



# LOS TIPOS DE INTEGRALES CON SIGNO

## PRÁCTICA 1

Nataly Yhislaine Jaya Salazar

GPraLab1

09/03/2024

**UNIVERSIDAD DE LLEIDA**

Escola Politècnica Superior

Grado en ingeniería Informàtica

Programación II

## ÍNDICE

<b>Apartado 1: boolean allBits ( int [ ] num ) .....</b>	<b>1</b>
<b>Apartado 2: void copy ( String from, int [ ] to ) .....</b>	<b>4</b>
<b>Apartado 3: int [ ] narrow ( int [ ] num, int toLength ).....</b>	<b>6</b>
<b>Apartado 4: int [ ] widen ( int [ ] num, int toLength ).....</b>	<b>8</b>
<b>Apartado 5: int [ ] cast (int [ ] from, int toLength ).....</b>	<b>10</b>
<b>Apartado 6: int [ ] and ( int [ ] arg1, int [ ] arg2 ).....</b>	<b>11</b>
<b>Apartado 7: int [ ] or ( int [ ] arg1, int [ ] arg2) .....</b>	<b>14</b>
<b>Apartado 8; int [ ] leftShift ( int [ ] num, int numPos).....</b>	<b>15</b>
<b>Apartado 9: int [ ] unsignedRightShift ( int [ ] num, int numPos).....</b>	<b>17</b>
<b>Apartado 10: int [ ] signed RightShift ( int [ ] num, int num Pos) .....</b>	<b>18</b>
<b>Conclusiones .....</b>	<b>19</b>
<b>Referencias .....</b>	<b>20</b>
<b>Anexo de Imágenes .....</b>	<b>21</b>

# APARTADO 1:

## BOOLEAN ALLBITS (INT[] NUM)

Ilustración 1 Código empleado en allBits

```
public boolean allBits(int[] num) {  
    //throw new UnsupportedOperationException("apartado 1");  
    for (int i = 0; i < num.length; i++) {  
        if (!(num[i] == 0 || num[i] == 1)) {  
            return false;  
        }  
    }  
    return true;  
}
```

Fuente 1: Elaboración propia (2024). Captura del código

El propósito de la función *allBits* es validar que la matriz de enteros *num* esté formada enteramente por dígitos binarios: ceros y unos. Para lograr esto, debemos implementar una función que examina meticulosamente cada elemento de la matriz y confirma su valor.

Como la función devuelve una salida booleana, sólo debe generar valores "verdaderos" o "falso". Las actualizaciones recientes del código implicaron realizar modificaciones en la declaración *if*. Se examinó y probó meticulosamente una variedad de escenarios que se explicaran a continuación;

Ilustración 2 Código considerado anteriormente

```
if(num[i]==0) || num[i] == 1) {  
    return true;  
}  
return false;
```

Fuente 2: Elaboración propia (2024). Captura del código

La implementación inicial del código no logró el objetivo deseado ya que solo reconocía los valores binarios de 1 y 0, sin considerar la presencia de dígitos adicionales. En respuesta, se llevó a cabo un programa de prueba separado para evidenciar el comportamiento de cada escenario y comprender los mecanismos. Este enfoque permitió una comprensión del tema en cuestión.

### ESQUEMA DEL PROGRAMA DE PRUEBA:

Es un programa sencillo que sigue una estructura similar. Permite jugar más directamente con los valores de entrada y de esta forma entender íntegramente el comportamiento del programa.

Ilustración 3 Estructura del programa prueba allBits

```
public void run () {  
    int[] nums = {1,9,5,1};  
    println(allbits(nums));  
}  
public boolean allbits (int[] nums) {  
    for (int i = 0; i < nums.length; i++) {  
        if (nums[i] == 0 || nums[i] == 1) {  
            return true;  
        }  
    }  
    return false;  
}  
public static void main(String[] args) {  
    new pruebabebe().start(args);  
}
```

Fuente 3: Elaboración propia (2024). Captura del programa prueba

Ilustración 4 Tabla de comportamiento del código prueba

VALORES DE ENTRADA	
<code>int[] nums = {1,9,5,1};</code>	<code>int[] nums = {1,0,0,1};</code>
RESPUESTA	
<code>true</code>	<code>true</code>

Fuente 4: Elaboración propia (2024). Tabla del canvas que resume el comportamiento del programa. Véase anexos

Implementado el código siguiente en el programa de prueba logra el objetivo deseado. Sin embargo, notamos que era posible una mayor optimización. Al analizar su composición, identificamos el uso del operador *and*, que requirió dos verificaciones, y el operador "!=", que se utilizó en ambas verificaciones. Con estos datos, se contempla la posibilidad de derivar un código más optimo.

Ilustración 5 Segundo código considerado

```
if(num[i] != 0 && num[i] != 1) {  
    return false;  
}  
}  
return true;
```

Fuente 5: Elaboración propia (2024). Captura del segundo programa considerado.

Esta variación supone una mejora significativa con respecto a la anterior. Al utilizar un operador *or*, garantiza la evaluación con una sola verificación. Además, un solo "!=" se utiliza para ambos casos, haciéndolo más eficiente. Cabe destacar que esta variación responde correctamente a la prueba del programa. En consecuencia, ambos programas tienen el mismo comportamiento la tabla siguiente hace referencia a la respuesta ambos casos.

Ilustración 6 Captura del código resultante e implementado en el programa

```
if (!(num[i] == 0 || num[i] == 1)) {  
    return false;  
}  
}
```

Fuente 6: Elaboración propia (2024). Captura del código resultante.

Ilustración 7 Tabla de comportamiento de los dos últimos programas

VALORES DE ENTRADA	
<code>int[] nums = {1,9,5,1};</code>	<code>int[] nums = {1,0,0,1};</code>
RESPUESTA	
<code>false</code>	<code>true</code>

Fuente 7: Elaboración propia (2024). Tabla del canvas que resume el comportamiento de ambos códigos. Véase anexos

## APARTADO 2: VOID COPY (STRING FROM, INT[] TO)

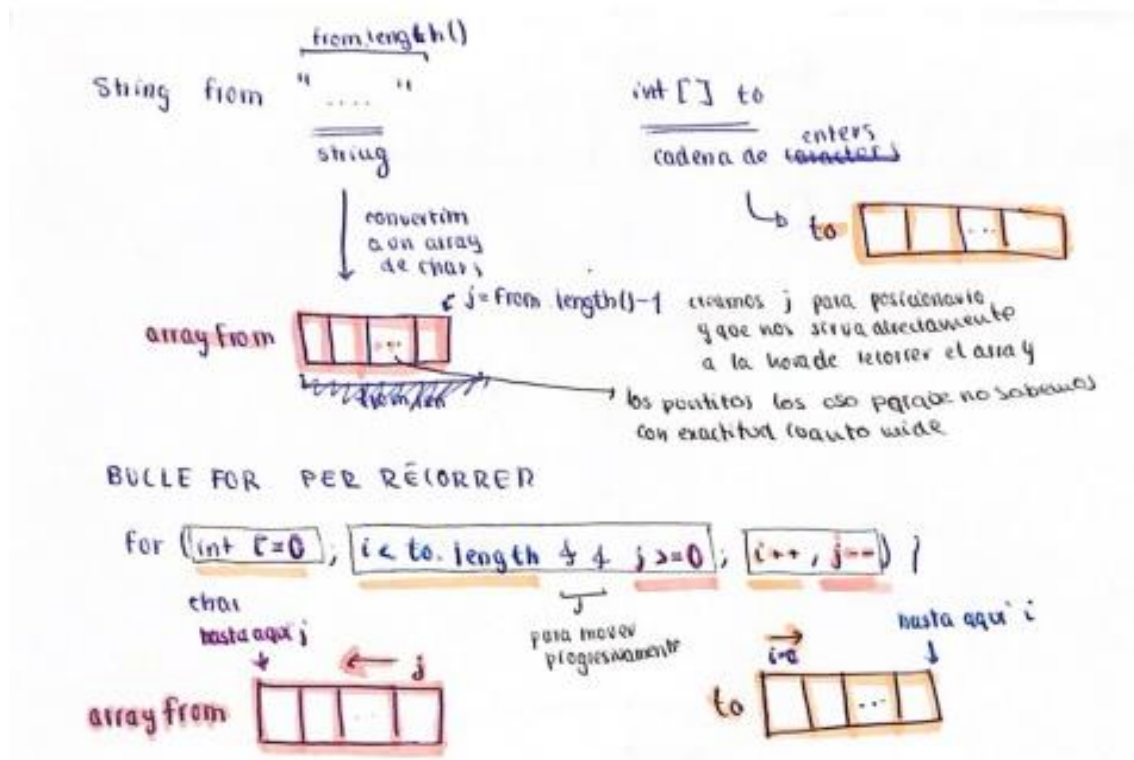
El objetivo de esta sección es realizar un recorrido inverso de la cadena *from* y posteriormente copiar los elementos de izquierda a derecha en el array de enteros *to*. Es importante tener en cuenta que este proceso está limitado por la capacidad máxima del array *to*, y cualquier espacio restante sin llenar se rellenará con ceros. Además, este mecanismo de copia sólo acepta valores binarios, específicamente 1 o 0.

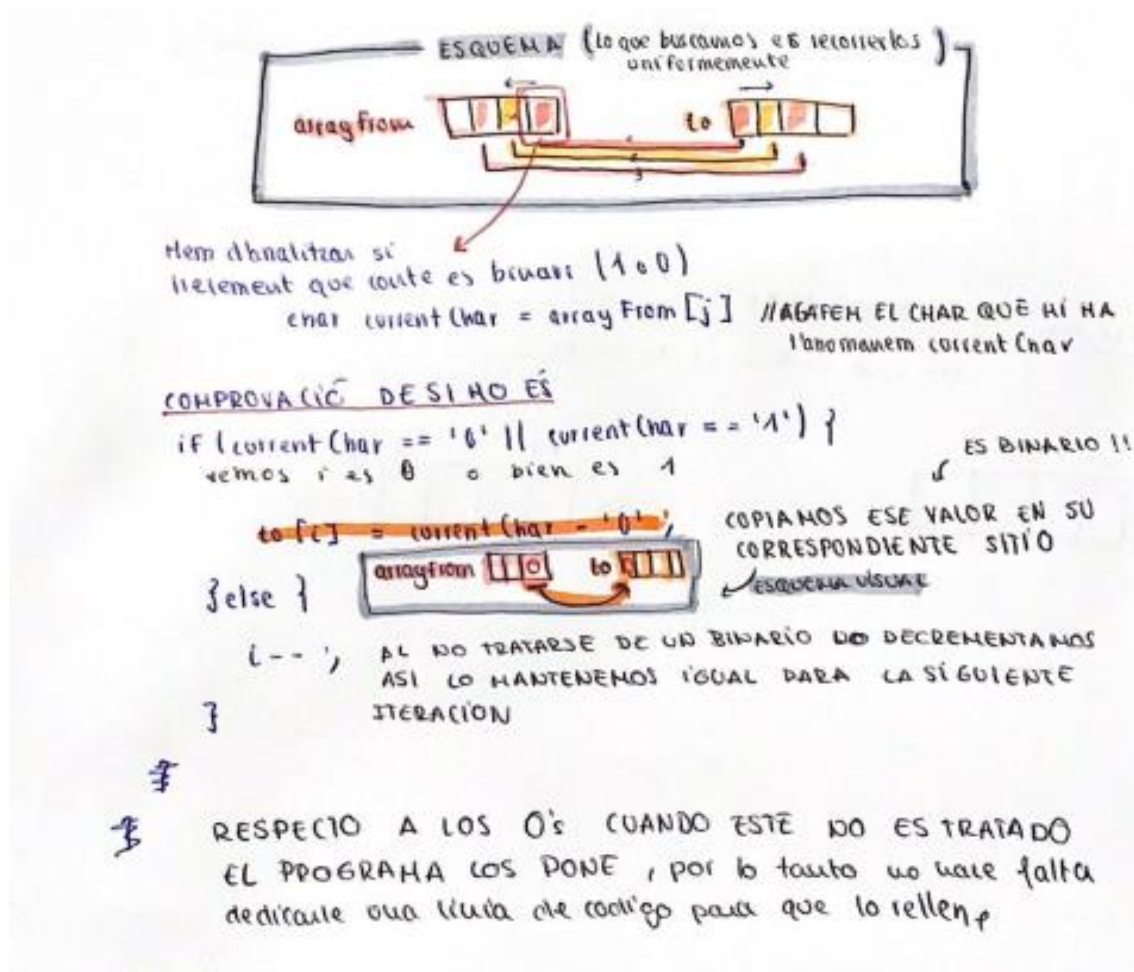
Ilustración 8 Código implementado en la función copy

```
public void copy(String from, int[] to) {  
    //throw new UnsupportedOperationException("apartado 2");  
    char[] arrayFrom = from.toCharArray();  
    int j = from.length() - 1;  
  
    for (int i = 0; i < to.length && j >= 0; i++, j--) {  
        char currentChar = arrayFrom[j];  
        if (currentChar == '0' || currentChar == '1') {  
            to[i] = currentChar - '0';  
        } else {  
            i--;  
        }  
    }  
}
```

Fuente 8: Elaboración propia (2024). Captura del código implementado

La explicación se hará mediante el uso de diagramas que representaran los bucles el comportamiento de los bucles, junto a un respectivo color para facilitar su ubicación además de comentarios adicionales y esquemas simplificados que visualizan la acción realizada.





Una observación del programa implementado se refiere al uso de un bucle *for* que atraviesa dos arrays simultáneamente. Inicialmente, esto se logró mediante el uso de bucles y condicional (Ilustración 9). Sin embargo, más tarde se consolidó en un único bucle *for* (Ilustración 8), ya que se descubrió [1] que atravesar dos matrices simultáneamente reducía la complejidad del código y mejoraba su eficiencia. Aunque el código cumplió con éxito su función, consideré implementar el otro método porque a mi parecer es más fácil de comprender. Además, aunque estos programas no requieren un alto nivel de abstracción, es posible que pueda comprender su funcionamiento en el futuro.

Ilustración 9 Código previo al implementado en copy

```
int j = from.length() - 1;
char[] charArray = from.toCharArray();

for (int i = 0; i < to.length && j >= 0; i++, j--) {
    while (j >= 0 && !(charArray[j] == '0' || charArray[j] == '1')) {
        j--;
    }

    if (j >= 0) {
        to[i] = charArray[j] - '0';
    }
}
```

Fuente 9: Elaboración propia(2024). Captura del código previo.

## APARTADO 3: INT[] NARROW (INT []FROM, INT TOLENGTH)

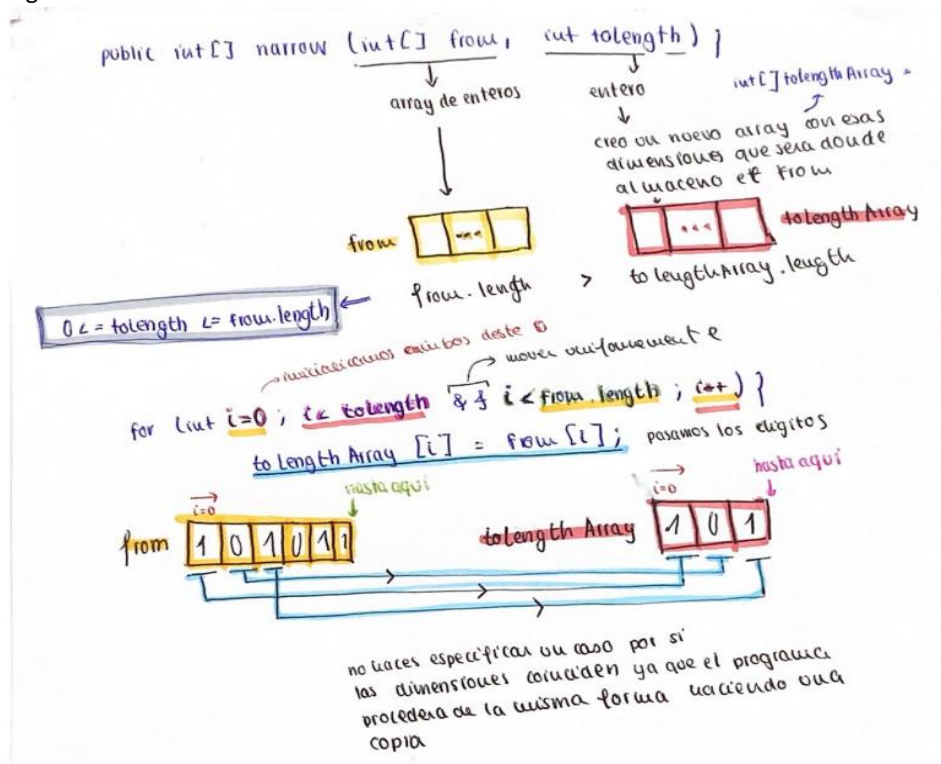
A partir de esta sección, discutiremos las conversiones en código. Específicamente, profundizaremos en la función *narrow* que toma un array de números enteros llamados *from* y un número entero *toLength* como dimensión, que será estrictamente menor a la de *from*. El objetivo principal de la función es convertir el array *from* en otro array con dimensiones de *toLength*, en la que inserta los números enteros de *from*. Esta función se puede implementar en varios escenarios, como cuando las longitudes son idénticas, se requiere que el programa realice una copia. De manera similar, si *toLength* es 0, el programa devolverá un array vacío.

Ilustración 10 Código implementado en la función narrow

```
public int[] narrow(int[] from, int toLength) {  
    // throw new UnsupportedOperationException("apartado 3");  
    int[] toLengthArray = new int[toLength];  
  
    for (int i = 0; i < toLength && i < from.length; i++) {  
        toLengthArray[i] = from[i];  
    }  
  
    return toLengthArray;  
}
```

Fuente 10: Elaboración propia (2024). Captura del código implementado en narrow.

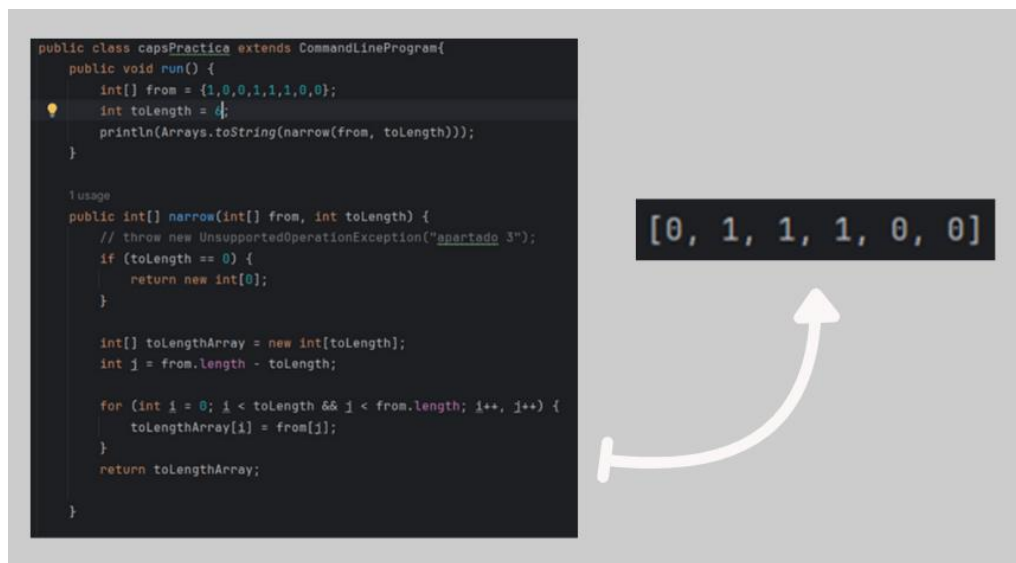
Del mismo modo que el anterior apartado, explicaremos el funcionamiento del programa mediante el uso de diagramas.





En cuanto a los desafíos que presentó el apartado, me gustaría resaltar que tomé la iniciativa de crear un programa de prueba antes de implementarlo en *IntegralTypes*. Sin embargo, la función no sucedió como se esperaba cuando se probó con *IntegralTypes*. Aunque inicialmente fue frustrante, esto me brindó una valiosa oportunidad para profundizar mi comprensión de las funciones de depuración (*debug* [2]) y *test*. Esta experiencia me ha brindado una base sólida sobre la cual construir en futuros proyectos de programación. En consecuencia el proximo programa insertado es el codigo que implemente en el código implementado en el porgrama prueba. Para abreviar, el código derivado a la malinterpretación propia del enunciado

Ilustración 11 Código erróneo implementado en el programa prueba



Fuente 11: Elaboración propia (2024). Tabla del canvas que expone las capturas de la estructura del código y programa prueba y a la derecha el resultado al compilar. Véase anexos.

## APARTADO 4:

# INT [] WIDEN(INT[] NUM, INT TOLENGTH)

El siguiente enunciado guarda cierta semejanza con la función *narrow*, aunque con un efecto inverso. Se trata de la conversión de la función de ampliación a un tipo con mayor número de bits. El objetivo principal es preservar el valor numérico de los datos, para lo cual el bit más significativo es el bit de signo que representa el valor entero original. Por tanto, es fundamental considerar la importancia del bit de signo en el proceso de conversión.

Ilustración 12 Código implementado en widen

```
public int[] widen(int[] from, int toLength) {  
    int[] result = new int[toLength];  
  
    if (from.length != 0) {  
        int importantBit = from[from.length - 1];  
  
        for (int i = 0; i < from.length; i++) {  
            result[i] = from[i];  
        }  
  
        for (int i = from.length; i < toLength; i++) {  
            result[i] = importantBit;  
        }  
    }  
    return result;  
}
```

Fuente 12: Elaboración propia (2024). Captura del código implementado en widen.

La explicación del código se hará mediante diagramas, del mismo modo se explica la observación distintiva entre la conversión respecto los arrays y strings.

```
public int[] widen (int[] from, int toLength)
```

array de enteros

entero

creamos un array

`int[] result = new int [toLength]`  
en la dimension pedida

from

result

`0 <= from.length <= toLength`

from.length

< result.length

`if (from.length != 0) {` → nos aseguramos que from tenga contenido si no lo retornamos  
}  
return result  
y retornamos vacío porque esta vacío

`int importantBit = from[from.length - 1];`

→ OBSERVACION

En el string el bit significativo esta al inicio y en el array al final

STRING  
"0101101"

ARRAY  
1011010

```
for (int i = 0; i < from.length; i++) {  
    result[i] = from[i];  
}
```

COPIA / TRASPASO DE ELEMENTOS

```
for (int i = from.length; i < toLength; i++) {  
    result[i] = from[i];  
    result[i] = importantBit;  
}
```

RELLENO CON EL BIT SI SIGNIFICATIVO

## APARTADO 5: CAST (INT[] FROM, INT TOLENGTH)

La función *cast* realiza un análisis de los parámetros para determinar la acción adecuada a realizar, ya sea para realizar una operación *narrow* o *widen*. Independientemente del curso de acción, no es necesario devolver la matriz original. Cabe señalar que una vez completada esta tarea, se podrán utilizar las siguientes conversiones:

```
public int[] toByte(int[] from) {  
    return cast(from, BYTE_SIZE);  
}  
  
public int[] toShort(int[] from) {  
    return cast(from, SHORT_SIZE);  
}  
  
public int[] toInteger(int[] from) {  
    return cast(from, INTEGER_SIZE);  
}  
  
public int[] toLong(int[] from) {  
    return cast(from, LONG_SIZE);  
}
```

El proceso de implementación del código actual no ha sido más complicado en comparación con esfuerzos anteriores. El objetivo principal ha sido garantizar que los parámetros de entrada se evalúen minuciosamente mediante la ejecución de un condicional if.

Ilustración 13 Código implementado en cast

```
public int[] cast(int[] from, int toLength) {  
    // throw new UnsupportedOperationException("apartado 5");  
    if (from.length > toLength) return narrow(from, toLength);  
    return widen(from, toLength);  
}
```

Fuente 13: Elaboración propia (2024). Captura del código implementado en cast.

## APARTADO 6:

### INT[] AND(INT[] ARG1, INT[] ARG2)

El apartado consiste en un cálculo que se basa en operaciones lógicas, específicamente el operador *and*. Este operador se utiliza para realizar una operación de multiplicación elemento a elemento. Para ejecutar este proceso, es crucial considerar los parámetros de entrada y sus dimensiones. Si los parámetros de entrada se encuentran dentro de las dimensiones de BYTE, SHORT o INTEGER, se convertirán a un entero de 32 bits. Sin embargo, si las dimensiones requeridas son mayores, se transformarán en un entero de 64 bits de longitud. La salida del proceso debe tener las mismas dimensiones que los parámetros de entrada

Ilustración 14 Código implementado en *and*

```
public int[] and(int[] arg1, int[] arg2) {  
  
    int[] multResult;  
  
    if ( arg1.length > INTEGER_SIZE || arg2.length > INTEGER_SIZE) {  
        multResult = new int[LONG_SIZE];  
        arg1 = toLong(arg1);  
        arg2 = toLong(arg2);  
    } else {  
        multResult = new int[INTEGER_SIZE];  
        arg1 = toInteger(arg1);  
        arg2 = toInteger(arg2);  
    }  
  
    for (int i = 0; i < arg1.length; i++) {  
        multResult[i] = arg1[i] * arg2[i];  
    }  
  
    return multResult;  
}
```

Fuente 14: Elaboración propia (2024). Captura del código implementado en *and*.

En el siguiente apartado pretendemos explicar el código que se ha implementado mediante diagramas similares a los casos anteriores. Sin embargo, es importante señalar que este código en particular implica un proceso complejo, que ha dado lugar a diversas interpretaciones y métodos de procedimiento en determinadas circunstancias. Estos enfoques divergentes se detallarán en la siguiente sección, utilizando un tono violeta para distinguirlos del contenido principal.

```

public int[] and (int[] arg1, int[] arg2) {
    Declaro la el array
    de la resultante
    de la multiplicación
    int[] multResult;
    if (arg1.length > INTEGER_SIZE || arg2.length > INTEGER_SIZE) {
        // no acordamos dimensiones
        // porque no sabemos los bits de
        // arg1 i arg2
        // Podríamos usar los 32 pero reusamos la constante definida
        // de INTEGER_SIZE al inicio del programa Integer-Types
    }
    // Este método que implemente fue el
    if (arg1.length <= INTEGER_SIZE && arg2.length <= INTEGER_SIZE)
        pero este era significativamente mejorable así que de aquí derive la condición implementada
        multResult = new int[LONG_SIZE]; ya definimos las dimensiones de la multResult
        arg1 = toLong(arg1);
        arg2 = toLong(arg2);
        // Respecto a la implementación de la
        // función toLong. Antes de darme cuenta, había
        // implementado otra vez una operación de
        // transformación. Asimismo en el toInteger
        // convertí los argumentos a su respectiva dimensión
        En ambas funciones toInteger i toLong llaman a la función
        así que se encarga de la conversión respectiva usando el BigInteger y
        width
    } else {
        // así sigue la misma estructura que la anterior pero en este
        // caso trataremos dimensiones de 32 bits (INTEGER_SIZE)
    }
    for (int i=0; i<arg1.length; i++) {
        En este bucle for también llegue a hacerlo dos veces ya que basaba
        recorrer ambos pero Esto es debido a una mal organización de
        pasos que trataremos más adelante
        for (int i=0; i<arg1.length; i++) {
            multResult[i] = arg1[i] * arg2[i];
            // Recorremos los arrays y aprovechamos
            // solo un iterador ya que se recorren
            // en el mismo sentido y misma
            // longitud
        }
    }
    Cabe mencionar que llegue a considerar una operación por función
    booleanas ya que había una patron en operac. funciones
    Más adelante trataremos estos casos de errores.

```

Mientras escribía el código, encontré algunos desafíos para determinar el mejor enfoque para estructurar el proceso. Sin embargo, aprendí mucho de la experiencia y pude desarrollar diferentes interpretaciones y métodos que finalmente me ayudaron a crear una solución eficaz. Aunque al principio me costó establecer una pauta fija. Por ejemplo, llegué a considerar en operar primero y luego transformar a los bits correspondientes, partir de esta idea derivó a la realización de bucles y muchas funciones auxiliares. El proceso de *prueba y error* que me comportado este apartado a significado hacer un cambio que mejoró significativamente el proceso en general. Además, comprendí mejor las instrucciones y pude corregir mi malentendido sobre la función de multiplicación<sup>1</sup>, lo que resultó en una implementación más precisa.

Ilustración 15 Código de multiplicar con booleanos

```
private boolean multiply (int[] arg1, int[] arg2) {
    for (int i = 0; i < arg1.length; i++) {
        for(int j = 0; j < arg2.length; j++) {
            if(arg1[i] == 0 || arg2[j] == 0) {
                return false;
            }
        }
    }
    return true;
}
```

Fuente 15: Elaboración propia (2024). Captura de la función booleana de multiplicar.

Ilustración 16 Tabla y explicación de la multiplicación con booleanos

0 * 0	=	0
0 * 1	=	0
1 * 0	=	0
1 * 1	=	1

**FALSE**

**TRUE**

En una función aparte en caso de false insertaría un 0 y en caso de true insertaría un 1. Esto es debido al patrón que hay en la multiplicación, la presencia de un 0 ya da como resultado 0

Fuente 16: Elaboración propia (2024). Tabla de canvas sobre el patrón encontrado y como este se procedía dentro de una función booleana de multiplicación.

<sup>1</sup> Esto deriva a que interprete que el operar booleanos significaba que debía realizar una función auxiliar de multiplicación que retornara true o false.

## APARTADO 7: OR (INT[] ARG1, INT ARG2)

En este ejercicio el objetivo es conseguir el mismo resultado que el anterior, pero utilizando la suma en lugar de la multiplicación. Se repitió la misma estructura y el único inconveniente que surgió fue la suma de 1 + 1, que equivale a 2 (10 en binario). Esto se resolvió empleando un *if* para sustituir la respuesta apropiadamente.

Ilustración 17 Código implementado en or

```
public int[] or(int[] arg1, int[] arg2) {  
    //throw new UnsupportedOperationException("apartado 7");  
    int[] sumResult;  
    if ( arg1.length > INTEGER_SIZE || arg2.length > INTEGER_SIZE) {  
        sumResult = new int[LONG_SIZE];  
        arg1 = toLong(arg1);  
        arg2 = toLong(arg2);  
    } else {  
        sumResult = new int[INTEGER_SIZE];  
        arg1 = toInteger(arg1);  
        arg2 = toInteger(arg2);  
    }  
  
    for (int i = 0; i < arg1.length; i++) {  
        sumResult[i] = arg1[i] + arg2[i];  
        if ( sumResult[i] == 2) {  
            sumResult[i] = 1;  
        }  
    }  
    return sumResult;  
}
```

Fuente 17: Elaboración propia (2024). Captura del código implementado en or.



## APARTADO 8:

### INT[] LEFT-SHIFT(INT[] NUM, INT NUMPOS)

Esta sección pertenece al proceso de desplazamiento a la izquierda de un número determinado en un número predeterminado de posiciones, cuyo valor está dictado por la variable entera *numPos*. Las posiciones restantes en el lado derecho del número resultante se rellenan con ceros. Es importante tener en cuenta que la transformación del número está influenciada por si se puede representar como un valor de 32 o 64 bits. Como tal, se debe utilizar el % del operador para garantizar que se ejecute un turno apropiado del valor de *numPos*.

Ilustración 18 Código implementado en leftShift

```
public int[] leftShift(int[] num, int numPos) {  
    //throw new UnsupportedOperationException("apartado 8");  
  
    numPos = getNumPos(num, numPos);  
    int[] result = new int[num.length + numPos];  
  
    for (int i = 0; i < num.length; i++) {  
        result[i + numPos] = num[i];  
    }  
  
    return cast(result, isLong(num) ? LONG_SIZE : INTEGER_SIZE);  
}
```

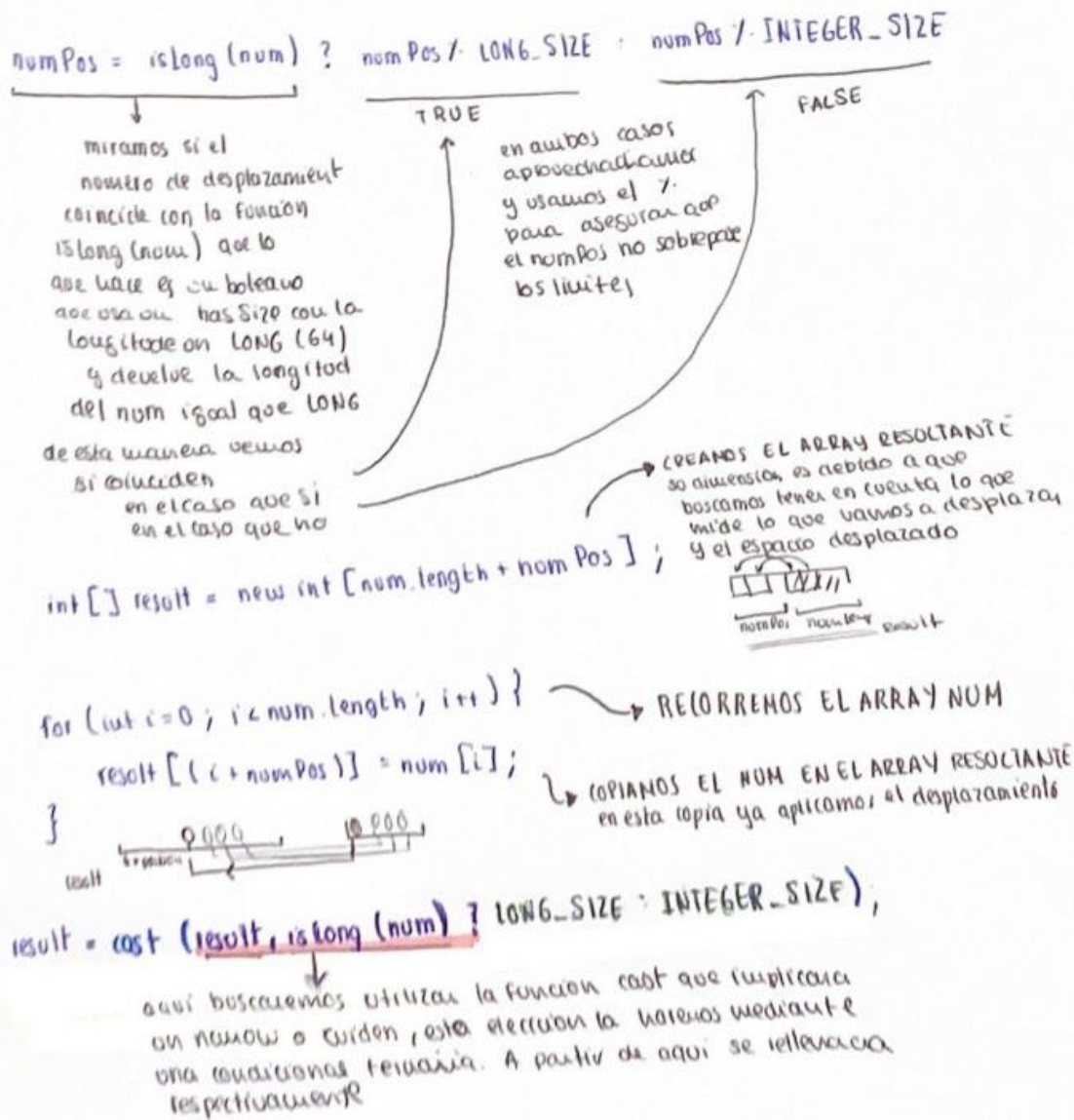
Ilustración 18: Elaboración propia (2024). Captura del código implementado en leftShift

Con el fin de ilustrar este ejercicio, cabe destacar la utilización de semejantes iteraciones de la declaración *if* [3]. Esto tiene como objetivo simplificar el código y mejorar su atractivo estético para las preferencias personales. En consecuencia, procederemos a exponer el enunciado mediante el empleo de diagramas para facilitar la comprensión visual de su ejecución.

Diagram illustrating the parameters of the `leftShift` method:

```
public int[] leftShift (int[] num, int numPos) {
```

The parameter `int[] num` is annotated as "array de enteros" (array of integers). The parameter `int numPos` is annotated as "entero (posiciones para mover)" (integer (positions to move)).



## APARTADO 9: UNSIGNEDRIGHTSHIFT(INT[] NUM, INT NUMPOS)

Esta sección tiene el mismo propósito que la sección 8; sin embargo, se diferencia en que el desplazamiento se realiza en sentido contrario, concretamente, hacia la derecha. En este escenario, las posiciones de la izquierda se rellenan con ceros. Si el bit más significativo se establece en 1, las posiciones se llenan con 1, pero con el número de posiciones especificado en *numPos*.

Ilustración 19 Código implementado en `unsignedRightShift`

```
public int[] unsignedRightShift(int[] num, int numPos) {  
    //throw new UnsupportedOperationException("apartado 9");  
  
    numPos = isLong(num) ? numPos % LONG_SIZE : numPos % INTEGER_SIZE;  
    int Size = isLong(num) ? LONG_SIZE : INTEGER_SIZE;  
  
    int[] result = new int[Size];  
    int[] newNum = widen(num, Size);  
    for (int i = numPos; i < result.length; i++) {  
        result[i - numPos] = newNum[i];  
    }  
  
    return result;  
}
```

Fuente 19: Elaboración propia (2024). Captura del código implementado en `unsignedRightShift`

El código actual sigue una estructura similar a la versión anterior. Implementa el mismo procedimiento, al respecto de *numPos* en los casos en que excede la longitud de INTEGER (32 bits) o LONG (64 bits). Sin embargo, esta versión permite la selección de una constante para especificar la longitud del array resultante. La función de *widen* se reutilizará para garantizar que la función numérica se complete con el tamaño adecuado. Posteriormente, un bucle *for* iterará sobre la matriz resultante a partir de *numPos*, que representará el número de desplazamiento. De esta forma se cubren las posiciones iniciales indicadas por *numPos*. Finalmente, se aplica un desplazamiento de matriz y se devuelve el resultado.

## APARTADO 10: INT[] SIGNEDRIGHTSHIFT (INT[] NUM, INT NUMPOS)

En este apartado también se realiza un desplazamiento que se realiza con la derecha, pero en este caso a diferencia con el anterior todo a partir de la izquierda se rellena con el bit significativo declarado en el array de enteros de *num*.

Ilustración 20 Código implementado en *signedRightShift*

```
public int[] signedRightShift(int[] num, int numPos) {  
    // throw new UnsupportedOperationException("apartado 10");  
  
    numPos = getNumPos(num, numPos);  
    int[] result = new int[num.length - numPos];  
  
    for (int i = 0; i < result.length; i++) {  
        result[i] = num[i + numPos];  
    }  
  
    return widen(result, isLong(num) ? LONG_SIZE : INTEGER_SIZE);  
}
```

Fuente 20: Elaboración propia(2024). Captura del código implementado en *signedRightShift*

Una vez realizada la función, me di cuenta de que hay un código que se repite en los tres últimos apartados. Se trata del código;

Ilustración 21 Código del *numPos*

```
numPos = isLong(num) ? numPos % LONG_SIZE : numPos % INTEGER_SIZE;
```

Fuente 21: Elaboración propia(2024). Captura del código *numPos*

Con el objetivo de optimizar y simplificar el resto de las funciones cree una única función denominada *getNumPos* de tipo privado que haga esta tarea, de este modo en los tres últimos casos hago una llamada ahorrando repetir la misma operación.

Ilustración 22 Código de la función *numPos*

```
private int getNumPos(int[] num, int numPos) {  
    return isLong(num) ? numPos % LONG_SIZE : numPos % INTEGER_SIZE;  
}
```

Fuente 22: Elaboración propia(2024). Captura de la función *numPos*

El proceso para llegar a este resultado no fue tan complicado a comparación de los otros, debido a que este código fue resultante en lo que llegue a realizar el código del anterior apartado, por ese motivo la dificultad del anterior nacía del hecho de añadir los primeros 0 en las posiciones de la izquierda según indicaba el *numPos*. En consecuencia, la estructura llega a asimilarse al anterior con la única excepción que no vi necesario crear una variable de medida ya que el uso del *widen* se encargaría de aplicar el relleno con el bit significativo y la única tarea a realizar sería hacer un traspaso de *num* con el debido desplazamiento, esta acción la realice con el bucle *for* (Ilustración 20).

## CONCLUSIONES

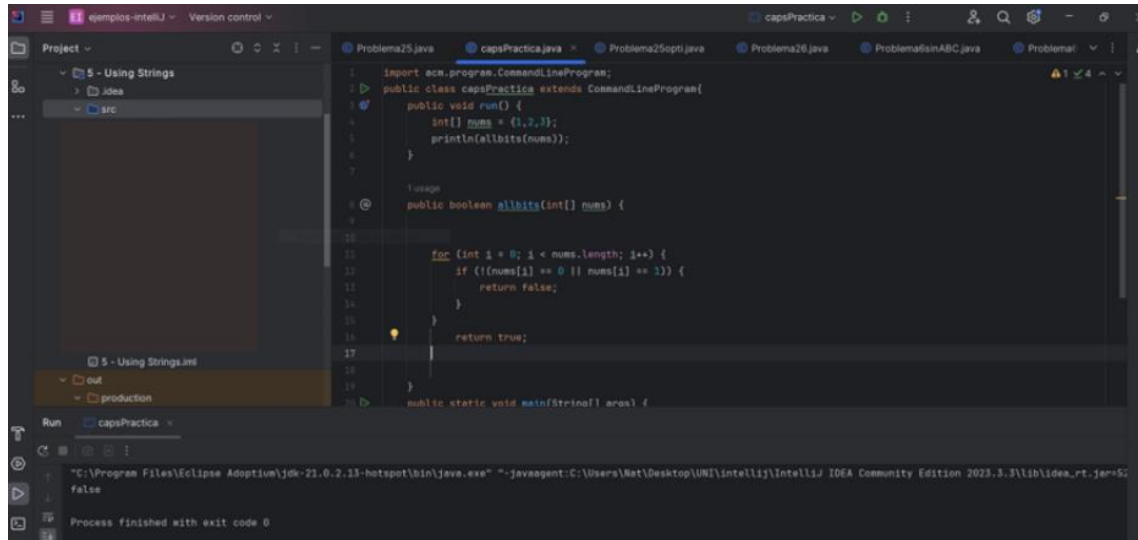
Estoy encantada de haber tenido la oportunidad de trabajar con Java, un lenguaje que no había utilizado antes. Esta práctica ha sido una experiencia valiosa, ya que me ha permitido desarrollar significativamente mis habilidades en programación. El proceso de profundizar en los niveles de abstracción, investigar recursos y metodologías y aprender de *prueba y error* ha sido a la vez desafiante y gratificante, y estoy agradecida por el crecimiento personal que me ha brindado.

Una de las lecciones más valiosas que he aprendido de esta experiencia es la importancia de adoptar un enfoque organizado para la codificación. Al estructurar y seguir un plan bien definido para implementar cada código, he podido lograr mejores resultados y mejorar mi comprensión general del lenguaje. Además, comprender el propósito de cada función en *IntegralTypes* y profundizar en los procedimientos de depuración ha mejorado aún más mis conocimientos de programación.

## REFERENCIAS

- [1] R. Yadav, «Java for bucle con dos variables,» DelftStack, Desconocida, 2023.
- [2] JetBrains s.r.o, «JetBrains,» 22 Febrero 2024. [En línea]. Available: <https://www.jetbrains.com/help/idea/debugging-your-first-java-application.html#setting-breakpoints>. [Último acceso: Marzo 2024].
- [3] M. C. Berenguer, «Java Autodidacta,» 2019. [En línea]. Available: <https://javautodidacta.es/if-else-en-java/>. [Último acceso: Marzo 2024].

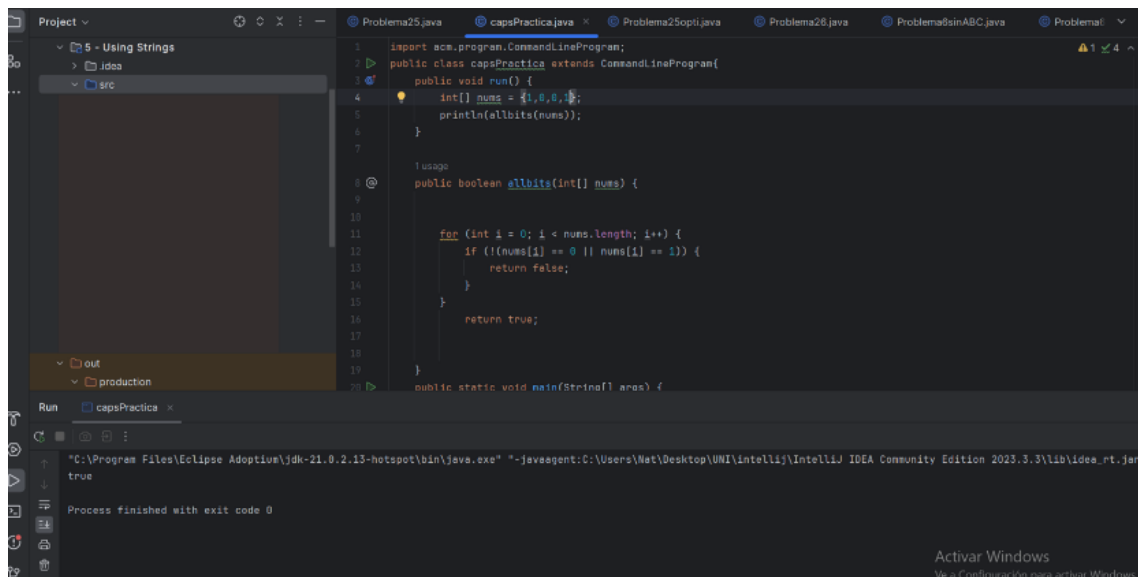
## ANEXO DE IMÁGENES



This screenshot shows the IntelliJ IDEA interface with the file `capsPractica.java` open. The code defines a class `capsPractica` that extends `CommandLineProgram`. It includes a `run()` method that initializes an array `nums = {1, 2, 3}` and calls `println(allbits(nums))`. The `allbits` method is a static utility that checks if all bits in the array are 1. The Run console at the bottom shows the command executed and the output `false`.

```
1 import acm.program.CommandLineProgram;
2 public class capsPractica extends CommandLineProgram {
3     public void run() {
4         int[] nums = {1, 2, 3};
5         println(allbits(nums));
6     }
7
8     // Usage
9     public boolean allbits(int[] nums) {
10
11         for (int i = 0; i < nums.length; i++) {
12             if (!(nums[i] == 0 || nums[i] == 1)) {
13                 return false;
14             }
15         }
16         return true;
17     }
18
19     public static void main(String[] args) {
```

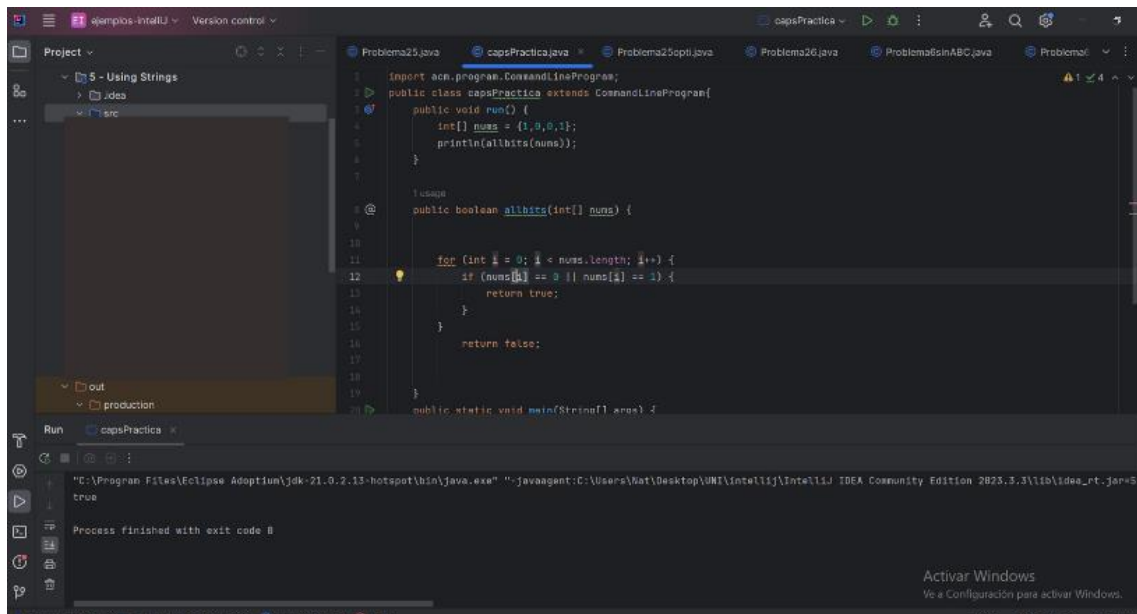
Run console output:  
"C:\Program Files\Eclipse Adoptium\jdk-21.0.2-hotspot\bin\java.exe" -javaagent:C:\Users\Nati\Desktop\UNI\IntelliJ\IntelliJ IDEA Community Edition 2023.3.3\lib\idea\_rt.jar=5;false  
Process finished with exit code 0



This screenshot shows the IntelliJ IDEA interface with the file `capsPractica.java` open. The code is the same as in the previous image, but the array `nums` is now initialized as `{1, 0, 0, 1}`. The Run console at the bottom shows the output `true`.

```
1 import acm.program.CommandLineProgram;
2 public class capsPractica extends CommandLineProgram {
3     public void run() {
4         int[] nums = {1, 0, 0, 1};
5         println(allbits(nums));
6     }
7
8     // Usage
9     public boolean allbits(int[] nums) {
10
11         for (int i = 0; i < nums.length; i++) {
12             if (!(nums[i] == 0 || nums[i] == 1)) {
13                 return false;
14             }
15         }
16         return true;
17     }
18
19     public static void main(String[] args) {
```

Run console output:  
"C:\Program Files\Eclipse Adoptium\jdk-21.0.2-hotspot\bin\java.exe" -javaagent:C:\Users\Nati\Desktop\UNI\IntelliJ\IntelliJ IDEA Community Edition 2023.3.3\lib\idea\_rt.jar=true  
Process finished with exit code 0



```
import acm.program.CommandLineProgram;

public class capsPractica extends CommandLineProgram {

    public void run() {
        int[] nums = {1, 0, 0, 1};
        println(allbits(nums));
    }

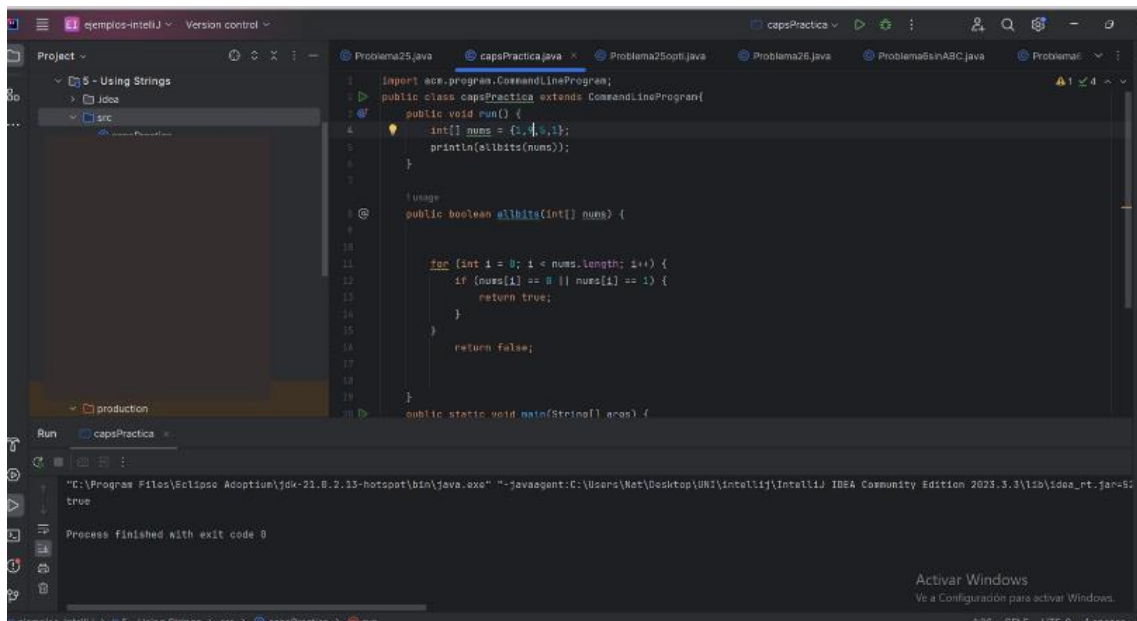
    // usage
    public boolean allbits(int[] nums) {
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == 0 || nums[i] == 1) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        new capsPractica().run();
    }
}
```

Run capsPractica

"C:\Program Files\Eclipse Adoptium\jdk-21.0.2-hotspot\bin\java.exe" "-javaagent:C:\Users\Naty\Desktop\UNI\IntelliJ\IntelliJ IDEA Community Edition 2023.3.3\lib\idea\_rt.jar=5121:21.0.2-hotspot\bin\java.exe" true

Process finished with exit code 0



```
import acm.program.CommandLineProgram;

public class capsPractica extends CommandLineProgram {

    public void run() {
        int[] nums = {1, 0, 0, 1};
        println(allbits(nums));
    }

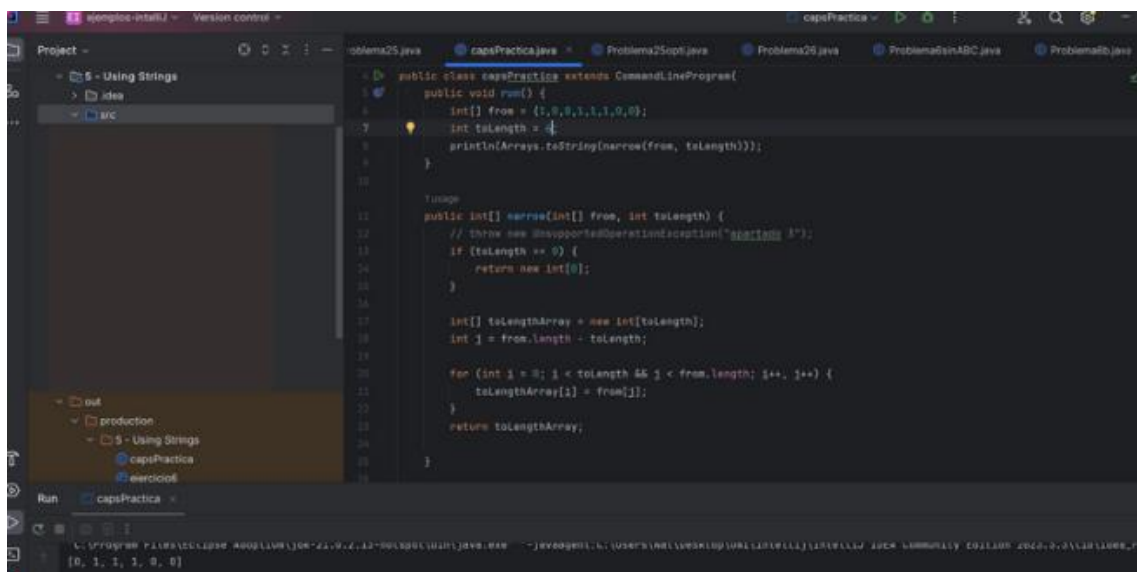
    // usage
    public boolean allbits(int[] nums) {
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == 0 || nums[i] == 1) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        new capsPractica().run();
    }
}
```

Run capsPractica

"C:\Program Files\Eclipse Adoptium\jdk-21.0.2-hotspot\bin\java.exe" "-javaagent:C:\Users\Naty\Desktop\UNI\IntelliJ\IntelliJ IDEA Community Edition 2023.3.3\lib\idea\_rt.jar=5121:21.0.2-hotspot\bin\java.exe" true

Process finished with exit code 0



```
import acm.program.CommandLineProgram;

public class capsPractica extends CommandLineProgram {

    public void run() {
        int[] from = {1, 0, 0, 1, 1, 0, 0};
        int toLength = 4;
        println(Arrays.toString(narrow(from, toLength)));
    }

    // usage
    public int[] narrow(int[] from, int toLength) {
        // throw new UnsupportedOperationException("not implemented");
        if (toLength > 9) {
            return new int[0];
        }

        int[] toLengthArray = new int[toLength];
        int i = from.length - toLength;

        for (int j = 0; j < toLength && j < from.length; j++, i++) {
            toLengthArray[i] = from[j];
        }

        return toLengthArray;
    }

    public static void main(String[] args) {
        new capsPractica().run();
    }
}
```

Run capsPractica

"C:\Program Files\Eclipse Adoptium\jdk-21.0.2-hotspot\bin\java.exe" "-javaagent:C:\Users\Naty\Desktop\UNI\IntelliJ\IntelliJ IDEA Community Edition 2023.3.3\lib\idea\_rt.jar=5121:21.0.2-hotspot\bin\java.exe" true

Process finished with exit code 0



```

1  import aca.program.CommandLineProgram;
2  public class capsPractice extends CommandLineProgram {
3      public void run() {
4          int[] nums = {1, 9, 5, 1};
5          println(allbits(nums));
6      }
7
8      @Usage
9      public boolean allbits(int[] nums) {
10
11         for (int i = 0; i < nums.length; i++) {
12             if (nums[i] != 0 && nums[i] != 1) {
13                 return false;
14             }
15         }
16         return true;
17     }
18
19     public static void main(String[] args) {
20

```

Run capsPractice

"C:\Program Files\Eclipse Adoptium\jdk-21.0.2-hotspot\bin\java.exe" "-javaagent:C:\Users\Nat\Desktop\UNI\IntelliJ\IntelliJ IDEA Community Edition 2023.3.3\lib\idea\_rt.jar=52:false"

Process finished with exit code 0

```

1  public class capsPractice extends CommandLineProgram {
2      public void run() {
3          int[] from = {1, 0, 0, 1, 1};
4          int toLength = 4;
5          println(Arrays.toString(narrow(from, toLength)));
6      }
7
8      @Usage
9      public int[] narrow(int[] from, int toLength) {
10         // throw new UnsupportedOperationException("not implemented");
11         if (toLength == 0) {
12             return new int[0];
13         }
14
15         int[] toLengthArray = new int[toLength];
16         int j = from.length - toLength;
17
18         for (int i = 0; i < toLength && j < from.length; i++, j++) {
19             toLengthArray[i] = from[j];
20         }
21         return toLengthArray;
22     }
23

```

Run capsPractice

"C:\Program Files\Eclipse Adoptium\jdk-21.0.2-hotspot\bin\java.exe" "-javaagent:C:\Users\Nat\Desktop\UNI\IntelliJ\IntelliJ IDEA Community Edition 2023.3.3\lib\idea\_rt.jar=52:false"

Process finished with exit code 0

```

1  public class capsPractice extends CommandLineProgram {
2      public void run() {
3          int[] from = {1, 0, 0, 1, 1};
4          int toLength = 4;
5          println(Arrays.toString(narrow(from, toLength)));
6      }
7
8      @Usage
9      public int[] narrow(int[] from, int toLength) {
10         // throw new UnsupportedOperationException("not implemented");
11         if (toLength == 0) {
12             return new int[0];
13         }
14
15         int[] toLengthArray = new int[toLength];
16         int j = from.length - toLength;
17
18         for (int i = 0; i < toLength && j < from.length; i++, j++) {
19             toLengthArray[i] = from[j];
20         }
21         return toLengthArray;
22     }
23

```

Run capsPractice

"C:\Program Files\Eclipse Adoptium\jdk-21.0.2-hotspot\bin\java.exe" "-javaagent:C:\Users\Nat\Desktop\UNI\IntelliJ\IntelliJ IDEA Community Edition 2023.3.3\lib\idea\_rt.jar=52:false"

Process finished with exit code 0