

Elementos básicos del lenguaje Java

Ricardo Pérez López

IES Doñana, curso 2020/2021



Generado el 12 de febrero de 2021 a las 12:24:00

1. Tipos y valores en Java
2. Variables en Java
3. Estructuras de control
4. Entrada/salida

1. Tipos y valores en Java

1.1 Introducción

1.2 Tipos primitivos

1.3 Tipos referencia

1.1. Introducción

Introducción

- ▶ El lenguaje de programación Java es un **lenguaje de tipado estático**, lo que significa que cada variable y cada expresión tienen un tipo que se conoce en tiempo de compilación.
- ▶ El lenguaje de programación Java también es un **lenguaje fuertemente tipado**, porque los tipos limitan las operaciones que se pueden realizar sobre unos valores dependiendo de sus tipos y determinan el significado de dichas operaciones.
- ▶ El tipado estático fuerte ayuda a **detectar errores en tiempo de compilación**, es decir, antes incluso de ejecutar el programa.

- ▶ Los **tipos** del lenguaje de programación Java se dividen en **dos categorías**:
 - Tipos **primitivos**.
 - Tipos **referencia**.
- ▶ Consecuentemente, en Java hay dos categorías de **valores**:
 - Valores primitivos.
 - Valores referencia.

- ▶ Los **tipos primitivos** son:
 - El tipo **booleano** (`boolean`).
 - Los tipos **numéricos**, los cuales a su vez son:
 - Los tipos **integrales**: `byte`, `short`, `int`, `long` y `char`.
 - Los tipos **de coma flotante**: `float` y `double`.
- ▶ Los **tipos referencia** son:
 - Tipos **clase**.
 - Tipos **interfaz**.
 - Tipos **array**.
- ▶ Además, hay un tipo especial que representa el valor **nulo** (`null`).

- ▶ En Java, un objeto sólo puede ser una de estas dos cosas:
 - Una instancia creada en tiempo de ejecución a partir de una clase.
 - Un *array* creado en tiempo de ejecución.
- ▶ Los valores de un tipo referencia son referencias a objetos.
- ▶ Todos los objetos, incluidos los *arrays*, admiten los métodos de la clase `Object`.
- ▶ Las cadenas literales representan objetos de la clase `String`.

1.2. Tipos primitivos

1.2.1 Booleanos

1.2.2 Integrales

1.2.3 De coma flotante

1.2.4 Subtipado

1.2.5 Conversiones entre datos primitivos

1.2.6 Promociones numéricas

Tipos primitivos

- ▶ Los **tipos primitivos** están predefinidos en Java y se identifican mediante su nombre, el cual es una palabra clave reservada en el lenguaje.
- ▶ Un valor primitivo es un valor de un tipo primitivo.
- ▶ Los valores primitivos **no son objetos** y no comparten estado con otros valores primitivos.
- ▶ En consecuencia, los valores primitivos no se almacenan en el montículo y, por tanto, **las variables que contienen valores primitivos** no guardan una referencia al valor, sino que **almacenan el valor mismo**.
- ▶ Los tipos primitivos son los **booleanos**, los **integrales** y los tipos de **coma flotante**.

Booleanos

- ▶ El tipo booleano (`boolean`) contiene dos valores, representados por los literales booleanos `true` (verdadero) y `false` (falso).
- ▶ Un literal booleano siempre es de tipo `boolean`.
- ▶ Sus operaciones son:

Igualdad: `==, !=`

Complemento lógico (*not*): `!`

And, *or* y *xor* estrictos: `&, |, ^`

And y *or* perezosos: `&&, ||`

Condicional ternario: `? :`

```
jshell> true && false
$1 ==> false

jshell> false == false
$2 ==> true

jshell> true ^ true
$3 ==> false
```

```
jshell> !true
$4 ==> false

jshell> true ? 1 : 2
$5 ==> 1

jshell> false ? 1 : 2
$6 ==> 2
```

Integrales

- ▶ Los **tipos integrales** son:
 - **Enteros** (`byte`, `short`, `int` y `long`): sus valores son **números enteros con signo** en complemento a dos.
 - **Caracteres** (`char`): sus valores son **enteros sin signo** que representan caracteres Unicode almacenados en forma de *code units* de UTF-16.
- ▶ Sus tamaños y rangos de valores son:

Tipo	Tamaño	Rango
<code>byte</code>	8 bits	-128 a 127 inclusive
<code>short</code>	16 bits	32768 a 32767 inclusive
<code>int</code>	32 bits	2147483648 a 2147483647 inclusive
<code>long</code>	64 bits	9223372036854775808 a 9223372036854775807 inclusive
<code>char</code>	16 bits	'\u0000' a '\uffff' inclusive, es decir, de 0 a 65535

- ▶ Los literales que representan números enteros pueden ser de tipo `int` o de tipo `long`.
- ▶ Un literal entero será de tipo `long` si lleva un sufijo `L` o `L`; en caso contrario, será de tipo `int`.
- ▶ Se pueden usar caracteres de subrayado (`_`) como separadores entre los dígitos del número entero.
- ▶ Los literales de tipos enteros se pueden expresar en:
 - **Decimal:** no puede empezar por `0`, salvo que sea el propio número `0`.
 - **Hexadecimal:** debe empezar por `0x` o `0X`.
 - **Octal:** debe empezar por `0`.
 - **Binario:** debe empezar por `0b` o `0B`.

► Ejemplos de literales de tipo `int`:

0
2
0372

0xDada_Cafe
1996
0x00_FF__00_FF

► Ejemplos de literales de tipo `long`:

0l
0777L
0x100000000L

2_147_483_648L
0xC0B0L

- ▶ Un literal de tipo `char` representa un carácter o secuencia de escape.
- ▶ Se escriben encerrados entre comillas simples (también llamadas *apóstrofes*).
- ▶ Los literales de tipo `char` sólo pueden representar *code units* de Unicode y, por tanto, sus valores deben estar comprendidos entre `'\u0000'` y `'\uffff'`.
- ▶ Ejemplos de literales de tipo `char`:

```
'a'  
'%'  
'\t'  
'\\'  
'\''
```

```
'\u03a9'  
'\uFFFF'  
'\177'  
'™'
```

En Java, los **caracteres** y las **cadenas** son **tipos distintos**.

Operadores integrales

- ▶ Java proporciona una serie de operadores que actúan sobre valores integrales.
- ▶ Los **operadores de comparación** dan como resultado un valor de tipo `boolean`:

Comparación numérica: `<, <=, >, >=`

Igualdad numérica: `==, !=`

```
jshell> 2 <= 3
$1 ==> true

jshell> 4 != 4
$2 ==> false
```


- Los **operadores numéricos** dan como resultado un valor de tipo `int` o `long`:

Signo más y menos (unarios):	<code>+, -</code>
Multiplicativos:	<code>*, /, %</code>
Suma y resta:	<code>+, -</code>
Preincremento y postincremento:	<code>++</code>
Predecremento y postdecremento:	<code>--</code>
Desplazamiento con y sin signo:	<code><<, >>, >>></code>
Complemento a nivel de bits:	<code>~</code>
<i>And, or y xor</i> a nivel de bits:	<code>&, , ^</code>

- ▶ Si un operador integral (que no sea el desplazamiento) tiene al menos un operando de tipo `long`, la operación se llevará a cabo en precisión de 64 bits y el resultado de la operación numérica será de tipo `long`.

Si el otro operando no es `long`, se convertirá primero a `long`.

- ▶ En caso contrario, la operación se llevará a cabo usando precisión de 32 bits y el resultado de la operación numérica será de tipo `int`.

Si alguno de los operandos no es `int` (por ejemplo, `short` o `byte`), se convertirá primero a `int`.

- ▶ Ciertas operaciones pueden lanzar excepciones. Por ejemplo, el operador de división entera (`/`) y el resto de la división entera (`%`) lanzan una excepción `ArithmeticException` si el operando derecho es cero.

```
jshell> 2 + 3
$1 ==> 5           // Devuelve un int

jshell> 2 + 3L
$2 ==> 5           // Devuelve un long

jshell> 8 >> 1
$3 ==> 4

jshell> -8 >> 1
$4 ==> -4

jshell> -8 >>> 1
$5 ==> 2147483644

jshell> 2 == 3 ? 5 : 6L
$6 ==> 6
```

De coma flotante

- ▶ Los **tipos de coma flotante** son valores que representan **números reales** almacenados en el formato de coma flotante **IEEE-754**.
- ▶ Existen dos tipos de coma flotante:
 - **float**: sus valores son números de coma flotante de 32 bits (simple precisión).
 - **double**: sus valores son números de coma flotante de 64 bits (doble precisión).
- ▶ Un literal de coma flotante tiene las siguientes partes en este orden (que algunas son opcionales según el caso):
 1. Una parte entera.
 2. Un punto (.).
 3. Una parte fraccionaria.
 4. Un exponente.
 5. Un sufijo de tipo.

- ▶ Los literales de coma flotante se pueden expresar en decimal o hexadecimal (usando el prefijo `0x` o `0X`).
- ▶ Todas las partes numéricas del literal (la entera, la fraccionaria y el exponente) deben ser decimales o hexadecimales, sin mezclar algunas de un tipo y otras de otro.
- ▶ Se permiten caracteres de subrayado (`_`) para separar los dígitos de la parte entera, la parte fraccionaria o el exponente.
- ▶ El exponente, si aparece, se indica mediante el carácter `e` o `E` (si el número es decimal) o el carácter `p` o `P` (si es hexadecimal), seguido por un número entero con signo.
- ▶ Un literal de coma flotante será de tipo `float` si lleva un sufijo `f` o `F`; si no lleva ningún sufijo (o si lleva opcionalmente el sufijo `d` o `D`), será de tipo `double`.

- ▶ El literal positivo finito de tipo `float` más grande es `3.4028235e38f`.
- ▶ El literal positivo finito de tipo `float` más pequeño distinto de cero es `1.40e-45f`.
- ▶ El literal positivo finito de tipo `double` más grande es `1.7976931348623157e308`.
- ▶ El literal positivo finito de tipo `double` más pequeño distinto de cero es `4.9e-324`.

► Ejemplos de literales de tipo `float`:

`1e1f`
`2.f`
`.3f`

`0f`
`3.14f`
`6.022137e+23f`

► Ejemplos de literales de tipo `double`:

`1e1`
`2.`
`.3`
`0.0`

`3.14`
`1e-9d`
`1e137`

- ▶ El estándar IEEE-754 incluye números positivos y negativos formados por un signo y una magnitud.
- ▶ También incluye:
 - Ceros positivo y negativos:

`+0``-0`

- Infinitos positivos y negativos:

`Float.POSITIVE_INFINITY``Double.POSITIVE_INFINITY``Float.NEGATIVE_INFINITY``Double.NEGATIVE_INFINITY`

- Valores especiales *Not-a-Number* (o *NaN*), usados para representar ciertas operaciones no válidas como dividir entre cero:

`Float.NaN``Double.NaN`

Operadores de coma flotante

- ▶ Los operadores que actúan sobre valores de coma flotante son los siguientes:
- ▶ Los **operadores de comparación** dan como resultado un valor de tipo `boolean`:

Comparación numérica: `<, <=, >, >=`

Igualdad numérica: `==, !=`

```
jshell> 2.0 <= 3.0
$1 ==> true

jshell> 4.0 != 4.0
$2 ==> false
```

- Los **operadores numéricos** dan como resultado un valor de tipo `float` o `double`:

Signo más y menos (unarios):	<code>+</code> , <code>-</code>
------------------------------	---------------------------------

Multiplicativos:	<code>*</code> , <code>/</code> , <code>%</code>
------------------	--

Suma y resta:	<code>+</code> , <code>-</code>
---------------	---------------------------------

Preincremento y postincremento:	<code>++</code>
---------------------------------	-----------------

Predecremento y postdecremento:	<code>--</code>
---------------------------------	-----------------

- ▶ Si al menos uno de los operandos de un operador binario es de un número de coma flotante, la operación se realizará en coma flotante, aunque el otro operando sea un integral.
- ▶ Si al menos uno de los operandos de un operador numérico es de tipo `double`, la operación se llevará a cabo en aritmética de coma flotante de 64 bits y el resultado de la operación numérica será de tipo `double`.

Si el otro operando no es `double`, se convertirá primero a `double`.

- ▶ En caso contrario, la operación se llevará a cabo usando aritmética de coma flotante 32 bits y el resultado de la operación numérica será de tipo `float`.

Si el otro operando no es `float`, se convertirá primero a `float`.

```
jshell> 4 / 3
$1 ==> 1

jshell> 4 / 3.0
$2 ==> 1.3333333333333333

jshell> 4 / 3.0f
$3 ==> 1.3333334

jshell> 4 / 0.0
$4 ==> Infinity

jshell> 4.0 * Double.NaN
$5 ==> NaN
```

- ▶ Una operación de coma flotante que produce *overflow* devuelve un infinito con signo.
- ▶ Una operación de coma flotante que produce *underflow* devuelve un valor desnormalizado o un cero con signo.
- ▶ Una operación de coma flotante que no tiene un resultado matemáticamente definido devuelve NaN.
- ▶ Cualquier operación numérica que tenga un NaN como operando devuelve NaN como resultado.

Subtipado

- ▶ Se dice que un tipo S es **supertipo directo** de un tipo T cuando esos dos tipos están relacionados según unas reglas que veremos luego. En tal caso, se escribe:

$$S >_1 T$$

- ▶ Se dice que un tipo S es **supertipo** de un tipo T cuando S se puede obtener de T mediante clausura reflexiva y transitiva sobre la relación de *supertipo directo*. En tal caso, se escribe:

$$S :> T$$

- ▶ S es un **supertipo propio** de T si $S :> T$ y $S \neq T$. En tal caso, se escribe:

$$S > T$$

- ▶ Los **subtipos** de un tipo T son todos aquellos tipos S tales que T es un supertipo de S , **más el tipo nulo**. Cuando S es un subtipo de T se escribe:

$$S <: T$$

- ▶ S es un **subtipo propio** de T si $S <: T$ y $S \neq T$. En tal caso, se escribe:

$$S < T$$

- ▶ S es un **subtipo directo** de T si $T >_1 S$. En tal caso, se escribe:

$$S <_1 T$$

- ▶ Las relaciones de **subtipo** y **supertipo** son muy importantes porque:
 - Un valor de un tipo se puede convertir en un valor de un supertipo suyo sin perder información (es lo que se denomina **ampliación** o *widening*).
 - En cualquier expresión donde se necesite un valor de un cierto tipo, se puede usar un valor de un subtipo suyo.

Subtipado entre tipos primitivos

- ▶ Las siguientes reglas definen la relación de **subtipo directo** entre los tipos primitivos de Java:
 - `float` $<_1$ `double`
 - `long` $<_1$ `float`
 - `int` $<_1$ `long`
 - `char` $<_1$ `int`
 - `short` $<_1$ `int`
 - `byte` $<_1$ `short`

Conversiones entre datos primitivos

- ▶ Es posible convertir valores de un tipo a otro, siempre y cuando se cumplan ciertas condiciones y teniendo en cuenta que, en determinadas ocasiones, puede haber pérdida de información.
- ▶ Por ejemplo, no es posible convertir directamente valores numéricos en booleanos o viceversa.
- ▶ Pero sí es posible convertir valores numéricos a otro tipo numérico, aunque es posible que se pueda perder información, según sea el caso.
- ▶ Por ejemplo, convertir un número de coma flotante en un entero supondrá siempre la pérdida de la parte fraccionaria del número.
- ▶ Igualmente, es posible que haya pérdida de información al convertir un número de más bits en otro de menos bits.

Casting

- ▶ El **casting** o *moldeado* de tipos es una operación de conversión entre tipos.
- ▶ En el caso de tipos primitivos, el *casting* se usa para:
 - Convertir, en tiempo de ejecución, un valor de un tipo numérico a un valor similar de otro tipo numérico.
 - Garantizar, en tiempo de compilación, que el tipo de una expresión es `boolean`.
- ▶ El *casting* se escribe anteponiendo a una expresión, y entre paréntesis, el nombre del tipo al que se quiere convertir el valor de esa expresión.
- ▶ Por ejemplo, si queremos convertir a `short` el valor de la expresión `4 + 3`, hacemos:

```
(short) (4 + 3)
```

- ▶ Los paréntesis alrededor de la expresión `4 + 3` son necesarios para asegurarnos de que el *casting* afecta a toda la expresión y no sólo al `4`.

De ampliación (*widening*)

- ▶ Existen 19 conversiones de ampliación o *widening* sobre tipos primitivos:
 - De `byte` a `short`, `int`, `long`, `float` o `double`.
 - De `short` a `int`, `long`, `float` o `double`.
 - De `char` a `int`, `long`, `float` o `double`.
 - De `int` a `long`, `float` o `double`.
 - De `long` a `float` o `double`.
 - De `float` a `double`.

- ▶ Una conversión primitiva de ampliación nunca pierde información sobre la magnitud general de un valor numérico.
- ▶ Una conversión primitiva de ampliación de un tipo integral a otro tipo integral no pierde ninguna información en absoluto; el valor numérico se conserva exactamente.
- ▶ En determinados casos, una conversión primitiva de ampliación de `float` a `double` puede perder información sobre la magnitud general del valor convertido.
- ▶ Una conversión de ampliación de un valor `int` o `long` a `float`, o de un valor `long` a `double`, puede producir pérdida de precisión; es decir, el resultado puede perder algunos de los bits menos significativos del valor. En este caso, el valor de coma flotante resultante será una versión correctamente redondeada del valor entero.

- ▶ Una conversión de ampliación de un valor entero con signo a un tipo integral simplemente extiende el signo de la representación del complemento a dos del valor entero para llenar el formato más amplio.
- ▶ Una conversión de ampliación de `char` a un tipo integral rellena con ceros la representación del valor `char` para llenar el formato más amplio.
- ▶ A pesar de que puede producirse una pérdida de precisión, una conversión primitiva de ampliación nunca da como resultado una excepción en tiempo de ejecución.

De restricción (*narrowing*)

- Existen 22 conversiones de restricción o *narrowing* sobre tipos primitivos:
 - De `short` a `byte` o `char`
 - De `char` a `byte` o `short`
 - De `int` a `byte`, `short` o `char`
 - De `long` a `byte`, `short`, `char` o `int`
 - De `float` a `byte`, `short`, `char`, `int` o `long`
 - De `double` a `byte`, `short`, `char`, `int`, `long` o `float`

- ▶ Una conversión primitiva de restricción puede perder información sobre la magnitud general de un valor numérico y además también puede perder precisión y rango.
- ▶ Las conversiones primitivas de restricción de `double` a `float` se llevan a cabo mediante las reglas de redondeo del IEEE-754. Esta conversión puede perder precisión y también rango, por lo que puede resultar un `float` cero a partir de un `double` que no es cero, y un `float` infinito a partir de un `double` finito. Los `NaN` se convierten en `NaN` y los infinitos en infinitos.
- ▶ Una conversión de restricción de un entero con signo a un integral T simplemente descarta todos los bits excepto los n menos significativos, siendo n el número de bits usados para representar un valor de tipo T . Por tanto, además de poder perder información sobre la magnitud del valor numérico, también puede cambiar el signo del valor original.
- ▶ Una conversión de restricción de un `char` a un integral T se comporta igual que en el caso anterior.

- Las conversiones de restricción de un número en coma flotante a un integral T se realizan en dos pasos:
 1. El número en coma flotante se convierte a `long` (si T es `long`) o a `int` (si T es `byte`, `short`, `char` o `int`). Para ello:
 - Si el número flotante es `NaN`, el resultado del primer paso de la conversión es `0`.
 - Si el número flotante no es infinito, el valor se redondea a entero truncando a cero la parte fraccionaria.
 - Si T es `long` y ese entero cabe en un `long`, el resultado es `long`.
 - Si cabe en un `int`, el resultado es `int`.
 - Si es demasiado pequeño (o grande), el resultado es el valor más pequeño (o grande) que se pueda representar con `int` o `long`.
 2. Si T es `int` o `long`, el resultado final será el del primer paso.
Si T es `byte`, `char` o `short`, el resultado final será el resultado de convertir al tipo T el valor del primer paso.

- ▶ Al convertir un valor de `byte` a `char`, se produce una doble conversión:
 1. Primero, una conversión de ampliación de `byte` a `int`.
 2. Después, una conversión de restricción de `int` a `char`.

Promociones numéricas

- ▶ Es posible que se apliquen **promociones numéricas unarias** o **binarias** a los operandos de un operador aritmético.
- ▶ **Promociones numéricas unarias:**
 - Se llevan a cabo sobre expresiones en las siguientes situaciones:
 - El índice de un *array*.
 - El operando de un `+` o `-` unario.
 - El operando de un operador de complemento de bits `~`.
 - Cada operando, por separado, de los operadores `>>`, `>>>` y `<<`.
 - En tales casos, se lleva a cabo una promoción numérica que consiste en lo siguiente:
 - Si tipo del operando es `byte`, `short`, o `char`, su valor se promociona a `int` mediante una conversión primitiva de ampliación.

► **Promociones numéricas binarias:**

- Se llevan a cabo sobre los operandos de ciertos operadores:
 - Los operadores `*`, `/` y `%`.
 - Los operadores de suma y resta de tipos numéricos `+` y `-`.
 - Los operadores de comparación numérica `<`, `<=`, `>` y `>=`.
 - Los operadores de igualdad numérica `==` y `!=`.
 - Los operadores enteros a nivel de bits `&`, `^` y `|`.
 - En determinadas situaciones, el operador condicional `?:`.
- En tales casos, se lleva a cabo una promoción numérica que consiste en lo siguiente, en función del tipo de los operandos del operador:
 1. Si algún operando es `double`, el otro se convierte a `double`.
 2. Si no, si alguno es `float`, el otro se convierte a `float`.
 3. Si no, si alguno es `long`, el otro se convierte a `long`.
 4. Si no, ambos operandos se convierten a `int`.

1.3. Tipos referencia

1.3.1 Nulo

Tipos referencia

- ▶ Los **tipos referencia** son:
 - Tipos **clase**.
 - Tipos **interfaz**.
 - Tipos **array**.
- ▶ Un tipo clase o interfaz consiste en un identificador (o una secuencia de identificadores separados por `.`)
- ▶ Cada identificador de un tipo clase o interfaz puede ser el nombre de un paquete o el nombre de un tipo.
- ▶ Opcionalmente puede llevar *argumentos de tipo*. Si un argumento de tipo aparece en alguna parte de un tipo clase o interfaz, se denomina *tipo parametrizado*.

- ▶ Un objeto es una instancia de una clase o un *array*.
- ▶ Las referencias son punteros a esos objetos.
- ▶ Existe una referencia especial llamada *referencia nula* (o `null`) que no apunta a ningún objeto.

Nulo

- ▶ Existe un tipo especial llamado **tipo nulo**.
- ▶ El tipo nulo es el tipo de la expresión `null`, la cual representa la **referencia nula**.
- ▶ La referencia nula es el único valor posible de una expresión de tipo nulo.
- ▶ El tipo nulo no tiene nombre y, por tanto, no se puede declarar una variable de tipo nulo o convertir un valor al tipo nulo.
- ▶ La referencia nula siempre puede asignarse o convertirse a cualquier tipo referencia.
- ▶ En la práctica, el programador puede ignorar el tipo nulo y suponer que `null` es simplemente un literal especial que pertenece a cualquier tipo referencia.

2. Variables en Java

2.1 Introducción

2.2 Variables de tipos primitivos

2.3 Variables de tipos referencia

2.4 Declaración de variables

2.1. Introducción

Introducción

- ▶ Una **variable** es un lugar donde se almacena un valor.
- ▶ Cada variable tiene un **tipo** asociado, denominado **tipo en tiempo de compilación** o **tipo estático**.
- ▶ Ese tipo puede ser un **tipo primitivo** o un **tipo referencia**.
- ▶ Las variables se deben **declarar** antes de usarlas, y en la declaración se indica su tipo.
- ▶ El valor de una variable se puede cambiar mediante **asignación** o usando los operadores de pre y post incremento (**++**) o decremento (**--**).
- ▶ El diseño del lenguaje Java garantiza que el valor de una variable es compatible con el tipo de la variable.

2.2. Variables de tipos primitivos

Variables de tipos primitivos

- ▶ Una variable de un tipo primitivo siempre contiene un valor primitivo de exactamente ese tipo primitivo.
- ▶ Las variables de tipos primitivos contienen al valor primitivo, es decir, que almacenan ellas mismas el valor primitivo.
- ▶ Por tanto, los valores primitivos no se almacenan en el montículo, sino directamente en la propia variable.

2.3. Variables de tipos referencia

Variables de tipos referencia

- ▶ Una variable de un tipo referencia T puede contener:
 - La referencia nula (`null`).
 - Una referencia a una instancia de S , siendo S un subtipo de T .
- ▶ Una variable de un tipo «array de T », puede contener:
 - Si T es un tipo primitivo:
 - La referencia nula (`null`).
 - Una referencia a un array de tipo «array de T ».
 - Si T es un tipo referencia:
 - La referencia nula (`null`).
 - Una referencia a un array de tipo «array de S », siendo S un subtipo de T .

- ▶ En Java, `Object` es supertipo de cualquier tipo referencia.
- ▶ Por tanto, una variable de tipo `Object` puede contener una referencia a cualquier valor referencia de cualquier tipo referencia.

2.4. Declaración de variables

2.4.1 Inicialización y asignación de variables

Declaración de variables

- ▶ La forma más común de declarar variables es mediante la **sentencia de declaración de variables**.
- ▶ La **sentencia de declaración de variables** es una sentencia mediante la cual anunciamos al compilador la existencia de unas determinadas variables e indicamos el tipo que van a tener esas variables.
- ▶ La sintaxis de la declaración de variables es:

```
<decl_vars> ::= <tipo> identificador(, identificador)* ;
```

- ▶ Todas las variables que aparecen en la declaración tendrán el mismo tipo (el tipo indicado en la declaración).

- Por ejemplo:

```
int x, y; // Declara las variables «x» e «y» de tipo «int»  
float z; // Declara la variable «z» de tipo «float»
```

- En Java, los identificadores son sensibles a mayúsculas y minúsculas. Por tanto, la variable `x` no es la misma que `X`.
- La declaración de una variable siempre debe aparecer antes de su uso.
- La variable empieza a existir en el momento en el que se declara.
- Se almacena en el marco del ámbito actual, donde también se crea la ligadura entre el identificador y la propia variable.
- El ámbito de la variable es el bloque (porción de código encerrado entre `{` y `}`) dentro del cual se ha declarado la variable, ya que cada bloque introduce un nuevo ámbito.
- Como los bloques se pueden anidar unos dentro de otros, sus ámbitos correspondientes también estarán anidados.

```
public static void main(String[] args) {  
    int x;  
    System.out.println("Bloque externo");  
    {  
        int y;  
        System.out.println("Bloque interno");  
    }  
}
```

- El ámbito de `x` es el bloque más externo (el que define el cuerpo del método `main`).
- El ámbito de `y` es el bloque más interno.
- Por tanto, `x` se puede usar dentro del bloque más interno, pero `y` no se puede usar fuera del bloque más interno.

```
public static void main(String[] args) { // Empieza el cuerpo del método  
    int x;  
    System.out.println("Bloque externo");  
    { // Empieza el bloque más interno  
        int y;  
        System.out.println("Bloque interno");  
    } // Termina el bloque más interno  
} // Termina el cuerpo del método
```

- ▶ Las variables declaradas dentro de un método son **locales al método**, independientemente del nivel de anidamiento del bloque donde se haya declarado la variable.

Por tanto, tanto `x` como `y` son **variables locales al método** `main`.

- ▶ Los **parámetros** de un método también se consideran variables locales al método.
- ▶ Una vez recién creadas, las variables locales a un método no tienen un valor inicial.
- ▶ Por tanto, cuando se declara una variable local, ésta permanece sin ningún valor hasta que se le asigna uno.
- ▶ Una variable sin valor se denomina **variable no inicializada**.
- ▶ Si se intenta usar una variable local no inicializada, provoca un error:

```
public static void main(String[] args) {  
    int x;    // La variable «x» es local al método «main»  
    System.out.println(x); // Da error porque «x» no está inicializada  
}
```

Inicialización y asignación de variables

- Para darle un valor a una variable, podemos:
 - **Inicializar la variable** en el momento de la declaración, de la siguiente forma:

```
int x = 25; // Declara la variable entera «x» y la inicializa con 25
```

- Asignarle un valor después de haberla declarado, usando la **sentencia de asignación**:

```
int x;      // Declara la variable entera «x»  
x = 25;     // Le asigna el valor 25 a la variable «x»
```

- La asignación puede ser **simple** o **compuesta**:

```
int x;      // Declaración de la variable  
x = 25;     // Asignación simple  
x = x + 5;  // Asignación simple  
x += 5;     // Asignación compuesta, equivalente a la anterior
```

► Ejemplo:

```
public static void main(String[] args) {  
    int x = 25;  
    System.out.println("Bloque externo");  
    System.out.println(x);  
    {  
        int y = 14;  
        System.out.println("Bloque interno");  
        System.out.println(x);  
        System.out.println(y);  
        x += 1;  
        System.out.println(x);  
    }  
}
```

- También se dispone de los operadores de pre y post incremento y decremento:

```
int x = 5, y = 4;  
y = x++;
```

3. Estructuras de control

3.1 Bloques

3.2 `if`

3.3 `switch`

3.4 `while`

3.5 `for`

3.6 `do ... while`

3.7 `break` y `continue`

3.1. Bloques

Bloques

- ▶ Un **bloque** es una secuencia de una o más sentencias encerradas entre llaves { y }.
- ▶ Java es un lenguaje **estructurado en bloques**, lo que significa que los bloques pueden anidarse unos dentro de otros y cada bloque define un ámbito, que iría anidado dentro del ámbito del bloque que lo contiene.
- ▶ Los bloques también puede incluir declaraciones, que serán locales al bloque.
- ▶ En cualquier parte del programa donde se pueda poner una sentencia, se podrá poner un bloque, que actuará como una sentencia compuesta.
- ▶ El cuerpo de un método siempre es un bloque.
- ▶ Todas las sentencias simples deben acabar en punto y coma (;) pero los bloques no.

3.2. if

if

- ▶ La palabra clave **if** introduce una sentencia condicional o estructura alternativa (simple o doble).
- ▶ Su sintaxis es:

```
⟨sentencia_if⟩ ::= if (⟨condición⟩) ⟨sentencia_si_verdadero⟩ [else ⟨sentencia_si_falso⟩]
```

- ▶ La ejecución de la ⟨sentencia_if⟩ implica evaluar la ⟨condición⟩. Si evalúa a **true**, se ejecutará la ⟨sentencia_si_verdadero⟩. En caso contrario, se ejecutará la ⟨sentencia_si_falso⟩, si es que existe.
- ▶ Aunque las ⟨sentencia_si_verdadero⟩ y ⟨sentencia_si_falso⟩ pueden ser sentencias simples, se aconseja (por regla de estilo) usar siempre bloques.
- ▶ Recordar que los bloques no hay que acabarlos en ;.

3.3. switch

switch

- ▶ La palabra clave **switch** introduce una estructura alternativa múltiple.
- ▶ Su sintaxis es:

```
⟨sentencia_switch⟩ ::=  
    switch (⟨expresión⟩) {  
        (case ⟨expresión_case⟩:  
            ⟨sentencia_case⟩*)+  
        [default:  
            ⟨sentencia_default⟩]  
    }
```

- ▶ Se evalúa la *⟨expresión⟩* y se compara con las distintas *⟨expresión_case⟩*, de una en una y de arriba abajo.
- ▶ Cuando se encuentra una que sea igual, se ejecuta su *⟨sentencia_case⟩* correspondiente y se sigue comparando con las siguientes *⟨expresión_case⟩* (salvo que haya un **break**).
- ▶ Si no hay ninguna *⟨expresión_case⟩* que coincida, y existe la cláusula **default**, se ejecuta la *⟨sentencia_default⟩*.

► Ejemplo:

```
switch (k) {  
    case 1: System.out.println("uno");  
           break; // sale del switch  
    case 2: System.out.println("dos");  
           break; // sale del switch  
    case 3: System.out.println("muchos");  
           break; // no hace falta, pero es conveniente  
}
```

3.4. while

while

- ▶ La palabra clave **while** introduce una sentencia o estructura repetitiva.
- ▶ Su sintaxis es:

```
⟨sentencia_while⟩ ::= while (⟨condición⟩) ⟨sentencia⟩
```

- ▶ Se evalúa la ⟨condición⟩ y, si evalúa a **true**, se ejecuta la ⟨sentencia⟩ y se vuelve de nuevo a evaluar la ⟨condición⟩ hasta que evalúa a **false**.
- ▶ Si la ⟨condición⟩ evalúa a **false** desde el principio, la ⟨sentencia⟩ no se ejecuta ninguna vez.
- ▶ La sentencia **while** implementa un **bucle**, y cada ejecución de la ⟨sentencia⟩ representa una **iteración** del bucle.
- ▶ Aunque la ⟨sentencia⟩ puede ser simple, se aconseja (por regla de estilo) usar siempre un bloque.

► Ejemplo:

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

Imprime:

0
1
2
3
4

3.5. for

for

- ▶ La palabra clave **for** introduce un variante del bucle **while** donde los elementos de control del bucle aparecen todos en la misma línea al principio de la estructura.
- ▶ Su sintaxis es:

```
⟨sentencia_for⟩ ::= for ( [⟨inicialización⟩]; [⟨condición⟩]; [⟨actualización⟩] ) ⟨sentencia⟩
```

- ▶ Equivale a hacer:

```
⟨inicialización⟩  
  while (⟨condición⟩) {  
    ⟨sentencia⟩  
    ⟨actualización⟩  
  }
```

- ▶ La **⟨inicialización⟩** es una sentencia que puede ser también una declaración. En tal caso, esa declaración tendrá un ámbito local a la sentencia **for** y no existirá fuera de ella.

- ▶ La *⟨inicialización⟩*, la *⟨condición⟩* y la *⟨actualización⟩* son todas opcionales.
- ▶ Aunque la *⟨sentencia⟩* puede ser simple, se aconseja (por regla de estilo) usar siempre un bloque.
- ▶ Ejemplo:

El siguiente bucle **while**:

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}
```

se puede escribir como un bucle **for**:

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

3.6. do ... while

do ... while

- ▶ La palabra clave **do** introduce un tipo especial de bucle donde la condición de continuación se comprueba al final, y no al principio como es habitual.
- ▶ Su sintaxis es:

```
⟨sentencia_do_while⟩ ::= do ⟨sentencia⟩ while (⟨condición⟩);
```

- ▶ Se ejecuta la *⟨sentencia⟩* y, a continuación, se comprueba la *⟨condición⟩*. Si evalúa a **true**, se vuelve a ejecutar la *⟨sentencia⟩* y a evaluar la *⟨condición⟩*, y así sucesivamente hasta que la *⟨condición⟩* evalúa a **false**.
- ▶ Se garantiza que la *⟨sentencia⟩* se ejecutará, al menos, una vez.
- ▶ Aunque la *⟨sentencia⟩* puede ser simple, se aconseja (por regla de estilo) usar siempre un bloque.

► Ejemplo:

```
do {  
    System.out.println(i);  
    i++;  
} while (i < 5);
```

Independientemente de lo que valga `i` al empezar a ejecutar el `do`, el `println` se va a ejecutar, al menos, una vez.

3.7. break y continue

break y continue

- ▶ Por tanto, la sentencia **break** sólo puede aparecer dentro de un **switch**, **while**, **do** o **for**.
- ▶ La sentencia **break** produce un salto incondicional que hace que el control salga de la sentencia **switch**, **while**, **do** o **for** más interna en la que se encuentra el **break**.
- ▶ La sentencia **continue** transfiere el control a la siguiente iteración del bucle actual más interno. Por tanto, sólo puede aparecer dentro de un **while**, **for** o **do**.

4. Entrada/salida

4.1 Flujos `System.in`, `System.out` y `System.err`

4.2 `java.util.Scanner`

4.1. Flujos `System.in`, `System.out` y `System.err`

4.2. java.util.Scanner