



Universidad Nacional Autónoma de México
Facultad de Ingeniería
Programación orientada a objetos
Tema 2:
TIPOS, EXPRESIONES Y CONTROL DE FLUJO

2 Tipos, expresiones y control de flujo

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

2 Tipos, expresiones y control de flujo

2.1 Generalidades.

2.2 Tipos de datos.

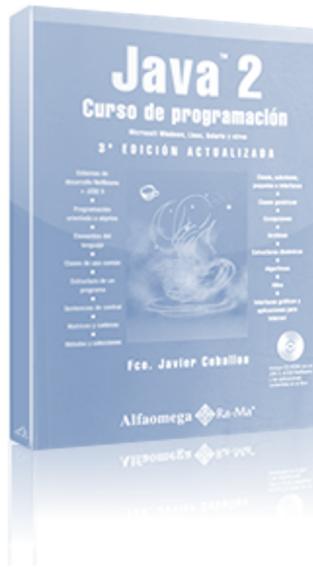
2.4 Tipos y ámbito de las variables.

2.5 Tipos de clases (públicas, sin modificador, abstractas, finales e internas).

2.6 Estructuras de selección.

2.7 Estructuras de repetición.

Bibliografía



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

Asking what is the best programming language is like asking what is the best chess move. Neither question has an answer without some context.

@CompSciFact

¿Qué es lenguaje de programación aprender?

Elegir un lenguaje para aprender a programar bajo el paradigma orientado a objetos es un tema de discusión entre profesionales, ya que no existe un criterio unánime respecto a qué lenguaje es el ideal para aprender todos los conceptos necesarios.

En este curso utilizaremos como caso de estudio el lenguaje de programación JAVA.

¿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.

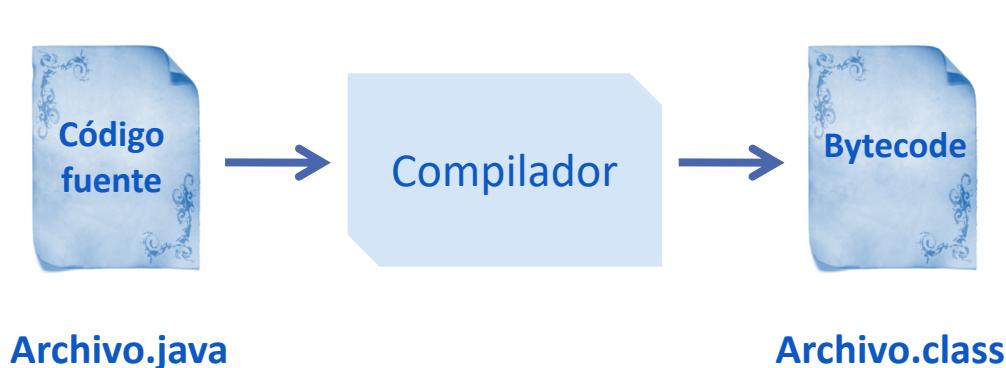
Java se ejecuta millones de computadoras personales de todo el mundo y en miles de millones de dispositivos, como dispositivos móviles y aparatos de televisión.



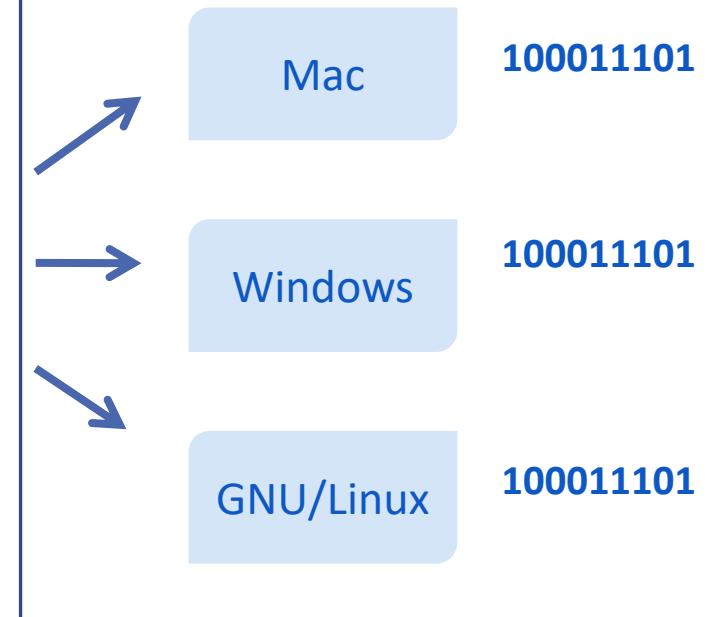
Máquina virtual

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.

Máquina virtual



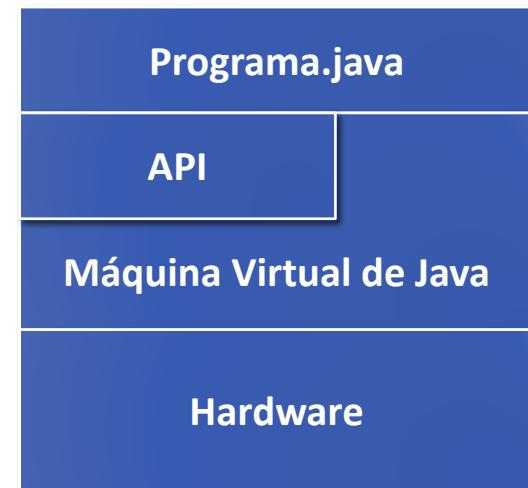
Intérprete o JVM



Máquina virtual

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

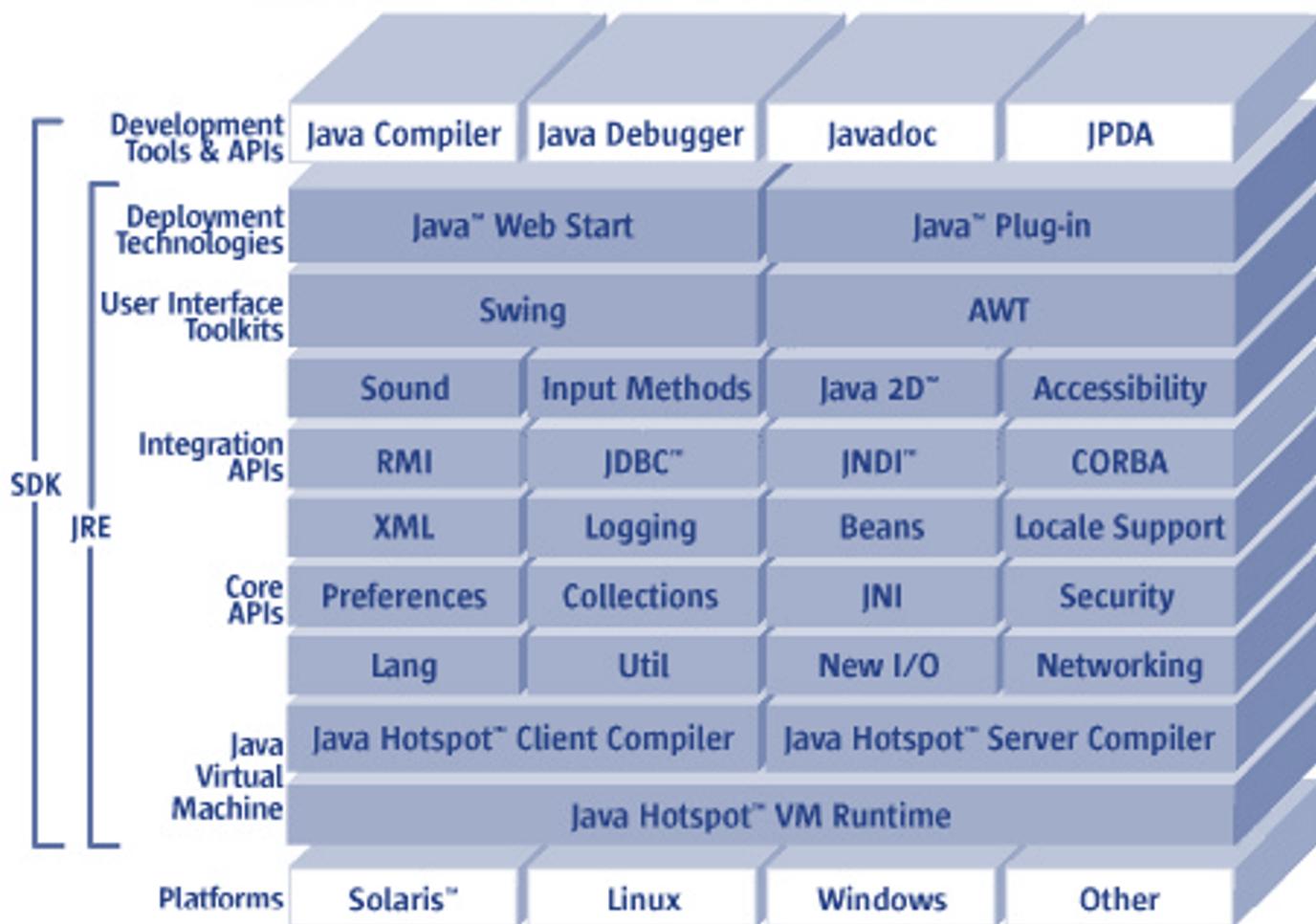
- Java 2 Enterprise Edition (J2EE)
- Java 2 Standard Edition (J2SE)
- Java 2 Micro Edition (J2ME)

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

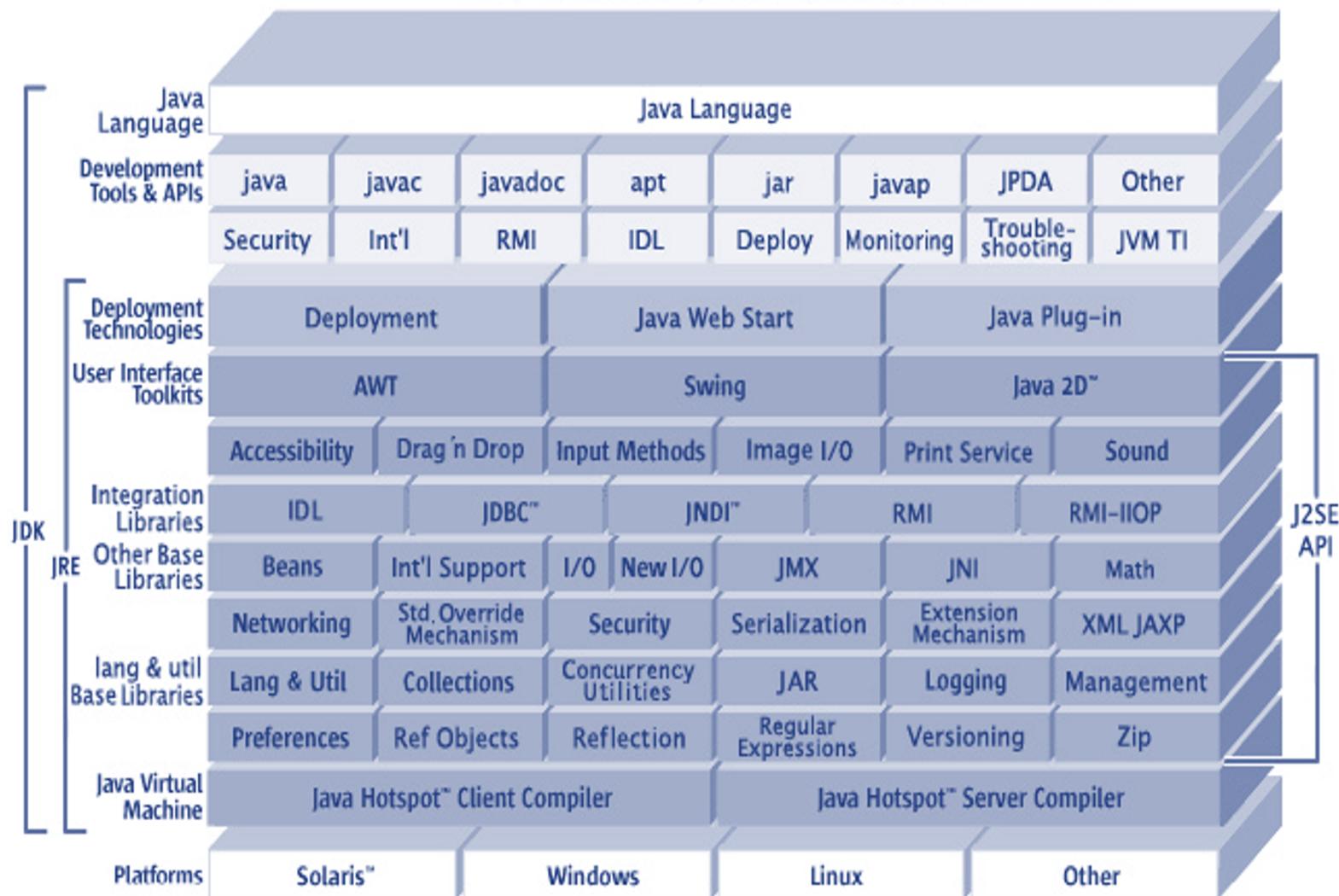
Para instalar Java 2 Standard Edition se debe descargar el JDK (Java Development Kit) que a su vez incluye el JRE.

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

Java™ 2 Platform, Standard Edition v 1.4



Java™ 2 Platform Standard Edition 5.0



Java Platform Standard Edition v6

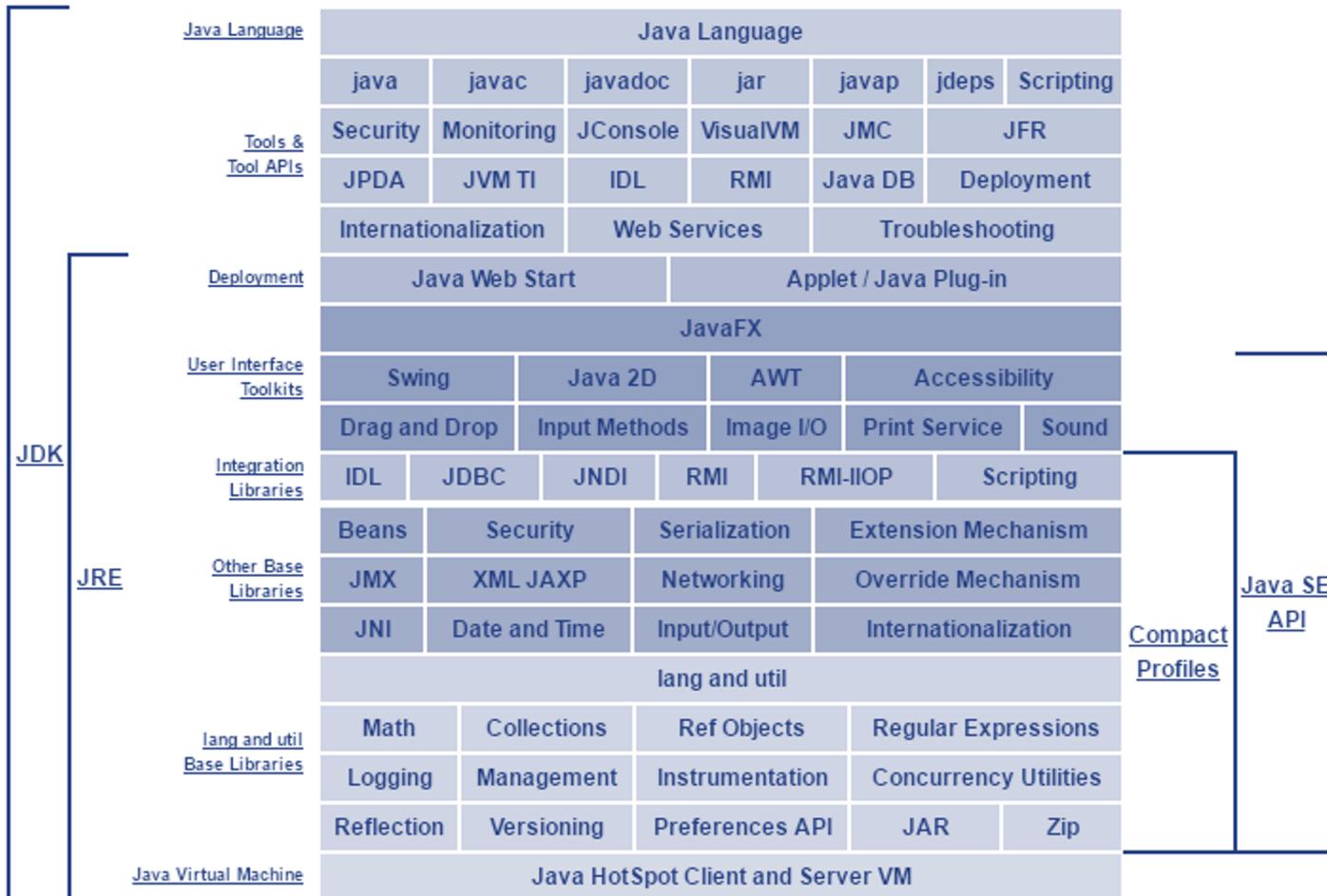
		Java Language								
		java	javac	javadoc	apt	jar	javap	JPDA	jconsole	
JDK	Tools & Tool APIs	Security	Int'l	RMI	IDL	Deploy	Monitoring	Troubleshoot	Scripting	JVM TI
JRE	Deployment Technologies	Deployment			Java Web Start			Java Plug-in		
	User Interface Toolkits	AWT			Swing			Java 2D		
	Integration Libraries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

Java SE API

Java Platform Standard Edition v7

		Java Language					
		java	javac	javadoc	jar	javap	JPDA
JDK	Tools & Tool APIs	JConsole	Java VisualVM	Java DB	Security	Int'l	RMI
		IDL	Deploy	Monitoring	Troubleshoot	Scripting	JVM TI
						Web Services	
JRE	Deployment	Java Web Start			Applet / Java Plug-in		
		JavaFX					
		Swing		Java 2D		AWT	Accessibility
Java SE API	User Interface Toolkits	Drag and Drop		Input Methods		Image I/O	Print Service
		IDL	JDBC	JNDI	RMI	RMI-IIOP	
		Beans	Int'l Support		Input/Output		JMX
JRE	Integration Libraries	JNI	Math		Networking		Override Mechanism
		Security	Serialization		Extension Mechanism		XML JAXP
		lang and util	Collections		Concurrency Utilities		JAR
Java Virtual Machine	lang and util Base Libraries	Logging	Management		Preferences API		Ref Objects
		Reflection	Regular Expressions		Versioning		Zip
		Java HotSpot Client and Server VM					

Java Platform Standard Edition v8





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 necesario 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 esta instalado JDK.
- **CLASSPATH:** Ruta que contiene la ubicación de las bibliotecas de JDK.
- **PATH:** Ruta que contiene la ubicación de los binarios de JDK.

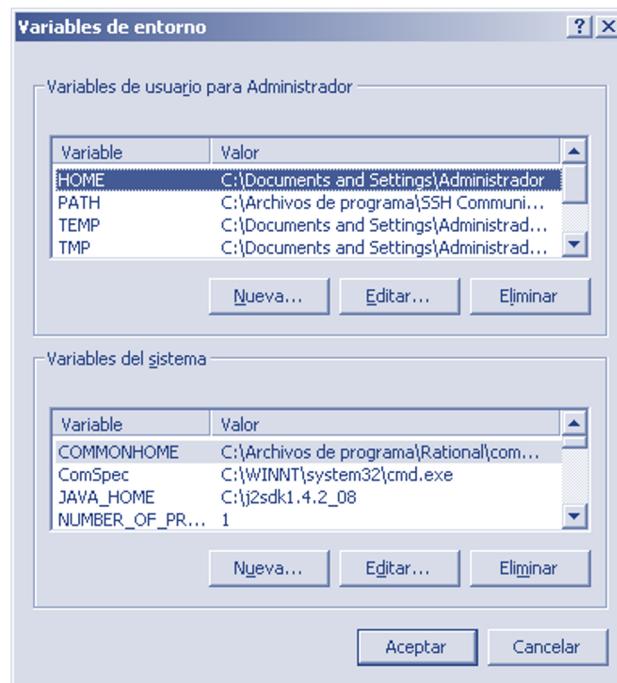
Una vez instalado Java Development Kit se realiza lo siguiente:

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

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.

Variables de entorno en Windows

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 las variables CLASSPATH y JAVAPATH.



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**

Tipos, expresiones y control de flujo

TEMA 2

2.1 Generalidades.

Para empezar a programar en un lenguaje es importante conocer los identificadores válidos para nombrar atributos y métodos, así como las palabras reservadas (ya que estas palabras no pueden ser utilizadas como nombres).

Todo código debe incluir comentarios (tanto por buena práctica como por buen karma).

Un programa orientado a objetos trabaja, normalmente, con objetos. Existen objetos de diferentes tipos, los cuales están modelados en un elemento base llamado clase.

Por lo tanto, es necesario definir el concepto de clase y el concepto de objeto a nivel de programación, es decir, saber cómo se declara una clase y cómo se crean objetos.

2.1.1 Identificadores

Un identificador es el nombre con el que se van a llamar a las entidades (objetos, atributos y métodos) del programa.

En java los nombres siguen las siguientes reglas:

- Comienzan con una letra (A-Z, a-z), el carácter guion bajo (_) 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.

Además, Java es case sensitive, es decir, sensible a mayúsculas y minúsculas.

En la práctica, existen dos convenciones muy útiles y utilizadas para nombrar elementos: la notación de camello y la notación húngara.

La notación de camello es un tipo de declaración que se caracteriza porque cada palabra que compone al nombre inicia con mayúscula. Dentro de esta notación existen dos tipos: lower camell case y upper camell case.

La notación lower camell case se caracteriza porque la primera letra del nombre inicia con minúscula y las siguientes palabras inician con mayúscula: area, identificadorProducto, totalCuenta.

La notación upper camell case se caracteriza porque la primera letra de todas las palabras que componen al nombre inician con mayúscula: Valor, EstadoCuenta, CalcularArea.

La notación húngara se caracteriza porque a cada nombre de variables se le antepone el tipo de dato al que pertenece, permitiendo identificar de manera rápida los valores que pueden adquirir estos atributos: strNombre, entTotal, RealSumaPotencias.

Los lenguajes de programación suelen usar convenciones para nombrar a los elementos (las cuales se deben respetar), sin embargo, el uso de notación húngara es de gran ayuda sobre todo para depurar código extenso.

Java utiliza, por convención, la notación de camelCase, tanto lower como upper camelCase, así como letras mayúsculas de la siguiente manera:

Tipo de identificador	Convención	Ejemplo
Clase	Upper camelCase	Rectangulo, Automovil, CuentaBancaria
Método	Lower camelCase	calcularArea, getValor, setValor, area
Atributo / variable	Lower camelCase	area, perimetro, objPoligono, triangulo1,
Constantes	Letras mayúsculas	PI, MAX_TAM, RADIO

2.1.2 Palabras reservadas

Dentro de un programa no se pueden definir palabras reservadas del lenguaje como nombres de variables, por lo tanto, es necesario conocer las palabras reservadas que maneja el lenguaje.

Es posible que palabras reservadas sean comunes en diferentes lenguajes de programación (como los tipos primitivos: int, float, etc.), sin embargo no siempre es así y no todos los lenguajes comparten estas sintaxis (en Python no es necesario declarar el tipo de dato; c# tiene 3 modificadores de acceso más que Python).

Las palabras reservadas que posee java son:

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

Además, null, true y false son literales reservadas.

2.1.3 Comentarios

Los comentarios dentro de un código son importantes, necesarios y trascendentales.

Un código que no posee comentarios puede complicar la depuración, corrección y mejora de un proceso, generando retrasos en la entrega.

Es recomendable realizar comentarios al iniciar la clase, para saber el objetivo general de la misma, además del creador y la fecha de creación. También se deben comentar todos los métodos de la clase, describiendo el proceso que realiza, los valores que recibe y lo que regresa al final.

Java maneja dos tipos de comentarios: comentarios simples y comentarios por bloque. Dentro de los comentarios por bloque se puede diferenciar el comentario multilínea y el comentario de documentación.

// comentario simple de una sola línea

**/* comentario multilínea que inicia
con diagonal asterisco
y termina con asterisco diagonal */**

/ comentario de documentación que inicia
con diagonal asterisco-asterisco
y termina con asterisco diagonal */**

2.1.4 Descripción de una clase.

Las clases son un tipo de dato abstracto (TDA), permiten modelar cualquier ente de la vida real tanto físico (una persona, un vehículo, un cheque, etc.) como abstracto (cuenta bancaria, inscripción a un club, cuenta de correo, etc.). Por tanto, las clases pueden contener:

- Atributos (se denominan Datos Miembro). Estos pueden ser de tipo primitivo o referencia (a otra(s) clase(s)).
- Métodos (se denominan Métodos Miembro).

En java la sintaxis general para declarar una clase es la siguiente:

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

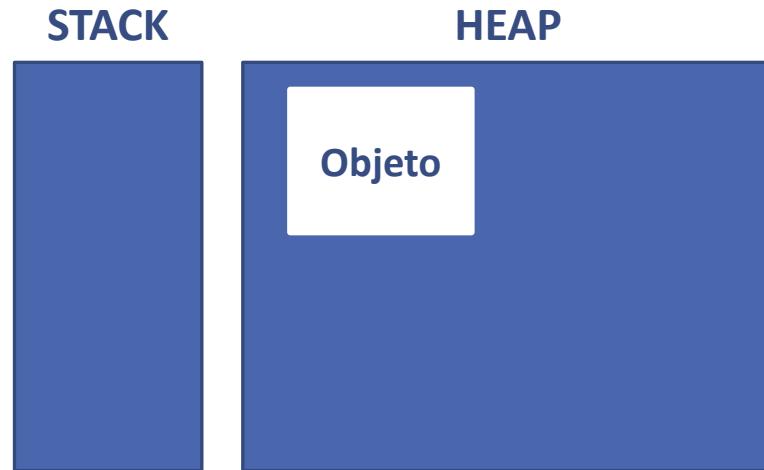
Por ejemplo:

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

NOTA: En estas notas los elementos que se encuentran entre corchetes indican que no son obligatorios.

2.1.5 Descripción de un objeto

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



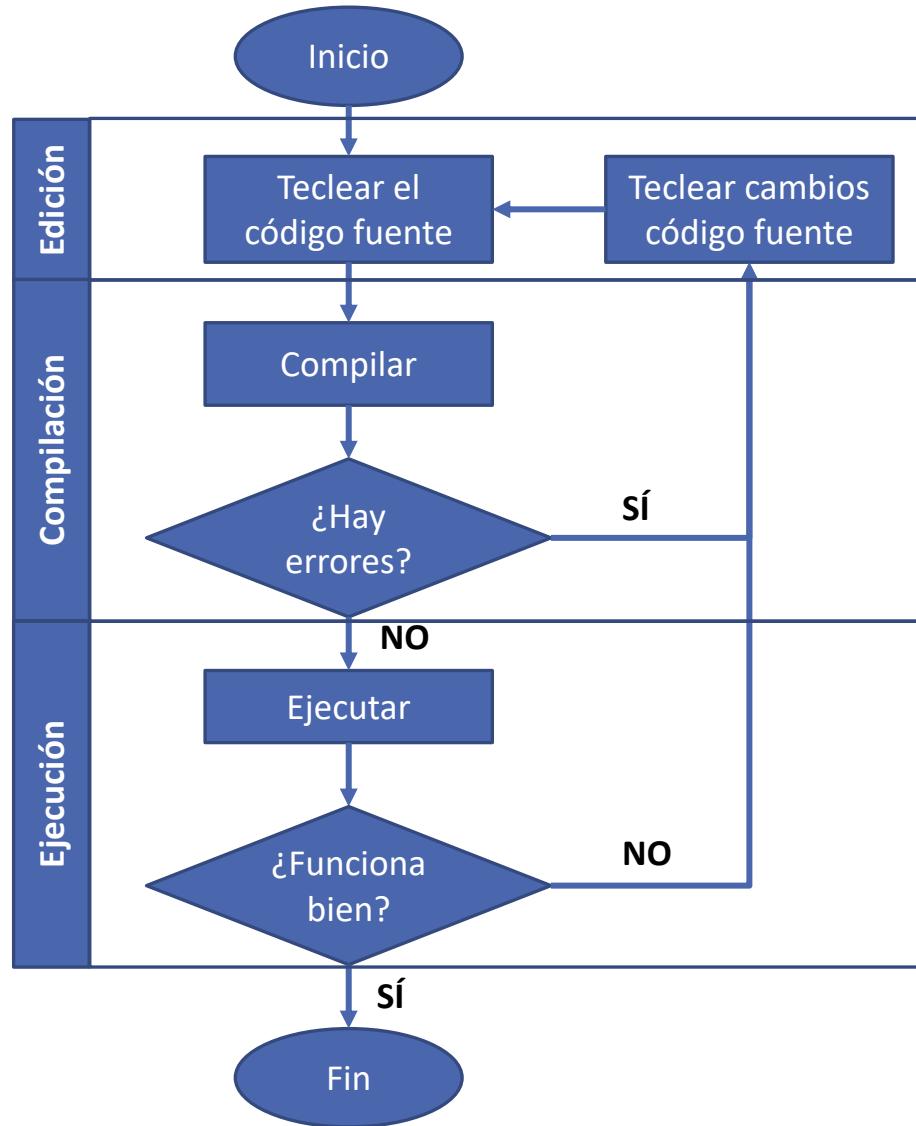
Los objetos se declaran utilizando la palabra reservada *new* seguido por el nombre de la clase:

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

A los miembros de un objeto se puede acceder a través de su referencia, es decir:

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

El proceso de creación de un programa compilado es el siguiente:



Para ejecutar un programa en Java se debe proporcionar al intérprete del lenguaje el nombre del archivo bytecode.

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

El método principal lleva por nombre main (método pivote) y tiene la siguiente forma:

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

Hola Mundo!!



Palabra reservada
para crear clases

Nombre de la clase

```
class HolaMundo {
```

```
    String saludo;
```

```
    String getSaludo(){  
        return saludo;  
    }
```

```
    void setSaludo(String cad){  
        // validar que cad sea un valor válido  
        saludo = cad;  
    }
```

atributo

Métodos
get y set

Palabra reservada
para crear clases



Nombre de la clase



```
class PruebaHolaMundo {  
    public static void main (String [] args) {  
        HolaMundo h = new HolaMundo();  
        h.setSaludo("Bonjour monde!!");  
        System.out.println (h.getSaludo());  
    }  
}
```

Método principal

Para compilar clases en java se utiliza el comando javac. Para ejecutar el bytecode generado por la compilación se debe invocar a la máquina virtual a través del comando java.

En general, solo es necesario compilar la clase que contiene el método principal y ésta a su vez mandará compilar las clases que se utilicen dentro del método principal, en este caso:

```
javac PruebaHolaMundo.java
```

Para ejecutar el programa se debe cargar el bytecode que posee el método principal (`main`), en este caso, `PruebaHolaMundo`, sin su extensión (`.class`):

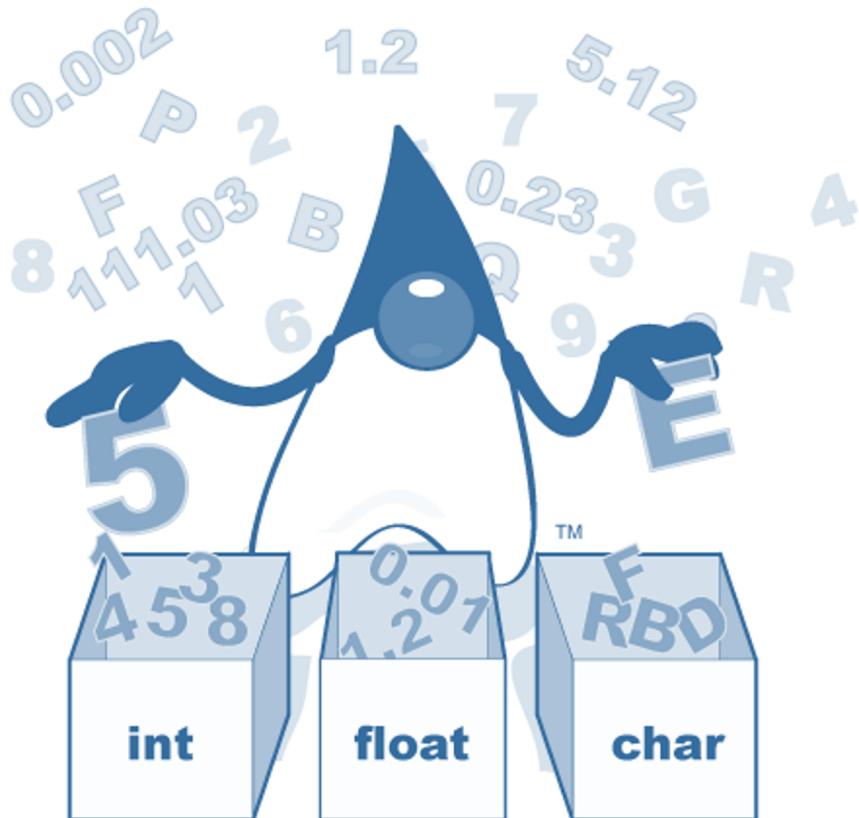
```
java PruebaHolaMundo
```

Tips para 'debugear'

- Los mensajes de error proporcionan el número de la línea donde se presenta el error. Esa línea puede no ser siempre la fuente del problema, también se recomienda revisar la línea anterior.
- Asegurar que se tiene ; al final de cada línea donde se requiere y no en donde no se ocupa.
- Asegurar que se tiene un número par de llaves. Cada llave de apertura { debe tener su correspondiente llave de cierre }.
- Usar identación ayuda a encontrar los errores más rápido.

2.2

Tipos de datos



Los lenguajes de programación poseen diversos tipos de datos primitivos. Así mismo, se pueden crear tipos de datos abstracto para manipular información más específica.

En este subtema se verán las diferentes operaciones que se pueden realizar tanto con los tipos de dato primitivo como con los tipos de dato abstracto.

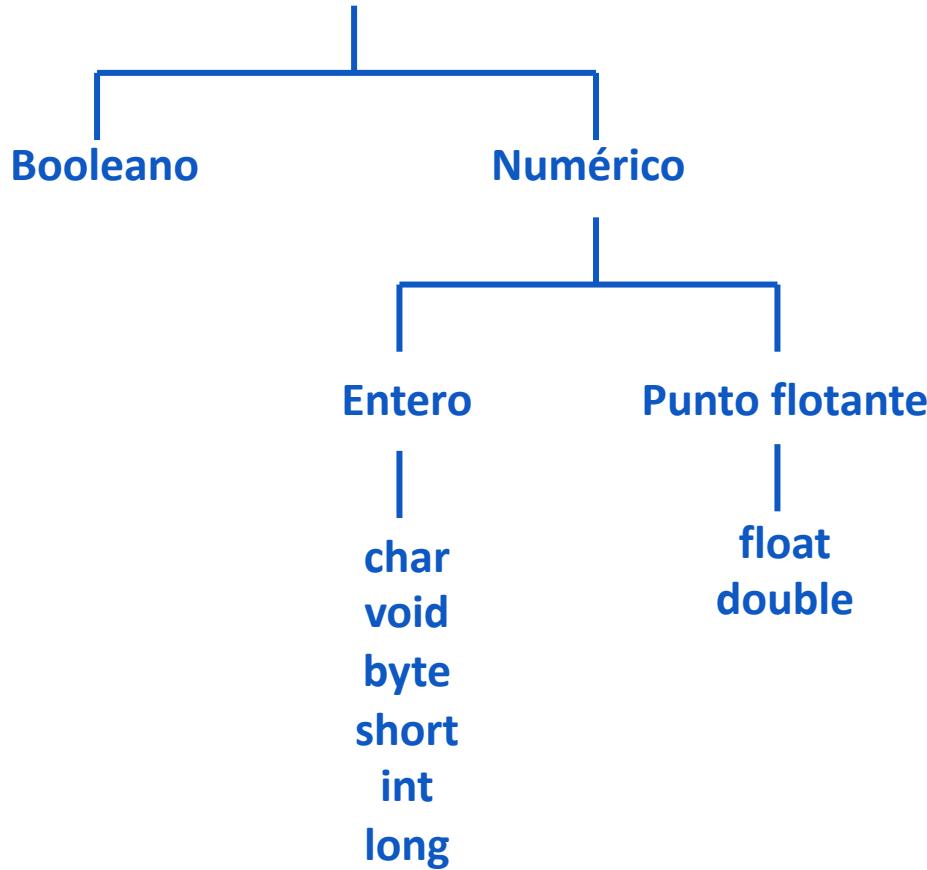
2.2.1 Primitivos y su jerarquía

Los tipos de datos primitivos que maneja Java se pueden agrupar en dos grandes conjuntos: Booleano y numérico.

A su vez, dentro del conjunto de los numéricos se encuentran dos grandes conjuntos: los tipos de dato entero y los tipos de dato flotante.

Dentro de los elementos primitivos existe el concepto de widening, esto es, un valor puede contener a otro de menor tamaño.

Tipos de dato primitivo



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

Los caracteres especiales que posee Java son los siguientes:

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

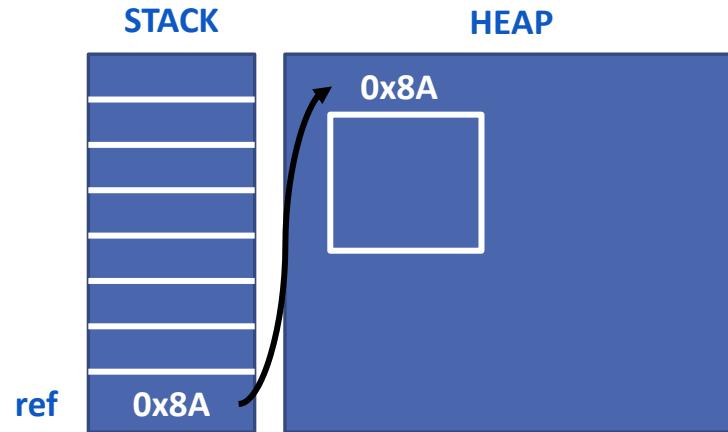
2.2.2 Referencias o instancias

Como ya se mencionó, la sintaxis para crear objetos en Java es la siguiente:

NombreClase nombreReferencia = new NombreClase()

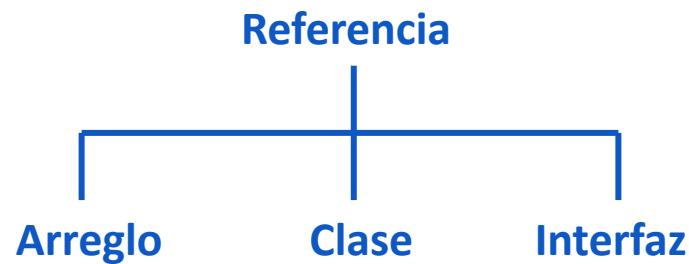
Donde el lado izquierdo de la igualdad corresponde a la referencia del objeto, el lado derecho corresponde a la materialización del objeto en memoria (instancia) y la igualdad corresponde a la asignación del objeto a la referencia.

La memoria se divide en dos grandes conjuntos: el stack y el heap. El stack es un espacio limitado de memoria RAM que permite guardar las referencias. El heap es un espacio más amplio dentro de la memoria RAM donde se crea el objeto.

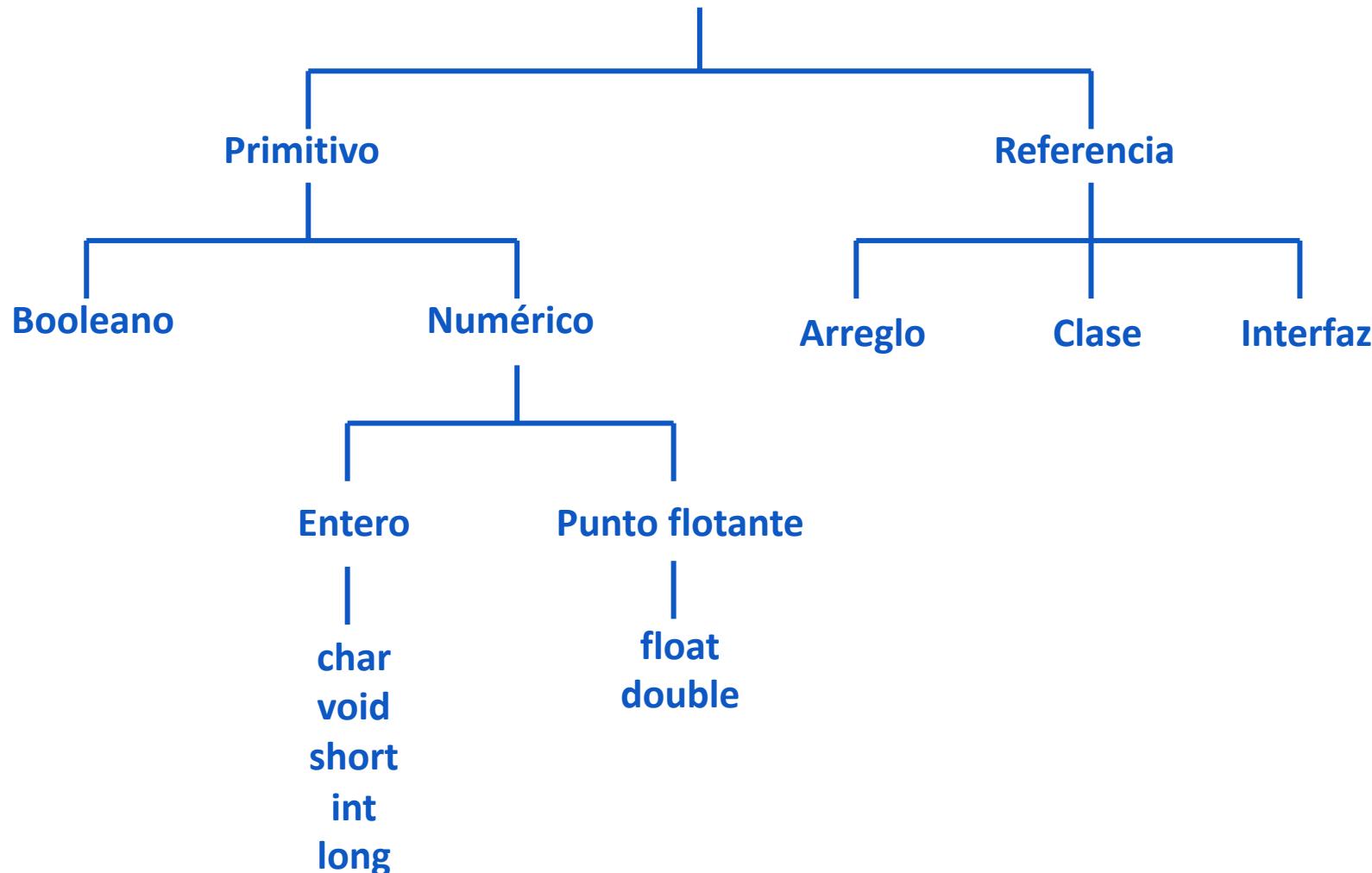


Por lo tanto, para acceder a un objeto a lo largo del tiempo se debe tener una referencia al mismo.

En Java se pueden crear tres tipos de referencias: referencia a arreglo, referencia a objeto y referencia a interfaz.



Tipos de datos en Java

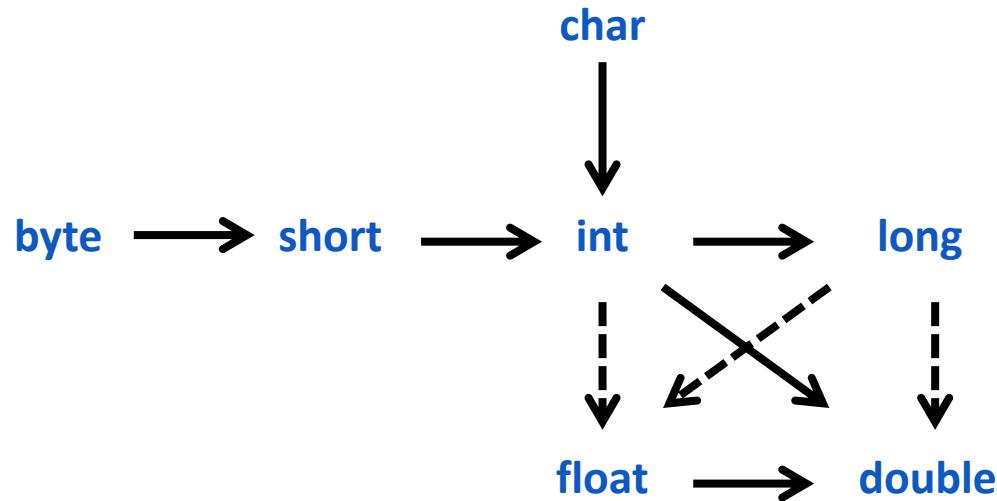


3.2.3 Conversiones entre tipos primitivos (moldeado o casting).

Un moldeado o casting se refiere a la posibilidad que tiene un tipo de dato para comportarse de manera diferente. Existe moldeado a nivel de tipos primitivos numéricos y a nivel de referencias.

En los tipos de datos primitivos el moldeado permite aumentar o reducir el tamaño en memoria del tipo de dato.

Las flechas sólidas muestran conversiones sin pérdida de información y el moldeado se puede realizar de manera implícita, en las conversiones con flechas punteadas puede existir pérdida de información y el casting se debe realizar de manera explícita.



```
public class Cast {  
    public static void main (String [] args){  
        // Se genera un valor de doble precision se moldea a entero  
        double x = 9.997;  
        System.out.println("x = " + x);  
        int castX = (int)x;  
        System.out.println("castX = " + castX);  
    }  
}
```

También es posible transformar una cadena numérica en un tipo de dato primitivo, para ello se ocupan las clases envolventes (o clases wrappers). Existe una clase envolvente para cada uno de los tipos de datos primitivos:

Primitivo	Clase envolvente
char	Character
boolean	Boolean
Byte	Byte
Short	Short
Int	Integer
float	Float
double	Double

```
public class Wrapper {  
    public static void main (String [] args){  
        String y = "5";  
        System.out.println("Cadena y -> " + y);  
        int castY = Integer.parseInt(y);  
        System.out.println("Entero castY = " + castY);  
    }  
}
```

2.2.4 Operadores aritméticos.

Los operadores aritméticos permiten realizar las operaciones básicas suma, resta, multiplicación, división y módulo.

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

3.2.5 Operadores de asignación.

Los operadores de asignación permiten asignar un valor directamente, además de asignar realizando un operación básica previamente.

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

3.2.6 Operadores relacionales.

Los operadores relacionales permiten validar valores cuando se utilizan con tipos primitivos y referencias cuando se utilizan objetos.

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

3.2.7 Operadores especiales (in/decremento (post o pre), concatenación, acceso a variables y métodos y de agrupación).

Los operadores de incremento o decremento permiten aumentar o disminuir, respectivamente, en una unidad un valor. Así mismo, es posible decrementar o incrementar previamente o después de haber realizado un cálculo:

Operadores especiales			
++	Incremento	a++ (postincremento) ++a (preincremento)	
--	Decremento	a-- (postdecremento) --a (predecremento)	

// Postincremento: El incremento que se realiza en la
// línea 3 se lleva a cabo después de la asignación, por lo
// tanto, y = 5, después de la línea 3 x = 6.

```
1 int x, y;  
2 x = 5;  
3 y = x++;
```

// Preincremento: el incremento de la línea 3 se realiza
// antes de la asignación, por lo tanto, a = 3 y b = 3.

```
1 int a, b;  
2 b = 2;  
3 a = ++b
```

La concatenación (o unión) de cadenas se lleva a cabo a través del operador +. Cuando a una cadena se le suma un valor primitivo, se convierte el primitivo a cadena y se une. Cuando a una cadena se le suma un objeto, el objeto se convierte a cadena (a través del método `toString()`) y se une.

Operadores especiales

+

Concatenación de cadenas

`a = "cad1" + "cad2"`

El acceso a los elementos de un objeto (en el heap) se realiza a través de la referencia (desde el stack), utilizando el nombre de la referencia, seguido de un punto, seguido del nombre del atributo o método al que se desea acceder.

Operadores especiales

.

Acceso a variables y métodos

a = obj.var1

Es posible generar un orden de ejecución a través de la agrupación de expresiones.

Operadores.especiales

()

Agrupación de expresiones

$a = (a + b) * c$

2.2.8 Operadores a nivel de bits.

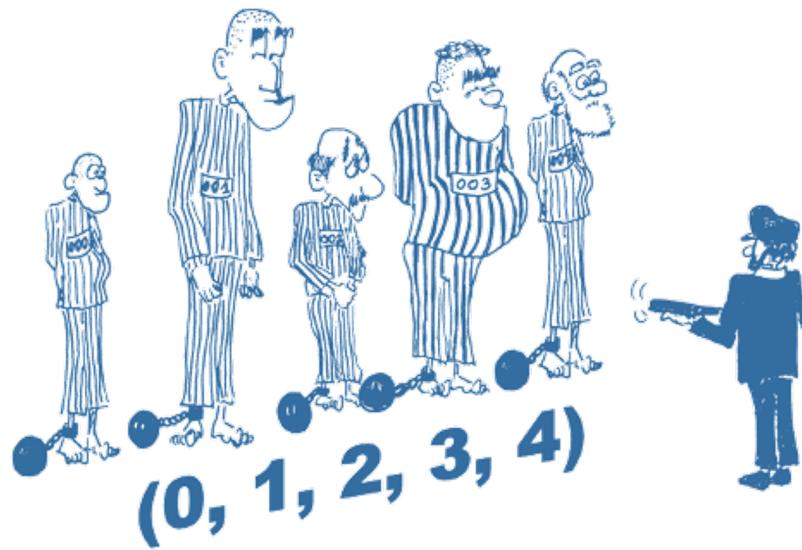
También se pueden realizar operaciones a nivel de bits, lo que hace el cómputo más rápido porque se trabaja directamente en la memoria.

Operadores 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	$-8 >>> 1$

3.2.9 Operadores lógicos.

Dentro de las estructuras de control los operadores lógicos toman gran importancia, Java posee los operadores and, or y not.

Operadores lógicos		
&&	AND	a && b
	OR	a b
!	NOT	! a



2.3

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;  
char s[],t,r;  
int [] s = new int[]{3,4};  
int [][] t = new t[4][];  
int [][] t = new t[][4];
```

- Todas las variables son arreglos.
- Solo s es un arreglo.
- Se declara y se inicializa.
- Declaración legal
- 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
```

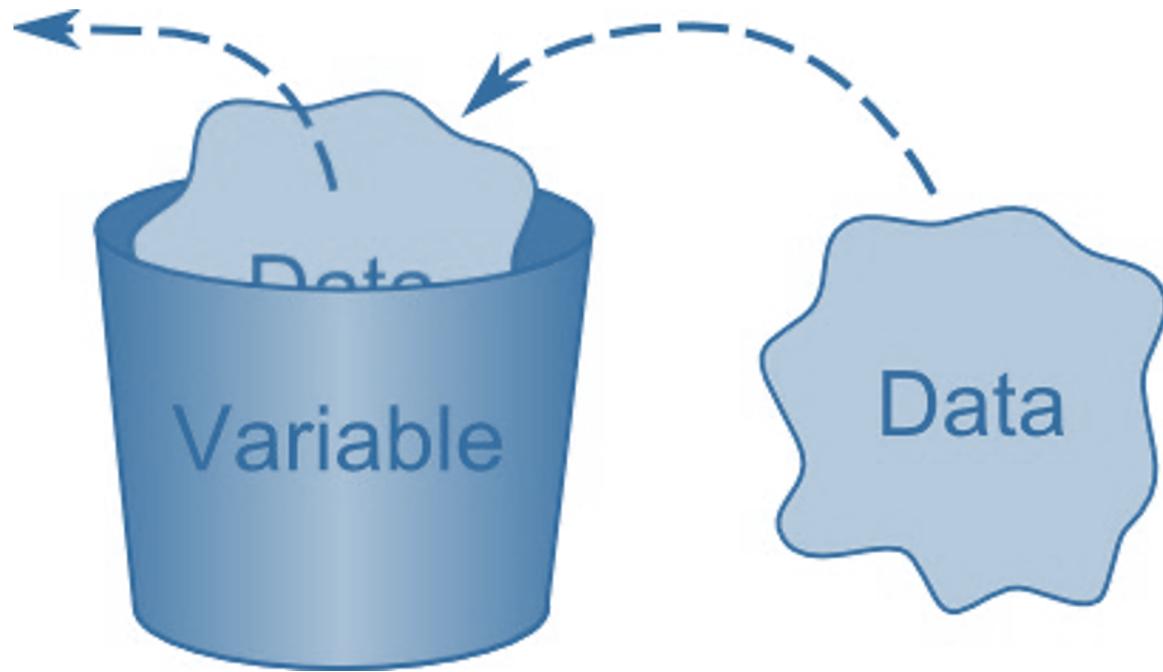
Un método puede recibir como parámetro y/o devolver como valor de retorno un arreglo.

**String [] metodo (Punto []){}
String [] resultado = metodo (puntos);**

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);  
    }  
}
```



2.4. Tipos y ámbito de las variables.

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

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

Las variables locales (también llamadas variables automáticas, temporales 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).

```
public class TiposPrimitivos {  
    double salario;  
    int a;  
    char c;  
    boolean exito;  
    String cad;  
  
    public void imprimirTiposPrimitivos(){  
        System.out.println("Salario=" + salario);  
        System.out.println("\na=" + a);  
        System.out.println("\tc=" + c);  
        System.out.println("\n\texito=" + exito);  
        System.out.println("\n\tcad=" + cad);  
    }  
}
```

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 (x == 0) {  
    p = 5 ;  
}  
int q = p; // error
```

El compilador genera el mismo error (variable p might not have been initialized), 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.

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

strictfp

Punto flotante estricto (strict float point) es un modificador de clase y de método (no de atributo) que indica que la clase o método se rige por el estándar IEEE 754.

El estándar IEEE 754 describe cómo se deben almacenar los números en punto flotante en un equipo binario. Se utiliza en gran medida porque permite que estos números se almacenen en una cantidad de espacio razonable y que los cálculos se realicen con relativa rapidez.

```
public strictfp class EstrictoFP{
    public static void main(String[] args){
        float aFlotante = 0.123456789f;
        double aDoble = 0.04150553411984792d;
        double sum = aFlotante + aDoble;
        float cociente = (float)(aFlotante / aDoble);
        System.out.println("aFlotante: " + aFlotante);
        System.out.println("aDoble: " + aDoble);
        System.out.println("sum: " + sum);
        System.out.println("cociente: " + cociente);
    }
}
```

3.4.1 Elementos estáticos.

Un elemento estático es una variable o método que se mantiene en la memoria durante todo el tiempo de ejecución. Los elementos estáticos no se asocian a un objeto (instancia), sino a la clase en sí, es decir, no se crea una copia del dato para cada objeto sino una sola copia que es compartida por todos los objetos y es propiedad de la clase.

Para declarar un elemento estático se debe agregar el modificador static a la definición del mismo, es decir:

```
// variable estatica  
static tipoVariable nombreVariable;  
// metodo estatico  
static valorRetorno nombreMetodo([Param]);
```

```
class Punto {  
    int x , y ;  
    int numPuntos = 0;  
  
    public void iniciarPunto (int equis , int ye ) {  
        x = equis;  
        y = ye;  
        numPuntos++ ;  
    }  
}
```

El acceso a las variables estáticas desde fuera de la clase donde se definen se puede realizar tanto a través del nombre de la clase como a través del nombre del objeto.

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

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

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;
```

Las variables estáticas existen, se inicializan y pueden usarse antes de que se cree ningún objeto. Por lo tanto, se dice que los elementos estáticos existen en tiempo de compilación.

```
class PruebaPunto {  
    public static void main (String [] args){  
        Punto pto1 = new Punto();  
        Punto pto2 = new Punto();  
        System.out.println(pto1.numPuntos);  
        pto1.iniciarPunto(5, 7);  
        System.out.println(pto2.numPuntos);  
        pto2.iniciarPunto(15, 10);  
        System.out.println(Punto.numPuntos);  
    }  
}
```

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([Parámetros]);
```

```
public class Punto {  
    int x , y ;  
    static int numPuntos = 0;  
  
    public void iniciarPunto (int equis , int ye ) {  
        x = equis;  
        y = ye;  
        numPuntos++ ;  
    }  
  
    static int getNumPuntos() {  
        return numPuntos;  
    }  
}
```

```
class PruebaPunto2 {  
    public static void main (String [] args){  
        Punto pto1 = new Punto();  
        System.out.println(Punto.getNumPuntos());  
        Punto pto2 = new Punto();  
        System.out.println(pto1.numPuntos);  
        pto1.iniciarPunto(5, 7);  
        System.out.println(pto2.getNumPuntos());  
        pto2.iniciarPunto(15, 10);  
        System.out.println(Punto.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. getNumPuntos();
```

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.

Java posee un bloque estático que permite acceder a elementos estáticos antes de iniciar el programa. El bloque está compuesto por la palabra reservada static y está delimitado por llaves:

```
static {  
    // bloque estático  
}
```

El bloque estático verificar valores de las variables estáticas antes de iniciar la aplicación.

```
class BloqueEstatico {  
    static boolean acceso;  
  
    static {  
        System.out.println("Dentro del bloque estático");  
        String so = System.getProperty("os.name");  
        if (so.equals("Linux"))  
            acceso = true;  
    }  
  
    public static void main (String [] args){  
        System.out.println("Método main");  
        if (acceso){  
            System.out.println("Bienvenido al sistema.");  
        } else {  
            System.out.println("Acceso denegado.");  
        }  
    }  
}
```

3.4.2 Elementos constantes.

Un elemento constante es una variable, un método o una clase cuyo valor o funcionamiento no puede ser modificado durante la ejecución del programa.

El concepto de clase final se analizará en el siguiente subtema (Tipos de clase) y el concepto de método final se abordará el siguiente tema (Herencia).

Atributo final

En un atributo (dato miembro), la palabra reservada final impide que éste se pueda redefinir, por lo tanto, para los atributos el modificador final se utiliza para definir constantes. En java, por convención, los atributos constantes se escriben con mayúscula y las palabras se separan con guión bajo.

El modificador final se puede utilizar para definir constantes de instancia o constantes locales.

La palabra const está en desuso, sin embargo, no se puede utilizar ni como palabra reservada y ni como identificador.

```
public class AtributoFinal {  
    public static final float PI = 3.14159265f;  
  
    public static void main(String [] args){  
        final int equis = 2;  
        // Una variable final no se puede modificar  
        //equis = 5;  
        System.out.println(PI*equis);  
    }  
}
```



2.5 Tipos de clases (públicas, sin modificador, abstractas, finales e internas).

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 de el o los primero(s)).

Al orden o clasificación de abstracciones en una estructura de árbol se conoce como jerarquía de clases. Una jerarquía de clases es un conjunto de clases que modelan un sistema 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.

0

Polígono

1

Triángulo

Cuadrilátero

2

Triángulo
equilátero

Triángulo
isósceles

Triángulo
rectángulo

Cuadrado

Rombo

Trapecio

3

Rectángulo

Romboide

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.

NOTA. Todas las clases definidas en java son implícitamente subclases de la clase Object.



2.6 Estructuras de selección.

Las estructuras de control de flujo determinan el orden en el que se ejecutan 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, lo cual sería muy restrictivo, debido a que no se podrían evaluar diferentes soluciones o secuencias de código en función de alguna condición.

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

Por suerte existen dos tipos de instrucciones de control de flujo: las sentencias de control alternativas y las 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.

2.6.1 Estructura 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 estructura es la instrucción if: el bloque de código gobernado por if se ejecuta si la condición evaluada es verdadera, su sintaxis es la siguiente:

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

La estructura completa permite ejecutar otro bloque de código si la condición lógica no se cumple, es decir, si la condición lógica es falsa.

```
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 estructura (el bloque if o el bloque else) pueden contener, a su vez, otra estructura if o if-else (estructuras anidadas).

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

```
public class Incrementos {  
    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);  
    }  
}
```

2.6.3 Estructura ternaria.

El operador ternario es una estructura parecida a if-else, para cuando el bloque de código está constituido por una sola instrucción. La sintaxis del operador ternario es la siguiente:

`condicion_logica ? expresion1: expresion2`

El operador ternario evalúa la condición lógica, si se cumple (true) se ejecuta la instrucción que está a la derecha del ? (expresion1); si no se cumple (false) se ejecuta la instrucción que está a la derecha de los : (expresion2).

```
public class Modulo {  
    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");  
        }  
    }  
}
```

Número mágico

Piensa en un número entre 0 y 7...

$$\begin{array}{r} 1 \quad 3 \quad 5 \quad 7 \\ \hline 7 \quad 6 \quad 2 \quad 3 \\ \hline 6 \quad 5 \quad 7 \quad 4 \end{array}$$

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

2.6.2 Estructura switch-case.

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.

La sentencia switch evalúa la expresión dada (generalmente un entero, aunque es posible evaluar también un carácter o un enumerador) 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. Los casos (case) deben ser constantes, es decir, no pueden cambiar su valor.

La sintaxis de la estructura de selección switch-case es la siguiente:

```
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  
}
```

```
public class SwitchInt {  
    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;  
            default:  
                System.out.println("Opción inválida.");  
        }  
    }  
}
```

```
public 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;  
            default:  
                System.out.println("Opción inválida.");  
        }  
    }  
}
```

Enumerador

Los enumeradores son listas que permiten almacenar valores constantes. Todos los valores que le pertenecen a un enumerador son estáticos y finales.

Para definir una enumeración en java se utiliza la palabra reservada enum. La sintaxis es la siguiente:

```
[modificadorAcceso] enum Nombre {  
    VALOR1, VALOR2, VALOR3,...  
}
```

Debido a que los elementos de las enumeraciones son estáticas, el acceso a los mismos se realiza a través del nombre de la enumeración seguido de punto y después el nombre del valor.

```
public enum Valores {  
    VALOR1, VALOR2, VALOR3  
}
```

```
Valores valor = Valores.VALOR1;
```

Los nombres de las enumeraciones siguen la misma convención que los nombres de las clases, es decir, ocupan notación Upper Camell Case. Así mismo, como los valores son constantes se escriben en mayúscula.

```
public class DiasSemana{
    public enum Dia {
        DOMINGO, LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO
    }

    public void describirDia(Dia diaElegido) {
        switch (diaElegido) {
            case LUNES:
                System.out.println("Inicio de semana.");
                break;
            case VIERNES:
                System.out.println("Inicia fin de semana.");
                break;
            case SABADO:
            case DOMINGO:
                System.out.println("Día de descanso.");
                break;
            default:
                System.out.println("Entre semana.");
                break;
        }
    }
}
```

```
public class PruebaDiasSemana{
    public static void main(String[] args) {
        DiasSemana dias = new DiasSemana();
        System.out.println("LUNES");
        dias.describirDia(DiasSemana.Dia.LUNES);
        System.out.println("MIÉRCOLES");
        dias.describirDia(DiasSemana.Dia.MIERCOLES);
        System.out.println("VIERNES");
        dias.describirDia(DiasSemana.Dia.VIERNES);
        System.out.println("SÁBADO");
        dias.describirDia(DiasSemana.Dia.SABADO);
    }
}
```

```
public class SwitchNoBreak {  
    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");  
    }  
}
```

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. Se pueden utilizar las funciones de la clase Math.



2.6 Estructuras de repetición.

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

2.7.1 Estructura 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.

```
public class While {  
    public static void main (String [] args) {  
        int x = 0;  
        int y = 10;  
        while(x < y) {  
            System.out.println("x = " + x++);  
        }  
    }  
}
```

2.7.2 Estructura do-while.

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 hasta  
    // que se cumpla la condición lógica  
} 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 mismo.

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

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

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 de la estructura switch.

Dentro de un ciclo, la sentencia break rompe la ejecución del ciclo, evitando así que este siga iterando.

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

La sentencia break permite utilizar etiquetas para validar el salto, es decir, en lugar de saltar a la condición inmediata es posible evaluar una condición de un ciclo más externo.

```
public class BreakEtiqueta {  
    public static void main (String [] a){  
        int x = 0;  
        fuera:  
        do {  
            do {  
                x++;  
                if (x>2) {  
                    break fuera;  
                }  
                System.out.println("if");  
            } while (x<3);  
            x++;  
            System.out.println("Do while interno");  
        } while (x<4);  
        System.out.println("Do while externo");  
    }  
}
```

Continue

La sentencia continue rompe el ciclo pero, a diferencia de break, en vez de forzar la finalización, continue permite volver al ciclo si la condición se sigue cumpliendo.

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

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

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.

2.7.3 Estructura for.

El ciclo for es una sentencia o bucle repetitivo que permite ejecutar un conjunto de instrucciones de manera cíclica mientras se cumpla la condición establecida. Su sintaxis es la siguiente:

```
for (asignación_inicial ; condición_lógica ; paso){  
    // Código a ejecutarse mientras se  
    // cumpla la condición lógica  
}
```

Asignación inicial: es una sentencia que se ejecuta la primera vez que se entra en el ciclo for. Normalmente es una asignación. Es un campo opcional.

Expresión lógica: es una expresión que se evalúa antes de ejecutar el bloque de código en cada iteración. La sentencia o bloque de sentencias se ejecutan mientras 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.

```
public class Argumentos{
    public static void main(String args[]){
        if ( args.length == 0 ){
            System.out.println("ERROR!, al ejecutar el programa!");
            System.out.println("Uso: java ForArgs ARG1 ARG2 ...");
        }

        for ( int i = 0 ; i < args.length ; i++ ){
            System.out.println("Parametro " + (i+1) + ":" + args[i]);
        }
    }
}
```

```
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);  
        }  
    }  
}
```

```
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]+\n");  
            }  
            System.out.println();  
        }  
    }  
}
```

Foreach

Existe una variante del ciclo for conocida como foreach. Foreach es una estructura que permite recorrer de forma rápida un arreglo o un enumerador. Su sintaxis es la siguiente:

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

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

```
public class ForeachEnum {  
  
    public enum Planeta {  
        MERCURIO, VENUS, TIERRA, MARTE,  
        JUPITER, SATURNO, URANO, NEPTUNO  
    }  
  
    public static void main (String [] a){  
        for(Planeta x : Planeta.values()){  
            System.out.println(x);  
        }  
    }  
}
```

Argumentos variables

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

```
[modificador] valorRetorno nombre (tipoDatos ... nombre) {  
    // Código método  
}
```

Indica que el método puede recibir de 0 a n variables de tipo Dato 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.

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

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

**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”

“If you lie to the compiler, it will get its revenge.”

Henry Spencer

**(A Canadian computer programmer, he wrote "regex",
a widely used software library for regular expressions)**

UTILERÍAS

Clase Math

La clase Math posee métodos para realizar operaciones numéricas básicas, así como funciones básicas como exponente, logaritmo, raíz cuadrada y funciones trigonométricas. Es importante aclarar que las funciones matemáticas que pertenecen a la clase Math se invocan de la siguiente manera:

Math.funcion(argumentos)

Donde Math es el nombre de la clase, función es el nombre de la función a la que se quiera acceder y, entre paréntesis, los argumentos que requiera la función.

```
public class UtileriaMath {  
    public static void main(String [] args){  
        double equis = 8.2;  
        System.out.println(equis + "2 = " + Math.pow(equis, 2));  
        System.out.println("Raiz cuadrada de " + equis + " = " + Math.sqrt(equis));  
        System.out.println("seno(" + equis + ") = " + Math.sin(equis));  
        System.out.println("Número aleatorio = " + Math.random());  
        System.out.println("logaritmo(" + equis + ") = " + Math.log(equis));  
        System.out.println("PI * " + equis + "2 = " + Math.PI * Math.pow(equis, 2.0));  
    }  
}
```

Clase String

La clase String representa cadenas de caracteres. Como todo en Java las cadenas son objetos, por lo tanto, se puede crear una instancia de un String con la palabra reservada new:

```
String cad = new String("Aquí va la cadena de caracteres");
```

Sin embargo, también es posible asignar una cadena directamente a una variable tipo String:

```
String cad2 = "Otra manera de declarar una cadena";
```

```
public class ClaseString {  
    public static void main (String [] args){  
        String cad1 = new String ("hola");  
        String cad2 = "hola";  
        String cad3 = new String ("hola");  
        String cad4 = "hola";  
        if (cad1 == cad2){  
            System.out.println("cad1 == cad2");  
        }  
        if (cad1 == cad3){  
            System.out.println("cad1 == cad3");  
        }  
        if (cad2 == cad3){  
            System.out.println("cad2 == cad3");  
        }  
        if (cad2 == cad4){  
            System.out.println("cad2 == cad4");  
        }  
    }  
}
```

Concatenación de cadenas.

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

```
String cad = "hola " + "mundo!";
```

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 concatena una cadena (String) con una variable primitiva, la JVM transforma la variable primitiva en una cadena (String) y las une.

Cuando se concatena una cadena (String) con una referencia a objeto, la JVM transforma al objeto en una cadena, invocando el método `toString()`, el cual existe en todos los objetos debido a que es un método declarado en la clase Object.

Algunos métodos importantes de la clase String son:

Método	Descripción
char charAt(int index)	Devuelve el carácter en la posición indicada por index. El rango de index va de 0 a length() - 1.
boolean equals(Object obj)	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.
boolean equalsIgnoreCase(String s)	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.
char[] toCharArray() String toLowerCase() String toUpperCase()	Transforman el string en un array de caracteres, o a mayúsculas o a minúsculas.

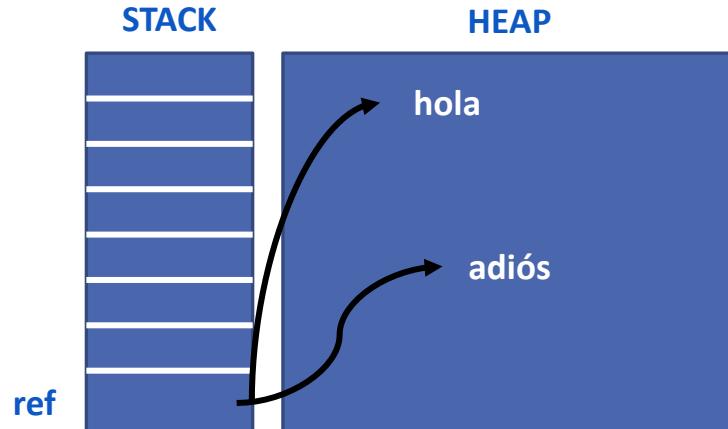
Algunos métodos importantes de la clase String son:

Método	Descripción
int indexOf(char c)	Devuelve el indice donde se produce la primera aparición de c. Devuelve -1 si c no está en el string.
int indexOf(String s)	Igual que el anterior pero buscando la subcadena representada por s.
int length()	Devuelve la longitud del String (número de caracteres)
String substring(int begin, int end)	Devuelve un substring desde el índice begin hasta el end
static String valueOf(int i)	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.

Algunos métodos importantes de la clase String son:

Método	Descripción
String toLowerCase()	Devuelve una cadena con todas las letras mayúsculas convertidas a minúsculas.
String toUpperCase()	Devuelve una cadena con todas las letras minúsculas convertidas a mayúsculas.
String trim()	Devuelve una cadena con todos los espacios en blanco al inicio o al final eliminados.

Las cadenas son inmutables, es decir, una vez que se ha creado una cadena su valor no puede cambiar. Sin embargo, la referencia sí es mutable y puede cambiar al valor al que apunta.



Clase StringBuffer y StringBuilder

El uso de estas clases se recomienda cuando la información contenida en los objetos va a cambiar constantemente, debido a que los objetos de la clase String son inmutables, entonces, si se generan varios objetos con la clase String al final de la ejecución del programa se tendrán un montón de Strings creados en el pool de Strings.

Por otro lado, los objetos creados con las clases StringBuffer y StringBuilder pueden ser modificados una y otra vez sin dejar regados los objetos por el heap.

Un uso común de los StringBuffer y de los StringBuilder se presenta al leer o escribir en un archivo grande.

La clase StringBuilder fue agregada en la versión 5 de Java. Tiene las mismas funcionalidades que StringBuffer con la salvedad de que StringBuilder no es thread-safe, es decir, sus métodos no son sincronizados (este concepto se verá cuando se aborde el tema de Hilos).

```
public class ClaseStringBuffer {  
    public static void main(String [] args){  
        StringBuffer sb = new StringBuffer("Somos o no somos");  
        sb.reverse();  
        System.out.println(sb);  
        sb.append(8.5f);  
        System.out.println(sb);  
        sb.insert((int)(sb.length()/2),"lol");  
        System.out.println(sb);  
    }  
}
```

Colecciones

Una colección es un conjunto de datos, es decir, una referencia que permite manejar un grupo de objetos, a este grupo de objetos se les conoce como elementos de la colección.

Todas las colecciones en Java tiene métodos generales como lo son:

- **add**
- **Contains**
- **isEmpty**
- **iterator**
- **remove**
- **size**

Conjunto

Un conjunto (Set) es una colección desordenada (no mantiene un orden de inserción) que no permite elementos duplicados.

Dentro de los conjuntos se encuentran:

- **HashSet**
- **SortedSet**
- **TreeSet**

```
import java.util.Set;
import java.util.HashSet;

public class Conjunto {
    public static void main(String[] args) {
        Set c = new HashSet();
        c.add("uno");
        c.add("segundo");
        c.add("3ro");
        c.add(new Integer(4));
        c.add(new Float(5.0F));
        c.add("segundo"); // duplicado
        c.add(new Integer(4)); // duplicado
        System.out.println(c);
    }
}
```

Lista

Una lista (List) es una colección ordenada (mantiene el orden de inserción) que permite elementos duplicados.

Dentro de las listas se encuentran:

- **ArrayList**
- **LinkedList**

```
import java.util.List;
import java.util.ArrayList;

public class Lista {
    public static void main(String[] args) {
        List l = new ArrayList();
        l.add("uno");
        l.add("segundo");
        l.add("3ro");
        l.add(new Integer(4));
        l.add(new Float(5.0F));
        l.add("segundo"); // duplicado
        l.add(new Integer(4)); // duplicado
        System.out.println(l);
    }
}
```

Mapa

Un mapa (también llamado arreglo asociativo) es un conjunto de elementos agrupados con una llave y un valor:

<llave, valor>

Las llaves de un diccionario deben ser únicas y a cada valor le corresponde una. La columna de valores sí puede repetir elementos.

El contenido de un mapa (ya sea su valor o su llave) puede ser manipulado como colecciones gracias a los métodos que posee la interfaz Map:

- **entrySet():** regresa un conjunto con todos los pares <llave-valor>.
- **keySet():** regresa un conjunto con todas las llaves del mapa.
- **values():** regresa una colección con todos los valores en el mapa.

Dentro de los mapas más utilizados se encuentran:

- **SortedMap**
- **Hashtable**
- **HashMap**
- **TreeMap**
- **Properties**
- **LinkedHashMap**

```
import java.util.Map;
import java.util.Set;
import java.util.Collection;
import java.util.HashMap;

public class Mapa {
    public static void main(String [] args) {
        Map m = new HashMap();
        m.put("uno","1ro");
        m.put("segundo", new Integer(2));
        m.put("tercero","3rd");
        // Se sobre-escribe un valor
        m.put("tercero","III");
        // Regresa un conjunto con las llaves del mapa
        Set llaves = m.keySet();
        // Regresa una colección con los valores del mapa
        Collection valores = m.values();
        // Regresa un conjunto con los pares <llave, valor>
        Set pares = m.entrySet();
        System.out.println("Llaves: \n " + llaves);
        System.out.println("Valores: \n " + valores);
        System.out.println("Pares: \n " + pares);
    }
}
```

Garbage collector

El garbage collection es un término que se utiliza para referirse al manejo automático de la memoria en Java.

Como ya se mencionó, el heap es la parte de la memoria donde se crean los objetos, es ahí donde el proceso del garbage collection tiene su razón de ser. El garbage collector libera la memoria que está en el heap y que no tiene referencia alguna.

El garbage collector se encuentra bajo el control de la JVM, la cual decide cuando se debe ejecutar. Se puede hacer una llamada explícita al garbage collector para que se ejecute, pero no existe garantía de que esto suceda.

```
public class Garbage {  
    public static void main (String [] args){  
        System.out.println("Procesadores disponibles: "  
                           + Runtime.getRuntime().availableProcessors());  
        System.out.println("Memoria al iniciar el programa:");  
        System.out.println("Memoria total: " + Runtime.getRuntime().totalMemory());  
        System.out.println("Memoria libre: " + Runtime.getRuntime().freeMemory());  
  
        TreeSet []t = new TreeSet[1000];  
        for (int cont = 0 ; cont < t.length ; cont++)  
            t[cont] = new TreeSet();
```

```
System.out.println();
System.out.println("Memoria despues de reservar TreeSet[1000]:");
System.out.println("Memoria total: " + Runtime.getRuntime().totalMemory());
System.out.println("Memoria libre: " + Runtime.getRuntime().freeMemory());

for (int cont = 0 ; cont < t.length ; cont++)
    t[cont] = null;

Runtime.getRuntime().gc();
//System.gc();
System.out.println();
System.out.println("Memoria despues de liberar los recursos del Arbol:");
System.out.println("Memoria total: " + Runtime.getRuntime().totalMemory());
System.out.println("Memoria libre: " + Runtime.getRuntime().freeMemory());
}

}
```

Método equals

Para saber si dos objetos son iguales se utiliza el método equals. Si no se sobrescribe el método, no se podrá utilizar el objeto como una llave de una HashTable y, probablemente, no se consiga un Conjunto preciso, ya que conceptualmente no habrá duplicados.

Método hashCode

El método hashCode es utilizado para mejorar el desempeño de grandes colecciones de datos. Es utilizado por algunas Collections en Java.

Colecciones como HashMap o HashSet utilizan este método para determinar en qué posición de la colección se guardará el dato.

```
public class Point {  
  
    private int x;  
    private int y;  
  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public void setX(final int x) {  
        this.x = x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setY(final int y) {  
        this.y = y;  
    }  
  
    @Override  
    public String toString() {  
        final StringBuilder sb = new StringBuilder();  
        sb.append("Point = {" );  
        sb.append(super.toString());  
        sb.append(", x = ").append(x);  
        sb.append(", y = ").append(y);  
        sb.append('}');  
        return sb.toString();  
    }  
}
```

```
import java.util.HashSet;
import java.util.Set;

public class TestPoint {

    public static void main(String[] args) {

        final Point p1 = new Point(10, 20);
        final Point p2 = new Point(60, 30);
        final Point p3 = new Point(10, 20);

        final Set<Point> points = new HashSet<Point>();
        points.add(p1);
        points.add(p2);
        points.add(p3);

        System.out.println(points);
        System.out.println(p1.hashCode());
        System.out.println(p3.hashCode());
    }
}
```

```
@Override  
public boolean equals(final Object object) {  
    if (this == object)  
        return true;  
    if (!(object instanceof Point))  
        return false;  
    final Point point = (Point) object;  
    return (this.x == point.x) && (this.y == point.y);  
}
```

```
@Override  
public int hashCode() {  
    return x * 31 + y;  
}
```

2 Tipos, expresiones y control de flujo

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

2.1 Generalidades.

2.2 Tipos de datos.

2.3 Arreglos.

2.4 Tipos y ámbito de las variables.

2.5 Tipos de clases (públicas, sin modificador, abstractas, finales e internas).

2.6 Estructuras de selección.

2.7 Estructuras de repetición.