a diferencia de las clases, una interfaz puede heredar de una o más interfaces (herencia múltiple), siguiendo la siguiente sintaxis:

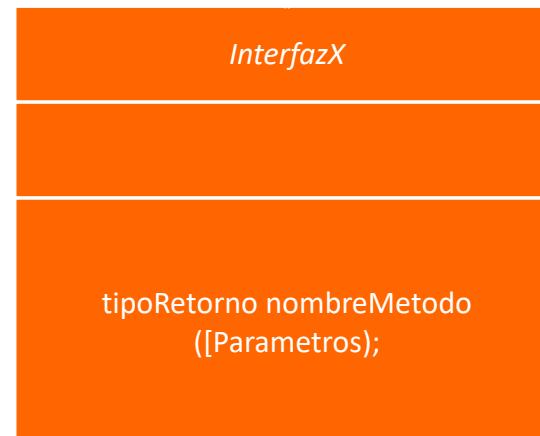
```
interface HeredalInterfaz extends Interfaz1, Interfaz2, ..., InterfazN {  
    tipoRetorno nombreMetodo([Parametros]);  
}
```

En este caso, se aplican las mismas reglas que en la herencia, es decir, la interfaz que hereda de otras interfaces posee todos los métodos definidos en ellas.

Empero, la clase que implemente esta interfaz (la interfaz que hereda de otras interfaces) debe definir los métodos de todas las interfaces.



extends



Dado que, por definición, todos los datos miembros (atributos) que se definen en una interfaz son estáticos y finales y dado que las interfaces no pueden instanciarse, resultan una buena herramienta para implantar grupos de constantes.

```
public interface Meses {  
    int UNO = 1, DOS = 2, TRES = 3, CUATRO = 4, CINCO = 5, SEIS = 6;  
    int SIETE = 7, OCHO = 8, NUEVE = 9, DIEZ = 10, ONCE = 11, DOCE =  
    12;  
  
    String [] NOMBRES_MESES = {"", "Enero", "Febrero", "Marzo",  
    "Abril", "Mayo", "Junio", "Agosto", "Septiembre", "Octubre",  
    "Noviembre", "Diciembre"};  
}
```

Se puede acceder a las variables de la interfaz por medio de la clase:

```
System.out.println (Meses.NOMBRES_MESES[Meses.DOS]);
```

PROGRAMACIÓN ORIENTADA A OBJETOS

ENCAPSULAMIENTO

HERENCIA

POLIMORFISMO

ABSTRACCIÓN

Paquetes

Un paquete es una agrupación de clases afines, equivalente al concepto de librería existente en otros lenguajes.

Una clase puede agruparse en un paquete y puede usar otras clases definidas en ese o en otros paquetes.

Los paquetes delimitan el espacio de nombres (space name) de las clases. El nombre de una clase debe ser único dentro del paquete donde se define.

Dos clases con el mismo nombre en dos paquetes distintos pueden coexistir e incluso pueden ser usadas en el mismo programa.

Para definir que una clase pertenece a un paquete en específico se utiliza la palabra reservada `package`, esta es su sintaxis:

```
package nombrepaquete;
```

La cláusula `package` debe ser la primera sentencia del archivo fuente. Cualquier clase declarada dentro de ese archivo pertenece al paquete indicado. Por ejemplo:

```
package mipaquete;
```

```
class MiClase {  
    // Código de MiClase  
}
```

declara que `MiClase` pertenece al paquete `mipaquete`.

La cláusula *package* es opcional, si no se utiliza las clases declaradas en el archivo fuente simplemente no pertenecen a ningún paquete en concreto (pertenecen a un paquete por defecto sin nombre).

Los paquetes se utilizan para organizar las clases y, por ende, mantener en una ubicación común las clases relacionadas. También resultan importantes para el alcance de los modificadores de acceso vistos anteriormente.

Por ejemplo, si se define la siguiente clase:

```
package geometria;  
  
public class Circulo {  
    Punto centro;  
}
```

El compilador y la máquina virtual asumen que la clase Punto pertenece al paquete geometría y, por tanto, para que la clase Punto sea accesible (al compilador) es necesario que esté definida en el mismo paquete.

Si la clase Punto no pertenece al mismo paquete, por ejemplo:

```
package geometriabase;  
  
public class Punto {  
    int x, y;  
}
```

es necesario hacer accesible el espacio de nombres donde se define la clase Punto en la clase Circulo. Esto se realiza mediante el uso de la clausula *import*.

```
package geometria;  
import geometriabase.*;  
  
public class Circulo {  
    Punto centro;  
}
```

Con la sentencia *import geomtriabase.*;* se hacen accesibles todas las clases que pertenecen al paquete geometriabase. Si únicamente se quisiera importar la clase punto, se podría utilizar la sentencia:

```
import geometriabase.Punto;
```

También es posible hacer accesibles las clases de un paquete sin utilizar la cláusula *import* y en tiempo de ejecución de la siguiente manera:

```
package geometria;  
  
public class Circulo {  
    geometriabase.Punto centro;  
}
```

Sin embargo, si no se utiliza la clase *import* el paquete, cada vez que se genere una instancia de ese paquete hay que especificar el nombre del mismo.

Los paquetes que utilizan nombres compuestos se separan por puntos, de manera similar a como se componen las direcciones URL, por ejemplo:

```
package geometria.base;
```

El API de java está estructurado de esta forma, con un primer calificador (java o javax) que indica la base, un segundo calificador (awt, util, swing, etc.) que indica el grupo funcional de las clases y subpaquetes en un tercer nivel, dependiendo de la amplitud del grupo.

Los paquetes también tienen un significado físico ya que sirven para almacenar los módulos ejecutables (archivos con extensión .class) en el sistema de archivos de la máquina.

Ejemplo 4.20

```
package figs;  
  
public abstract class Poligono {  
    public abstract float area();  
}
```

```
package figs;
```

```
public class Triangulo extends Poligono {  
    public float base, altura;  
    public float area(){  
        return (base*altura)/2;  
    }  
}
```

```
javac -d . *.java
```

Ejemplo 4.20

```
import figs.Triangulo;

public class FigurasGeometricas {

    public static void main (String [] args) {
        Triangulo t = new Triangulo();
        t.base = 10F;
        t.altura = 5F;
        System.out.println("El Area del triAngulo es: " +
t.area());
    }
}
```

Excepciones

Es posible hacer que un método lance una excepción propia si falla una instrucción en su bloque de código. Para ello es necesario habilitar al método para arrojar la excepción así como declarar el tipo de excepción que va a arrojar. La sintaxis es la siguiente:

```
valorRetorno nombreMetodo() throws NombreExpcion {  
    // Bloque de código del método  
}
```

La excepción se controla con la palabra reservada **throw**, instanciando la excepción correspondiente

```
throw new NombreExpcion();
```

Ejemplo 4.21

```
public class NuevaExcepcion extends Exception {  
    NuevaExcepcion (String msg) {  
        super (msg);  
    }  
}
```

```
public class DisparaExcepcion {  
    void dispara() throws NuevaExcepcion {  
        throw new NuevaExcepcion("Descripción del error");  
    }  
}
```

Ejemplo 4.21

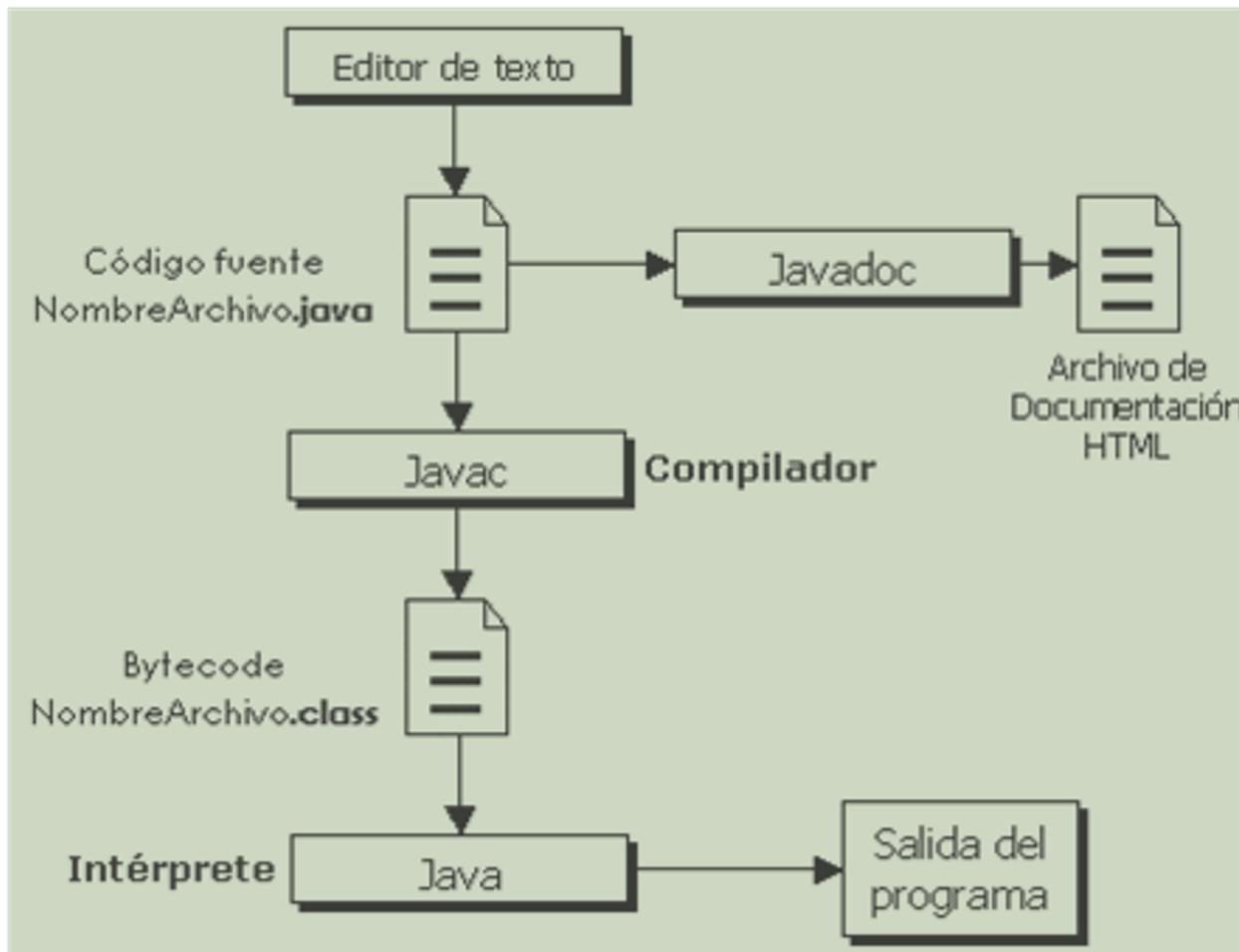
```
public class PruebaExpcion {  
  
    public static void main(String[] args) {  
        DisparaExpcion de = new DisparaExpcion();  
        try {  
            de.disparar();  
        } catch (NuevaExpcion e) {  
            System.out.println(e.getMessage());  
            System.out.println("-----");  
            e.printStackTrace();  
        }  
    }  
}
```

Documentación

JDK proporciona una herramienta para generar la documentación de las clases creadas a partir de los comentarios incluidos en el código fuente.

Los comentarios de documentación deben empezar con `/**` y terminar con `*/`. Se pueden documentar clases, atributos (datos miembro) y métodos.

La documentación se genera en código HTML utilizando el comando javadoc.



Existen etiquetas especiales para documentar aspectos concretos, como los parámetros o los valores devueltos.

Etiqueta	Formato	Descripción
Todos	@see	Permite crear una referencia a la documentación de otra clase o método.
Clases	@version	Comentario con datos indicativos del número de versión.
Clases	@author	Nombre del autor.
Clases	@since	Fecha desde la que está presente la clase.
Métodos	@param	Parámetros que recibe el método.
Métodos	@return	Significado del dato devuelto por el método
Métodos	@throws	Comentario sobre las excepciones que lanza.
Métodos	@deprecated	Indicación de que el método es obsoleto.

Comentarios en una clase

Ejemplo 4.22

```
import java.util.*;  
  
/**  
 * Clase que calcula el perímetro de un triángulo isósceles  
 * @author Jorge A. Solano  
 * @author jorge.a.solano@hotmail.com  
 * @version 1  
 */  
class Triangulososceles extends Triangulo {  
  
    /** Método que calcula el perímetro del triángulo isósceles  
     * @param Ninguno  
     * @return Valor del perímetro (tipo flotante)  
     * @exception No se genera error alguno  
     */  
    float perimetro () {  
        return 2 * ladoA + ladoB;  
    }  
}
```

Ejemplo 4.22

La documentación se puede guardar en cualquier ruta. Para especificar la ruta de destino se utiliza la bandera `-d`.

```
javadoc -d ruta -version -author -use  
NombreClase.java
```

Es posible generar la documentación de todas las clases (`*.java`) que se encuentran en la ruta actual. Se hace explícita la ruta donde se desea generar y se especifica los parámetros deseados (`-version`, `-author`, `-use`). La documentación se genera en HTML.

4.4 Resolución de problemas matemáticos, físicos y químicos sencillos

Elaborar diversos programas en lenguaje orientado a objetos:

- Resolución de derivadas e integrales por los métodos de Taylor y Simpson.
- Resolver ecuaciones diferenciales por los métodos de Euler y Runge-Kutta.

Tema 4: Conocer y aplicar las técnicas y herramientas de la programación orientada a objetos para la solución de problemas.

- **Subtema 4.1: Describir cómo es la jerarquía de clases.**
- **Subtema 4.2: Describir la sintaxis de las instrucciones de control de flujo en un lenguaje de programación orientado a objetos.**
- **Subtema 4.3: Explicar los diferentes tipos de clases existentes.**
- **Subtema 4.4: Elaborar diversos programas para la resolución de problemas científicos.**

Análisis

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Data.SqlClient;

namespace SP {
    public class Logica {
        SqlConnection con = new SqlConnection();
        SqlCommand comando = new SqlCommand();
        string servidor, tabla, seguridad, usuario, pwd;
        public string error;

        public Logica() {
            iniciaCadCon();
        }
    }
}
```

```
public void iniciaCadCon() {  
    servidor = "omega";  
    tabla = "AdventureWorks";  
    seguridad = "True";  
    usuario = "labmicrosoft";  
    pwd = "q1w2e3r4t5y6,k.l";  
}  
  
public void conexion() {  
    con.ConnectionString = "Data Source=" + servidor +  
    "; Initial Catalog=" + tabla + "; Persist Security Info=" +  
    seguridad + "; User ID=" + usuario + "; Password=" +  
    pwd;  
    con.Open();  
}
```

```
public DataSet buscarId(int id) {  
    try {  
        conexion();  
        comando.CommandType = CommandType.StoredProcedure;  
        comando.Connection = con;  
        comando.CommandText = "HumanResources.sp_getEmployeed";  
        comando.CommandTimeout = 10;  
        comando.Parameters.Clear();  
        comando.Parameters.AddWithValue("@EmployeedID", id);  
        SqlDataAdapter adapter = new SqlDataAdapter(comando);  
        DataSet ds = new DataSet();  
        adapter.Fill(ds);  
        con.Close();  
        return ds;  
    } catch (Exception e){  
        return null;  
    } finally {  
        if (con != null)  
            con.Close();  
    }  
}
```

```
public DataSet mostrarTodo() {
    try {
        conexion();
        comando.CommandType = CommandType.StoredProcedure;
        comando.Connection = con;
        comando.CommandText = "seleccionaTodo";
        comando.CommandTimeout = 10;
        comando.Parameters.Clear();
        SqlDataAdapter adapter = new SqlDataAdapter(comando);
        DataSet ds = new DataSet();
        adapter.Fill(ds);
        con.Close();
        return ds;
    } catch (Exception){
        return null;
    } finally {
        if (con != null)
            con.Close();
    }
}
```