

2021.05.13 Java

<10진수를 2진수로 변환하는 법>

~~79~~

$2^0 = 1$	$2^7 = 128$
$2^1 = 2$	$2^8 = 256$
$2^2 = 4$	$2^9 = 512$
$2^3 = 8$	$2^{10} = 1024$
$2^4 = 16$	
$2^5 = 32$	
$2^6 = 64$	

$79 = 2^6 + 15$
 $15 = 2^3 + 7$
 $7 = 2^2 + 3$
 $3 = 2^1 + 1$

$1 = 2^0 + 0$
 $\therefore 79 \rightarrow \frac{1}{2^6} \frac{0}{2^5} \frac{0}{2^4} \frac{1}{2^3} \frac{1}{2^2} \frac{1}{2^1} \frac{1}{2^0}$
 ~~$\therefore 79 \rightarrow 1001111$~~

79는 이진수로

1001111

```
// 이 문제를 다루려면 2진수에 대한 이해가 필요함.  
// 십진수 10을 이진수로 변환해보면  
// 1. 10에 가장 근접하면서 10보다 작은 2^n을 찾는다 > 8  
// 2. 찾는 숫자는 10에 8을 뺀 값인 2를 적는다.  
// 3. 값이 0이 나올때까지 이 절차를 반복.  
// 4. 0이 된 이후 뺀 값들의 2^n에 해당하는 n 값들을 열거 >> 3, 1  
// 5. 구한 숫자들이 각각 이진수의 자리수에 해당한다.  
// 6. 2^3 2^2 2^1 2^0  
// 1 0 1 0  
// 7. 계산 >>> (2^3 * 1) + (2^2 * 0) + (2^1 * 1) + (2^0 * 0) = 10
```

```
// 2^0 + 2^1 + 2^2 + ... + 2^n = 2^(n+1) - 1

// 2진수 10101000 을 10진수로 바꿔보자!
// 2^7 + 2^5 + 2^3 = 8 + 32 + 128 = 168

// (2진수)11111100 을 10진수로 바꿔보자!
// (2진수)11111111 =(2진수)100000000 -1 = 2^0 + 2^1 + ... + 2^7 => 2^8 - 1 = 256 - 1 = 255
// (2진수)11111100 = (2진수)11111111 - (2^1+2^0)
// 255 - 3 = 252
```

특정 2진수들 쉽게 10진수로 바꾸는 Tip

```
// 21 ---> 16(2^4) + 4(2^2) + 1(2^0)
//          10101
// 1, 3, 5 번째 비트지만
// 실제 표현할때는 0번 비트, 2번 비트, 4번 비트로 표현해주도록 한다.

// 73 ---> 64(2^6) + 8(2^3) + 1(2^0)
//          1001001
// 🟡 0번 비트, 3번 비트, 6번 비트로 표현됨
```

2진수 표현 할 때 주의 할 점.

<비트연산자 AND &>

```
_2nd_BitAndTest.java × _1st_NonDuplicateWithoutArrayTest.java ×
1 public class _2nd_BitAndTest {
2     public static void main(String[] args) {
3         int num1 = 10, num2 = 8;
4         System.out.printf("%d & %d = %d\n", num1, num2, num1 & num2);
5         num2 = 138;
6         System.out.printf("%d & %d = %d\n", num1, num2, num1 & num2);
7     }
8 }
9 // & 이 비트연산자 AND
10 // 관계연산자에서는 && 형태로 나타냄.
11 // 10 ==> 1010
12 // 8 ==> 1000 AND
13 // -----
14 // 8 ==> 1000
15
16 // 138 ==> 10001010
17 // 10 ==> 1010 AND
18 // -----
19 // 10 ==> 00001010
```

Run: _2nd_BitAndTest ×

```
"C:\Program Files\Java\jdk-16\bin\java.exe" -javaagent:C:\Users\Samuel\Ap
10 & 8 = 8
10 & 138 = 10
Process finished with exit code 0
```

<비트연산자 OR | >

```
_2nd_BitAndTest.java × _2nd_BitOrTest.java × _1st_NonDuplicateWithoutArrayTest.java ×
1 public class _2nd_BitOrTest {
2     public static void main(String[] args) {
3         int num1 = 10, num2 = 5;
4         System.out.printf("%d | %d = %d\n", num1, num2, num1 | num2);
5         num2 = 136;
6         System.out.printf("%d | %d = %d\n", num1, num2, num1 | num2);
7     }
8 }
9 // | 는 비트연산자 OR
10 // 관계 연산자에서는 || 형태로 존재하였음
11 // 10 ==> 1010
12 // 5 ==> 0101 OR
13 // -----
14 // 15 ==> 1111
15
16 // 10 ==> 00001010
17 // 136 ==> 10001000 OR
18 // -----
19 // 138 ==> 10001010
20
21 // OR 연산은 합집합 개념, AND 연산은 교집합 개념
```

Run: _2nd_BitOrTest ×

```
"C:\Program Files\Java\jdk-16\bin\java.exe" -javaagent:C:\Users\Samuel\
10 | 5 = 15
10 | 136 = 138

Process finished with exit code 0
```

// 관계연산자 AND와 비트연산자 AND는 서로 동작 방식에 약간의 차이가 있다.
// 십진수 10과 십진수 5의 AND 연산은 아래와 같이 이루어진다.

```
// 1010 ---- 10
// 0101 ---- 5   AND
// -----
// 0000 ---- 0
```

```
// 1010 ---- 10
// 0101 ---- 5   OR
// -----
// 1111 ---- 15
```

// 비트 연산자 AND는 각 비트의 자리수가 1(참)인 녀석들끼리만 1(참)이 된다.
// 하나라도 0(거짓)이 있으면 해당 자리수는 0(거짓)이 된다.

// 비트 연산자 OR는 각 비트의 자리수중 하나라도 1(참)이 있으면 1(참)이 된다.
// 양쪽 모두 0(거짓)을 가지고 있는 경우에만 0(거짓)이 된다.

// Q: 비트연산자 OR 연산은 덧셈가요 ?

// A: NO

```
// 1010 - 10
```

```
// 0111 - 7   OR
```

```
// -----
```

```
// 1111 - 15 ==> 8 + 4 + 2 + 1
```

```
// 10000 (2^4) - 1 = 1111(2) = 15
```

<이동연산자 SHIFT>

```
public class _2nd_BitShift {  
    public static void main(String[] args) {  
        int num1 = 2, num2 = 5, num3 = 10;  
  
        //  $2^1 \times 2^5 = 2^6(64)$   
        System.out.printf("%d << %d = %d\n", num1, num2, num1 << num2);  
        //  $5 \times 2^5 = 160$   
        System.out.printf("%d << %d = %d\n", num2, num2, num2 << num2);  
        //  $10 \times 2^5 = 320$   
        System.out.printf("%d << %d = %d\n", num3, num2, num3 << num2);  
  
        //  $2^1 \times 2^2 = 2^3(8)$   
        System.out.printf("%d << %d = %d\n", num1, num1, num1 << num1);  
        //  $5 \times 2^2 = 20$   
        System.out.printf("%d << %d = %d\n", num2, num1, num2 << num1);  
        //  $10 \times 2^2 = 40$   
        System.out.printf("%d << %d = %d\n", num3, num1, num3 << num1);  
  
        //  $2^1 \times 2^{10} = 2^{11}(2048)$   
        System.out.printf("%d << %d = %d\n", num1, num3, num1 << num3);  
        //  $5 \times 2^{10} = 5120$   
        System.out.printf("%d << %d = %d\n", num2, num3, num2 << num3);  
        //  $10 \times 2^{10} = 10240$   
        System.out.printf("%d << %d = %d\n", num3, num3, num3 << num3);  
  
        // 왼쪽 쉬프트의 경우 단순히  $2^n$ 을 곱하면 되지만  
        // 오른쪽 쉬프트의 경우 단순히  $2^n$ 을 나누면 안된다.  
        //  $5 / 2^2 = 1.25$   
        // 결론: 오른쪽 쉬프트는  $2^n$ 으로 나누되 소수점을 버려야 한다.  
        System.out.printf("%d >> %d = %f\n", num2, num1, (float)(num2 >> num1));  
  
        // 이유:  
        // 0101 ----> 5  
        // 0001 ----> 1  
  
        // 종합적 결론:  
        // 쉬프트 연산은  $2^n$ 을 곱하거나 나눈다.  
        // 안타깝게도 쉬프트 연산은 정수형끼리밖에 안된다.  
        // 최근에 나온 휴대폰 전용 ARM 프로세서에서는 소수점에 대한 쉬프트 연산을 지원하기도 한다.  
    }  
}
```

<< 이동연산자와

>> 이동연산자 계산시 주의 할 것 인지

```
_2nd_BitShift x
"C:\Program Files\Java\jdk-16\bin\ja
2 << 5 = 64
5 << 5 = 160
10 << 5 = 320
2 << 2 = 8
5 << 2 = 20
10 << 2 = 40
2 << 10 = 2048
5 << 10 = 5120
10 << 10 = 10240
5 >> 2 = 1.0000000
|
Process finished with exit code 0
```

```
// 쉬프트 연산의 결과 (비트를 왼쪽으로 이동시킴)
// 1 << 2 ==> 2^0 x 2^2 = 4   (비트를 왼쪽으로 2칸 이동시킴)
// 1 << 4 ==> 2^0 x 2^4 = 16  (비트를 왼쪽으로 4칸 이동시킴)
// 1 << 8 ==> 2^0 x 2^8 = 256 (비트를 왼쪽으로 8칸)
// 1 << 9 ==> 2^0 x 2^9 = 512 (왼쪽으로 9칸)

//      10000   (1이라는 숫자를 왼쪽으로 2칸 이동시키면 ?)
//      1000000  (10^2 이 곱해진다)
//      10000   (1이라는 숫자를 왼쪽으로 4칸 이동시키면 ?)
// 1000000000   (10^4 이 곱해진다)

// 십진수기 때문에 위치가 이동할 때마다 10씩 곱해졌다.
// 이진수는 위치가 이동할 때마다 2씩 곱해진다.
```

<중복없는 Math.random()>

(단, 배열 사용 없이)

```
1 public class _1st_NonDuplicateWithoutArrayTest {
2     public static void main(String[] args) throws InterruptedException {
3
4         // 0 ~ 9까지의 숫자가 중복이 없게 나오도록(단, 배열 없이)
5         // 2진 비트 AND 연산자와 OR 연산자를 활용
6         // 또한 쉬프트 연산자를 함께 활용해서 각각의 비트를 채우는 형식으로 코드를 구현.
7         final int BIN = 1;
8         int testBit = 0;
9         int randNum;
10
11         for (int i = 0; i < 10; i++){
12             randNum = (int)(Math.random() * 10); // int형이므로 0 ~ 9 가 나올 것
13             // 나온 randNum에 대한 중복 판정을 어떻게 할 것인가 >> bit 연산
14             while((testBit & (BIN << randNum)) != 0){
15                 randNum = (int)(Math.random() * 10);
16             }
17             System.out.printf("randNum = %d\n", randNum);
18             testBit |= (BIN << randNum); // A |= B ==> A = A | B
19         }
20         Thread.sleep(500);
21     }
22 }
23
24 // 2^9 2^8 2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
25 // 1 0 0 0 0 0 0 0 0 0 2^9(512)
26 // 0 1 0 0 0 0 0 0 0 0 2^8(256)
27 // 0 0 1 0 0 0 0 0 0 0 2^7(128)
28 // 0 0 0 1 0 0 0 0 0 0 2^6(64)
29 // 0 0 0 0 1 0 0 0 0 0 2^5(32)
30 // 0 0 0 0 0 1 0 0 0 0 2^4(16)
31 // 0 0 0 0 0 0 1 0 0 0 2^3(8)
32 // 0 0 0 0 0 0 0 1 0 0 2^2(4)
33 // 0 0 0 0 0 0 0 0 1 0 2^1(2)
34 // 0 0 0 0 0 0 0 0 0 1 2^0(1)
```

```
_1st_NonDuplicateWithoutArrayTest
"C:\Program Files\Java\jdk-
randNum = 2
randNum = 9
randNum = 0
randNum = 8
randNum = 7
randNum = 1
randNum = 6
randNum = 3
randNum = 4
randNum = 5
|
Process finished with exit
```



```

DuplicateWithoutArrayTest.java × _2nd_BitShift.java × _1st_NonDuplicateWithoutArrayTest_forSelfStudy.java ×
public class _1st_NonDuplicateWithoutArrayTest_forSelfStudy {
    public static void main(String[] args) throws InterruptedException {

        final int BIN = 1;
        int testBit = 0;
        int randNum;

        for (int i = 0; i < 10; i++){
            randNum = (int)(Math.random() * 10);
            System.out.println("while 문 들어가기 전 randNum: " + randNum); // 나중에 지을 것

            while((testBit & (BIN << randNum)) != 0){
                System.out.println("중복된 randNum: " + randNum); // 나중에 지을 것
                randNum = (int)(Math.random() * 10);
            }

            System.out.printf("randNum = %d\n", randNum);
            testBit |= (BIN << randNum);
        }
        Thread.sleep( millis: 500);
        System.out.println("마지막 출력된 testbit는 2^9+2^8+2^7+.....2^1+2^0 = " + testBit);
    }
}

```

중간에 sout 사용해서 어떤 식으로 진행이 되는지 살펴보면

```

_1st_NonDuplicateWithoutArrayTest_forSelfStudy ×
"C:\Program Files\Java\jdk-16\bin\jav
while 문 들어가기 전 randNum: 7
randNum = 7
while 문 들어가기 전 randNum: 2
randNum = 2
while 문 들어가기 전 randNum: 4
randNum = 4
while 문 들어가기 전 randNum: 2
중복된 randNum: 2
중복된 randNum: 2
randNum = 6
while 문 들어가기 전 randNum: 5
randNum = 5
while 문 들어가기 전 randNum: 0
randNum = 0
while 문 들어가기 전 randNum: 0
중복된 randNum: 0
중복된 randNum: 5
randNum = 3

```

<for문 진입>

i = 0

randNum = 7

<while문 진입>

testbit = 0

$(BIN \ll randNum) = (2^0 \ll 7) = 2^7$

$\therefore 0 \& 2^7 = 0$

<while문 false - out, for문 return>

testBit |= (BIN << randNum)

>> testBit = testBit | (BIN << randNum)

testBit = 0 | $2^7 = 2^7$

<for문 반복>

```

_1st_NonDuplicateWithoutArrayTest_forSelfStudy x
"C:\Program Files\Java\jdk-16\bin\ja
while 문 들어가기 전 randNum: 7
randNum = 7
while 문 들어가기 전 randNum: 2
randNum = 2
while 문 들어가기 전 randNum: 4
randNum = 4
while 문 들어가기 전 randNum: 2
중복된 randNum: 2
중복된 randNum: 2
randNum = 6
while 문 들어가기 전 randNum: 5
randNum = 5
while 문 들어가기 전 randNum: 0
randNum = 0
while 문 들어가기 전 randNum: 0
중복된 randNum: 0
중복된 randNum: 5
randNum = 3

```

<for문 진입>

i = 1

randNum = 2

<while문 진입>

testbit = 2^7

$(BIN \ll randNum) = (2^0 \ll 2) = 2^2$

$$\therefore 2^7 \& 2^2 = 0$$

<while문 false - out, for문 return>

testBit = testBit | $(BIN \ll randNum)$

testBit = $2^7 \mid 2^2 = 2^7 + 2^2$

<for문 반복>

<for문 진입>

i = 2

randNum = 4

<while문 진입>

testbit = $2^7 + 2^2$

$(BIN \ll randNum) = (2^0 \ll 4) = 2^4$

$$\therefore (2^7 + 2^2) \& 2^4 = 0$$

<while문 false - out, for문 return>

testBit = testBit | $(BIN \ll randNum)$

testBit = $2^7 + 2^2 \mid 2^4 = 2^7 + 2^4 + 2^2$

<for문 반복>

```
_1st_NonDuplicateWithoutArrayTest_forSelfStudy ×
"C:\Program Files\Java\jdk-16\bin\jav
while 문 들어가기 전 randNum: 7
randNum = 7
while 문 들어가기 전 randNum: 2
randNum = 2
while 문 들어가기 전 randNum: 4
randNum = 4
while 문 들어가기 전 randNum: 2
중복된 randNum: 2
중복된 randNum: 2
randNum = 6
while 문 들어가기 전 randNum: 5
randNum = 5
while 문 들어가기 전 randNum: 0
randNum = 0
while 문 들어가기 전 randNum: 0
중복된 randNum: 0
중복된 randNum: 5
randNum = 3
```

이런 방법으로 중복된 것을
제외시키면서 i = 9까지 진행됨.

<for문 진입>

i = 3

randNum = 2

<while문 진입>

testbit = $2^7 + 2^4 + 2^2$

(BIN << randNum) = (2^0 << 2) = 2^2

$\therefore (2^7 + 2^4 + 2^2) \& 2^2 \neq 0$

while문 true 즉, 중복 발생 >> while문 실행
2번의 재실행에서 또 2가 나왔기 때문에 반복
3번째 재실행에서 randNum = 6이 나왔음.

$\therefore (2^7 + 2^4 + 2^2) \& 2^6 = 0$

<while문 false - out, for문 return>

testBit = testBit | (BIN << randNum)

testBit = $2^7 + 2^4 + 2^2 \mid 2^6$

= $2^7 + 2^6 + 2^4 + 2^2$

<for문 반복>

<Interrupt>

```
Quiz18_21.java × 4th Quiz23.java × _3rd_InterruptComment.java ×
public class _3rd_InterruptComment {
    public static void main(String[] args) throws InterruptedException {
        for(int i = 0 ; ; i++){
            if (i % 2 ==0){
                System.out.println("안녕 난 짝수야");
            } else {
                System.out.println("하이 난 홀수야");
            }
            Thread.sleep(500);
        }
    }
}

// Interrupt:
//     >> 하드웨어 개발자들이 주로 사용하는 단어.
//     >> 보통 Java나 GUI 개발자들 혹은 App 개발자들은 Event라고 표현한다.
//     >> 결국 Event와 Interrupt는 동의어
// Event:
//     >> 기본적으로 Event라는 것은 최우선적으로 처리해야 하는 작업으로
//     >> 어떤 작업보다도 우선순위가 높은 것들을 말 한다.
//     >> 마찬가지로 위에서 Thread.sleep()을 하는 작업도 일종의 Event(Interrupt)에 해당한다
//     >> 따라서 이 작업이 시작되면 다른 모든 작업들을 제쳐두고 이것을 최우선적으로 처리하게 된다.
//     >> 물론 조금 더 정확한 것은 cpu의 동작과 Thread의 동작 과정에 대해 설명할 때 자세히 기술할 예정.

// 결국 Thread.sleep(500)이 가장 중요한 작업이므로
// 이 작업을 완료하기 전까지는 어떠한 작업도 수행하지 않는다는 뜻.
// 그래서 0.5(millis:500)초 동안은 무조건적인 대기를 하게 된다.

// throw에 대한 건 나중에 설명
```