Práctica: Variables, tipos y operadores

3

The subtlest bugs cause the greatest damage and problems.

- *Corollary - A subtle bug will modify storage thereby masquerading as some other problem.*

– Murphy's Laws of Computer Programming #3

Meta

Que el alumno aprenda a utilizar los tipos básicos de Java y sus operadores, y que entienda el concepto de referencia.

Objetivos

Al finalizar la práctica el alumno será capaz de:

- utilizar la clase Consola para hacer salida en pantalla;
- manejar variables locales, tipos básicos y literales de Java;
- entender lo que son referencias y
- manejar operadores y expresiones.

Desarrollo

Un programa (en Java o cualquier lenguaje de programación) está escrito para resolver algún problema, realizando una o más tareas.

En el caso particular de Java, se utiliza la orientación a objetos para resolver estos problemas. Utilizar la orientación a objetos quiere decir, entre otras cosas, abstraer el mundo en objetos y a estos objetos agruparlos en clases. En la práctica anterior, por ejemplo, utilizamos las clases ClaseReloj y VistaRelogAnalogico, que abstraían el funcionamiento de un reloj y su representación externa al implementar las interfaces Reloj y VistaReloj.

Vamos a comenzar a resolver problemas utilizando orientación a objetos. Esto quedará dividido en dos partes fundamentales: la primera será identificar los objetos que representan nuestro problema, agruparlos en clases y definir precisamente el comportamiento (las funciones) que tendrán para que resuelvan nuestro problema.

La segunda parte es un poco más detallada. Una vez definido el comportamiento de una clase, hay que sentarse a escribirlo. Esto se traduce en implementar los métodos de la clase. Para implementarlos, es necesario manejar las partes básicas o atómicas del lenguaje: las variables, sus tipos, y sus operadores.

En esta práctica jugaremos con esta parte fundamental del lenguaje Java; en la siguiente utilizaremos lo que aprendamos aquí para comenzar a escribir nuestras clases.

Una clase de prueba

Para jugar con variables, vamos a necesitar un bloque de ejecución donde ponerlas, o sea un método, y ya que necesitamos un método, vamos a necesitar también una clase donde ponerlo.

Un problema es que no sabemos hacer todavía métodos, así que utilizaremos el método main, que ya utilizamos en las prácticas pasadas. Escribamos pues una clase llamada Prueba, y utilicemos el método main de ella.

Nuestra clase Prueba quedaría como sigue:

```
1  package icc1.practica3;
2
3  public class Prueba {
4
5      public static void main (String[] args) {
6
7      }
8  }
```

Aún no hemos visto paquetes, así que por ahora no explicaremos qué significa la línea

```
1  package icc1.practica3;
```

Por ahora sólo necesitamos saber que eso significa que el archivo Prueba.java debe estar en un directorio llamado practica3, y que éste a su vez debe estar en un directorio llamado icc1.

Actividad 3.1 Baja el archivo practica3.tar.gz, y descomprímelo como lo hemos hecho en prácticas pasadas.

El archivo contiene ya un build.xml para esta práctica, y la estructura de directorios necesaria (o sea src/icc1/practica3). Dentro de ese directorio crea tu archivo Prueba.java para tu clase Prueba.

Si estás usando XEmacs (y éste está bien configurado), y pones esto:
/ -*- jde -*- */*
en la primera línea, cada vez que abras el archivo, XEmacs hará disponibles varias funcionalidades especiales para editar archivos de Java. Entre otras cosas, coloreará de forma especial la sintaxis del programa, y te permitirá llamar a Ant con `C-c C-v C-b`.

Como recordarás de la práctica anterior, utilizamos el directorio src para poner nuestro código fuente (y así también lo especifiacmos en el build.xml). Y como nuestro paquete es icc1.practica3, dentro de src creamos el directorio icc1, y dentro de éste otro llamado practica3.

Con el build.xml proporcionado podemos compilar y ejecutar el programa como veníamos haciéndolo hasta ahora. Podemos comprobar (viendo el archivo build.xml) que ahora la clase que ejecutamos en el objetivo run es icc1.practica3.Prueba. Esto es "la clase Prueba del paquete icc1.practica3".

Varios deben estar ahora haciéndose algunas preguntas: ¿Qué quiere decir exactamente **public class** Prueba? ¿Para qué es la palabra **static**? ¿De dónde salió el constructor de Prueba?

Todas esas preguntas las vamos a contestar en la siguiente práctica; en ésta nos vamos a concentrar en variables, tipos y operadores.

La clase Prueba se ve un poco vacía; sin embargo, es una clase completamente funcional y puede ser compilada y ejecutada.

Actividad 3.2 Compila y ejecuta la clase (desde el directorio donde está el archivo build.xml):

```
# ant compile
# ant run
```

No va a pasar nada interesante (de hecho no va a pasar nada), pero con esto puedes comprobar que es una clase válida, pues compila bien y no termina con un mensaje de rror al ejecutarse.

Ahora, queremos jugar con variables, pero no va a servir de mucho si no vemos los resultados de lo que estamos haciendo. Para poder ver los resultados de lo que hagamos, vamos a utilizar a la clase Consola.

La clase *Consola*

Para tener acceso a la pantalla y ver los resultados de lo que le hagamos a nuestras variables, vamos a utilizar la clase Consola. Esta clase nos permitirá, entre otras cosas, escribir en la pantalla.

Actividad 3.3 De la página referida al final de las prácticas, baja el archivo icc1.jar, y ponlo en el directorio de la práctica 3.

¿Cómo vamos a utilizar la clase Consola dentro de nuestra clase Prueba? Primero, debemos decirle a nuestra clase acerca de la clase Consola. Esto lo logramos escribiendo

```
import icc1.interfaz.Consola;
```

antes de la declaración de nuestra clase. Veremos más detalladamente el significado de **import** cuando veamos paquetes.

En segundo lugar, cuando compilemos también tenemos que decirle al compilador dónde buscar a la clase Consola. Para poder hacer esto, necesitamos modificar nuestro build.xml.

Actividad 3.4 Modifica el archivo build.xml cambiando la línea:

```
6  <javac srcdir="src" destdir="build" debug="true"
7         debuglevel="source" />
```

por

```
6  <javac srcdir="src" destdir="build" debug="true"
7         debuglevel="source" classpath="icc1.jar" />
```

El *classpath* es donde Java (el compilador y la máquina virtual) buscan clases extras.

En último lugar, cuando ejecutemos de nuevo tenemos que decirle a la JVM dónde está nuestra clase Consola; para esto también modificamos nuestro build.xml.

Actividad 3.5 Modifica el archivo build.xml cambiando el objetivo run de la siguiente manera:

```
10 <target name="run" depends="compile">
11   <java classname="icc1.practica3.Prueba" fork="true">
12     <classpath>
13       <pathelement path="build" />
14       <pathelement location="icc1.jar"/>
15     </classpath>
16   </java>
17 </target>
```

De la misma manera, estamos añadiendo el archivo icc1.jar al *classpath* para que la JVM lo encuentre al ejecutar el programa.

¿Qué podemos hacer con la clase Consola? De hecho, podemos hacer muchas cosas, que iremos descubriendo poco a poco conforme avancemos en el curso; pero por ahora, nos conformaremos con utilizarla para mostrar los valores de nuestras variables.

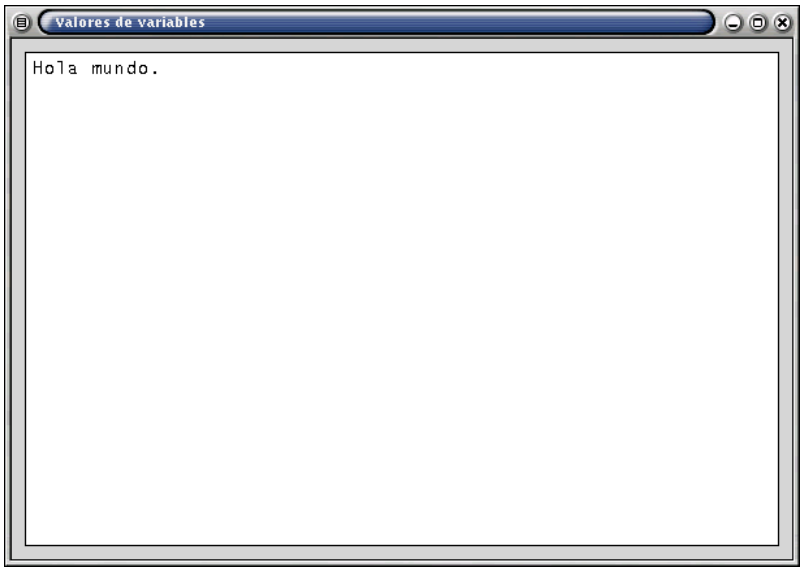
Actividad 3.6 Con la ayuda de un navegador, ve la documentación generada de la clase Consola. Se irán explicando y utilizando todos los métodos de la clase conforme se avance en el material.

Para mostrar información con la consola, necesitamos crear un objeto de la clase consola y con él llamar a los métodos imprime o imprimeln. Por ejemplo, modifica el método main de la clase Prueba de la siguiente manera:[1]

```
1  package icc1.practica3;
2
3  import icc1.interfaz.Consola;
4
5  public class Prueba {
6
7      public static void main (String[] args) {
8          Consola c;
9          c = new Consola ("Valores de variables");
10         c.imprime ("Hola mundo.");
11     }
12 }
```

Compila y ejecuta la clase. El resultado lo puedes ver en la figura 3.1.

Figura 3.1 Consola



Para terminar el programa, cierra la ventana de la consola.
Analicemos las tres líneas que añadimos a la función main:

```
8  Consola c;
```

[1] Observa que estamos utilizando ␣ para representar los espacios dentro de las cadenas.

En esta línea sólo declaramos un nuevo objeto de la clase Consola llamado c.

```
9   c = new Consola ("Valores de variables");
```

En esta línea construimos un objeto de la clase Consola y lo asociamos a c. El constructor que usamos recibe como parámetro una cadena, que es el título de la ventana de nuestra consola. Al construir el objeto, la ventana con nuestra consola se abre automáticamente.

```
10  c.imprime ("Hola mundo.");
```

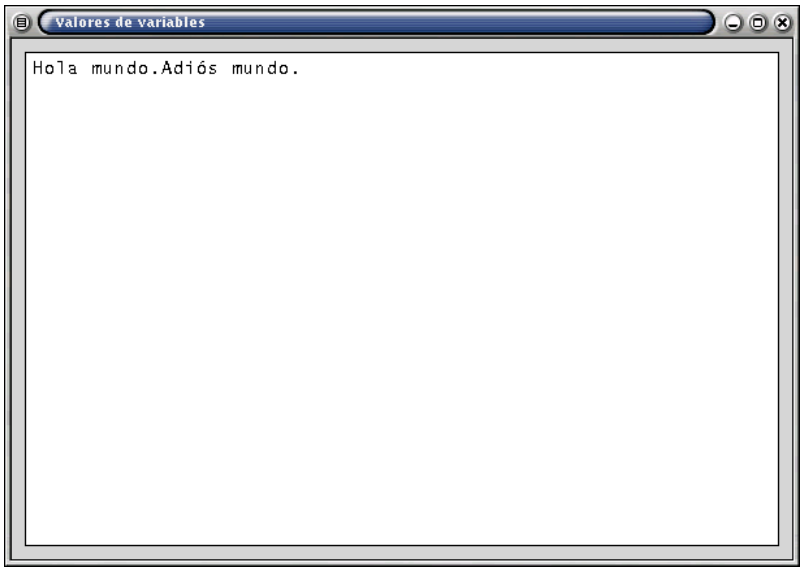
El método imprime de la clase Consola hace lo que su nombre indica; imprime en la consola la cadena que recibe como parámetro, que en nuestro ejemplo es `"Hola mundo."`.

Dijimos que podíamos utilizar los métodos imprime e imprimeln para mostrar información en la consola. Para ver la diferencia, imprime otra cadena en la consola:

```
9   Consola c;
10  c = new Consola ("Valores de variables");
11  c.imprime ("Hola mundo.");
12  c.imprime ("Adiós mundo.");
```

Compila y ejecuta la clase. El resultado lo puedes ver en la figura 3.2.

Figura 3.2 Resultado de imprime en la consola

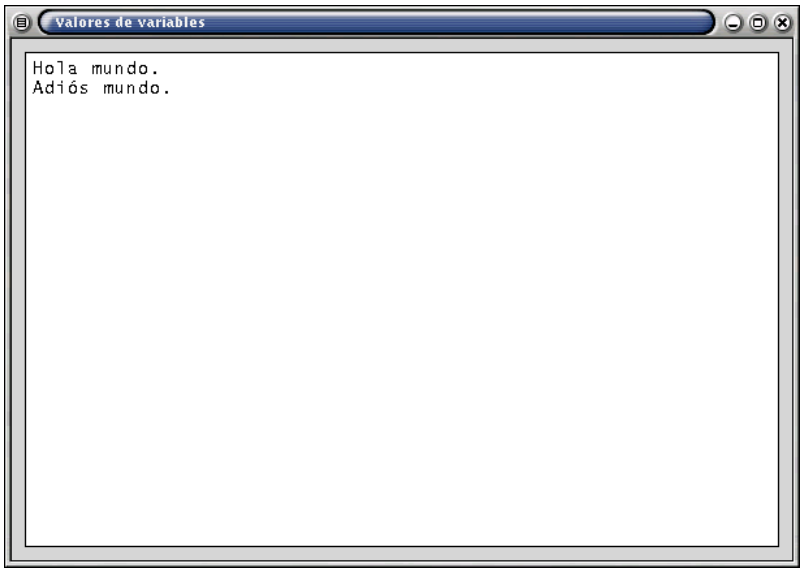


Ahora, utilicemos el método imprimeln en lugar de imprime:

```
13  c.imprimeln ("Hola mundo.");
14  c.imprimeln ("Adiós mundo.");
```

Si compilas y ejecutas la clase, verás algo parecido a la figura 3.3.

Figura 3.3 Resultado de *imprimeln* en la consola



La diferencia entre imprime e imprimeln es que la segunda imprime un salto de línea después de haber impreso la cadena que le pasamos como parámetro. El salto de línea en Java está representado por el carácter `'\n'`. Otra forma de conseguir el mismo resultado es hacer

```
13  c.imprime ("Hola mundo.\n");
14  c.imprime ("Adiós mundo.\n");
```

o incluso

```
13  c.imprime ("Hola mundo.\nAdiós mundo.\n");
```

La clase Consola ofrece muchos métodos, que iremos utilizando a lo largo de las demás prácticas. En ésta sólo veremos imprime e imprimeln.

Actividad 3.7 Crea la clase Prueba en el archivo Prueba.java y realiza (valga la redundancia) pruebas con imprime e imprimeln.

Variables locales, tipos básicos y literales

Ya hemos trabajado con variables locales. En la práctica pasada, reloj y vista eran variables locales del método main en la clase UsoReloj. En esta práctica, c es un variable local de la función main en la clase Prueba.

Las variables locales se llaman así porque sólo son visibles *localmente*. Cuando escribamos funciones, no van a poder ver (y por lo tanto no podrán usar) a c porque esta variable sólo puede ser vista dentro de la función main. Se conoce como el *alcance* de una variable a los lugares dentro de un programa desde donde puede ser vista la variable.

Recordemos la declaración de reloj y vista de la práctica anterior:

```
Reloj reloj;
...
VistaReloj vista;
```

Las dos son variables locales, pero tienen una diferencia muy importante: son de *tipos diferentes*. Java es un lenguaje de programación *fuertemente tipificado*, lo que significa que una variable tiene un único tipo durante toda su vida, y que por lo tanto nunca puede cambiar de tipo.

Esto quiere decir que lo siguiente sería inválido:

```
vista = new ClaseReloj ();
```

Si intentan compilar algo así, javac se va a negar a compilar la clase; les va a decir que vista no es de *tipo* ClaseReloj.

Vamos a empezar a declarar variables. Para declarar una variable, tenemos que poner el *tipo* de la variable, seguido del nombre de la variable:

```
int a;
```

Si queremos usar varias variables de un solo tipo, podemos declararlas todas juntas:

```
int a, b, c, d;
```

Con eso declaramos una variable de cierto tipo, pero, ¿qué son los tipos? Para empezar a ver los tipos de Java, declaremos un entero llamado a en el método main.

```
7       public static void main (String[] args) {
8           Consola c;
9           c = new Consola ("Valores de variables");
10
11          // Declaramos un entero "a";
12          int a;
13      }
```

La variable a es de tipo *entero* (**int**). Java tiene algunos tipos especiales que se llaman *tipos básicos*. Los tipos básicos difieren de todos los otros tipos en que ***NO*** son objetos; no tienen métodos ni variables de clase, y no tienen constructores. Para inicializar (que no es igual que construir) una variable de tipo básico, sencillamente se le asigna el valor correspondiente, por ejemplo:

```
a = 5;
```

Java tiene ocho tipos básicos, que puedes consultar en la tabla 3.1.

Tabla 3.1 Tipos básicos de Java

Nombre	Descripción
byte	Enteros con signo de 8 bits en complemento a dos
short	Enteros con signo de 16 bits en complemento a dos
int	Enteros con signo de 32 bits en complemento a dos
long	Enteros con signo de 64 bits en complemento a dos
float	Punto flotante de 32 bits de precisión simple
double	Punto flotante de 64 bits de precisión doble
char	Carácter de 16 bits
boolean	Valor booleano (verdadero o falso)

En el ejemplo de arriba, ¿qué representa el 5? El 5 representa una *literal*, en este caso de tipo entero. Las literales son valores que no están "atados" a una variable. Todos los tipos básicos tienen literales:

```
int     entero;
double  doble;
char    caracter;
boolean booleano;
```

Continúa en la siguiente página

Continúa de la página anterior

```
entero   = 1234;
doble    = 13.745;
caracter = 'a';
booleano = true;
```

En general, a cualquier serie de números sin punto decimal Java lo considera una literal de tipo **int**, y a cualquier serie de números con punto decimal Java lo considera una literal de tipo **double**. Si se quiere una literal de tipo **long** se tiene que poner una `L` al final, como en `1234L`, y si se quiere un flotante hay que poner una `F` al final, como en `13.745F`. No hay literales explícitas para **byte** ni para **short**.

Las únicas literales de tipo **boolean** son **true** y **false**. Las literales de carácter son `'a'`, `'Z'`, `'+'`, etc. Para los carácteres como el salto de línea o el tabulador, las literales son especiales, como `'\n'` o `'\t'`. Si se necesita la literal del carácter \, se utiliza `'\\'`. Si se necesita un carácter internacional, se puede utilizar su código Unicode[2], por ejemplo, `'\u0041'` representa el carácter `'A'` (el entero que sigue a `'\u'` está escrito en hexadecimal, y deben escribirse los cuatro dígitos).
La fuerte tipificación de Java se aplica también a sus tipos básicos:

```
int a = 13;   // Código válido
int b = true; // ¡Código inválido!
```

Sin embargo, el siguiente es código válido también:

```
float x = 12;
byte  b = 5;
int   a = b;
```

Si somos precisos, la literal `12` es de tipo **int** (como dijimos arriba); y sin embargo se lo asignamos a una variable de tipo **float**. De igual forma, `b` es de tipo **byte**, pero le asignamos su valor a `a`, una variable de tipo **int**. Dejaremos la explicación formal de por qué esto no viola la fuerte tipificación hasta que veamos *conversión explícita de datos*; por ahora, basta con entender que un entero con signo que cabe en 8 bits (los de tipo **byte**), no tiene ninguna dificultad hacer que quepa en un entero con signo de 32 bits, y que todo entero con signo que quepa en 32 bits, puede ser representado por un punto flotante de 32 bits.

Por la misma razón, el siguiente código sí es inválido:

[2]Unicode es un código de carácteres pensado para reemplazar a ASCII. ASCII sólo tenía (originalmente) 127 caracteres y después fue extendido a 255. Eso basta para el alfabeto latino, con acentos incluidos, y algunos carácteres griegos; pero es completamente inútil para lenguas como la china o japonesa. Unicode tiene capacidad para 65,535 caracteres, lo que es suficiente para manejar estos lenguajes. Java es el primer lenguaje de programación en soportar nativamente Unicode.

```
int  a = 1.2;
int  b = 5;
byte c = b;
```

El primero debe ser obvio; no podemos hacer que un punto flotante sea representado por un entero. Sin embargo, ¿por qué no podemos asignarle a c el valor b, cuando 5 sí cabe en un entero de 8 bits con signo? No podemos porque al compilador de Java *no le interesa* qué valor en particular tenga la variable b. Lo único que le importa es que es de tipo **int**, y un **int** *es muy probable* que no quepa en un **byte**. El compilador de Java juega siempre a la segura.

Actividad 3.8 Prueba los pequeños ejemplos que se han visto, tratando de compilar todos (utiliza el método main para tus ejemplos). Ve qué errores manda el compilador en cada caso, si es que manda.

¿Cómo vamos a imprimir en nuestra consola nuestros tipos básicos? Pues igual que nuestras cadenas:

```
int a = 5;
c.imprimeln(a);
```

Los métodos imprime e imprimeln de la clase Consola soportan ***todos*** los tipos de Java.

Una última observación respecto a variables locales (sean éstas de tipo básico u objetos). Está claro que para usar una variable local en un método, tiene que estar declarada *antes de ser usada*. Esto es obvio porque no podemos usar algo antes de tenerlo. Empero, hay otra restricción: no podemos usar una variable antes de *inicializarla*; el siguiente código no compila:

```
int a;
int b = a; // No podemos usar la variable a todavía...
```

Estamos tratando de usar a sin que haya sido inicializada. Esto es inválido para el compilador de Java.

Dijimos que esto se aplica a tipos básicos y objetos, y más arriba dijimos que inicializar no era lo mismo que construir. Esto es porque también podemos inicializar variables que sean objetos sin necesariamente construir nada. Para entender esto, necesitamos comprender qué son las *referencias*.

Referencias

Cuando hacemos

```
Consola c;
```

dijimos que estamos declarando un objeto de la clase Consola. Así se acostumbra decir, y así lo diremos siempre a lo largo de estas prácticas. Sin embargo, formalmente hablando, estamos declarando una variable de tipo *referencia* a un objeto de la clase Consola.

Una referencia es una dirección en memoria. Al construir en c con **new**

```
c = new Consola ();
```

lo que hace **new** es pedirle a la JVM que asigne memoria para poder guardar en ella al nuevo objeto de la clase Consola. Entonces **new** regresa la dirección en memoria del nuevo objeto, y se guarda en la variable c.

Esto es importante; las variables de referencia a algún objeto tienen valores que son direcciones de memoria, así como las variables de tipo entero tienen valores que son enteros en complemento a dos, o las variables de tipo booleano tienen valores que son **true** o **false**.

Si después de construir p hacemos

```
Consola c2;
c2 = c;
```

entonces ya no estamos *construyendo* a c2 (no se está asignando memoria a ningún nuevo objeto). Lo que estamos haciendo es *inicializar* a c2 con el valor de c, o sea, con la dirección de memoria a la que hace referencia c. Después de eso, c y c2 son dos variables *distintas*, que tienen *el mismo* valor, o sea, que hacen referencia al mismo objeto[3].

Todas las referencias (no importa de qué clase sean) tienen una única literal, que es **null**. Ésta es una referencia que por definición es inválida; no apunta nunca a ninguna dirección válida de la memoria. La referencia **null** es muy utilizada para varios propósitos, y en particular sirve para inicializar variables de objetos que aún no deseamos construir. Veremos más ejemplos con **null** en las siguientes prácticas.

Una vez definidas las referencias, podemos dividir a las variables de Java en dos; toda variable de Java es de un tipo básico, o alguna referencia. No hay otras. Además de esto, todas las referencias apuntan a algún objeto, o tienen el valor **null**.

[3] Se suele decir también que *apuntan* al mismo objeto.

Operadores

Sabemos ya declarar todas las posibles variables de Java. Una vez que tengamos variables, podemos guardar valores en ellas con el operador de asignación =; pero además debemos poder hacer cosas interesantes con los valores que puede tomar una variable.

Para hacer cosas interesantes con nuestras variables, es necesario utilizar operadores. Los operadores de Java aparecen en la tabla 3.2.

Tabla 3.2 Operadores de Java.

Operandos	**Asociatividad**	**Símbolo**	**Descripción**
posfijo unario	derecha	[ ]	arreglos
posfijo unario	derecha	.	selector de clase
posfijo n-ario	derecha	(<parámetros>)	lista de parámetros
posfijo unario	izquierda	<variable>++	auto post-incremento
posfijo unario	izquierda	<variable>−−	auto post-decremento
unario prefijo	derecha	++<variable>	auto pre-incremento
unario prefijo	derecha	−−<variable>	auto pre-decremento
unario prefijo	izquierda	+<expresión>	signo positivo
unario prefijo	izquierda	−<expresión>	signo negativo
unario prefijo	izquierda	~<expresión>	complemento en bits
unario prefijo	izquierda	!<expresión>	negación
unario prefijo	izquierda	**new** <constructor>	constructor
unario prefijo	izquierda	(<tipo>)<expresión>	*casting*
binario infijo	izquierda	*	multiplicación
binario infijo	izquierda	/	división
binario infijo	izquierda	%	módulo
binario infijo	izquierda	+	suma
binario infijo	izquierda	−	resta
binario infijo	izquierda	<<	corrimiento de bits a la izquierda
binario infijo	izquierda	>>	corrimiento de bits a la derecha llenando con ceros
binario infijo	izquierda	>>>	corrimiento de bits a la derecha propagando signo
binario infijo	izquierda	<	relacional “menor que”
binario infijo	izquierda	<=	relacional “menor o igual que”
binario infijo	izquierda	>	relacional “mayor que”

Continúa en la siguiente página

Tabla 3.2 Operadores de Java

Continúa de la página anterior

Operandos	Asociatividad	Símbolo	Descripción
binario infijo	izquierda	>=	relacional “mayor o igual que”
binario infijo	izquierda	**instanceof**	relacional “ejemplar de”
binario infijo	izquierda	==	relacional, igual a
binario infijo	izquierda	!=	relacional, distinto de
binario infijo	izquierda	&	AND de bits
binario infijo	izquierda	^	XOR de bits
binario infijo	izquierda	|	OR de bits
binario infijo	izquierda	&&	AND lógico
binario infijo	izquierda	||	OR lógico
ternario infijo	izquierda	<exp log > ? <exp> : <exp>	Condicional aritmética
binario infijo	derecha	=	asignación
binario infijo	derecha	+=	autosuma y asignación
binario infijo	derecha	−=	autoresta y asignación
binario infijo	derecha	*=	autoproducto y asignación
binario infijo	derecha	/=	autodivisión y asignación
binario infijo	derecha	%=	automódulo y asignación
binario infijo	derecha	>>=	autocorrimiento derecho y asignación
binario infijo	derecha	<<=	autocorrimiento izquierdo y asignación
binario infijo	derecha	>>>=	autocorrimiento derecho con propagación y asignación
binario infijo	derecha	&=	auto-AND de bits y asignación
binario infijo	derecha	^=	auto-XOR de bits y asignación
binario infijo	derecha	|=	auto-OR de bits y asignación

Los operadores en general se comportan como uno esperaría; si uno hace

```
int a = 3 + 2;
```

entonces a toma el valor 5. Todos los operadores tienen un dominio específico; cosas como

```
true + false;
```

no tienen sentido. Cuando se opera con tipos distintos, pero compatibles, se asume el tipo más general. Por ejemplo, si tenemos

```
int a = 2;
float x = 1.5;
```

entonces `a+x` es de tipo **float**.

Los operadores funcionan de acuerdo al tipo de los operandos. Si se dividen dos enteros con el operador /, se hace división entera; el resultado se trunca al mayor entero menor o igual al resultado de la división. Si se dividen flotantes o dobles, el resultado es la división real (o lo más cercano, de acuerdo a la precisión simple o doble de cada tipo).

Hay varios operadores en la tabla que funcionan sobre bits. Esto quiere decir que operan sobre un valor de tipo entero, pero fijándose en los bits, más que en el valor en sí.
Veamos un ejemplo.

```
int a = 13;
int b = 19;
int c = a | b;
```

La variable `a` vale `13` y la `b` vale `19`. En bits esto es

$$\begin{aligned} a &= 00000000000000000000000000001101 \\ b &= 00000000000000000000000000010011 \end{aligned}$$

(recordemos que los enteros tienen 32 bits). Para no tener que escribir tanto, usamos hexadecimal para representar los valores y podemos verlos como `0x0000000D` y `0x00000013`.

¿Cuál es valor de la variable `c` entonces? Estamos usando el OR para bits; se aplica un OR a cada uno de los bits del entero, y el resultado es

$$c = 00000000000000000000000000011111$$

o bien `0x0000001F`, que a su vez es el valor 31 en base 10. Siempre hay que tener en cuenta que los tipos enteros de Java (**byte**, **short**, **int**, **long**) tienen signo; eso quiere decir que el bit más significativo en estos tipos guarda el signo.

La tabla 3.2 está ordenada en orden de precedencia. La precedencia de los operadores determina el orden en que serán aplicados. Por ejemplo en

```
int a = 3 + 2 * 5;
```

el valor de a es 13, no 25, ya que la precedencia de la multiplicación es mayor que la de la suma. Si queremos que la suma se aplique antes, tenemos que hacer

```
int a = (3 + 2) * 5;
```

Pueden utilizar la tabla de operadores para saber qué operador tiene precedencia sobre cuáles; sin embargo, en general es bastante intuitivo. En caso de duda, poner paréntesis nunca hace daño (pero hace que el código se vea horrible; los paréntesis son malos^{MR}).[4]

Los operadores relacionales (<, >, <=, >=) funcionan sobre tipos numéricos, y regresan un valor booleano. La igualdad y la desigualdad (==, !=) funcionan sobre *todos* los tipos de Java. En el caso de las referencias, == regresará **true** si y sólo si las referencias apuntan a la misma posición en la memoria; al mismo objeto.

Hay que notar que la asignación (=) es muy diferente a la igualdad (==). La asignación toma el *valor* de la derecha (se hacen todos los cómputos que sean necesarios para obtener ese valor), y lo asigna en la *variable* de la izquierda. El valor debe ser de un tipo compatible al de la variable. También hay algunos operadores de conveniencia, los llamados "auto operadores". Son +=, −=, *=, /=, %=, >>=, <<=, >>>=, &=, ^=, y |=. La idea es que si tenemos una variable entera llamada a, entonces

```
a += 10;
```

es exactamente igual a

```
a = a + 10;
```

E igual con cada uno de los auto operadores. Como el caso de sumarle (o restarle) un uno a una variable es muy común en muchos algoritmos, los operadores ++ y −− hacen exactamente eso. Pero hay dos maneras de aplicarlos: prefijo y post fijo. Si tenemos

```
int a = 5;
int b = a++;
```

entonces a termina valiendo 6, pero b termina valiendo 5. Esto es porque el operador regresa el valor de la variable *antes* de sumarle un uno, y hasta después le suma el uno a la variable. En cambio en

[4] "Parenthesis are evil^{TM}".

```
int a = 5;
int b = ++a;
```

las dos variables terminan valiendo 6, porque el operador *primero* le suma un uno a a, y después regresa el valor de la variable.

Los operadores lógicos && (AND) y || (OR) funcionan como se espera que funcionen ($p \wedge q$ es verdadero si y sólo si p y q son verdaderos, y $p \vee q$ es verdadero si y sólo si p es verdadero o q es verdadero), con una pequeña diferencia: funcionan en *corto circuito*. Esto quiere decir que si hacemos

```
if (a < b && b < c)
```

y a < b es falso, entonces el && ya no comprueba su segundo operando. Ya sabe que el resultado va a dar falso, así que ya no se molesta en hacer los cálculos necesarios para ver si b < c es verdadero o falso. Lo mismo pasa si el primer operando de un || resulta verdadero. Es importante tener en cuenta esto, porque aunque en lógica matemática nos dijeron que $p \wedge q \equiv q \wedge p$, a la hora de programar esto no es 100 % cierto.

Un operador bastante útil es la condicional aritmética (?:). Funciona de la siguiente manera:

```
int z = (a < b) ? 1 : 0;
```

Si a es menor que b, entonces el valor de z será 1. Si no, será 0. La condicional aritmética funciona como un **if** chiquito, y es muy útil y compacta. Por ejemplo, para obtener el valor absoluto de una variable (digamos a) sólo necesitamos hacer

```
int b = (a < 0) ? -a : a;
```

Hay varios operadores que funcionan con referencias. Algunos funcionan con referencias y con tipos básicos al mismo tiempo (como = y ==, por ejemplo); pero otros son exclusivos de las referencias.

Los operadores más importantes que tienen las referencias son **new**, que las inicializa, y . (punto), que nos permite interactuar con las partes de un objeto.

Hay otros operadores para los objetos, y otros para tipos básicos, que iremos estudiando en las prácticas que siguen.

Expresiones

Una vez que tenemos variables (tipos básicos u objetos), literales y operadores, obtenemos el siguiente nivel en las construcciones de Java. Al combinar una o más variables o literales con un operador, tenemos una *expresión*.

Las expresiones no son atómicas; pueden partirse hasta que lleguemos a variables o literales y operadores. Podemos construir expresiones muy rápido:

```
a++;                                                    // Esto es una expresión.
(a++);                                                  // Esto también.
(a++)*3;                                                // También.
5+((a++)*3/2+5/reloj.dameMinuto()*reloj.dameHoras());
                                                        // Sí, también.
```

En general, todas las expresiones regresan un valor. No siempre es así, sin embargo, como veremos en la siguiente práctica.

Podemos hacer expresiones tan complejas como queramos (como muestra el último de nuestros ejemplos arriba), siempre y cuando los tipos de cada una de sus partes sean compatibles. Si no lo son, el compilador de Java se va a negar a compilar esa clase.

El siguiente nivel en las construcciones de Java lo mencionamos en la práctica pasada: son los bloques. En la próxima práctica analizaremos nivel de construcción más importante de Java: las clases.

Ejercicios

Todas estas pruebas las tienes que realizar dentro del método `main` de tu clase `Prueba`.

1. Trata de obtener el módulo[5] (`%`) de un flotante.

2. Declara un **`float`** llamado `x`, y asígnale la literal `1F`. Declara un **`float`** llamado `y` y asígnale la literal `0.00000001F`. Declara un tercer **`float`** llamado `z` y asígnale el valor que resulta de restarle `y` a `x`. Imprime `z` en tu consola.

3. Imprime el valor de la siguiente expresión: `1>>1`. Ahora imprime el valor de la siguiente expresión `-1>>1`.

4. Declara dos variables booleanas `p` y `q`.

   Asígnales a cada una un valor (**`true`** o **`false`**, tú decide), e imprime el valor de la expresión $\neg(p \wedge q)$ en sintaxis de Java. Después imprime el valor de la expresión $\neg p \vee \neg q$ utilizando la consola.

   Cambia los valores de `p` y `q` para hacer las cuatro combinaciones posibles, imprimiendo siempre el resultado de las dos expresiones. Con esto demostrarás la regla de De Morgan caso por caso.

[5] El resultado que se obtiene de la expresión `a % b`, con `a` y `b` enteros, es el *residuo entero* – lo que "sobra" – que resulta al dividir `a` entre `b`.

Preguntas

1. ¿Crees que sea posible asignar el valor de un flotante a un entero? ¿Cómo crees que funcionaría?

2. Observa el siguiente código

```
int a = 1;
int b = 2;
int c = 3;

(a > 3 && ++a <= 2) ? b++ : c--;
```

   Sin compilarlo, ¿cuál es el valor final de `a`, `b`, `c`? Compila y compara lo que pensaste con el resultado real. Explica porqué cada variable termina con el valor que termina.