Sumario

Operadores y Conversiones.

Operadores

Los operadores realizan operaciones entre uno, dos o tres operandos.
Los operadores que requieren un operador se llaman operadores unarios. Por ejemplo, ++ es un operador unario que incrementa el valor su operando en uno.
Los operadores que requieren dos operandos se llaman operadores binarios. El operador = es un operador binario que asigna un valor del operando derecho al operando izquierdo.
Existe un único operador ternario que trabaja con tres operandos y funciona como un condicional y dependiendo de cómo evalúa la condición devuelve un valor u otro. Este operador es :?
Los operadores unarios en Java pueden utilizar la notación de prefijo o de sufijo (también conocidas como pre y post, por ejemplo, un pre incremento de la variable i es ++i, mientras que un post incremento es i++). La notación de prefijo significa que el operador aparece antes de su operando, por ejemplo,

```
operador operando
```

La notación de sufijo significa que el operador aparece después de su operando:

```
operando operador
```

Todos los operadores binarios de Java tienen la misma notación, es decir aparecen entre los dos operandos:

```
op1 operator op2
```

Además de realizar una operación también devuelve un valor. El valor y su tipo dependen del tipo del operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos (realizan las operaciones de aritmética básica como la suma o la resta) devuelven números, el resultado típico de las operaciones aritméticas.
El tipo de datos devuelto por los operadores aritméticos depende del tipo de sus operandos: si se suma dos enteros, se obtiene un entero. Se dice que una operación evalúa su resultado.
Es muy útil dividir los operadores Java en las siguientes categorías: aritméticos, relacionales y condicionales, lógicos y de desplazamiento y de asignación.

Operadores aritméticos

El lenguaje Java soporta varios operadores aritméticos, incluyendo

Operador	Descripción
+	suma
-	resta
*	multiplicación
/	división
%	módulo

En todos los números enteros y de coma flotante. Por ejemplo, se puede utilizar este código Java para sumar dos números:

```
sumarEsto + aEsto
```

O este código para calcular el resto de una división:

```
dividirEsto % porEsto
```

Esta tabla resume todas las operaciones aritméticas binarias en Java:

Operador	Uso	Descripción
+	op1 + op2	Suma op1 y op2
-	p1 – op2	Resta op2 de op1
*	op1 * op2	Multiplica op1 y op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Obtiene el resto de dividir op1 por op2

Nota: El lenguaje Java extiende la definición del operador + para incluir la concatenación de cadenas (Strings).

Los operadores + y - tienen versiones unarias que seleccionan el signo del operando:

Operador	Uso	Descripción
+	+op	Indica un valor positivo
-	- op	Niega el operando

Además, existen dos operadores de incremento y decremento aritméticos, ++ que incrementa en uno su operando, y -- que decrementa en uno el valor de su operando.

Operador	Uso	Descripción
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
–	op –	Decrementa op en 1; evalúa el valor antes de decrementar
–	-- op	Decrementa op en 1; evalúa el valor después de decrementar

Operadores de asignación

Se puede utilizar el operador de asignación =, para asignar un valor a otro.
Además del operador de asignación básico, Java proporciona varios operadores de asignación que permiten realizar operaciones aritméticas, lógicas o de bits y una operación de asignación al mismo tiempo.

Específicamente, suponiendo que se quiere añadir un número a una variable y asignar el resultado dentro de ella misma, se puede hacer.

```
i = i + 2;
```

Se puede lograr el mismo resultado en la sentencia utilizando el operador +=.

```
i += 2;
```

Las dos líneas de código anteriores son equivalentes.
La siguiente tabla lista los operadores de asignación y sus equivalentes:

Operador	Uso	Equivale a
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	p1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Operadores relacionales

Los valores relacionales comparan dos valores y determinan la relación entre ellos. Por ejemplo, != devuelve true si los dos operandos son distintos.
La siguiente tabla resume los operadores relacionales de Java:

Operador	Uso	Devuelve true si
>	op1 > op2	op1 es mayor que op2
>=	op1 >= op2	op1 es mayor o igual que op2
<	op1 < op2	op1 es menor que op2
<=	op1 <= op2	op1 es menor o igual que op2
==	op1 == op2	op1 y op2 son iguales
!=	op1 != op2	op1 y op2 son distintos

Operadores lógicos o condicionales

Se utilizan para construir expresiones de decisión complejas. Uno de estos operadores es && que realiza la operación Y lógico o booleano. Por ejemplo, se puede utilizar dos operadores relacionales diferentes y evaluar la totalidad de la expresión uniéndola con && para determinar si ambas relaciones son ciertas. La siguiente línea de código utiliza esta técnica para determinar si un valor esta entre dos límites, esto es, para determinar si el índice es mayor que 0 o menor que NUM_ENTRADAS (que se ha definido previamente como un valor constante):

```
0 < index && index < NUM_ENTRADAS
```

Se puede observar que en algunas situaciones, el segundo operando de un operador relacional no será evaluado. Consideremos esta sentencia:

```
((contador > NUM_ ENTRADAS) && (System.in.read() != -1))
```

Si contador es menor que NUM_ ENTRADAS en la parte izquierda del operando &&, la parte derecha no se evalúa. El operador && sólo devuelve true si los dos operandos son verdaderos.

Por eso, en esta situación se puede determinar el valor de && sin evaluar el operador de la derecha y en un caso como este, Java ignora el operando de la derecha. Así no se llamará a System.in.read() y no se leerá un carácter de la entrada estándar.

Los operadores condicionales lógicos son:

Operador	Uso	Devuelve true si
&&	op1 && op2	op1 y op2 son verdaderos
||	op1 || op2	uno de los dos es verdadero
!	! op	op es falso

El operador & se puede utilizar como un sinónimo de && si ambos operadores son bits. Similarmente, | es un sinónimo de || si ambos operandos son bits. En definitiva, estos operadores están diseñados para actuar como sus contrapartidas, pero en lugar de evaluar todo el operando, lo hace bit a bit entre cada uno de los operandos. Para entender mejor lo afirmado, se realiza la siguiente clasificación

Operadores lógicos a nivel de bit entre operandos (Binarios)

Operador	Descripción
~	NOT
|	OR
&	AND
^	XOR

Conversión de tipo de datos

Promoción de expresiones en las conversiones de tipo

Las variables se promueven automáticamente a tipos más grandes (como por ejemplo, un int a long). Esto se debe a que Java permite que si una variable se asigna a otra del mismo tipo (por ejemplo, dos tipos enteros) automáticamente verifica que el tamaño en donde se va a asignar el valor sea mayor o igual que el mismo.
Por lo tanto, las asignaciones de expresiones son compatibles si el tipo de variable que recibe el valor es al menos del mismo tamaño en bits que el resultado que la expresión arroja

```
long valorGrande = 6;          // 6 es un tipo entero, esta bien
int valorPequeño = 99L;        // 99L es un long, esta mal
double z = 12.414F;            // 12.414F es float, esta bien
float z1 = 12.414;             // 12.414 es double, esta mal
```

Las promociones ocurren también en expresiones aritméticas como pasos intermedios. Por ejemplo, si se tiene el siguiente código

```
long valorGrande = 6;   // 6 es un tipo entero, esta bien
double z = 12.414F;     // 12.414F es float, esta bien
float resultado = 4 + valorGrande + (float)z;
```

Concatenación de Strings

Java permite concatenar cadenas fácilmente utilizando el operador +. Este operador puede crear un nuevo String para resolver una concatenación. Ejemplo

```
String titulo = "Dr.";
String nombre = "Pedro" + " " + "Ramirez";
String saludo = titulo + " " + nombre;
```

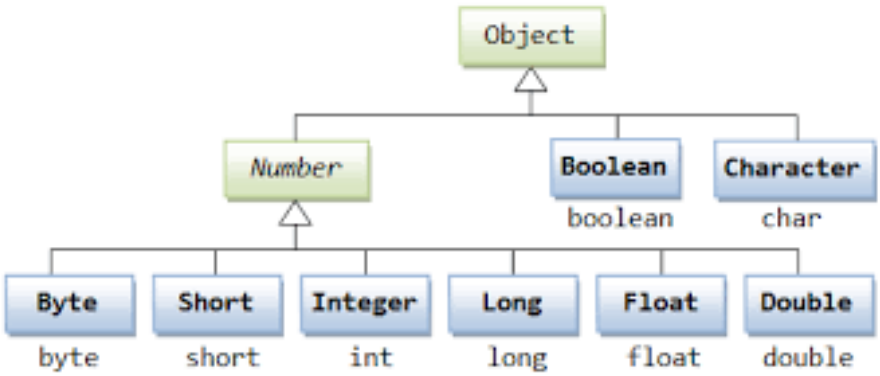
En una concatenación, si algún elemento no es String, se lo convierte automáticamente, pero se debe tener en cuenta que al menos un elemento debe ser de tipo String. El lenguaje trata a este tipo en particular como uno superior respecto de los demás y realiza una promoción de otro tipo de datos a String
El siguiente fragmento de código concatena tres cadenas

```
"La entrada tiene " + contador + " caracteres."
```

Dos de las cadenas concatenadas son literales: "La entrada tiene " y " caracteres.". El tercer String, el del medio, es realmente un entero que primero se convierte a tipo String y luego se concatena con las otras cadenas.

Clase Number

En Java existen Clases que heredan de la **clase Numer** del paquete **java.lang** (incluido por defecto en Java) y que sirven de envoltorio a los tipos primitivos ya conocidos.
Estas clases aparte de proporcionar métodos útiles, sirven en la conversión a objetos de un tipo primitivo y viceversa, acción que en determinados contextos puede ser realizada por el compilador de java de forma automática, aunque usando estas clases el programador puede realizarlo de forma explícita.
La siguiente imagen muestra el diagrama de herencia de clase Number en Java.



Subclases de la clase Number.

En la tabla siguiente se exponen las clases principales que heredan de la clase Number:

Clase	**Tipo primitivo.**
Byte	byte.

Double	double.
Float	float.
Integer	int.
Short	short.
Long	long

Declaración de la clase Number.

```
package java.lang;
public abstract class Number implements java.io.Serializable {
    public abstract int intValue();
    public abstract long longValue();
    public abstract float floatValue();
    public abstract double doubleValue();
    public byte byteValue() {
        return (byte)intValue();
    }
    public short shortValue() {
        return (short)intValue();
    }
    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    private static final long serialVersionUID = -8742448824652078965L;
}
```

Clases de Envoltorio (Wrapers)

Las clases de envoltorio sirven para utilizar valores almacenados en datos primitivos como si fueran objetos. La razón de estos es que los datos primitivos no son manejados como objetos internamente, por lo tanto para utilizarlos así, se deben crear objetos que en su construcción reciben como argumento el dato primitivo y permiten el manejo del valor como si fueran objetos. Las variables primitivas tienen mecanismos de reserva y liberación de memoria más eficaces y rápidos que los objetos por lo que deben usarse datos primitivos en lugar de sus correspondientes envolturas siempre que se pueda.
Java provee una clase de envoltorio para cada tipo de dato primitivo, como se muestra en la siguiente tabla

Las clases envoltorio existentes son:

- `Byte` para `byte`.
- `Short` para `short`.
- `Integer` para `int`.
- `Long` para `long`.
- `Boolean` para `boolean`
- `Float` para `float`.
- `Double` para `double y`
- `Character` para `char`.

Observese que las clases envoltorio tienen siempre la primera letra en mayúsculas.

Las clases de envoltorio son útiles también para las conversiones de tipos porque definen métodos para este fin. Cada tipo tiene una serie de métodos específicos para la conversión que se necesite realizar.

En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, etc.) como objetos. Por ejemplo, los contenedores definidos por el API en el package java.util (Arrays dinámicos, listas enlazadas, colecciones, conjuntos, etc.) utilizan como unidad de almacenamiento la clase Object. Dado que Object es la raíz de toda la jerarquía de objetos en Java, estos contenedores pueden almacenar cualquier tipo de objetos. Pero los datos primitivos no son objetos, con lo que quedan en principio excluidos de estas posibilidades.

Para resolver esta situación el API de Java incorpora las clases envoltorio (wrapper class), que no son más que dotar a los datos primitivos con un envoltorio que permita tratarlos como objetos. Por ejemplo podríamos definir una clase envoltorio para los enteros, de forma bastante sencilla, con:

```
public class Entero {
	private int valor;
	Entero(int valor) {
		this.valor = valor;
	}
	int intValue() {
		return valor;
	}
}
```

La API de Java hace innecesario esta tarea al proporcionar un conjunto completo de clases envoltorio para todos los tipos primitivos. Adicionalmente a la funcionalidad básica que se muestra en el ejemplo las clases envoltorio proporcionan métodos de utilidad para la manipulación de datos primitivos (conversiones de / hacia datos primitivos, conversiones a String, etc.)

Las clases envoltura se usan como cualquier otra:

```
Integer i = new Integer(5);
int x = i.intValue();
```

Hay que tener en cuenta que las operaciones aritméticas habituales (suma, resta, multiplicación ...) están definidas solo para los datos primitivos por lo que las clases envoltura no sirven para este fin.

Las variables primitivas tienen mecanismos de reserva y liberación de memoria más eficaces y rápidos que los objetos por lo que deben usarse datos primitivos en lugar de sus correspondientes envolturas siempre que se pueda.

Conversión automática de tipos primitivos en objetos: autoboxing

Si se necesita cambiar los tipos de datos primitivos en su objeto equivalente (operación denominada boxing), hay que utilizar las clases envoltorio.

Igualmente, para obtener el tipo de dato primitivo a partir de la referencia del objeto (lo que se denomina unboxing), también necesita usar los métodos de las clases envoltorio. Todas estas operaciones de conversión pueden complicar el código y dificultar su comprensión. En la versión 5.0 de J2SE se ha introducido una función de conversión automática denominada autoboxing que permite asignar y recuperar los tipos primitivos sin necesidad de usar clases envoltorio.
El ejemplo siguiente contiene dos casos sencillos de conversión y recuperación automática de primitivos (autoboxing y autounboxing).

```
int entero = 420;
Integer envoltorio = entero; // esto se denomina autoboxing
System.out.println(envoltorio);
int p2 = envoltorio; // esto se denomina autounboxing
System.out.println(p2);
```

Un compilador de J2SE versión 5.0 o superior ahora creará el objeto de envoltorio automáticamente cuando se asigne un primitivo a una variable del tipo de la clase de envoltorio.

Asimismo, el compilador extraerá el valor primitivo cuando realice la asignación de un objeto de envoltorio a una variable de tipo primitivo.
Esto puede hacerse al pasar parámetros a métodos o, incluso, dentro de las expresiones aritméticas.

Nota: *No abusar de la función de autoboxing. El rendimiento se ve afectado cada vez que se utiliza la conversión o recuperación automática de tipos primitivos. La combinación de tipos primitivos y objetos de envoltorio en expresiones aritméticas dentro de un ciclo cerrado podría tener efectos negativos sobre el rendimiento y la velocidad de transmisión de datos de las aplicaciones.*

Convirtiendo Strings a valores numéricos

Una necesidad común de un desarrollador Java es convertir cadenas a valores numéricos para luego manipularlos como un dato primitivo (por ejemplo, un int , float , doublé , etc...). Existen numerosas situaciones en la que esta necesidad se presenta (lecturas de archivos, parámetros de línea de comando, etc...), o tan sólo tomar un valor ingresado en el campo de texto de una interfaz gráfica para usuarios.
Para convertir un String a un dato primitivo, se debe comprender el uso de las clases de envoltorio. Cada tipo primitivo tiene asociado una clase de este tipo, como se muestra en la tabla anterior. Estas clases permiten tratar a los datos primitivos como objetos, dando además el beneficio de servicios extras para que se puedan manipular estos objetos a través de sus métodos en la forma apropiada para cada tipo primitivo.
Todas las conversiones excepto las del tipo boolean, se realizan con métodos que tienen nombres similares aunque diferentes.

Tipo Primitivo	**Método de Conversión**
byte	Byte.parseByte(unString)
short	Short.parseShort(unString)
int	Integer.parseInt(unString)
long	Long.parseLong(unString)
float	Float.parseFloat(unString)
double	Double.parseDouble(unString)
boolean	Boolean.valueOf(unString).booleanValue()

Cada una de estas clases de envoltorio, excepto la clase Character, tiene un método que permite

convertir un tipo primitivo a String (o sea, la operación inversa). Lo único que se debe hacer es llamar al correspondiente método de la clase de envoltorio apropiada para convertir el tipo de dato.

```
//Ejemplo
String miString = "12345";
int miInt = Integer.parseInt(miString);
```

Estas sentencias convierten el contenido de una variable de tipo String llamada miString a un tipo primitivo int llamado miInt. Si bien la conversión es simple, se debe tener en cuenta que para cada tipo de datos esta involucrado un único nombre de método en cada clase de envoltorio.
Existe una excepción, la clase Character no tiene un método de este tipo, en su lugar, se debe especificar que caracter se quiere rescatar del String indicando la posición que este ocupa en él con el método charAt. Si por ejemplo, se quiere obtener el caracter “o” en una cadena “Hola”, la forma es la siguiente:

```
//Ejemplo
String hello = "Hola";
char e = hello.charAt(1);
```

Si en cualquiera de los casos mencionados no se puede convertir el tipo String, se produce un error en tiempo de ejecución.

Nota: *Las clases de envoltorio y String generan objetos inmutables, esto quiere decir que si se le cambia el valor que contienen generan un nuevo objeto descartando el anterior para ser recolectado como basura del heap.*