CONVERSIÓN DE TIPOS

Hay muchas situaciones en que queremos asignar a una variable de un tipo, un valor de otro tipo.

```
int i=10;
float f;
f=i; //estamos asignando a una variable float un valor entero.
```

Este tipo de situaciones requieren *conversión de tipos*: el valor de la derecha tiene que convertirse en el tipo de la variable de la izquierda. Esta conversión puede hacerse:

- automáticamente por el compilador.

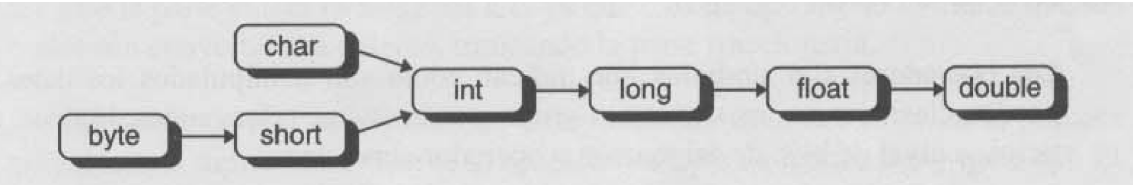
Esta conversión automática recibe indistintamente los nombres de conversión automática, conversión implícita o upcasting, lo de "up" recuerda que se pretende "subir" ("ensanchar" "promocionar") un tipo pequeño, a grande. Pequeño en este contexto es el tipo que utiliza menos bits de almacenamiento o que es más "sencillo".

- expresamente por el programador con operadores de moldeado.

Recibe indistintamente los nombres de conversión explícita o downcasting. Lo de "down" recuerda que se pretende "bajar" de un tipo grande, a uno pequeño .

Conversión automática, implícita, de ensanchamiento o upcasting.

En general puede haber conversión upcasting si se convierte algo "pequeño" en algo "grande". Esto ocurre cuando pasamos un tipo con menos bits a otro de más bits, con la excepción de que se permite pasar un long (64 bits) a un float(32 bits) o double(64 bits) ya que se sobreentiende que el programador es conscientes que esto son tipos imprecisos y que cuenta con ello. Esto se resume con el siguiente gráfico:



Es posible upcasting cuando los tipos están conectados por flechas de izquierda a derecha, por ejemplo:

- es posible convertir automáticamente un byte en un double
- no es posible convertir automáticamente un double en un byte
- es posible convertir automáticamente un char en un int
- no es posible convertir automáticamente un byte en un char
- etc...

Probaremos una serie de ejemplos para comprobar el gráfico anterior.

Ejemplo de conversión automática no posible de double a float.
Comprueba el error al compilar.

```
class App{
    public static void main(String[] args){
        long l;
        double d;
        d=20.3434;
        l=d; //error
```

```
   }
}
```

Ejemplo de conversión automática posible de long a double.
Comprueba que al compilar o.k.

```
class App{
   public static void main(String[] args){
      long l;
      double d;
      l=22222222L;
      d=l; //hay conversión automática
   }
}
```

Ejemplo de conversión automática posible de char a int
Comprueba que al compilar o.k.

```
class App{
   public static void main(String[] args){
      char ch='A';
      int i;
      i=ch;
      System.out.println("en la variable i tengo el código ASCII de A: " + i);
   }
}
```

Esto es posible ya que internamente se representa como un entero sin signo de 16 bits según el código UNICODE, y por tanto "cabe" en un int de 32 bits.

Ejemplo de conversión automática no posible de int a char.
Comprueba el error al compilar.

```
class App{
   public static void main(String[] args){
      char ch;
      int i = 65;
      ch = i; //error
      System.out.println("en la variable ch tengo el valor " + ch);

   }
}
```

Ejemplo de un caso muy particular que no recoge el gráfico de conversiones
se puede asignar un literal entero a una variables char, byte y short siempre y cuando los enteros especificados estén en rango con el tipo de la variable. Compila y comprueba

```
class App{
   public static void main(String[] args){
     char ch;
     ch =65;
     System.out.println("en la variable ch tengo el valor " + ch);
   }
}
```

Ahora bien, el literal entero debe tener un valor dentro del rango de los valores unicode, es decir debe ser menor de 2^{16}
Por tanto si en el ejemplo anterior utilizo:
ch=65535
no hay error de compilación
en cambio con

ch=65536
si hay error

Similar con byte y short

```
class App{
   public static void main(String[] args){
     byte b;
     short s;
     b=127;//ok
     b=128;//ERROR fuera de rango
     s=32767;//ok
     s=32768;//ERROR fuera de rango
   }
}
```

Ejemplo de conversión automática posible de long a float
Es un caso en principio curioso ya que un long tiene 64 bits y un float 32 bits. No obstante con la representación de mantisa y exponente se pueden representar enteros tan grandes con float como con un long, eso sí, en muchas situaciones se produce pérdida de precisión:

```
class App{
   public static void main(String[] args){
      long l=3000000000000000000L;
      float f;
      f=l;
      System.out.println(f);
   }
}
```

Conversión explícita(downcasting)

El operador de cast
El programador indica con el operador de *cast* que se fuerce una conversión. Los términos operador de *Cast*, de *proyección* y de *moldeado* se emplean indistintamente. El operador de cast consiste en una combinación de paréntesis y un tipo de datos, colocando simplemente el tipo de datos al que se quiere convertir entre paréntesis a la izquierda del valor que convertir. Este operador se puede aplicar tanto para hacer upcasting como downcasting. Ya que el upcasting, tal y como se vio anteriormente, es automático, no requiere operador de moldeado pero se puede querer incluir específicamente con la intención de clarificar el código. Un ejemplo:

```
class App{
   public static void main(String[] args){
     int i =200;
     long lng =(long) i;//upcasting: operador de cast superfluo, lo habitual es verlo sin cast como en la siguiente instrucción
     lng=i;
      long lng2=(long) 200; // upcasting: operador de cast superfluo, lo habitual es verlo sin cast como en la siguiente instrucción
     lng2=400;
     i=(int)lng2;//downcasting: operador de cast necesario si queremos hacer la conversión
   }
}
```

Observa que el operador de cast se aplica a un valor sea este valor literal "(long) 200" o variable "(long) i"
Por tanto, el uso del operador de moldeado:

- es obligatorio en downcasting
- y opcional (y muy poco habitual) en upcasting.

Downcasting y truncamiento

Al hacer downcasting es posible que haya pérdida de información. Hay que tener claro que si quiero meter un valor representado por ejemplo con 64 bits en otro con 8 bits, hay situaciones en las que es imposible no perder información. Las conversiones downcasting implican que hay que prescindir de ciertos bits del original. A esto se le llama truncamiento de bits.

convertir un double/float a int.

El truncamiento va a consistir en despreciar la parte decimal.

Ejemplo: Si tenemos el valor 29,7 y lo convertimos a int ¿el valor resultante será 30 o 29?

```
class App{
    public static void main(String[] args){
      double d=29.7;
      int i =(int)d;
      System.out.println(i);
    }
}
```

¡29! Es decir, java simplemente desprecia o "trunca" en este caso la parte decimal. Si quisiéramos obtener el valor 30 deberíamos utilizar el método round() pero no es algo interesante en este momento. Simplemente observar que "truncar" no es lo mismo que redondear.

truncamiento al convertir int en byte/short

Va a consistir en despreciar los bits más a la izquierda.
Por ejemplo, un valor tipo byte se almacena con 8 bits y un tipo int con 32 bits. Al convertir un int en un byte ya que sólo podemos quedarnos con 8 bits, java se queda con los 8 bits LSB(los bits menos significativos, es decir, los de menor peso o dicho de otro modo los de más a la derecha).

```
class App{
    public static void main(String[] args){
      int entero1=10;
      int entero2=300;
      System.out.println((byte)entero1);
      System.out.println((byte)entero2);
    }
}
```

En el código anterior al convertir el primer entero no hay pérdida de información y el segundo sí. Examinemos ambos casos.
Primer caso: convertir el entero 10 a byte

- Entero 10 en binario(32 bits): 00000000000000000000000000001010

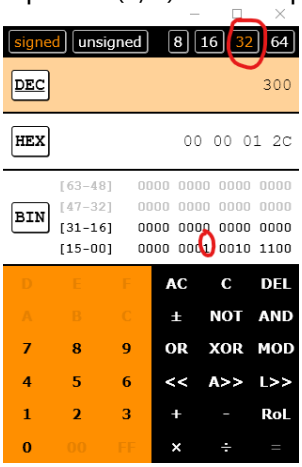
- Java coge los 8 bits más la derecha para obtener un valor tipo byte: 00001010 ¡y esto sigue siendo 10 al pasarlo a base 10!

Segundo caso: convertir el entero 300 a byte

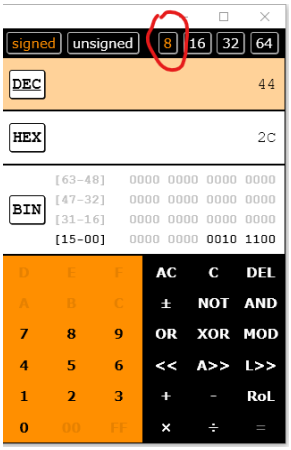
- Entero 300 en binario: 00000000000000000000000100101100
- Java coge los 8 bits más la derecha para obtener un valor tipo byte: 00101100 ¡y esto es 44 al pasarlo al sistema decimal!

Fijate que para 300 necesitamos el 9º bit

El ejemplo de (byte)entero2 lo podemos razonar con calculadora



Si ahora pasamos de 32 a 8 bits perdemos ese 9º bit



El truncamiento puede generar valores en un principio chocantes, pero si se razonan salen las cuentas. Por ejemplo, si el entero 18600 le hacemos un downcast a byte ise convierte en un número negativo -88!.

```
class App{
    public static void main(String[] args){
        byte b;
   int i= 18600;
        b= (byte) i; //habrá pérdida y se genera un número negativa porque el 8º bit codifica signo en el tipo byte.
        System.out.println("al convertir int 18600 a byte: " +b);

    }
}
```

Compruébalo con la calculadora y observa que el truncado genera un número con el 8º a bit a 1 que indica en un tipo byte que se trata de un número negativo, ya que el primer bit en la representación de enteros codifica el signo (0 es + y 1 es -), lo cual estudiarás con calma en otro módulo cuando veas la representación de números enteros en complemento a 2.

Conversiones en expresiones

Hasta ahora vimos casos de conversión de un valor, pero en una expresión puede haber varios valores con una mezcla de tipos ¿Qué tipo tendrá la expresión?

Caso general: el tipo de datos de mayor tamaño dentro de una expresión es el que determina el tipo del valor final de la expresión, por ejemplo, si se multiplica un float por un double el resultado será double, y si se suma un int con un long el resultado será long. Igual en las divisiones, un float dividido por un int genera un float.

Ejemplo: razona el error de javac

```
class App{
   public static void main(String[] args){
      double d=2.0;
         int i= d*4;
   }
}
```

Ejemplo: razona de nuevo el error de javac

```
class App{
   public static void main(String[] args){
         double d=2;
          int i= d*4;
   }
}
```

Ejemplo: razona de nuevo el error de javac

```
class App{
   public static void main(String[] args){
      double d=2;
         float f= d*4;
   }
}
```

Ejemplo: solucionar el error anterior

La expresión d*4 es de tipo double ya que d es double y el resultado de la expresión se almacena en el tipo mayor

La variable es float, por lo tanto, el resultado de la expresión tiene que ser float

```
class App{
   public static void main(String[] args){
      double d=2; //upcasting automático
       float f= d*(float)4; //mal, expresión sigue siendo double
       //float f= (float)d*4;  OK
       float f= (float)(d*4);
       System.out.println(f);
   }
}
```

Caso especial en división: al dividir un entero por un entero se asume división entera y hace java automáticamente el truncamiento cuando la división no es exacta como por ejemplo en el caso 10/3.

Ejemplo:

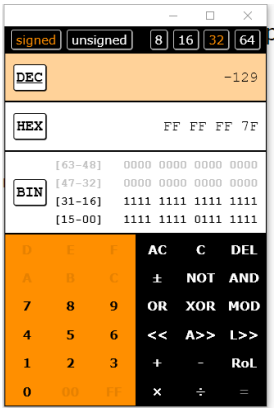
```
class App{
   public static void main(String[] args){
      double x, y;
      int i;
      x=10.0;
      y=3.0;

      System.out.println(10/3);//division entera
      System.out.println(10.0/3);//3 se promociona a double y la división es real

      //i = x/y; //error la división x/y genera un double y no se puede asignar a entero sin downcast
      i = (int) (x/y); //la division genera un double el cast lo pasa a int  truncando.
      System.out.println("Salida de entero de x/y: " +i);

      int m=10;
      int n=3;
      i =  m/n; //division entera.
      System.out.println("Salida de entero de m/n: " +i);


   }
}
```

Caso especial variables byte y short: en una expresión se promocionan automáticamente a int

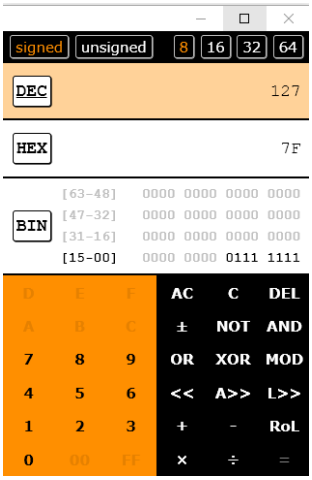
```
class App{
   public static void main(String[] args){
      byte bx=-128;
      byte by=1;
      System.out.println("bx-by sin hacer cast a byte es un entero: " + (bx-by) );
      byte bz = (byte)(bx - by);
      System.out.println("pasando -129 a byte:  " + bz );
   }
}

run:
bx-by sin hacer cast a byte es un entero: -129
pasando -129 a byte:  127
```

Observa que se tiene que calcular una expresión bx-by y ese valor ¿De que tipo será? Podríamos decir: " *es byte, ya que ambos operandos son byte"* PERO NO ES ASÍ ya que en las expresiones JAVA los valores short y byte promocionan automáticamente a int y el resultado final en la expresión anterior es un valor int -129

Ya que bx y by promocionan a int la expresión (bx-by)genera un int (32 bits) que representa a -129 (observa que el bit 31 es un 1 y por tanto representa un número negativo)



Si ahora con cast nos quedamos con 8 bits … nos quedamos con 8 bits que significan 127 (el 8º bit es ahora 0 y es por tanto un número positivo)



Un caso concreto del caso general: mezclar int y char

No es un caso especial pues cumple el caso general, si en una expresión aparecen char e int el char se promociona a int y el resultado será un int

```
class App{
   public static void main(String[] args){

      int i;
      char ch;

      i= 65; //código ascii para A
      ch = (char) i;
      System.out.println("ch: " + ch);

      //ch= ch+1; //error la expresion genera un int
      ch=(char)(ch+1);
      System.out.println("ch: " + ch);
   }
}
```

No olvides descomentar las instrucciones que dan error para observar el error

RESUMEN: La siguiente tabla resume las posibilidades de casting existentes:

Convertir desde / Convertir a...	boolean	byte	short	char	int	long	float	double
boolean		no	no	no	no	no	no	no
byte	no		si	cast	si	si	si	si
short	no	cast		cast	si	si	si	si
char	no	cast	cast		si	si	si	si
int	no	cast	cast	cast		si	si*	si*
long	no	cast	cast	cast	cast		si*	si*
float	no	cast	cast	cast	cast	cast		si
double	no	cast	cast	cast	cast	cast	cast	

Donde:

- no: indica que no hay posibilidad de conversión.
- si: indica que el casting es implícito.
- si*: indica que el casting es implícito pero se puede producir pérdida de precisión.
- cast: indica que hay que hacer casting explícito.

y además recuerda también los casos especiales de cast automático cuando asignamos literales enteros "en rango" a variables char, byte y short

BLUEJ.
En bluej se aprecia muy didácticamente el hecho de que en una expresión "tiene tipo". Recuerda que cuando una expresión se evalúa se reduce a un valor y ese valor tendrá un tipo.

Con el profesor prueba el code pad, una ventana que me permite probar expresiones e incluso "trozos" de código sin necesidad de escribir un programa completo.
Podemos contrastar las cuestiones analizadas en este boletín
Las más básicas ...

```
200
200   (int)
(long)200
200   (long)
2+3
5   (int)
2L+3
5   (long)
(long)(2+3)
5   (long)
```

El comportamiento del operador / ….

```
10/3
3  (int)
10.0/3
3.3333333333333335  (double)
```

La relación char/int...

```
'a'+1
98  (int)
(char)('a'+1)
'b'  (char)
```

Asignación a variables byte y short con valores int pero dentro de rango. Observa que para declarar variables hay que acabar con ;

```
byte bx=100;
byte by=200;
Error: possible loss of precision
  required: byte
  found:    int
```

Promoción automática de variables byte y short a int …

```
byte b1=1;
byte b2=2;
b1+b2
3  (int)
```

Truncamientos

```
(byte)300
44  (byte)
```

etc...

Ejercicio U1_B4_E1: Escribe un programa que compruebe en la tabla anterior la fila de la tabla anterior referente a char

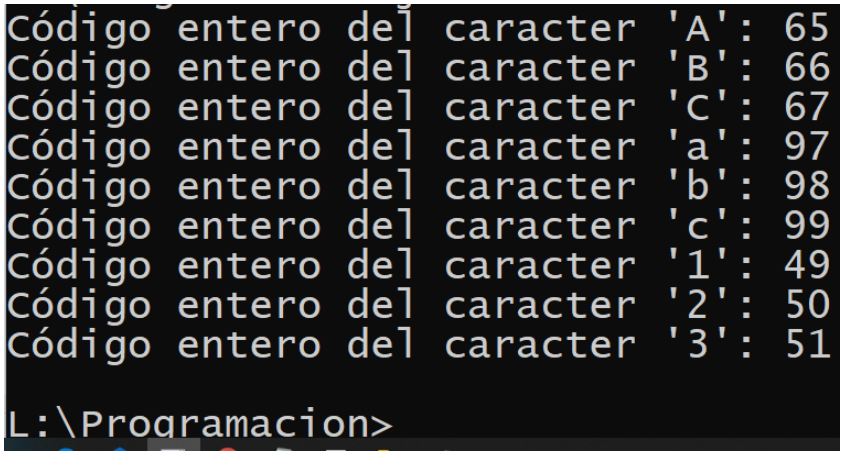
	Boolean	byte	short	char	int	long	float	double
char	no	cast	cast		si	si	si	si

Por ejemplo para el caracter 'a'

```
no se puede hacer cast a boolean
Downcasting a byte 97
Downcasting a short 97
Upcasting a int 97
Upcasting a long 97
Upcasting a float 97.0
Upcasting a double 97.0
```

Ejercicio U1_B4_E2: imprimir los códigos enteros de los siguientes caracteres: A B C a b c 1 2 3 . Observa que los números asignados tienen cierta lógica entre sí. Por ejemplo, entre las letras mayúsculas los códigos se van asignando por orden alfabético, idem para

las minúsculas y para los números se les van también asignando códigos con lógica (el código de '2', es el siguiente al de '1' etc.)l



Ejercicio U1_B4_E3:

```
class App{
  public static void main(String[] args){
        System.out.println("Código entero del caracter 'A': "+ (int)'A');
        System.out.println("Código entero del caracter 'B': "+ (int)'B');
        System.out.println("Código entero del caracter 'B': "+ ('A'+1));
        System.out.println("Código entero del caracter 'B': "+ (char)('A'+1));
  }
}
```

A la vista del código anterior responde a las siguientes preguntas:

1. La instrucción 2º y 3º hacen lo mismo, imprimen el código entero asociado al carácter 'B' ¿Porque la 2ª necesita (int) y la 3ª no?.
2. ¿Por qué la última instrucción imprime el carácter 'B'?

Ejercicio U1_B4_E4:

Haz un programa que produzca una tabla con valores *int* de 127 a 130 convertidos a *byte*. Razona el resultado. La salida será similar a…

int 127 se pasa a byte y es 127

int 128 se pasa a byte y es -128

int 129 se pasa a byte y es -127

int 130 se pasa a byte y es -126

Ejercicio U1_B4_E5: Todos sabemos que 32767+1 es 32. Entonces, ¿por qué obtenemos -32768 en el siguiente código?

```
class App{
   public static void main(String[] args){
      short sx = 32767;
      short sy = 1;
      short sz = (short)(sx + sy);
      System.out.println(" sz vale " + sz );
```

```
   }
}

run:
 sz vale -32768
```