Unidad 2

2.1 Identificadores

Tomemos como ejemplo el siguiente programa que calcula la media de tres números (tipo double) introducidos por teclado, calcula la media almecenándola en una variable y luego muestra el resultado:

```
import java.util.Scanner; // Scanner is in the java.util package
public class ComputeAverage {
    public static void main(String[] args) {
        // Create a Scanner object
        Scanner input = new Scanner(System.in);
        // Prompt the user to enter three numbers
        System.out.print("Enter three numbers: ");
        double number1 = input.nextDouble();
        double number2 = input.nextDouble();
        double number3 = input.nextDouble();
        // Compute average
        double average = (number1 + number2 + number3) / 3;
        // Display results
        System.out.println("The average of " + number1 + " " + number2 + " " +
number3 + " is " + average);
    }
}
```

Los **identificadores** son los nombres que identifican los elementos tales como las clases, métodos y las variables de un programa.

Según el programa anterior, **ComputeAverage**, **main**, **input**, **number1**, **number2**, **number3** son identificadores. Los cuales deben de cumplir las siguientes normas:

- Un **identificador** es una secuenca de caracteres que consisten en letras, dígitos, guiones bajos (_) y signo dollar (**$**).
- Un **identificador** de comenzar por una letra, guión bajo o dollar, nunca con un dígito o espacio en blanco.
- Un identificador no podrá ser ni **true**, ni **false** ni **null**.
- Un identificador puede tener cualquier longitud.

Por ejemplo: **$2, ComputeArea, ara, radius, input** son identificadores legales, mientras que **2A, d+4, mi variable** no lo son.

> Recuerda que Java es *case sensitive*: **area** no es igual a **Area**.

Los identificadores se utilizan para nombrar variables, métodos, clases y otros items de un programa. Identificadores descriptivos hacen programas fáciles de leer.

- Evita usar abreviaciones para identificadores
- Utiliza palabras completas: **numeroDeAlumno** es mejor que **numAlu,numDeAlu**.
- A veces es mejor utilizar nombres sencillos: **i, j, k, x, y, s** pero como método de aprendizaje ya que es más breve.

- No poner un **$** delante de un identificador normalmente ya que esto es para cuando se genera código automáticamente.la almacena en una variable

2.2 Variables

Las variables repesentan valores que pueden cambiar en un programa, dichos valores se almacenan en posiciones de memoria.

```
1  // Compute the first area
2  radius = 1.0;                                          radius:  1.0
3  area = radius * radius * 3.14159;                        area:  3.14159
4  System.out.println("The area is " + area + " for radius " + radius);
5
6  // Compute the second area
7  radius = 2.0;                                          radius:  2.0
8  area = radius * radius * 3.14159;                        area:  12.56636
9  System.out.println("The area is " + area + " for radius " + radius);
```

Podemos ver en el programa anterior, que `radius` es una variable con valor `1.0`, y con esa variable se puede realizar un cálculo que se asigne a la variable `area`. Y luego se imprime el resutlado de ambas variables. Esto lo puodemos hacer las veces que queramos, pero los valores pueden ir cambiando.

Para utilizar una variable tiene que declararla, de esta forma le dice al compilador que busque un espacio en memoria para esa variable según del tipo que sea.

La sintaxis de declaración de una variable es:

`TipoDeDatos NombreVariable;`

Ejemplos de declaraciones:

```
int count;          // Declaramos un contador tipo entero
double radius;      // Declaramos un radio de tipo doble
double tasaInteres; // Declaramos una tasa de interés de tipo double
```

Si las variables son del mismo tipo se pueden declarar juntas separadas por comas:

```
tipoDatos variable1, variable2, ...., variablen;
```

```
int i, j, k;
```

Se pueden inicializar las variables de la siguiente forma:

```
int count = 1;
//equivalente a
int count;
count = 1;
```

También puedes declarar e inicializar variables de forma conjunta para el mismo tipo:

```
int i = 1, j = 2;
```

> **Una variable debe ser declarada antes de que se le asigne el valor, y en general, antes de que pueda utilizarse en cualquier parte del programa. Cuando sea posible declarar la variable e inicializarla en un paso.**

Cada variable tiene lo que se llama **alcance de la variable** que es donde se puede utilizar o acceder. Es un concepto que iremos viendo en siguientes unidades.

¿Cuál es el error de este programa?:

```
1  public class Test {
2    public static void main(String[] args) {
3      int i = k + 2;
4      System.out.println(i);
5    }
6  }
```

2.3 Constantes

Una constante es un identificador que representa un valor permanente.

El valor de una variable puede cambiar, pero el de una constante nunca cambia. También se llama **variable final**.

En un programa que calcule el área se utiliza mucho el valor PI, el cual es constante, lo podemos utilizar como constante.

Al declarar una constante:

```
final TipoDeDatos NOMBRE_DE_LA_CONSTANTE = valor;
```

- Las constantes deben ser siempre inicializadas.
- La palabra `final` es para decir que es una variable.
- Por convención, se utilizan mayúsculas para declarar variables.

En el siguiente ejemplo se puede ver la declaración de la variable **CM_PER_INCH** :

```
public class Constants
{
   public static void main(String[] args)
   {
      final double CM_PER_INCH = 2.54;
      double paperWidth = 8.5;
      double paperHeight = 11;
      System.out.println("Tamaño del papel en cm: "
         + paperWidth * CM_PER_INCH + " por " + paperHeight * CM_PER_INCH);
   }
}
```

Existen tres beneficios de utilizar constantes:

- No tienes que repetir el mismo valor cuando lo uses.
- Si tienes que cambiar el valor de la variable, lo cambias en un sólo sitio.

- Los nombres descriptivos de las constantes ayudan a entender mejor el programa.

2.4 Convenciones en el uso de nombres

Seguir las convenciones de los nombres en Java hace los programas más fáciles de leer y evita errores.

- Utiliza **lowerCamelCase** para nombrar variables si es largo el nombre: `numberOfStudents`
- Para nombres sencillos de variables en minúscula: `area, radio`
- Utiliza la primera en mayúscula (**UpperCamelCase**) para nombrar las clases: `public class CalcularArea {...}` O también: `public class Sistema {...}`
- Capitalizar las constantes y separarlas si fuera necesario por `_`: `PI, VALOR_MAX`

2.5 Tipos de Datos Numéricos y sus operaciones

Java tiene seis tipos de datos numéricos y de tipo punto flotante con los operadores +, *, -, / y %

Tipos Numéricos

Cada tipo de datos tiene un rango de valores. El compilar asigna espacio de memoria para cada variable o constante de acuerdo a su tipo de datos. Java provee **ocho tipos de datos primitivos para valores numéricos, caracteres y booleanos: byte, short, int, long, float, double, char, boolean**.

En la siguiente tabla se ven los tipos de datos numéricos:

TABLE 2.1 Numeric Data Types

Name	*Range*	*Storage Size*	
`byte`	-2^7 to 2^7-1 (-128 to 127)	8-bit signed	byte type
`short`	-2^{15} to $2^{15}-1$ (-32768 to 32767)	16-bit signed	short type
`int`	-2^{31} to $2^{31}-1$ (-2147483648 to 2147483647)	32-bit signed	int type
`long`	-2^{63} to $2^{63}-1$	64-bit signed	long type
	(i.e., -9223372036854775808 to 9223372036854775807)		
`float`	Negative range: $-3.4028235E+38$ to $-1.4E-45$	32-bit IEEE 754	float type
	Positive range: $1.4E-45$ to $3.4028235E+38$		
`double`	Negative range: $-1.7976931348623157E+308$ to $-4.9E-324$	64-bit IEEE 754	double type
	Positive range: $4.9E-324$ to $1.7976931348623157E+308$		

> El estándar **IEEE 754** se utiliza para representar números en punto flotante. Este estándar se utiliza ampliamente. Java utiliza 32 bit IEEE 754 para los datos de tipo **float** y 64-bit IEEE 754 para el tipo **double**.

Java utiliza cuatro tipos de datos enteros: byte, short, int, long. Básicamente es el tamaño ocupado en memoria lo que va dar la elección de un tipo u otro.

Java utiliza dos tipos de números en punto flotante: float y double. El **double** es dos ves más grande que el **float**. También se dice que el double es de doble precisión y el float de simple precisión. Se suele utiizar el double para mayor precisión.

> Noticia de cómo existen grandes errores en los tipos de datos elegidos: link

Números enteros (literales enteros)

Un entero puede ser asignado a una variable de tipo entero pero tiene que tener el espacio suficiente. Por ejemplo un entero de tipo byte se puede declarar: `byte b = 128` y nos daría un error de compilación ya que si vemos el límite que tiene el tipo byte es de **-128 a 127**.

Un tipo **int** tiene los límites 2^{31} (-2147483648) y 2^{31-1} (2147483647) . No podrías exceder estos límites en una variable de tipo int.

Un tipo **long** se denota con una **L** ó **l (l minúscula)** al final del valor. Por ejemplo para escribir el entero 2147483648 en un programa Java tienes que escribir `2147483648L` ó `21474836481`.

Por defecto un entero es un número decimal entero. Para denotar otras bases:

- Entero binario: utilizar `0b` ó `0B`
- Entero octal: utilizar el `0` como prefijo del número.
- Entero Hexadecimal: usar `0x` ó `0X`

```
System.out.println(0B1111); // Muestra 15
System.out.println(07777); // Muestra 4095
System.out.println(0XFFFF); // Muestra 65535
```

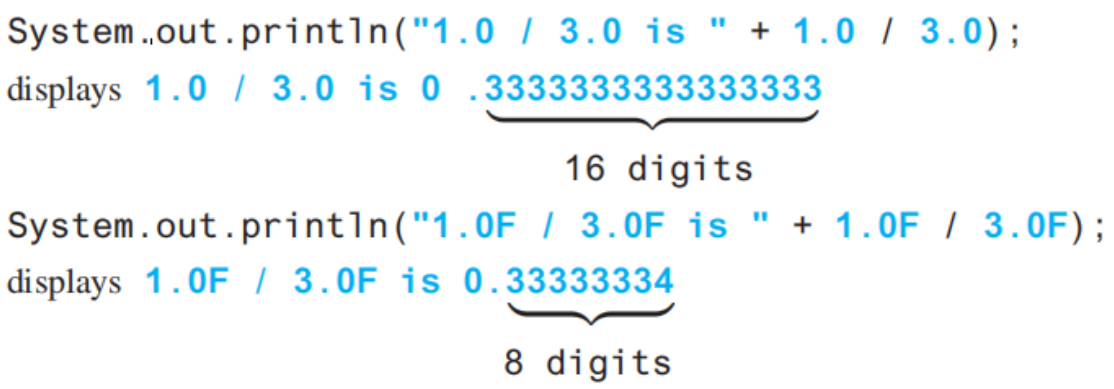
Para mejorar la lectura de números Java permite utilizar guiones bajo:

```
long ssn = 232_45_4519;
long creditCardNumber = 2324_4545_4519_3415L;
```

Números en Punto Flotante

- Son números con decimales.
- Normalmente este número es tratado como tipo **double**.
- Por ejemplo, **5.0** es un número **double** no es un valor **float**. Puedes hacerlo float agregando **F** ó **f** al final del número.: `100.2f`, `100.2F`.
- Y puedes hacer un número **double** agregando **D** ó **d** al final del número: `100.2.d`, `100.2D`

Los valores de tipo *double* son más precisos que los valores de tipo *float*, por ejemplo:



Un valor *float* tiene **7-8** números de dígitos significantes y un *double* tiene **15-17** números de dígitos significativos.

Los números punto flotante tienen unas constantes de números especiales que son las siguientes:

- Infinito Positivo: **Infinity**

- Infinito Negativo: **-Infinity**
- No es un número: **NaN**

Si hacemos una división **0.0/0.0** será un **NaN** y si hacemos un **1/0.0** será **Infinity** y **-1/0.0** será **-Infinity**.

Notación científica

Los números de punto flotante pueden ser escritos en **notación científica** de la forma **a * 10^b** . Por ejemplo:

- La notación científica de 123.456 es **1.23456 * 10^2**
- Para 0.0123456 es **1.23456 * 10^{-2}**

Se suele utilizar una notación especial para este tipo de números, por ejemplo:

- 1.23456 * 10^2 es escrito como **1.23456E2** ó **1.23456+E2** . Se puede utilzar **e** ó **E**.

> Se llaman números de punto flotante por que, por ejemplo, el número 50.534 se almacena internamente como 5.0534E+1, el punto decimal es **flotante** o movido a una nueva posición.

Verifica tu conocimiento:

- Cuánto de preciso en dígitos son el tipo float y double.
- Cuál de los siguientes números se pueden transformar para punto flotante:

  `12.3, 12.3e+2, 23.4e-2, –334.4, 20.5, 39F, 40D`
- Qué numeros son los mismos que el 52.534: `5.2534e+1, 0.52534e+2, 525.34e-1, 5.2534e+0`
- Qué números son correctos: `5_2534e+1, _2534, 5_2, 5_`

Operadores de asignación aumentada

Los operadores `+, -, *, /, %+` se pueden combinar con el operador de asignación para formar operadores aumentados.

A menudo la variable debe ser utilizada y reasignada asimisma. Por ejemplo, un contador:

```
int count;
count = count + 1;
```

Java permite utilizar unos operadores augmentados de la siguiente forma:

```
count += 1;
```

TABLE 2.4 Augmented Assignment Operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

Esta operación: `x /= 4 + 5.5 * 1.5;` es igual a `x = x / (4 + 5.5 * 1.5);`

> No hay espacios entre `+=`

Operadores de Incremento y Decremento

Los operadores de incremento `++` y decremento `--` son utiilzados para aumentar o decrementar la variable por 1

Por ejemplo:

```
int i = 3, j = 3;
i++; // i vale 4
j--; // j vale 2
```

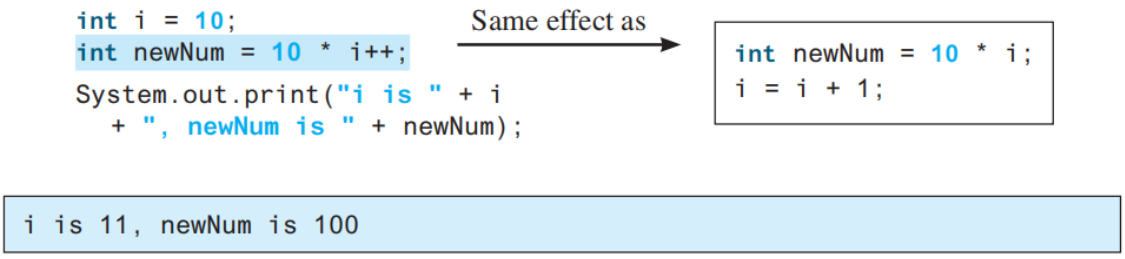
El operador `i++` se llama **postincremento** (postdecremento si es `i--`) y el siguiente que vamos a ver es el `++i` que se llama **preincremento** (preincremento si es `++i`).

```
int i = 3, j = 3;
++i; // i vale 4
--j; // j vale 2
```

Se ve que es el mismo valor, pero la diferencia se verá en sentencias. Aquí vemos un resument de los operadores:

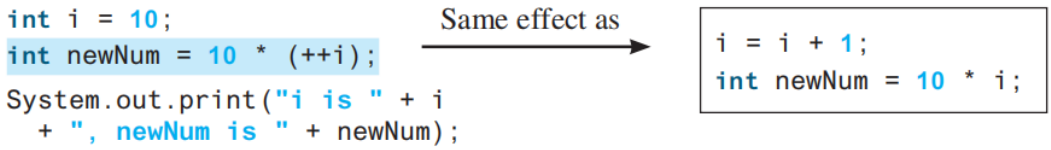
Operator	*Name*	*Description*	*Example* (assume i = 1)
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` // j is 2, i is 2
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` // j is 1, i is 2
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;` // j is 0, i is 0
`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;` // j is 1, i is 0

Y aquí para que veáis la diferencia entre usar pre o post, según el siguiente código:




```
i is 11, newNum is 100
```

Y ahora con el siguiente ejemplo:




```
i is 11, newNum is 110
```

Otro ejemplo:

```
double x = 1.0;
double y = 5.0;
double z = x-- + (++y);
```

y vale 6.0, **z** vale 7.0 y **x** vale 0.0. ¿Qué pasa si no está el paréntesis de ++y?

Tipo de datos carácter (char)

Un tipo de datos carácter representa un único carácter.

Un carácter siempre va entre *comillado simple* `' '`

El tipo primitivo **char** es un dato que se puede representar desde 0000 a FFFF en hexadecimal o también presentado como **\u0000** a **\uFFFF** la u le viene del estándar Unicode de 16 bits y su implementación informática UTF-16.

Puedes consultar el juego de caracteres Unicode aquí

Ejemplos:

```
char letter = 'A';
char numChar = '4';
```

No confundir con el tipo **String** o cadena de caracteres que va entre comillado doble `" "`. Un string es un tipo de datos **no primitivo** que consta de 0 o más caracteres seguidos. De hecho `"A"` es un String y `'A'` es un carácter. Son diferentes tipos. El String lo veremos más adelante.

Unicode y ASCII

Existen unos estándares de conjunto de caracteres que en un principio era el conjunto **ASCII** (American Standard Code for Information Interchange) y luego actualmente se utiliza el conjunto **Unicode** que engloba muchos más caracteres.

El estándar unicode se representa en Java mediante `\u` seguido de una numeración en Hexadecimal de 4 dígitos.

Si imprimimos estos caracteres:

```
System.out.println('\u6b22');
System.out.println("Símbolos del alfabeto chino: '\u6b22' y '\uu8fce");
```

También funciona sin comilla simple pero no en el JShell.

Estas sentencias tienen el mismo significado:

```
char letter = 'A';
char letter = '\u0041'; // Character A’s Unicode is 0041
```

El **Unicode** es un estándar pero no es un sistema de codificación aplicado a la computación, en ese caso utilizaremos el **UTF-8, UTF-16, UTF-32** ya que estos son una implementación del Unicode. **Concretamente nos referimos a UTF-16** pues es de 16 bits y representa todos los datos de tipo char.

Carácter de escape `\`

Los caracteres de escape también pertenecen al conjunto de caracteres y hemos visto que esta sentencia no es válida:

`System.out.println("Dicen por ahí que "Java es divertido"");`

La sentencia anterior arrojaría un error, para que imprimiese entre comillas "Java es divertido" deberíamos escapar o poner carácter de escape `\"` con la barra de esta forma:

`System.out.println("Dicen por ahí que \" Java es divertido \"");` Con salida:

`Dicen por ahí que " Java es divertido "`

TABLE 4.5 Escape Sequences

Escape Sequence	*Name*	*Unicode Code*	*Decimal Value*
`\b`	Backspace	`\u0008`	8
`\t`	Tab	`\u0009`	9
`\n`	Linefeed	`\u000A`	10
`\f`	Formfeed	`\u000C`	12
`\r`	Carriage Return	`\u000D`	13
`\\`	Backslash	`\u005C`	92
`\"`	Double Quote	`\u0022`	34

Tipo de datos lógico o booleano

Este tipo de datos puede tener dos valores: **true** o **false**

Sirve para verificar como vimos en el anterior tema condiciones mediante operadores relacionales en lo que se llama **expresión booleana**:

TABLE 3.1 Relational Operators

Java Operator	*Mathematics Symbol*	*Name*	*Example (radius is 5)*	*Result*
`<`	<	Less than	`radius < 0`	`false`
`<=`	≤	Less than or equal to	`radius <= 0`	`false`
`>`	>	Greater than	`radius > 0`	`true`
`>=`	≥	Greater than or equal to	`radius >= 0`	`true`
`==`	=	Equal to	`radius == 0`	`false`
`!=`	≠	Not equal to	`radius != 0`	`true`

Asimismo si utilizamos un `if (expresión booleanas)` lo que hace es verificar si la condición que chequea el if es verdadera.

Las expresiones booleanas pueden ser compuestas y acompañadas de los operadores booleanos: `|| (OR), && (AND), ! (NOT), ^ (XOR)` que veremos posteriormente.

Este tipo de datos se declara:

```
boolean b; // toma el valor de falso por defecto
boolean b1 = true;
boolean b2 = false;
```

Conversiones de tipos

Conversiones de tipos numéricos (casting)

A continuación introducimos el concepto de conversión llamado **casting de tipos** en el cual se puede convertir de un tipo a otro. Nos ocupamos ahora de los castings numéricos:

- **Los números en punto flotante se convierten a interos utilizando casting explícito**.

- Java convierte directamente de entero a punto flotante: `3 * 4.5` es igual a `3.0 * 4.5`
- Siempre se puede asignar un valor a una variable numérica que soporte un mayor rango de valores. Puedese asignar un **long** a un **float**.
- Si lo que se necesita hacer es pasar un valor a un tipo con un rango menor de valores, se realiza entonces **casting de tipos**
- Convertir un tipo con un rango pequeño a otro con rango mayor de valores se llama *ensanchar un tipo*
- Convertir un tipo con un rango mayor a uno menor se llama *estrechar un tipo* .
- Java ensancha tipos automáticamente pero para estrechar tienes que hacerlo explicitamente.

Para hacer casting explícito se utiliza la siguiente notación:

`(tipoDestino) variableAConvertir;`

Siendo **variableAConvertir** la variable origen que queremos convertir y el **tipoDestino** es el tipo destino que queremos que tenga nuestra variable.

`System.out.println( (int) 1.7 );`

Esto muestra un 1. Cuando un **double** se realiza un casting a un **int** la parte fracional es truncada quedándose sólo la parte entera.

Si yo reailzo lo siguiente:

`System.out.println((double) 1 / 2);`

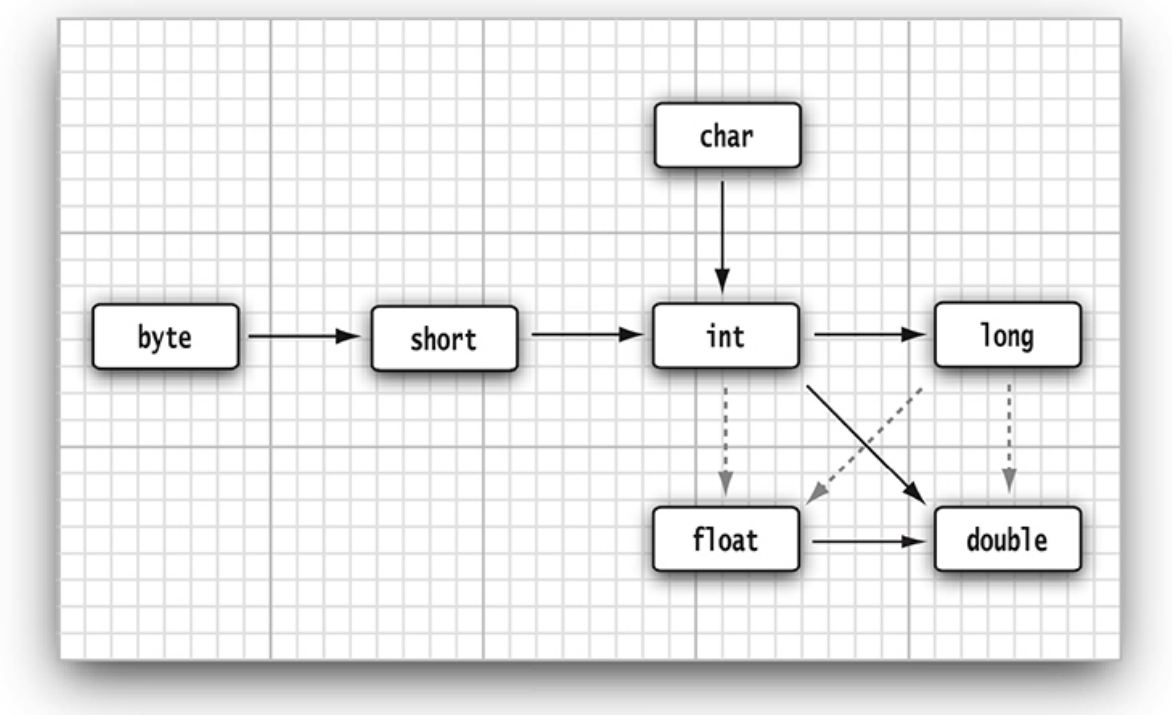
Muestra **0.5** porque tanto 1 como 2 se convierten a double. Sin embargo:

`System.out.println(1 / 2);` Va a mostrar un **0** porque 1 y 2 son enteros y el resultado va a ser un entero o la parte entera.

Siempre que hagas un casting de mayor tipo a menor tipo debes hacerlo explícitamente, si no, tendrás un error de compilación.

Cuando se realizan los castings, a veces se pueden tener pérdidas, lo cual hay que tener en cuenta:

En el siguiente gráfico se muestran las converisones numéricas legales:



- Las flechas continuas denotan las converiones sin pérdida de información
- Las flechas con línea de puntos denotan conversiones con posibles pérdidas.

Ejemplos:

```
// Ejemplo 1
double d = 4.5;
int i = (int) d; // i será 4, pero d sigue siendo 4.5
// Ejemplo 2
int sum = 0;
sum += 4.5; // sum será 4, una expresión equivalente será sum = (int)(sum + 4.5)

// Ejemplo 3
int i = 0;
byte b = 1;
b = i; // error, debes hacer casting (byte)i
```

Casting entre char y tipos numéricos

- Un **char** puede ser convertido a un tipo numéroco y viceversa.

```
int a;
char c = 'A';
a = (int) c; // a ==> 65 que es el código ASCII de A en número
a = c; // también es válido
```

- Cuando un entero se convierta a char sólo se utilizarán 16 bits.

```
// Los enteros hexadecimales se denotan con prefixo 0X
char ch = (char)OXAB0041; // los 16 bits menos significativos de código es 0041
y es lo
                          // que se asigna a ch
System.our.println(ch);   // ch es el carácter A
```

- **Cuando un punto flotante se convierte a char, el punto flotante primero pasa a entero y luego se pasa a char**

```
char ch = (char)65.25; // 65 que pasa a ch
System.out.println(ch); // ch es character A
```

- Cuando un char es convertido a un tipo numérico, el carácter Unicode se convierte al tipo número con el código numérico específico:

```
int i = (int)'A'; // El carácter Unicode de A se asigna a i
System.out.println(i); // i es 65
```

- Cast implícito entre byte char y tipos enteros:

```
byte b = 'a';
int i = 'a';
```

- Pero lo siguiente es incorrecto, porque Unicode `\uFFF4` no cabe en un byte:

```
byte b = '\uFFF4';
//Para forzarlo, se debe hacer:
byte b = (byte)'\uFFF4';
```

- Cualquier entero positivo entre 0 y FFFF en hexadecimal puede ser convertido a un carácter de forma implćita.

- Cualquier número que no esté en este rango, debe hacer cast explícito.
- Todos lo operadore numéricos puden ser aplicados operandos char.
- Un operando char automáticamente se convierte a número.
- Si el operando es un string, el carácter se concatena con el string.

```
int i = '2' + '3'; // (int)'2' es 50 y (int)'3' es 51
System.out.println("i es " + i); // i es 101
int j = 2 + 'a'; // (int)'a' es 97
System.out.println("j es " + j); // j es 99
System.out.println(j + " es el Unicode para el carácter  ")
 + (char)j); // 99 es el Unicode para el carácter c
System.out.println("Capítulo " + '2'); //Aquí lo pasa a un String
```

- También se pueden hacer operaciones como si fueran enteros:

```
'a' < 'b' es true porque el Unicode para 'a' (97) es menor que el Unicode para
'b' (98)
'a' < 'A' falso porque 'a' (97) es mayor que 'A'(65)
```

Anexo

Funciones Math

La clase **Math** tiene una serie de funciones o métodos para cálculos matemáticos que se pueden acceder con:

```
tipoResultado Math.función(argumento1,argumento2,etc)
```

Para saber más de la función Math, acceder a la API de Java

Exponentes

TABLE 4.2 Exponent Methods in the Math Class

Method	*Description*
`exp(x)`	Returns e raised to power of x (e^x).
`log(x)`	Returns the natural logarithm of x ($\ln(x) = \log_e(x)$).
`log10(x)`	Returns the base 10 logarithm of x ($\log_{10}(x)$).
`pow(a, b)`	Returns a raised to the power of b (a^b).
`sqrt(x)`	Returns the square root of x ($\sqrt{x}$) for $x >= 0$.

Ejemplos:

$e^{3.5}$ is `Math.exp(3.5)`, which returns `33.11545`
`ln(3.5)` is `Math.log(3.5)`, which returns `1.25276`
$\log_{10}$ `(3.5)` is `Math.log10(3.5)`, which returns `0.544`
2^3 is `Math.pow(2, 3)`, which returns `8.0`
3^2 is `Math.pow(3, 2)`, which returns `9.0`
$4.5^{2.5}$ is `Math.pow(4.5, 2.5)`, which returns `42.9567`
$\sqrt{4}$ is `Math.sqrt(4)`, which returns `2.0`
$\sqrt{10.5}$ is `Math.sqrt(10.5)`, which returns `3.24`

Redondeo

TABLE 4.3 Rounding Methods in the Math Class

Method	*Description*
`ceil(x)`	*x* is rounded up to its nearest integer. This integer is returned as a double value.
`floor(x)`	*x* is rounded down to its nearest integer. This integer is returned as a double value.
`rint(x)`	*x* is rounded to its nearest integer. If *x* is equally close to two integers, the even one is returned as a double value.
`round(x)`	Returns `(int)Math.floor(x + 0.5)` if *x* is a float and returns `(long)Math.floor(x + 0.5)` if *x* is a double.

Ejemplos:

```
Math.ceil(2.1) returns 3.0
Math.ceil(2.0) returns 2.0
Math.ceil(-2.0) returns -2.0
Math.ceil(-2.1) returns -2.0
Math.floor(2.1) returns 2.0
Math.floor(2.0) returns 2.0
Math.floor(-2.0) returns -2.0
Math.floor(-2.1) returns -3.0
Math.rint(2.1) returns 2.0
Math.rint(-2.0) returns -2.0
Math.rint(-2.1) returns -2.0
Math.rint(2.5) returns 2.0
Math.rint(4.5) returns 4.0
Math.rint(-2.5) returns -2.0
Math.round(2.6f) returns 3 // Returns int
Math.round(2.0) returns 2 // Returns long
Math.round(-2.0f) returns -2 // Returns int
Math.round(-2.6) returns -3 // Returns long
Math.round(-2.4) returns -2 // Returns long
```