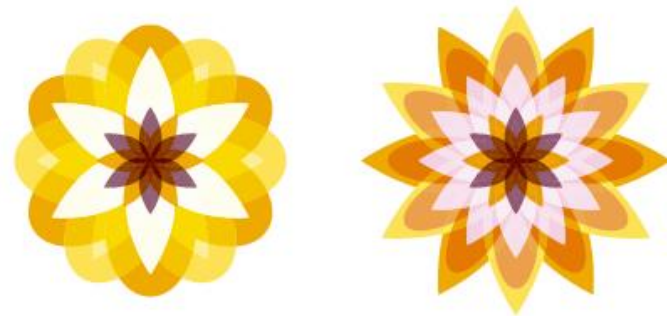


Chapter 04

숫자야구 게임



1. 이번에 만들 코드

- 숫자 야구 게임은 두 사람이 각각 3개의 숫자를 숨겨놓고 먼저 상대방의 숫자를 맞추는 편이 이기는 숫자 야구 놀이를 컴퓨터와 할 수 있도록 옮긴 것입니다.
- 물론 컴퓨터는 숫자를 맞히지 않고 숨겨두기 만하고, 플레이어(사람)가 제한된 횟수 내에 컴퓨터가 숨긴 숫자 3개를 모두 맞히면 적절한 칭찬을 해주도록 구현 하겠습니다.
- 규칙
 - 컴퓨터가 숨기는 숫자 3개는 1부터 9까지의 수로 0은 포함되지 않는다.
 - 3개의 숫자가 모두 다르다.
 - 플레이어는 3개의 숫자를 맞혀야 하는 것뿐만 아니라 그 위치까지 정확히 맞혀야 한다.
 - 예를 들어 컴퓨터가 1, 2, 3 의 3개의 숫자를 숨겨두었다면, 플레이어는 2, 3, 1 또는 3, 2, 1 이라고 해서는 안 되고 반드시 1, 2, 3 이라고 해야 한다.

1. 이번에 만들 코드

- 플레이어가 정답을 추측할 수 있도록 컴퓨터는 매 번 볼카운트를 알려줍니다.
- 이 볼카운트가 정답을 맞힐 수 있는 중요한 힌트가 됩니다.
- 볼카운트의 규칙은 플레이어가 입력한 숫자와 컴퓨터가 숨겨놓은 숫자가 같고 위치만 다르면 Ball 이고, 위치도 같으면 Strike으로 간주하는 것입니다.
- 예를 들어, 컴퓨터가 숨겨 놓은 숫자가 1, 2, 3 이고 플레이어가 입력한 숫자가 2, 1, 3 이면, 1과 2는 위치가 다르기 때문에 Ball 이고 3은 위치까지 같기 때문에 Strike 이어서, 볼카운트는 1 Strike, 2 Ball 이 됩니다.

컴퓨터가 숨긴 숫자	플레이어가 입력한 숫자	볼카운트
1, 2, 3	2, 5, 3	1Strike, 1Ball
4, 2, 1	4, 2, 7	2Strike, 0Ball
4, 5, 9	1, 3, 8	0Strike, 0Ball
2, 5, 8	5, 8, 2	0Strike, 3Ball
3, 6, 7	6, 7, 4	0Strike, 2Ball
3, 7, 2	3, 7, 2	3Strike, 0Ball→Game Over

1. 이번에 만들 코드

- 이 프로그램에서 컴퓨터가 숨겨두는 3개의 숫자와 플레이어가 입력하는 3개의 숫자는 배열에 저장되고, 규칙에 어긋나지 않는 숫자를 만들기 위해서 반복문을 사용합니다.
- 또 앞에서 배운 프로그램과는 달리 메서드를 정의하고 호출하기 때문에, 이 프로그램을 잘 이해하면 자바의 배열과 반복문, 메서드의 사용법을 익힐 수 있습니다.

2. 메서드 호출

- 프로그램을 개발하는 방법 중의 하나는 커다란 문제를 간단히 해결할 수 있는 작은 부분으로 쪼개어 각 부분을 구현함으로써 전체 문제를 해결하는 프로그램을 만드는 것입니다.
- 이러한 작은 부분을 모듈(module) 또는 서브루틴(subroutine) 이라고 부르는데, 모듈식으로 프로그램을 만들면, 개발하기에 용이할 뿐만 아니라 만든 프로그램을 이해하기 쉬워지고 반복적으로 나오는 부분을 하나의 모듈로 만들어서 계속 부를 수 있기 때문에 전체 프로그램의 크기가 줄어 드는 이점이 있습니다.
- C 언어나 C++ 언어에서는 이러한 모듈을 함수(function) 라고 부르고 파스칼 언어에서는 프로시저(procedure) 라고 하지만 자비에서는 메서드(method) 라고 부릅니다.

2. 메서드 호출

- 자바의 메서드는 반환하는 값의 데이터형(반환형), 메서드 이름, 인수인 매개 변수의 리스트를 정의하는 헤더 (header)와 처리할 일을 정의하는 바디 (body) 로 구성됩니다.

```
반환형 메서드이름([매개변수, 매개변수, ..., 매개변수])  → 메서드 헤더
{
    // 명령어들                                         → 메서드 바디
}
```

- 예를 들어 int형인 두 수 x와 y를 매개 변수로 받아서 두 수의 합을 int형으로 돌려주는 메서드 add는 다음처럼 정의할 수 있습니다.

```
반환형  메서드이름  매개변수  매개변수
  ↓         ↓         ↓         ↓
int      add      ( int x,   int y ) ← 메서드 헤더
{
    return ( x + y );                ← 메서드 바디
}
```

2. 메서드 호출

- 이 메서드 add를 호출하는 명령어는 다음과 같습니다.

```
z    =    add    ( 10  , 20 );
```

↑ ↑ ↑ ↑

반환값 메서드이름 인수 인수

- 위 명령어에서 10과 20은 변수가 아닌 상수지만, add의 매개변수 x와 y의 위치에 전달되어 각각 x와 y가 됩니다.
- 따라서 add가 돌려주는 값은 30 이 되고 z 에 저장되게 됩니다.
- 이때 Z는 add의 반환형인 int형이거나 int 형 값이 저장될 수 있는 데이터형이어야 합니다.
- 만일 z가 int형이 저장될 수 없는 데이터형인 경우는 형변환을 해야 합니다.

2. 메서드 호출

- 메서드를 호출하면 괄호 안에 주어진 매개 변수의 리스트가 해당 메서드에게 인수로 전달되고, 메서드 바디의 실행 결과는 메서드를 호출한 자리에 치환됩니다.
- 만일 인수가 필요 없는 메서드라면 매개 변수 리스트를 생략할 수 있습니다.
- 그러나 반환형은 반드시 명시해야 하는데, 만일 돌려줄 반환형이 없는 경우에도 생략해서는 안되고 void라고 표시해야 합니다.
- C 언어에서는 반환형이 int형일 때는 종종 반환형 선언을 생략하기도 하고, 바디에서 return을 빠뜨려도 됩니다만, 자바에서는 프로그래머의 실수를 방지하기 위해서 반드시 표시하도록 정해져 있습니다.
- 다음은 인수도 없고 반환값도 없는 메서드의 예입니다.

```
void printHello()  
{  
    System.out.println("Hello!");  
}
```


2. 메서드 호출

- 자바에서 인수로 매개 변수를 전달하는 방식은 크게 두 가지가 있는데, 기본 데이터형은 모두 Call by Value로 처리되고, 클래스의 객체는 Call by Reference로 처리됩니다.
- 두 방식의 차이점을 분명히 알아야 프로그램을 만들 수 있기 때문에 잘 알아두어야 합니다.
- Call by value
 - 자바에서 인수로 기본 데이터 형을 사용하면 모두 Call by Value가 됩니다.
 - Call by Value는 주어진 값을 복사하여 처리하는 방식입니다.
 - 즉 메서드 내에서 인수로 전달되는 데이터형과 동일한 종류의 데이터형 변수를 만들어 값을 복사한 후, 메서드 내의 변수만을 가지고 수행하는 방식입니다.
 - 따라서 메서드 내의 처리 결과는 메서드 밖의 변수에는 영향을 미치지 않습니다.
 - 다음은 Call by Value로 두 변수의 값을 바꾸려고 한 예제입니다.

2. 메서드 호출

■ Call by value

```
1 public class CallByValueTest {
2     public static void swap(int x, int y) {
3         int temp = x;
4         x = y;
5         y = temp;
6     }
7     public static void main(String[] args) {
8         int a = 10;
9         int b = 20;
10        System.out.println("swap() 메서드 호출 전: " + a + ", " + b);
11        swap(a, b);
12        System.out.println("swap() 메서드 호출 후: " + a + ", " + b);
13    }
14 }
```

2. 메서드 호출

■ Call by value

▪ 결과



```
<terminated> CallByValueTest [Java Application] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe (2021. 6. 26. 오후 3:12:58 - 오후 3:13:01)
swap() 메서드 호출 전: 10, 20
swap() 메서드 호출 후: 10, 20
```

- `swap(a, b);`로 호출했으므로 `x` 값은 `a`, `y` 값은 `b`의 값이 복사되지만, `swap()` 메서드가 끝난 후에는 돌려받지 못합니다.
- 위의 예제에서 `main()` 메서드의 `a`와 `b`는 `swap` 메서드 내의 `x`와 `y`에 각각 값이 복사되고, `swap()` 메서드에서는 `x`와 `y`만 다루어지기 때문에, `swap()` 내에서 `x`와 `y`의 값을 서로 바꾸지만 `main()` 메서드의 `a`와 `b`에는 아무런 영향을 미치지 않게 되는 것입니다.
- 만일 위 예제에서 `a`와 `b`의 값이 바뀌도록 하고 싶다면, 이어서 배울 `Call by Reference`를 쓰거나 다음처럼 `a`와 `b`를 전역변수로 선언하여 사용하면 됩니다.

2. 메서드 호출

■ Call by value

```
1 public class CallByValueTest2 {  
2     static int a;  
3     static int b;  
4  
5     public static void swap( ) {  
6         int temp = a;  
7         a = b;  
8         b = temp;  
9     }  
10 }
```

2. 메서드 호출

■ Call by value

```
11 public static void main(String[] args) {  
12     a = 10;  
13     b = 20;  
14  
15     System.out.println("swap() 메서드 호출 전: " + a + ", " + b);  
16     swap( );  
17     System.out.println("swap() 메서드 호출 후: " + a + ", " + b);  
18 }  
19 }
```

2. 메서드 호출

■ Call by value

▪ 결과



```
<terminated> CallByValueTest2 [Java Application] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe (2021. 6. 26. 오후 3:35:34 - 오후 3:35:35)
swap() 메서드 호출 전: 10, 20
swap() 메서드 호출 후: 20, 10
```

- swap();으로 호출했으므로 값의 전달은 일어나지 않고, swap() 메서드에서도 인수를 받지 않습니다.
- 하지만 swap() 메서드에서 전역 변수에 바로 값을 저장하기 때문에, 전역변수 a, b를 사용하는 모든 메서드에 영향을 미치게 됩니다.

■ Call by Reference

- Call by Value가 주어진 매개 변수의 값을 복사해서 처리하는데 비해 Call by Reference는 매개 변수의 원래 주소에 값을 저장하는 방식입니다.
- 따라서 Call by Reference로 인수를 전달하면, 메서드의 실행에 따라 인수로 전달한 변수의 값이 영향을 받게 됩니다.
- 자바에서는 클래스 객체를 인수로 전달한 경우에만 Call by Reference로 처리합니다.
- 다음은 Call by Reference로 두 변수의 값을 바꾼 예제입니다.

```
1 public class CallByReferenceTest {  
2     public static void swap(Number z) {  
3         int temp = z.x;  
4         z.x = z.y;  
5         z.y = temp;  
6     }  
7 }
```

2. 메서드 호출

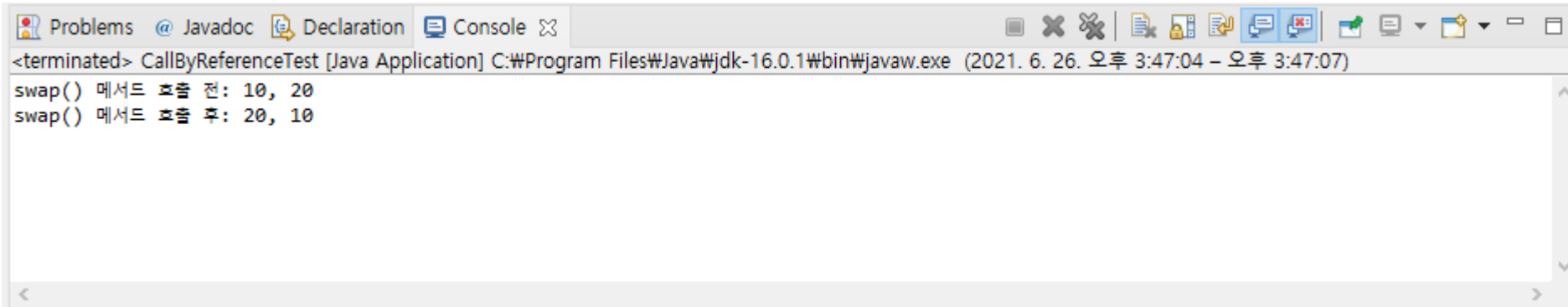
■ Call by Reference

```
8      public static void main(String[] args) {
9          Number n = new Number();    // Number 클래스로 n 생성
10         n.x = 10;
11         n.y = 20;
12
13         System.out.println("swap() 메서드 호출 전: " + n.x + ", " + n.y);
14         swap(n);
15         System.out.println("swap() 메서드 호출 후: " + n.x + ", " + n.y);
16     }
17 }
18 class Number{
19     public int x;
20     public int y;
21 }
```


2. 메서드 호출

■ Call by Reference

▪ 결과



```
<terminated> CallByReferenceTest [Java Application] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe (2021. 6. 26. 오후 3:47:04 - 오후 3:47:07)
swap() 메서드 호출 전: 10, 20
swap() 메서드 호출 후: 20, 10
```

- Number 클래스의 객체를 생성하여 값을 전달하게 되면 객체가 저장한 값이 주소 값이기 때문에, swap() 메서드에서 객체에 저장한 결과가 main() 메서드로 돌려지게 됩니다.
- 위의 예제에서는 main() 메서드에서 Number 클래스의 객체인 n을 만들어 인수로 전달하기 때문에, swap() 메서드 내에서 Number 클래스 내의 x와 y 값을 바꾼 결과가 main() 메서드에 영향을 미치게 됩니다.
- 이렇게 다른 메서드에서 현재의 메서드 내의 변수 값을 바꾸는 현상을 사이드 이펙트(side effect)라고 합니다.
- 사이드 이펙트는 메서드 간의 값 전달을 쉽게 하기 때문에 편리하지만, 실수로 프로그래머가 모르는 사이에 값이 바뀌면 심각한 문제를 일으킬 수 있기 때문에 위험하다고 알려져 있습니다.
- 그래서 자바는 모든 기본 데이터형은 Call by Value로 값을 주고받아 사이드 이펙트가 일어나지 않도록 했고, Call by Reference가 필요한 경우는 명시적으로 클래스 객체를 주고받도록 정해둔 것입니다.

3. 메서드 오버로딩

- 프로그래밍을 할 때 각각의 메서드를 모두 다른 이름으로 만드는 것은 당연하지만, 때때로 같은 이름의 메서드를 여러 개 정의하고 싶을 때가 있을 수 있습니다.
- c 언어에서는 모든 함수가 다른 이름이어야 하지만, 자바에서는 인수의 개수나 종류가 다르다면 같은 이름의 메서드를 얼마든지 정의할 수 있습니다.
- 이렇게 같은 이름의 메서드를 여러 개 정의할 수 있도록 해주는 것을 메서드 오버로딩(method over loading) 이라고 합니다.
- 메서드 오버로딩이 필요한 경우가 어떤 때일지 생각해봅시다.
- 예를 들어 int형의 두 수를 인수로 받아 합을 돌려주는 다음과 같은 add() 메서드를 만들었습니다.

```
int add(int x, int y){  
    return x + y;  
}
```

3. 메서드 오버로딩

- 그런데 나중에 `int`형이 아니고 `double`형의 두 수를 인수로 받아 돌려주는 메서드가 필요하게 되었다고 한다면, 메서드 오버로딩이 불가능한 경우에는 이미 `add()` 메서드가 있기 때문에 다음처럼 `d_add()` 메서드를 만들어야 합니다.

```
double d_add(double x, double y){  
    return x + y;  
}
```

- 비슷한 일을 하는 메서드인데도 같은 이름을 쓸 수 없다면 매번 다른 이름으로 정의해야 하고, 프로그래머는 다음과 같은 여러 개의 이름을 기억해야 합니다.

<code>add(int x, int y)</code>	→ <code>int</code> 형인 두 수를 더하는 <code>add</code>
<code>d_add(double x, double y)</code>	→ <code>double</code> 형인 두 수를 더하는 <code>add</code>
<code>f_add(float x, float y)</code>	→ <code>float</code> 형인 두 수를 더하는 <code>add</code>
<code>l_add(long x, long y)</code>	→ <code>long</code> 형인 두 수를 더하는 <code>add</code>

3. 메서드 오버로딩

- 그러나 자바의 경우는 메서드 오버로딩을 지원하기 때문에 모두 같은 이름으로 정의하는 것이 가능합니다.
- 예를 들어, 위의 여러 `add()` 메서드들도 다음처럼 같은 이름으로 정의할 수 있습니다.

<code>add(int x, int y)</code>	→ int형인 두 수를 더하는 add
<code>add(double x, double y)</code>	→ double형인 두 수를 더하는 add
<code>add(float x, float y)</code>	→ float형인 두 수를 더하는 add
<code>add(long x, long y)</code>	→ long형인 두 수를 더하는 add

- 이렇게 하면, `add()` 메서드를 호출하는 쪽에서는 여러 메서드 이름을 외울 필요 없이 `add(10, 3)`, `add(2.5, 4.2)`, `add(3F, 1.2F)`, `add(100L, 2000L)`, .. 등으로 부를 수 있습니다.

3. 메서드 오버로딩

- 그런데, 메서드의 이름이 같은데 자바는 어떻게 적절한 메서드를 호출할 수 있는 걸까요?
- 자바는 메서드의 이름 뿐만 아니라 인수를 함께 보고 판단하는 것입니다.
- 같은 이름의 메서드가 2 개 이상 있다면, 주어진 인수가 몇 개인지, 종류가 무엇 인지로 판단하게 됩니다.
- 예를 들어 `add(4, 5)` 는 2 개의 `int`형 인수가 든 `add()` 메서드이기 때문에, 자바 는 `add(int x, int y)`를 호출해줍니다.
- 따라서 같은 이름의 메서드를 정의하는 것은 괜찮지만 인수까지 동일한 메서드를 정의해서는 안됩니다.

3. 메서드 오버로딩

- 다음의 두 `add()`는 함께 정의 될 수 없습니다.

```
int add(int x, int y){  
    return x + y;  
}
```

```
int add(int a, int b){  
    return a + b;  
}
```

3. 메서드 오버로딩

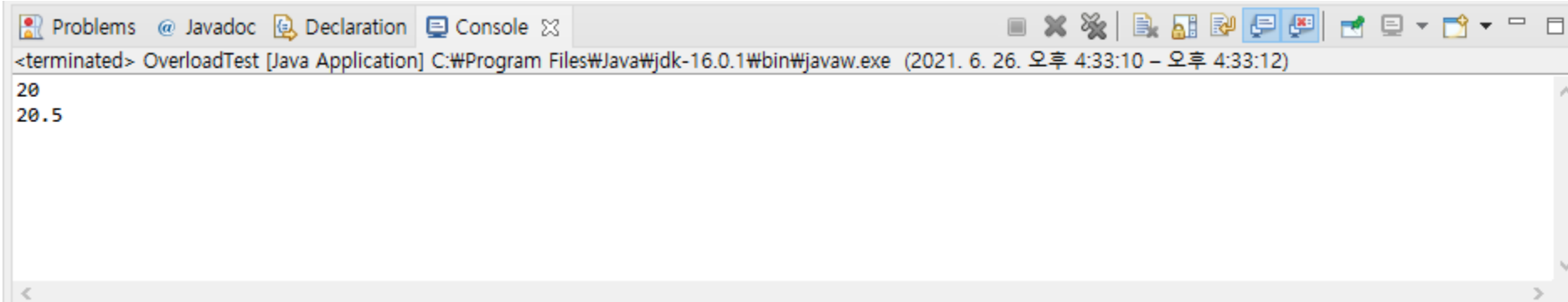
```
1 public class OverloadTest {  
2     public static int max(int x, int y) {  
3         if (x > y) {  
4             return x;  
5         }else {  
6             return y;  
7         }  
8     }  
9  
10    public static double max(double x, double y) {  
11        if (x > y) {  
12            return x;  
13        }else {  
14            return y;  
15        }  
16    }
```

3. 메서드 오버로딩

```
17     public static void main(String[] args) {
18         int a = 10;
19         int b = 20;
20
21         System.out.println(max(a, b)); // int형 인수 2개를 받는 max() 메서드
호출
22
23         double c = 10.5;
24         double d = 20.5;
25         System.out.println(max(c, d)); // double형 인수 2개를 받는 max() 메서드
호출
26     }
27 }
```


3. 메서드 오버로딩

■ 결과



```
<terminated> OverloadTest [Java Application] C:\Program Files\Java\jdk-16.0.1\bin\javaw.exe (2021. 6. 26. 오후 4:33:10 - 오후 4:33:12)
20
20.5
```

- 메서드의 이름이 같을 경우에는 전달하는 인수의 데이터형과 수로 메서드를 결정합니다.
- 만약 애매하게 인수를 주게 되면 메서드를 잘지 못하는 경우도 발생합니다.

4. 반복문

■ 반복문이란?

- 1부터 10까지 더하여 그 합을 계산해 볼까요?
- 지금까지 우리가 배운 것만으로 코드를 작성한다면 다음과 같을 것입니다.
- Ex) 1부터 10까지 더하기

```
1 package loopexample;
2
3 public class BasicLoop {
4     public static void main(String[] args) {
5         int num = 1;
6         num += 2;
7         num += 3;
8         num += 4;
9         num += 5;
10        num += 6;
11        num += 7;
```

4. 반복문

■ 반복문이란?

- Ex) 1부터 10까지 더하기

```
12         num += 8;
13         num += 9;
14         num += 10;
15
16         System.out.println("1부터 10까지의 합은 " + num + "입니다.");
17     }
18 }
```

- 그냥 보기에도 별로 효율적이지 않은 것 같죠?
- 이렇게 반복되는 일을 처리하기 위해 사용하는 것이 '반복문'입니다.
- 자바 프로그램에서 사용하는 반복문의 종류에는 while문, do-while문, for문 이렇게 세 가지가 있습니다.
- 모두 반복 수행을 한다는 것은 동일하지만, 사용 방법에 조금씩 차이가 있습니다.

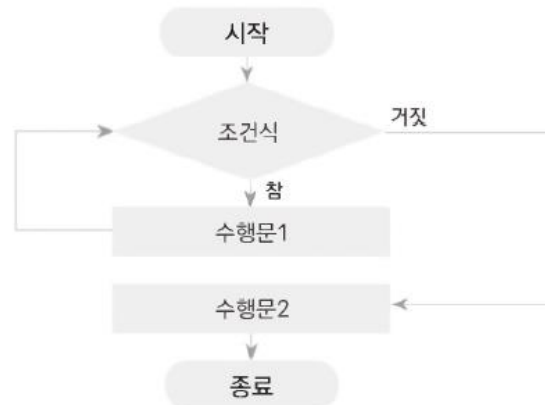
4. 반복문

■ while 문

- 반복문 중 먼저 while문을 살펴보겠습니다.
- while문은 조건식이 참인 동안 수행문을 반복해서 수행합니다.
- while문의 문법을 살펴보면 다음과 같습니다.

```
while(조건식) {  
    수행문1;  
    ...  
}  
수행문2;  
...
```

조건식이 참인 동안
반복 수행

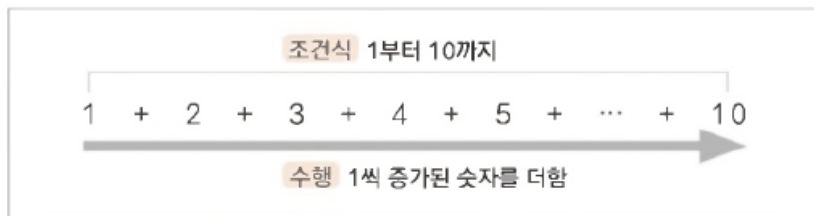


- 어떠한 조건식을 만족하는 동안 중괄호 {} 안의 수행문을 반복해서 처리합니다.
- 조건문과 마찬가지로 수행문이 하나인 경우에는 {}를 사용하지 않을 수 있습니다.

4. 반복문

■ while 문

- 그러면 우리가 앞에서 만든 1부터 10까지 더하는 프로그램을 while문으로 만들어 보겠습니다.
- 반복문은 조건식을 만족하는 동안에 수행문을 반복해서 처리한다고 했습니다.
- 그러면 조건식을 어떻게 만들면 될까요?

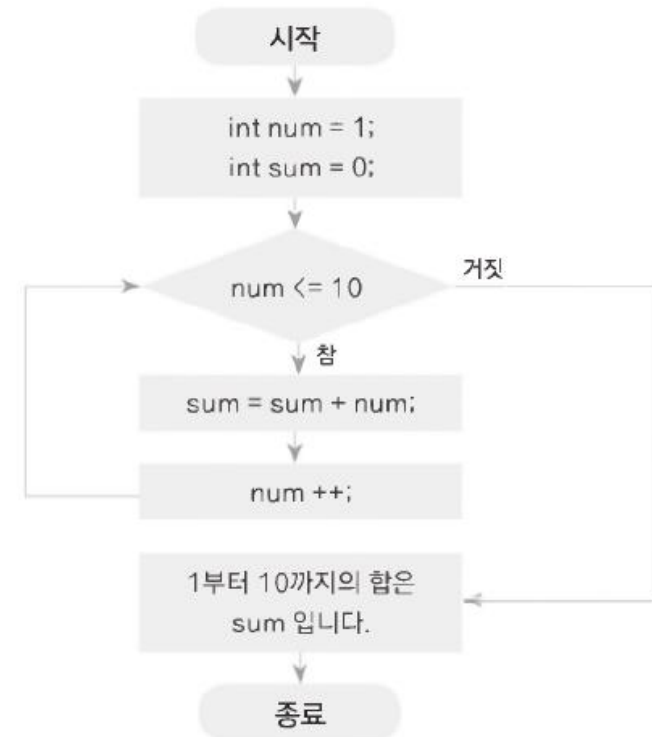


- '1부터 10까지 숫자가 커지는 동안'을 조건으로 하고, 1씩 증가한 숫자를 더하는 작업을 합니다.
- 1씩 올려 나갈 변수를 하나 선언하고, 증가한 숫자를 모두 더한 결과 값은 다른 변수에 저장하겠습니다.

4. 반복문

■ while 문

- 이 내용을 순서도로 보면 다음과 같습니다.
- num이 1 씩 증가하다가 숫자가 10을 넘어가는 순간 while문이 끝납니다.
- 즉 num이 10일 때까지 1씩 더한 값이 sum에 저장됩니다.



- 다음은 while문이 반복되는 과정을 보여 주는 표입니다.

num	num = 1	num = 2	num = 3	num = 4	num = 5
sum =	sum =	sum =	sum =	sum =	sum =
sum + num	0 + 1	1 + 2	3 + 3	6 + 4	10 + 5
sum	sum = 1	sum = 3	sum = 6	sum = 10	sum = 15

...

num = 9	num = 10	num = 11
sum =	sum =	while문 종료
36 + 9	45 + 10	
sum = 45	sum = 55	

4. 반복문

■ while 문

- 전체 코드는 다음과 같습니다.
- Ex) while문 활용하여 1부터 10까지 더하기

```
1 package loopexample;
2
3 public class WhileExample {
4     public static void main(String[] args) {
5         int num = 1;
6         int sum = 0;
7
8         while(num <= 10) { // num값이 10보다 작거나 같을 동안
9             sum += num;    // 합계를 뜻하는 sum에 num을 더하고
10            num++;         // num에 1씩 더해 나감
11        }
```

4. 반복문

■ while 문

- 전체 코드는 다음과 같습니다.
- Ex) while문 활용하여 1부터 10까지 더하기

```
12         System.out.println("1부터 10까지의 합은 " + sum + "입니다.");  
13     }  
14 }
```

- 위의 예제에서 5~6행을 보면 num 변수와 sum 변수를 선언하면서 동시에 초기값을 저장했습니다.
- 변수를 항상 초기화해야 하는 것은 아니지만, 이 예제에서는 반드시 초기화를 해야 합니다.
- 만약 변수를 초기화하지 않고 프로그램을 실행하면 오류가 납니다.
- 왜 그럴까요?
- while문 내부를 보면 sum에 num 값을 더해 줍니다.
- 그런데 num 값이 먼저 정해져 있지 않다면 sum에 무엇을 더해야 할지 알 수 없습니다.
- 또 sum 값도 정해져 있지 않다면 어떤 값에 num 값을 더해야 할지 알 수 없겠죠.
- 즉 변수를 사용하여 연산을 하거나 그 값을 가져다 사용하려면 변수는 반드시 어떤 값을 가지고 있어야 합니다.
- 따라서 이 예제에서는 num과 sum을 먼저 초기화해야 합니다.

■ while 문

■ while문이 무한히 반복되는 경우

- 앞에서 살펴본 while문은 특정 조건을 만족하는 동안 반복되는 명령을 수행하고, 그렇지 않으면 수행을 중단한 후 while문을 빠져나옵니다.
- 그런데 어떤 일을 수행할 때는 멈추면 안 되고 무한 반복해야 하는 경우도 있습니다.
- 가장 쉬운 예로 여러분이 자주 사용하는 인터넷 쇼핑몰을 생각해 봅시다.
- 인터넷 쇼핑몰이 24시간 서비스하기 위해서는 쇼핑몰의 데이터를 저장하고 있는 웹 서버가 멈추지 않고 끊임없이 돌아가야 합니다.
- 웹 서버가 멈추면 고객들의 항의가 많을 겁니다.
- while문의 구조를 보면 조건식이 참이면 반복합니다.
- 따라서 while문을 다음과 같이 사용하면 조건이 항상 '참'이 되어 '무한 반복'하겠죠?

```
while(true){  
    ...  
}
```

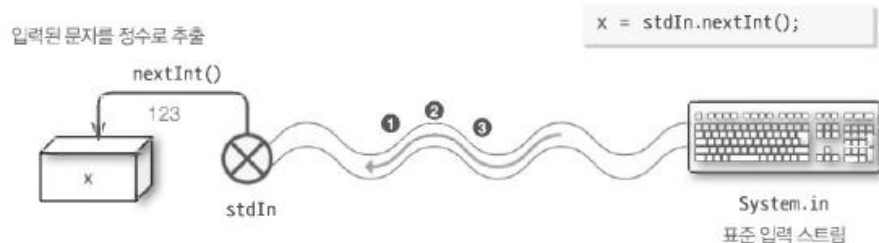
- 이렇게 끊임없이 돌아가는 시스템을 데몬(daemon)이라고 부릅니다.
- 데몬 시스템은 반복문을 이용하여 멈추지 않는 서비스를 구현할 수 있습니다.

4. 반복문

■ while 문

■ 연습문제

- 입력받은 정숫값부터 0까지 카운트다운하는 프로그램을 작성하라. 카운트다운 종료 후의 변수값을 확인할 수 있게 할 것.
- 키보드로 입력받는 것은 다음과 같이 수행한다.



```
import java.util.Scanner;          ---①
...
Scanner stdIn = new Scanner(System.in); ---②
...
int x = stdIn.nextInt( );          ---③
```

- 키보드로 값을 입력하려면 ①, ②, ③ 순서를 따른다.
- ②에서, System.in은 키보드와 연결된 표준 입력 스트림(standard input stream; STDIN)이다.
- 키보드와 연결된 표준 입력 스트림인 System.in에서 문자나 숫자를 꺼내는 '추출 장치'가 stdin이다.
- stdin은 다른 이름으로 변경할 수 있다

4. 반복문

■ while 문

■ 연습문제 4-1

- 입력받은 정숫값부터 0까지 카운트다운하는 프로그램을 작성하라. 카운트다운 종료 후의 변숫값을 확인할 수 있게 할 것.

```
1 package loopexample;
2
3 import java.util.Scanner;
4
5 public class Countdown {
6     public static void main(String[] args) {
7         Scanner stdIn = new Scanner(System.in);
8         System.out.println("카운트다운 합니다.");
9         int x = 0;
10    }
```

4. 반복문

■ while 문

■ 연습문제 4-1

- 입력받은 정숫값부터 0까지 카운트다운하는 프로그램을 작성하라. 카운트다운 종료 후의 변숫값을 확인할 수 있게 할 것.

```
11         while(x <= 0){
12             System.out.print("양의 정숫값: ");
13             x = stdIn.nextInt();
14         };
15
16         while(x >= 0)
17             System.out.println(x--);
18         System.out.println("x의 값이 " + x + "이 되었습니다.");
19     }
20 }
```

4. 반복문

■ do-while 문

- while문은 조건을 먼저 검사하기 때문에 조건식에 맞지 않으면 반복 수행이 한 번도 일어나지 않습니다.
- 하지만 do-while문은 { } 안의 문장을 무조건 한 번 수행한 후에 조건식을 검사합니다.
- 즉 조건이 만족하는지 여부를 마지막에 검사하는 것입니다.
- 따라서 중괄호 안의 문장을 반드시 한 번 이상 수행해야할 때 while문 대신 do-while문을 사용합니다.
- do-while문의 구조는 다음과 같습니다.

```
do {  
    수행문1;  
    ...  
} while(조건식);  
수행문2;  
...
```



4. 반복문

■ do-while 문

- while문으로 만든 1부터 10까지 더하는 프로그램을 do-while문으로 바꿔 봅시다.
- Ex) do-while문 예제

```
1 package loopexample;
2
3 public class DoWhileExample {
4     public static void main(String[] args) {
5         int num = 1;
6         int sum = 0;
7
8         do{
9             sum += num;
10            num++;
11        } while(num <= 10);
```

4. 반복문

■ do-while 문

- while문으로 만든 1부터 10까지 더하는 프로그램을 do-while문으로 바꿔 봅시다.
- Ex) do-while문 예제

```
12         System.out.println("1부터 10까지의 합은 " + sum + "입니다.");  
13     }  
14 }
```

■ for 문

- 반복문 중에서 가장 많이 사용하는 반복문이 for문입니다.
 - for문은 while문이나 do-while 문보다 구조가 조금 더 복잡합니다.
 - 왜냐하면 반복문을 구현하는 데 필요한 여러 요소(변수의 초기화식, 조건식, 증감식)를 함께 작성하기 때문이지요. 처음에는 for문이 좀 낯설겠지만, 익숙해지면 어떤 조건부터 어떤 조건까지 반복 수행하는지 한눈에 알아볼 수 있어 편리합니다.
-
- for문의 기본 구조
 - for문의 구조를 살펴보면서 반복문의 요소도 함께 알아보시다.
 - 초기화식은 for문이 시작할 때 딱 한 번만 수행하며 사용할 변수를 초기화합니다.
 - 조건식에서 언제까지 반복 수행할 것인지 구현합니다.
 - 증감식에서 반복 횟수나 for문에서 사용하는 변수 값을 1 만큼 늘리거나 줄입니다.

```
for(초기화식; 조건식; 증감식){  
    명령어;  
}
```


4. 반복문

■ for 문

■ for문의 기본 구조

- for문의 수행순서를 이해하기 쉽도록 간단한 예를 들어 보겠습니다.
- 1부터 5까지 출력하는 프로그램을 for문으로 만들어 볼까요?
- 화살표와 번호는 이 예제가 수행되는 순서입니다.
- 조건식이 참인 동안 순서로 반복문을 계속 수행합니다.
- for 문은 증감식에서 사용한 변수가 조건식의 참·거짓 여부를 결정합니다.

```
int num;
for(num = 1; num <= 5; num++)
{
    System.out.println(num);
}
```

```
graph LR
    1((1)) --> 2((2))
    2 --> 3((3))
    3 --> 4((4))
    4 --> 2
```

4. 반복문

■ for 문

▣ for문의 기본 구조

① 처음 for문이 시작할 때 출력할 숫자인 num을 1로 초기화합니다.



- ② 조건식 `num <= 5`를 검사했을 때 num은 1이므로 참입니다.
③ 조건식이 참이기 때문에 for문의 `System.out.println(num);`을 수행하고 1을 출력합니다.
④ 증감식 `num++`를 수행하여 num 값은 2가 됩니다.



- ② 조건식 `num <= 5`를 검사했을 때 num은 2이므로 참입니다.
③ 조건식이 참이기 때문에 for문의 `System.out.println(num);`을 수행하고 2를 출력합니다.
④ 증감식 `num++`를 수행하여 num 값은 3이 됩니다.



...



② 조건식 `num <= 5`를 검사했을 때 num은 6이므로 거짓입니다. for문이 끝납니다.

4. 반복문

■ for 문

▪ for문의 기본 구조

- 1부터 10까지 더하는 과정을 for문으로 구현한 전체 프로그램은 다음과 같습니다.
- Ex) for문 예제

```
1 package loopexample;
2
3 public class ForExample1 {
4     public static void main(String[] args) {
5         int i;
6         int sum;
7         for(i = 1, sum = 0; i <= 10; i++) {
8             sum += i;
9         }
10    }
```

4. 반복문

■ for 문

▪ for문의 기본 구조

- 1부터 10까지 더하는 과정을 for문으로 구현한 전체 프로그램은 다음과 같습니다.
- Ex) for문 예제

```
11         System.out.println("1부터 10까지의 합은 " + sum + "입니다.");
12     }
13 }
```

- 초기화 부분과 증감식 부분도 콤마(,)로 구분하여 여러 문장을 사용할 수 있습니다.
- 예를 들어 7행을 보면 $i = 1$, $sum = 0$ 으로 두 개의 변수를 초기화한 것을 볼 수 있습니다.
- 연습문제 : for 문 연습하기
 - for문과 변수를 사용하여 '안녕하세요1, 안녕하세요2..., 안녕하세요10'까지 차례로 출력하는 프로그램을 작성해 보세요.

4. 반복문

■ for 문

■ For문을 자주 사용하는 이유

- for문을 가장 많이 사용하는 이유는 반복 횟수를 관리할 수 있기 때문입니다.
- 물론 while문에서도 반복 횟수에 따라 구현할 수 있습니다.
- 1부터 10까지 더하는 프로그램을 while문과 for문으로 만들어 비교해 보겠습니다.

```
int num = 1;           //초기화
int sum = 0;
while(num <= 10) {     //조건 비교
    sum += num;
    num++;             //증감식
}
```

while문으로 구현



```
int sum = 0;
for(int num = 1; num <= 10; num++) {
    sum += num;
}
```

for문으로 구현

- while문으로 작성한 코드를 살펴보면 변수 num의 초기화와 조건 비교, 증감식을 따로 구현했습니다.
- 하지만 for문을 사용하여 구현하면 초기화, 조건 비교, 증감식을 한 줄에 쓸 수 있을 뿐더러 가독성도 좋습니다.

4. 반복문

■ for 문

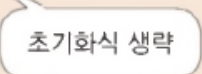
■ For문을 자주 사용하는 이유

- 또 for문은 배열과 함께 자주 사용합니다.
- 배열은 같은 자료형이 순서대로 모여 있는 구조인데, 배열 순서를 나타내는 색인은 항상 0부터 시작합니다.
- 따라서 배열의 전체 요소 개수가 n 개일 때, 요소 위치는 $n-1$ 번째로 표현할 수 있습니다.
- 이러한 배열의 특성과 증감에 따른 반복을 표현하는 데 적합한 for문의 특성 때문에 for문과 배열을 함께 자주 사용하는 것입니다.

■ for문 요소 생략하기

- for문을 구성하는 요소는 코드가 중복되거나 논리 흐름상 사용할 필요가 없을 때 생략할 수 있습니다.
- 초기화식 생략
 - 이미 이전에 다른 곳에서 변수가 초기화되어 중복으로 초기화할 필요가 없을 때 초기화 부분을 생략할 수 있습니다.

```
int i = 0;
for( ; i < 5; i++) {
    ...
}
```



■ for 문

■ for문 요소 생략하기

• 조건식 생략

- 어떤 연산 결과 값이 나왔을 때 바로 for문의 수행을 멈추려면 조건식을 생략하고 for문 안에 if문을 사용하면 됩니다.
- 예를 들어 1부터 시작해 수를 더해 나갈 때 더한 결과 값이 200을 넘는지 검사하려면 for문 안에 if문을 사용합니다.

조건식 생략

```
for(i = 0; ; i++) {  
    sum += i;  
    if(sum > 200) break;  
}
```

• 증감식 생략

- 증감식의 연산이 복잡하거나 다른 변수의 연산 결과 값에 좌우된다면 증감식을 생략하고 for문 안에 쓸 수 있습니다.

증감식 생략

```
for(i = 0; i < 5; ) {  
    ...  
    i = (++i) % 10;  
}
```

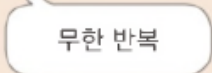
4. 반복문

■ for 문

■ for문 요소 생략하기

- 요소 모두 생략
 - 모든 요소를 생략하고 무한 반복하는 경우에 사용합니다.

```
for( ; ; ) {  
    ...  
}
```



무한 반복

■ 중첩된 반복문

- 반복문 안에 또 다른 반복문을 중첩해서 사용하는 경우가 종종 있습니다.
- 간단한 예로 구구단을 출력해 보겠습니다.

4. 반복문

■ for 문

- 중첩된 반복문

- Ex) 중첩된 반복문

```
1 package loopexample;
2
3 public class NestedLoop {
4     public static void main(String[] args) {
5         int dan;
6         int times;
7
8         for(dan = 2; dan <= 9; dan++) {
9             for(times = 1; times <= 9; times++) {
10                 System.out.println(dan + " X " + times + " = " + dan * times);
11             }
```

4. 반복문

■ for 문

▪ 중첩된 반복문

- Ex) 중첩된 반복문

```
1      System.out.println();
2      }
3      }
11     }
```

- 반복문을 중첩해서 사용할 때 외부 for문과 내부 for문이 어떤 순서로 실행되는지 잘 이해해야 합니다.
- 구구단은 2단부터 9단까지 단이 증가합니다.
- 그리고 각 단은 1부터 9까지 곱하는 수가 증가하죠.

■ for 문

■ 중첩된 반복문

- 그러면 '단이 증가'하는 부분과 '곱하는 수가 증가'하는 부분 중 무엇을 먼저 반복 수행 해야 할까요?



- 먼저 외부 for문의 초기화 값이 `dan = 2`이므로 구구단 2단부터 시작합니다.
- 그리고 내부 for문으로 들어가면 초기화 값 `times = 1`부터 시작해 1씩 증가하면서 9보다 작거나 같을 때까지 곱합니다.
- `times` 값이 10이 되면 내부 for문은 끝나고 외부 for문으로 돌아갑니다.
- 외부 for문에서 `dan++`를 수행하고 증가한 단의 값이 9보다 작은 지 확인합니다.
- 9보다 작으므로 다시 내부 for문으로 들어와 1부터 9까지 곱합니다.
- 정리하자면, 중첩 반복문을 쓸 때는 어떤 반복문을 먼저 수행해야 하는지 그리고 내부 반복문을 수행하기 전에 초기화해야 할 값을 잘 초기화했는지를 살펴야 합니다.
- for문 외의 다른 반복문도 중첩해서 사용할 수 있습니다.
- 연습문제 : 조금 전에 실습한 중첩 반복문 예제를 수정해 구구단을 3단부터 7단까지만 출력해 보세요.

■ for 문

■ 중첩된 반복문

- 우리는 지금까지 세 가지 반복문(while문, do-while문, for문)을 살펴보았습니다.
- 그러면 각 반복문을 언제, 어떤 경우에 사용하는 것이 가장 좋을까요?
- 반복 횟수가 정해진 경우에는 for문을 사용하는 것이 좋습니다.
- 그리고 수행문을 반드시 한 번 이상 수행해야 하는 경우에는 do-while문이 적합합니다.
- 이 두 경우 외에 조건의 참·거짓에 따라 반복문이 수행하는 경우에는 while문을 사용합니다.
- 물론 반복 횟수가 정해진 반복문을 while문으로 구현할 수도 있습니다.
- 그리고 조건의 참·거짓에 따른 반복문을 for문으로 구현할 수도 있죠.
- 하지만 좋은 프로그래밍 습관을 가지고 싶다면, 상황에 맞는 적절한 문법을 사용하는 것이 중요합니다.

■ for 문

▪ continue 문

- continue문은 반복문과 함께 쓰입니다.
- 반복문 안에서 continue문을 만나면 이후의 문장은 수행하지 않고 for문의 처음으로 돌아가 증감식을 수행합니다.
- 다음 예제를 봅시다.
- 1부터 100까지 수를 더할 때 홀수일 때만 더하고 짝수일 때는 더하지 않는 프로그램을 continue문으로 작성해 보겠습니다.
- Ex) continue문 예제

```
1 package loopexample;
2
3 public class ContinueExample {
4     public static void main(String[] args) {
5         int total = 0;
6         int num;
7     }
```

■ for 문

▪ continue 문

- Ex) continue문 예제

```
8      for(num = 1; num <= 100; num++) {  
9          if(num % 2 == 0)  
10             continue;  
11             total += num;  
12         }  
13         System.out.println("1부터 100까지의 홀수의 합은: " + total + "입니다.");  
14     }  
15 }
```

- 그러면 continue문은 언제 사용할까요?
- 예제를 보면 반복문 안의 조건문에서 변수 num이 짝수일 때는 이후 수행을 생략하고 for문의 증감식으로 돌아가서 num에 1을 더합니다.
- num이 홀수일 때는 계속 진행(continue)해서 total += num ; 문장을 수행합니다.
- 이렇듯 continue문은 반복문을 계속 수행하는데, 특정 조건에서는 수행하지 않고 건너뛰어야 할 때 사용합니다.

■ for 문

▪ break 문

- switch-case문에서 break문을 사용할 때 조건을 만족하면 다른 조건을 더 이상 비교하지 않고 switch문을 빠져 나왔지요?
- 반복문에서도 마찬가지입니다.
- 반복문에서 break문을 사용하면 그 지점에서 더 이상 수행문을 반복하지 않고 반복문을 빠져 나옵니다.
- 다음 예제를 살펴보겠습니다.
- 0부터 시작해 숫자를 1 씩 늘리면서 합을 계산할 때 숫자를 몇까지 더하면 100이 넘는지 알고 싶습니다.
- 지금까지 배운 반복문을 사용해 봅시다.
- Ex) break문 예제

```
1 package loopexample;
2
3 public class BreakExample1 {
4     public static void main(String[] args) {
5         int sum = 0;
6         int num = 0;
```

4. 반복문

■ for 문

▪ break 문

- Ex) break문 예제

```
7      for(num = 0; sum < 100; num++) {  
8          sum += num;  
9      }  
10     System.out.println("num: " + num);  
11     System.out.println("sum: " + sum);  
12 }  
13 }
```

- 이 코드를 실행해 보면 합은 105가 되었고 이 때 num 값은 15가 출력되었습니다.
- 그렇다면 1부터 15까지 더 했을 때 100이 넘는 걸까요?
- 그렇지 않습니다.
- 합이 105가 되는 순간 num 값은 14였습니다.
- 즉 1부터 14까지 더해져서 105가 되었고 num 값이 1씩 증가하여 15가 되었을 때 조건을 비교해 보니 합이 100보다 커서 반복문이 끝난 것입니다.

■ for 문

▪ break 문

- 따라서 우리가 원하는 정확한 값인 14를 얻으려면 증감이 이루어지기 전에 반복문을 끝내야 하죠.
- 그러면 반복문 안에 break문을 사용하여 수행을 중단해 보겠습니다.
- Ex) break문 예제

```
1 package loopexample;
2
3 public class BreakExample2 {
4     public static void main(String[] args) {
5         int sum = 0;
6         int num = 0;
7
8         for(num = 0; ; num++) {
9             sum += num;
```

4. 반복문

■ for 문

▪ break 문

- 따라서 우리가 원하는 정확한 값인 14를 얻으려면 증감이 이루어지기 전에 반복문을 끝내야 하죠.
- 그러면 반복문 안에 break문을 사용하여 수행을 중단해 보겠습니다.
- Ex) break문 예제

```
10         if(sum >= 100)           // sum이 100보다 크거나 같을 때(종료 조건)
11             break;                // 반복문 중단
12     }
13     System.out.println("num : " + num);
14     System.out.println("sum : " + sum);
15 }
16 }
```

- 위 예제는 0부터 시작해 1씩 늘린 숫자를 sum에 더합니다.
- 그리고 sum 값이 100보다 크거나 같으면 반복문을 바로 빠져 나옵니다.
- 프로그램을 실행하면 num 값이 14일 때 합이 105가 되는 것을 알 수 있습니다.

■ for 문

▪ break 문

- Ex) break문 예제
- 종료 조건을 for문 안에 사용하면 num 값을 늘리는 증감식을 먼저 수행하므로 num 값이 15가 됩니다.
- 따라서 프로그램 실행 중에 반복문을 중단하려면 break문을 사용해야 정확한 결과 값을 얻을 수 있습니다.

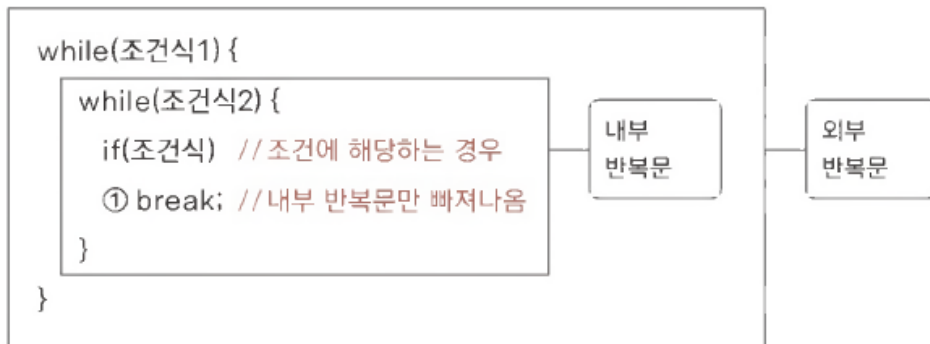
4. 반복문

■ for 문

▪ break 문

- break문의 위치

- 앞의 예제에서 봤듯이 반복문이 중첩된 경우가 있습니다.
- 이 경우에 break문을 사용하면 모든 반복문을 빠져 나오는 것이 아니고 break문을 감싸고 있는 반복문만 빠져나옵니다.



- 위 코드의 ① 위치에서 break문을 사용하면 if 조건문만 빠져나온다고 생각할 수도 있는데, 반복문 안의 break문은 해당 반복문 수행만 중지한다는 것을 기억하기 바랍니다.
- 즉 이러한 경우에는 내부 반복문만 빠져나오고 외부 반복문은 계속 수행합니다.
- 정리하자면, continue문은 반복문을 계속 수행하지만 특정 조건에서 수행문을 생략하는 경우에 사용하고, break문은 반복문을 더 이상 수행하지 않고 빠져 나올 때 사용합니다.

5. 배열

■ 자료를 순차적으로 관리하는 구조, 배열

- 학교에 학생이 100명 있습니다.
- 이 학생들 100명의 학번을 어떻게 관리할 수 있을까요?
- 학번의 자료형을 정수라고 하면 학생이 100명일 때 `int studentID1, int studentID2, int studentID3, ..., int studentID100` 이렇게 변수 100개를 선언해서 사용해야겠죠.
- 그런데 학번에 대한 여러 개 변수들을 일일이 쓰는 것은 너무 귀찮고 번거롭습니다.
- 이때 사용하는 자료형이 배열(array)입니다.
- 배열은 자료 구조의 가장 기초 내용입니다.



- 배열을 사용하면 자료형이 같은 자료 여러 개를 한 번에 관리할 수 있습니다.
- 위 그림으로 알 수 있듯이 배열은 자료가 연속으로 나열된 자료 구조입니다.

■ 배열 생성 - 1단계 배열 선언

- 배열을 선언할 때는 다음과 같이 2가지 방법으로 선언할 수 있다.

```
자료형[ ] 배열이름;  
자료형 배열이름[ ];
```

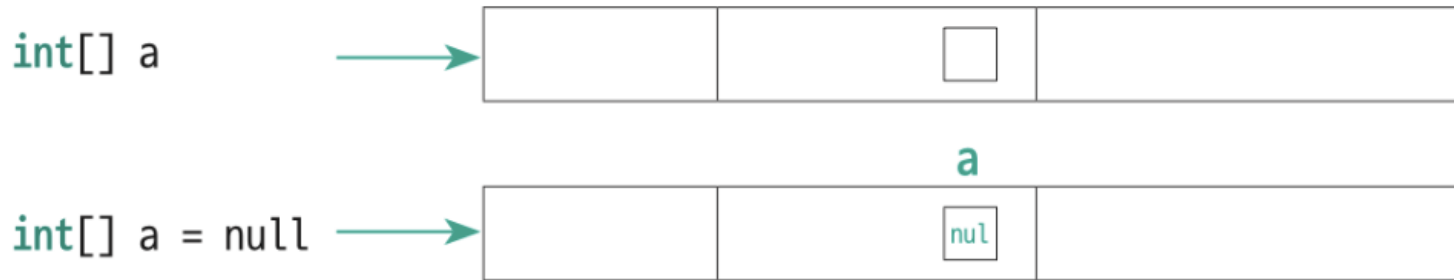
```
예)  
int[] a;  
double[] b;
```

- 정수 자료형을 `int`라는 이름으로 만들었고, 실수 자료형을 `double`이라는 이름으로 만든 것처럼 배열 자료형도 `array` 정도의 이름으로 만들면 편할 텐데 왜 '자료형 []' 형식을 사용하는 것일까?
- 배열은 동일한 자료형만 묶을 수 있는 자료형인데, 만일 `array a`와 같이 만들면 이 배열이 어떤 자료형을 묶은 것인지 알 길이 없다.
- 그래서 `int[] a`, `String[] b`와 같이 선언해 배열 자료형을 보자마자 어떤 타입을 묶은 것인지 알 수 있도록 하는 것이다.

5. 배열

■ 배열 생성 - 1단계 배열 선언

- 배열을 선언하면 스택 메모리에 변수의 공간만 생성하고, 공간 안은 비운 채로 둔다.
- 아직 배열의 실제 데이터인 객체를 생성하지 않았기 때문이다.
- 스택 메모리에 위치하고 있는 참조 자료형 변수의 빈 공간을 초기화할 때는 null(널) 값을 사용할 수 있다.
- null 값은 힙 메모리의 위치(번지)를 가리키고 있지 않다는 의미다.
- 즉, 연결된 실제 데이터가 없다는 것을 의미한다.



■ 배열 생성 – 2단계 힙 메모리에 배열의 객체 생성

- 모든 참조 자료형의 실제 데이터(객체)는 힙 메모리에 생성된다.
- 힙 메모리에 객체를 생성하기 위해서는 `new` 키워드를 사용해야 한다.

```
new 자료형[배열의 길이];
```

예)

```
new int[3];
```

```
new String[5];
```

- 배열을 생성할 때 `new int[3]` 또는 `new String[10]`과 같이 배열의 길이를 반드시 지정해야 한다.
- 예를 들어 배열의 길이를 지정하지 않고 `new int[]`와 같이 명령하면 오류가 발생한다.

■ 배열 생성 - 3단계 배열 자료형 변수에 객체 대입

- 선언된 배열 참조 자료형 변수에 생성한 객체를 대입하는 데는 2가지 방법이 있는데, 변수 선언과 값(참조 자료형은 객체)의 대입을 한 번에 작성해도 되고, 따로 구분해 작성해도 된다.

```
new 자료형[] 변수명 = new 자료형[배열의 길이];
```

예)

```
int a[] = new int[3];
```

```
자료형[] 변수명;
```

```
변수명 = new 자료형[배열의 길이];
```

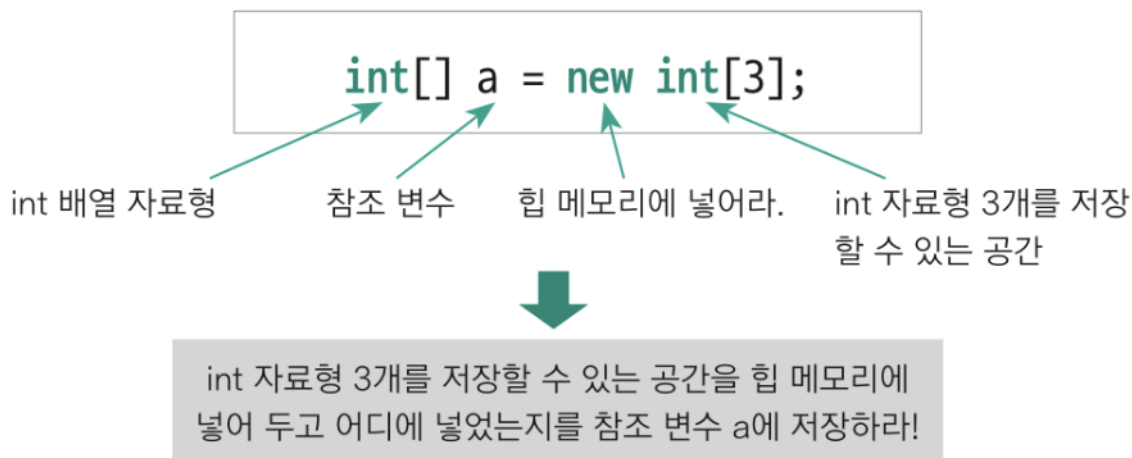
예)

```
int[] a;
```

```
a = new int[3];
```

■ 배열 생성 - 3단계 배열 자료형 변수에 객체 대입

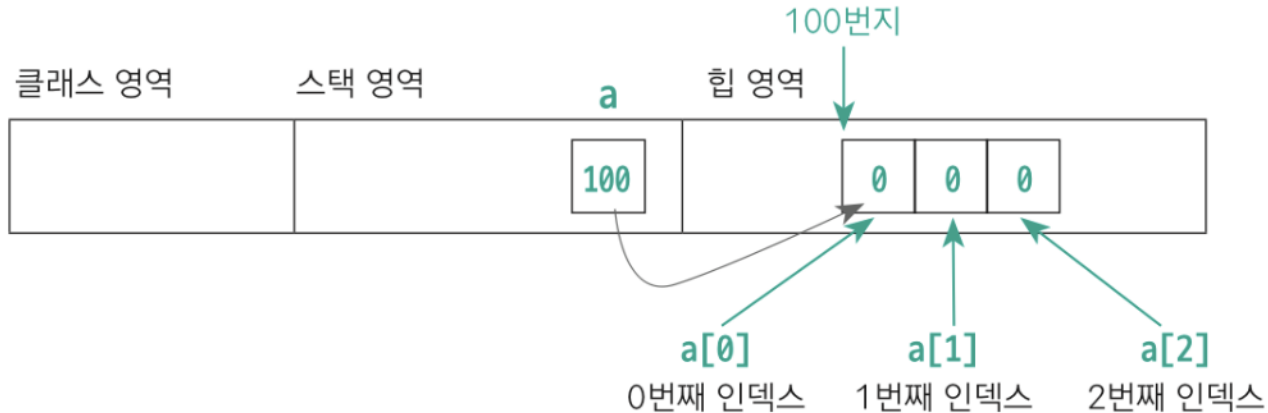
- 예시 코드중 배열 선언과 동시에 값을 대입한 첫 번째 코드를 좀 더 자세히 살펴보자.
- 각 구성 요소를 하나하나 뜯어 보면 먼저 `int[]`는 `int` 자료형만을 저장할 수 있는 배열을 의미한다.
- `A`는 참조 변수로, 실제 데이터값을 저장하는 것이 아니라 실제 데이터값의 위치값을 저장한다.
- `new` 키워드는 힙 메모리에 객체를 넣으라는 의미이고, `int[3]`은 정수 3개를 저장할 수 있는 공간을 만들라는 의미이다.
- 이를 정리하면 'int 자료형 3개를 저장할 수 있는 공간을 힙 메모리에 넣어 두고 어디에 넣었는지를 참조 변수 `a`에 저장하라!'는 의미인 것이다.



5. 배열

■ 배열 생성 - 3단계 배열 자료형 변수에 객체 대입

- 이때의 메모리 구조는 다음과 같다.



- 여기서 알고 넘어가야 할 점은 스택 메모리 공간은 값을 초기화하지 않으면 빈 공간으로 존재한다는 것이다.
- 반면 힙 메모리는 어떤 상황에서도 빈 공간이 존재하지 않는다.
- 그래서 값을 주지 않으면 컴파일러가 값을 강제로 초기화한다.
- 강제 초기화값은 자료형에 따라 다른데, 기본 자료형일 때 숫자는 모두 0(실수는 0.0), 불리언은 `false`로 값이 초기화되며 이외의 모든 참조 자료형은 `null`로 초기화된다.

■ 배열 생성 - 3단계 배열 자료형 변수에 객체 대입

- 실습 - 1차원 배열의 2가지 선언 방법 방법과 다양한 배열 선언 예

```
1 package sec01_array.EX01_ArrayDefinition;
2
3 public class ArrayDefinition {
4     public static void main(String[] args) {
5         //#1. 배열의 선언 방법 #1
6         int[] array1 = new int[3];
7         int[] array2;
8         array2 = new int[3];
9
10        //#2. 배열의 선언 방법 #2
11        int array3[] = new int[3];
12        int array4[];
13        array4 = new int[3];
14    }
```

■ 배열 생성 - 3단계 배열 자료형 변수에 객체 대입

- 실습 - 1차원 배열의 2가지 선언 방법 방법과 다양한 배열 선언 예

```
15      // #3. 다양한 배열 선언 (기본자료형 배열, 참조자료형 배열)
16      boolean[] array5 = new boolean[3];
17      int[] array6 = new int[5];
18      double[] array7 = new double[7];
19      String[] array8 = new String[9];
20  }
21 }
```

■ 배열 생성 - 4단계 객체에 값 입력

- 배열은 값을 저장할 수 있는 공간마다 방 번호가 있는데, 이 번호를 인덱스(index)라고 한다.
- 인덱스는 0부터 시작하며, 1씩 증가한다.
- 예를 들어 방이 3개일 때 방 번호, 즉 인덱스는 0, 1, 2다.
- 인덱스를 이용해 각 저장 공간에 값을 대입하는 방법은 다음과 같다.

참조 변수명[인덱스] = 값;

예)

```
int[] a = new int[3];
```

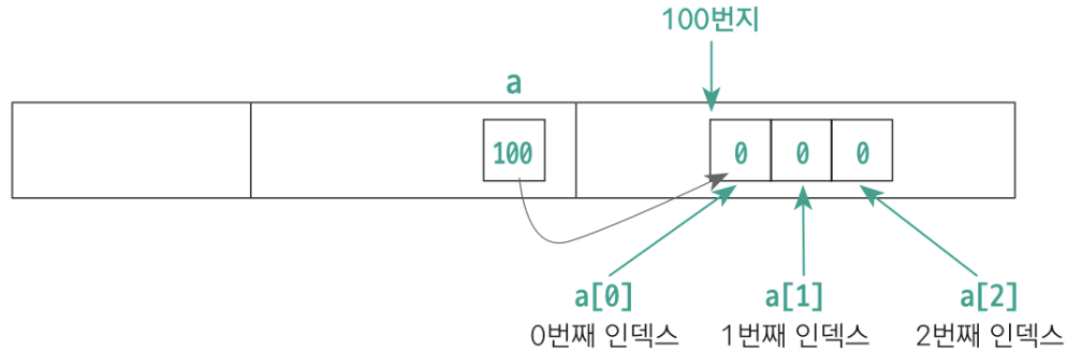
```
a[0] = 3;
```

```
a[1] = 4;
```

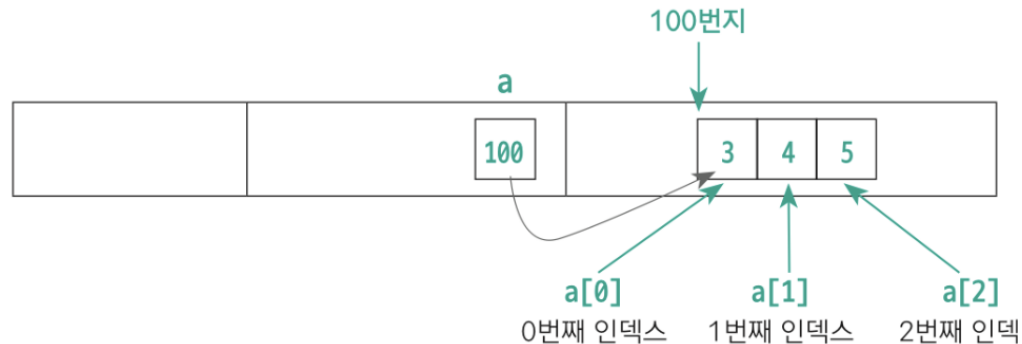
```
a[2] = 5;
```

■ 배열 생성 - 4단계 객체에 값 입력

- `int[] a = new int[3]`과 같이 처음 배열을 선언함과 동시에 객체를 생성하면 힙 메모리의 모든 값은 초기화된다.
- 이때 배열의 자료형이 정수이므로 초기화되는 값은 0이다.



- 이후 0번째 방에 3을 넣고(`a[0] = 3`), 1번째 방에 4를 넣고(`a[1] = 4`), 2번째 방에 5를 넣으면 (`a[2] = 5`) 다음과 같이 각각의 인덱스에 해당하는 공간에 값이 대입된다.



■ 배열 생성 - 4단계 객체에 값 입력

- 값을 읽을 때도 다음과 같이 인덱스를 사용한다.

참조 변수명[인덱스];

예)

```
System.out.println(a[0]);      // 3
System.out.println(a[1]);      // 4
System.out.println(a[2]);      // 5
```

- 배열의 저장 공간에 값을 대입하거나 읽을 때, 없는 인덱스를 사용하면 예외(exception)가 발생하고 프로그램이 종료된다.

```
System.out.println(a[2]);      // 5 출력
System.out.println(a[-1]);     // 예외 발생
System.out.println(a[3]);     // 예외 발생
```


■ 1차원 배열을 생성하는 다양한 방법

■ 방법 ① 배열 객체를 생성하고 값 대입하기

- 첫 번째 방법은 앞서 알아본 것처럼 배열의 객체를 먼저 선언하고, 이후 각 인덱스 위치마다 값을 대입하는 것이다.
- 객체를 생성할 때는 배열의 길이가 명확히 기술되어야 하고, 값을 입력할 때는 가용한 인덱스(0~배열의 길이 - 1)만을 사용해야 한다.

```
자료형 참조 변수명 = new 자료형[배열의 길이];
```

```
참조 변수명[0] = 값;
```

```
참조 변수명[1] = 값;
```

```
...
```

```
참조 변수명[배열의 길이-1] = 값;
```

예)

```
int[] a = new int[3];
```

```
a[0] = 3;
```

```
a[1] = 4;
```

```
a[2] = 5;
```

■ 1차원 배열을 생성하는 다양한 방법

▪ 방법 ② 배열 객체 생성과 함께 값 대입하기

- 두 번째 방법은 배열 객체를 생성함과 동시에 값을 대입하는 것이다.
- 이때 초깃값을 직접 넣어 주므로 컴파일러에 따른 강제 초기화 과정은 생략된다.
- 두 번째 방법에서는 객체를 생성할 때 오른쪽향의 대괄호([]) 안에 배열의 길이를 지정하지 않는다.
- 배열의 길이는 다음에 나오는 중괄호({ }) 안의 초기화 데이터 개수로 결정되기 때문이다.

```
자료형[] 참조 변수명 = new 자료형[]{값, 값, ..., 값};
```

예)

```
int[] a = new int[]{3, 4, 5};
```

■ 1차원 배열을 생성하는 다양한 방법

▪ 방법 ③ 대입할 값만 입력하기

- 마지막 방법은 new 키워드 없이 초기화할 값만 중괄호에 넣어 대입하는 것이다.
- 초기화 데이터의 개수가 배열의 길이를 결정한다.

```
자료형[] 참조 변수명 = {값, 값, ..., 값};
```

예)

```
int[] a = {3, 4, 5};
```

- 방법 ③은 방법 ②에서 new int[]를 생략한 형태이다.
- 만일 방법 ②와 방법 ③이 완벽히 동일하다면 굳이 상대적으로 복잡한 방법 ②를 쓸 필요는 없을 것이다.
- 하지만 방법 ③에는 변수 선언과 값의 대입을 분리할 수 없다는 제약 조건이 따른다.
- 즉, 선언과 동시에 값을 대입할 때만 사용할 수 있다.
- 방법 ③은 선언과 대입을 분리할 수 없다는 특징 때문에 메서드의 입력매개 변수값으로는 사용할 수 없다.

■ 실습. 1차원 배열을 생성하는 다양한 방법

■ ValueAssignment.java

```
1 package sec01_array.EX02_ValueAssignment;
2
3 public class ValueAssignment {
4     public static void main(String[] args) {
5         ///#1. 배열의 원소값 대입 방법 1
6         int[] array1 = new int[3];
7         array1[0]=3;
8         array1[1]=4;
9         array1[2]=5;
10        System.out.println(array1[0]+ " "+ array1[1]+ " " + array1[2]);
11
12        int[] array2;
13        array2 = new int[3];
14        array2[0]=3;
15        array2[1]=4;
16        array2[2]=5;
```

■ 실습. 1차원 배열을 생성하는 다양한 방법

■ ValueAssignment.java

```
17      System.out.println(array2[0]+ " "+ array2[1]+ " " + array2[2]);
18
19      // #2. 배열의 원소값 대입 방법 2
20      int[] array3 = new int[] {3, 4, 5};
21      System.out.println(array3[0]+ " "+ array3[1]+ " " + array3[2]);
22
23      int[] array4;
24      array4 = new int[] {3, 4, 5};
25      System.out.println(array4[0]+ " "+ array4[1]+ " " + array4[2]);
26
27      // #3. 배열의 원소값 대입 방법 3
28      int[] array5 = {3, 4, 5};
29      System.out.println(array5[0]+ " "+ array5[1]+ " " + array5[2]);
30
```

■ 실습. 1차원 배열을 생성하는 다양한 방법

■ ValueAssignment.java

```
31         //int[] array6;  
32         //array6 = {3, 4, 5}; // 불가능  
33         //System.out.println(array6[0]+ " "+ array6[1]+ " "+ array6[2]);  
34     }  
35 }
```

■ 참조 변수와 배열 객체의 값 초기화하기

- 스택 메모리 변수를 초기화하지 않으면 메모리 공간은 텅 비어 있다.
- 이 상태에서는 해당 변수를 출력할 때 오류가 발생한다.
- 기본 자료형 변수이든, 참조 자료형 변수이든 모든 변수는 스택 메모리에 위치하고 있다.
- 따라서 모든 변수는 초기화 이후에만 출력할 수 있다.

```
int a;           // 기본 자료형
int[] b;         // 참조 자료형
System.out.println(a); // 오류 발생
System.out.println(b); // 오류 발생
```

■ 참조 변수와 배열 객체의 값 초기화하기

- 기본 자료형 변수는 스택에 실제 데이터값을 저장하므로 초깃값 역시 실제 데이터값(0, -1, false 등)을 저장한다.
- 반면 참조 자료형 변수는 실제 데이터의 위치를 저장하므로 초깃값으로는 실제 데이터값이 아닌 '가리키고 있는 위치가 없음.'을 나타내는 null을 사용한다.
- 정리하면 기본자료형의 초깃값으로는 '값', 참조자료형의 초깃값으로는 'null'을 사용하면 된다.

```
int a = 0;           // 기본 자료형
Int[] b = null;      // 참조 자료형
System.out.println(a); // 0 출력
System.out.println(b); // null 출력
```


■ 참조 변수와 배열 객체의 값 초기화하기

- 실습. 스택 메모리의 초깃값과 참조 자료형의 강제 초깃값

Initialvalue.java

```
1 package sec01_array.EX03_InitialValue;
2 import java.util.Arrays;
3
4 public class InitialValue {
5     public static void main(String[] args) {
6         //#1. stack 메모리값 (강제초기화 되지 않음)
7         int value1;
8         // System.out.println(value1);           //오류
9         int[] value2;
10        // System.out.println(value2);           //오류
11
12        int value3 = 0;
13        System.out.println(value3);              //0
14        int[] value4 = null;
15        System.out.println(value4);              //null
16        System.out.println();
```

■ 참조 변수와 배열 객체의 값 초기화하기

- 실습. 스택 메모리의 초기값과 참조 자료형의 강제 초기값

Initialvalue.java

```
17
18     // #2. heap 메모리의 초기값 (강제초기화)
19     // @기본자료형 배열
20     boolean[] array1 = new boolean[3]; //false로 초기화
21     for(int i=0; i<3; i++) {
22         System.out.print(array1[i]+ " ");
23     }
24     System.out.println();
25
26     int[] array2 = new int[3]; //0으로 초기화
27     for(int i=0; i<3; i++) {
28         System.out.print(array2[i]+ " ");
29     }
30     System.out.println();
31
```

■ 참조 변수와 배열 객체의 값 초기화하기

- 실습. 스택 메모리의 초깃값과 참조 자료형의 강제 초깃값

Initialvalue.java

```
32      double[] array3 = new double[3];           //0.0으로 초기화
33      for(int i=0; i<3; i++) {
34          System.out.print(array3[i]+ " ");
35      }
36      System.out.println();
37
38      //@참조자료형 배열
39      String[] array4 = new String[3]; //null으로 초기화
40      for(int i=0; i<3; i++) {
41          System.out.print(array4[i]+ " ");
42      }
43      System.out.println();
44      System.out.println();
45
```

■ 참조 변수와 배열 객체의 값 초기화하기

- 실습. 스택 메모리의 초깃값과 참조 자료형의 강제 초깃값

Initialvalue.java

```
46      //Tip. 배열을 쉽게 출력할 수 있는 방법
47      System.out.println(Arrays.toString(array1));
48      System.out.println(Arrays.toString(array2));
49      System.out.println(Arrays.toString(array3));
50      System.out.println(Arrays.toString(array4));
51  }
52 }
```

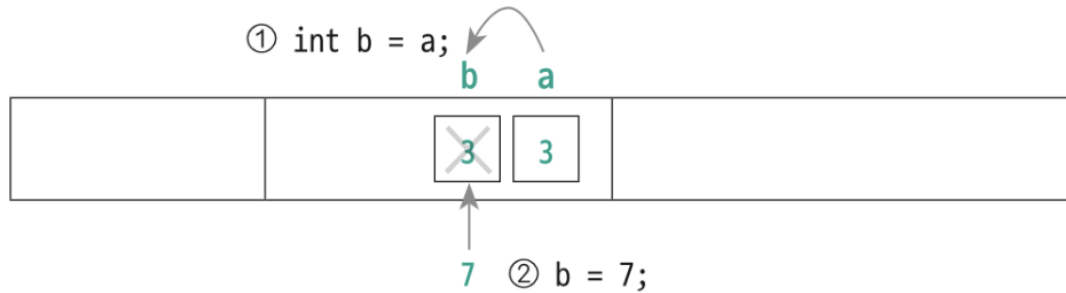
■ 참조 자료형으로서 배열의 특징

- 기본 자료형과 참조 자료형에서 변수를 복사할 때를 비교해 보자.
- 먼저 대입 연산자(=)를 이용해 변수가 복사되는 과정을 이해해야 한다.
- '변수의 복사'라는 말에는 목적어가 빠져 있다.
- 변수의 어떤 값을 복사한다는 의미일까?
- 바로 변수에 포함돼 있는 스택 메모리의 값이다.
- 그런데 기본 자료형과 참조 자료형이 스택 메모리에 저장하는 값의 의미가 다르므로 자연스럽게 둘 사이에 차이가 발생하는 것이다.
- 먼저 기본 자료형을 살펴보자.
- 기본 자료형은 스택 메모리에 실제 데이터값을 저장하고 있으므로 기본자료형 변수를 복사하면 실제 데이터값이 1개 더 복사된다.
- 이후 복사된 값을 아무리 변경해도 원본값은 아무런 영향을 받지 않는다.

5. 배열

■ 참조 자료형으로서 배열의 특징

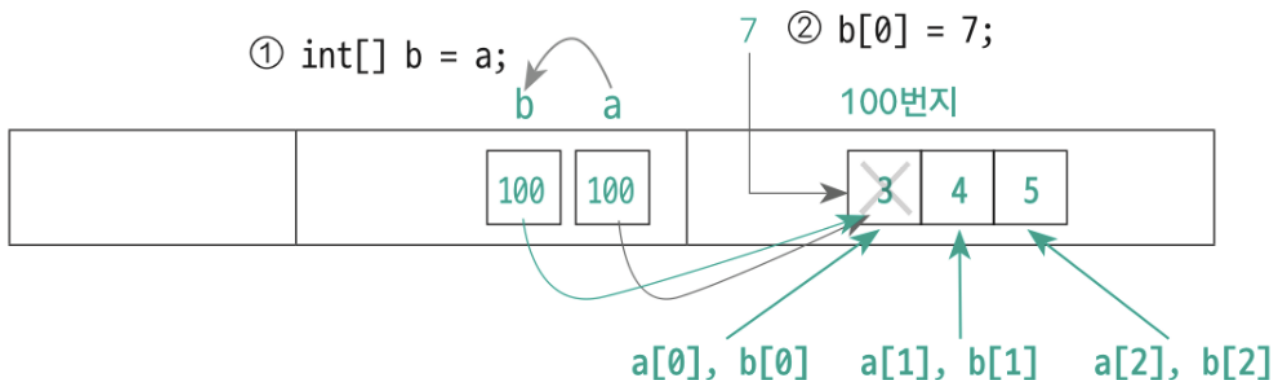
```
int a = 3;  
int b = a;  
b = 7;  
System.out.println(a);    // 3  
System.out.println(b);    // 7
```



■ 참조 자료형으로서 배열의 특징

- 그럼 이번에는 참조 자료형을 살펴보자.
- 참조 자료형은 스택 메모리에 실제 데이터값이 아닌 힙 메모리에 저장된 객체의 위치를 저장하고 있다.
- 따라서 참조 자료형 변수를 복사하면 실제 데이터가 복사되는 것이 아니라 실제 데이터의 위치값이 복사된다.
- 따라서 하나의 참조 변수를 이용해 데이터를 수정하면 다른 참조 변수가 가리키는 데이터도 변하게 되는 것이다.

```
int[] a = {3, 4, 5};  
int[] b = a;  
b[0] = 7;  
System.out.println(a[0]);    // 7  
System.out.println(b[0]);    // 7
```



■ 참조 자료형으로서 배열의 특징

- 실습. 기본 자료형과 참조 자료형의 특징 비교

PrimaryAndReferenceType.java

```
1 package sec01_array.EX04_PrimaryAndReferenceType;
2
3 public class PrimaryAndReferenceType {
4     public static void main(String[] args) {
5         //#1. 기본자료형의 대입연산 (stack 값 복사)
6         int value1 = 3;
7         int value2 = value1;
8         value2 = 7;
9         System.out.println(value1);           //3
10        System.out.println(value2);           //7
11        System.out.println();
12
13        //#2. 참조자료형의 대입연산 (stack 값 복사)
14        int[] array1 = new int[] {3, 4, 5};
15        int[] array2 = array1;
16        array2[0]=7;
```


■ 참조 자료형으로서 배열의 특징

- 실습. 기본 자료형과 참조 자료형의 특징 비교

PrimaryAndReferenceType.java

```
17      System.out.println(array1[0]);          //7
18      System.out.println(array2[0]);          //7
19  }
20 }
```

■ 반복문을 이용해 배열 데이터 읽기

- 배열은 동일한 자료형을 여러 개 묶어 저장한다고 했다.
- 따라서 배열의 모든 데이터를 출력하려면 다음처럼 배열의 길이만큼 출력해야 한다.

```
int[] a = new int[100];  
a[0] = 1, a[1] = 2, ..., a[99] = 100;  
  
System.out.println(a[0]);           // 1  
System.out.println(a[1]);           // 2  
// ...  
System.out.println(a[99]);          // 100
```

- 하지만 이건 아닌 듯하다.
- 하나의 배열 데이터를 출력하기 위해 무려 100줄이나 소비했다.
- 어쩌면 그나마 배열의 길이가 100이어서 다행인지도 모른다.
- 눈치챘겠지만 이럴 때 반복문을 사용하는 것이다.

■ 반복문을 이용해 배열 데이터 읽기

■ 배열의 길이

- 반복의 횟수를 결정하기 위해서는 먼저 배열의 길이를 알아야 한다.
- 물론 배열을 생성할 때 길이가 결정되므로 그 길이만큼 반복문을 수행하면 될 것이다.
- 하지만 많은 배열을 사용할 때 모든 배열의 길이를 일일이 외울 수도 없고, 외울 필요도 없다.
- 자바는 '배열 참조 변수.length'로 배열의 길이를 구할 수 있는 쉬운 방법을 제공한다.
- 여기서 포인트 연산자(.)는 '해당 참조 변수가 가리키는 곳으로 가라.'는 의미다.
- length는 객체에 포함된 읽기 전용 속성으로, 배열 객체의 방 개수에 해당하는 값을 지닌다.
- 따라서 '배열 참조 변수.length'를 풀어 설명하면 '배열 참조 변수가 가리키는 곳에 가면 배열 객체가 있는데, 그 배열의 방의 개수를 가져오라.'는 의미다.

배열 참조 변수.length

예)

```
int[] a = new int[] {3, 4, 5, 6, 7};  
System.out.println(a.length);           // 5
```

■ 반복문을 이용해 배열 데이터 읽기

■ 배열의 길이

- 배열의 길이를 알았으므로 이제 반복문을 활용해 배열의 데이터를 출력해 보자.
- 먼저 배열의 길이로 반복 횟수가 고정되므로 for 문이 적절할 것이다.
- 다음과 같이 작성하면 100줄의 코드가 3줄로 줄어든다.

```
a[0] = 1, a[1] = 2, ..., a[99] = 100;
```

```
for (int i = 0; i < a.length; i++) {  
    System.out.println(a[i]);  
}
```

- for-each 문을 사용하는 방법도 있다.
- for-each 문은 배열이나 컬렉션(collection) 등의 집합 객체에서 원소들을 하나씩 꺼내는 과정을 반복하는 구문으로, 집합 객체의 원소들을 출력할 때 사용한다.

```
for (원소 자료형 변수명: 집합 객체) {  
    ...  
}
```

```
예)  
int[] a = new int[100];  
a[0] = 1, a[1] = 2, ..., a[99] = 100;
```

```
for(int k: a) {  
    System.out.println(k);  
}
```

■ 반복문을 이용해 배열 데이터 읽기

■ 실습. 1차원 배열의 원소 값 출력

ReadArrayData.java

```
1 package sec01_array.EX05_ReadArrayData;
2
3 import java.util.Arrays;
4
5 public class ReadArrayData {
6     public static void main(String[] args) {
7         int[] array = new int[] {3, 4, 5, 6, 7};
8
9         // #1. 배열의 길이 구하기
10        System.out.println(array.length);           //5
11
12        // #2. 출력하기 1:
13        System.out.print(array[0] + " ");
14        System.out.print(array[1] + " ");
15        System.out.print(array[2] + " ");
16        System.out.print(array[3] + " ");
```

■ 반복문을 이용해 배열 데이터 읽기

■ 실습. 1차원 배열의 원소 값 출력

ReadArrayData.java

```
17      System.out.print(array[4]+ " ");
18      System.out.println();
19
20      // #3. 출력하기 2:
21      for(int i=0; i<array.length; i++)
22          System.out.print(array[i]+ " ");
23      System.out.println();
24
25      // #4. 출력하기 3:
26      // for(꺼낸 하나의 원소를 저장할 수 있는 변수:집합객체) {} for each 구문
27      for(int k : array) {
28          System.out.print(k+ " ");
29      }
30      System.out.println();
31
```

■ 반복문을 이용해 배열 데이터 읽기

■ 실습. 1차원 배열의 원소 값 출력

ReadArrayData.java

```
32      // #5. 출력하기 4 :  
33      System.out.println(Arrays.toString(array));  
34  }  
35 }
```

■ 2차원 정방 행렬 배열

- 가로 및 세로 방향의 2차원으로 데이터를 저장하는 배열이 2차원 배열이다.
- 그중 직사각형의 형태(모든 행의 길이가 같은 배열)를 띤 배열을 '2차원 정방 행렬 배열'이라고 한다.

`int[][] a;` →

	0	1	2	3
0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- 2차원 배열을 선언할 때도 배열을 대괄호([])로 표시한다.
- 다만 1차원 배열과 다른 점은 2차원이라는 것을 나타내기 위해 2개의 대괄호를 표시한다는 것이다.
- 대괄호의 위치는 자료형 다음에 올 수 있고, 변수명 뒤에도 올 수 있다.
- 자료형과 변수명 뒤에 각각 하나씩 써도 상관없지만, 일관성을 고려해 자료형 뒤에 쓰는 것을 권장한다.

■ 2차원 정방 행렬 배열

■ 2차원 배열의 선언 방법

2차원 배열의 선언 방법

자료형[][] 변수명	자료형 변수명[][]	자료형[] 변수명[]
예 <code>int[][] a;</code> <code>double[][] b;</code> <code>String[][] c;</code>	<code>int a[][];</code> <code>double b[][];</code> <code>String c[][];</code>	<code>int[] a[];</code> <code>double[] b[];</code> <code>String[] c[];</code>

- 2차원 배열의 선언을 보면 차원이 1개씩 늘어날 때마다 대괄호가 1개씩 늘어난다는 것을 알 수 있다.
- 따라서 3차원 이상의 배열을 선언하는 방법도 쉽게 유추할 수 있을 것이다.
- 대괄호 안에는 배열의 인덱스가 들어가는데, 2차원 배열은 각 위치 정보가 2개의 인덱스 쌍으로 이뤄져 있다.
- 배열의 위치 표현은 세로 방향으로 숫자가 늘어나는 행(row) 번호와 가로 방향으로 숫자가 늘어나는 열(column) 번호로 구성돼 있으며, 각 방향의 인덱스는 0부터 시작한다.
- 예를 들어 `a[2][1]`은 2차원 배열 `a`의 세 번째 행과 두 번째 열을 의미한다.

■ 2차원 정방 행렬 배열

- 2차원 배열의 3가지 선언 방법과 다양한 배열 선언

RectangleArrayDefinition.java

```
1 package sec01_array.EX06_RectangleArrayDefinition;
2
3 public class RectangleArrayDefinition {
4     public static void main(String[] args) {
5         // #1. 배열의 선언 방법 1
6         int[][] array1 = new int[3][4];
7         int[][] array2;
8         array2 = new int[3][4];
9
10        // #2. 배열의 선언 방법 2
11        int array3[][] = new int[3][4];
12        int array4[][];
13        array4 = new int[3][4];
14
15        // #3. 배열의 선언 방법 3
16        int[] array5[] = new int[3][4];
```

■ 2차원 정방 행렬 배열

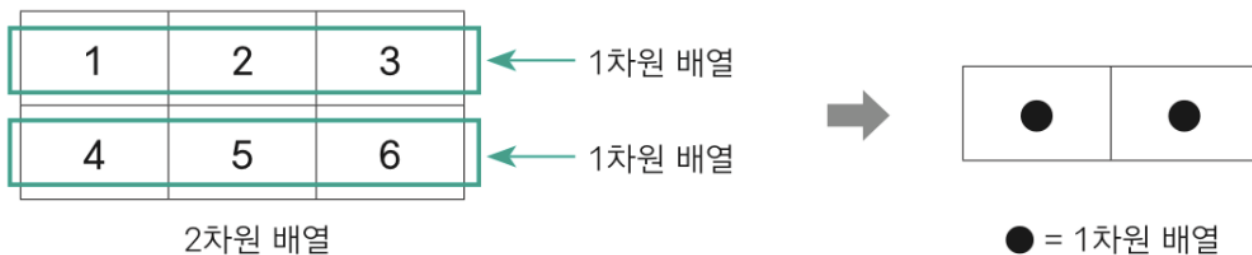
- 2차원 배열의 3가지 선언 방법과 다양한 배열 선언

RectangleArrayDefinition.java

```
17         int[] array6[];  
18         array6 = new int[3][4];  
19  
20         // #4. 다양한 배열 선언 (기본자료형 배열, 참조자료형 배열)  
21         boolean[][] array7 = new boolean[3][4];  
22         int[][] array8 = new int[2][4];  
23         double[][] array9 = new double[3][5];  
24         String[][] array10 = new String[2][6];           // 참조자료형 배열  
25     }  
26 }
```

■ 2차원 정방 행렬 배열

- 2차원 정방 행렬은 객체를 생성하는 데도 4가지 방법이 있다.
- 각 방법을 이해하는 것보다 더욱 중요한 사실은 '메모리는 2차원 데이터를 바로 저장할 수 없다.'는 것이다.
- 실제로 메모리는 1차원 형태의 데이터만 저장할 수 있다.
- 그렇다면 어떻게 2차원 데이터를 저장할까?
- 그 방법은 2차원 데이터를 1차원 데이터들로 나눠 저장하는 것이다.
- 그림과 같은 2X3 크기의 2차원 배열을 살펴보자.



■ 2차원 정방 행렬 배열

- 이 배열의 각 행은 1차원 배열이다.
- 배열의 첫 번째 특징은 동일한 자료형만 묶어 저장할 수 있다는 것이었다.
- 즉, 각각의 행이 1차원 배열이므로 '2차원 배열은 1차원 배열을 원소로 포함하고 있는 1차원 배열'이라고 생각할 수 있다.
- 이 개념을 3차원 배열로 확장하면 3차원은 2차원 배열을 원소로 포함하는 1차원 배열이라고 볼 수 있는 것이다.
- 이러한 개념을 이해해야 2차원 배열의 객체를 생성하는 방법과 메모리에서의 동작을 이해할 수 있다.

■ 2차원 정방 행렬 배열

■ 방법 ① 배열 객체를 생성하고 값 대입하기

- 첫 번째 방법은 2차원 배열 객체를 선언한 후 각각의 인덱스 위치에 값을 하나씩 대입하는 것이다.
- 여기서도 배열의 2가지 특징을 모두 만족한다는 것을 알 수 있다.
- 우선 어떤 자료형을 저장하는지가 선언에 나와 있고, 객체를 생성할 때 배열의 길이가 지정돼 있다.

자료형[][] 참조 변수명 = new 자료형[행의 길이] [열의 길이];

참조 변수명[0][0] = 값;

참조 변수명[0][1] = 값;

...

참조 변수명 [행의 길이 -1] [열의 길이 -1] = 값;

예)

```
int[][] a = new int[2][3];
```

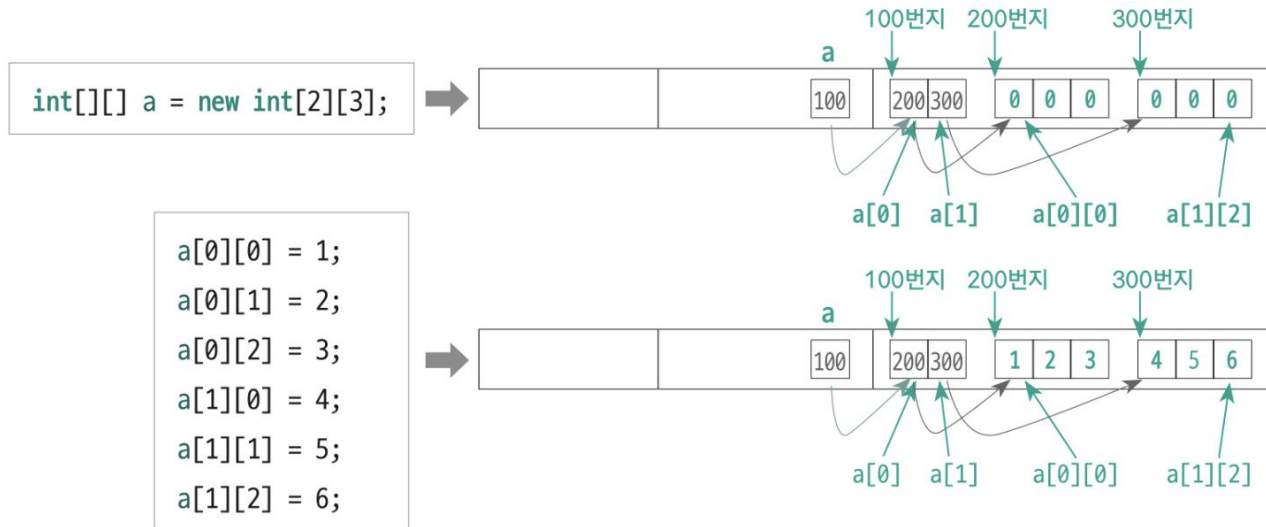
```
a[0][0] = 1; a[0][1] = 2; a[0][2] = 3;
```

```
a[1][0] = 4; a[1][1] = 5; a[1][2] = 6;
```

■ 2차원 정방 행렬 배열

■ 방법 ① 배열 객체를 생성하고 값 대입하기

- 위 예제에서 2차원 배열은 길이가 3인 1차원 배열을 2개 포함하고 있는 1차원 배열로 볼 수 있다.
- 즉, `int[]`가 `int`를 저장하는 1차원 배열인 것처럼 `int[][]`는 `int[]`를 저장하는 1차원 배열로 볼 수 있다는 말이다.
- 따라서 몇 차원의 배열이든 최종적으로는 1차원 배열로 분할할 수 있으며, 이것이 바로 1차원 데이터만 저장할 수 있는 메모리에 다차원 배열을 저장할 수 있는 이유다.
- 이제 다시 메모리의 구조로 돌아가 다음 예제를 이용해 생성되는 메모리의 구조를 살펴보자.



■ 2차원 정방 행렬 배열

■ 방법 ① 배열 객체를 생성하고 값 대입하기

- 2차원 배열의 참조 변수 a는 2개의 원소(1차원 배열)를 포함하고 있는 1차원 배열이므로 참조 변수가 가리키는 곳으로 가면 2개의 방이 있다.
- 이 2개의 방에는 서로 다른 1차원 배열의 위치값이 들어 있다.
- 이 위치값들이 가리키는 또 다른 힙 메모리의 공간에 객체의 실제 데이터값이 들어 있다.
- 메모리의 저장 구조를 이해해야 2차원 배열의 length 속성값을 알 수 있다. '배열 참조 변수.length'는 배열의 길이를 나타낸다고 했다.
- 다시 말해서 배열의 가리키는 곳으로 가서 방의 개수를 알아오는 것이 '배열 참조 변수.length' 명령어다.
- 그렇다면 앞의 예제에서 a.length는 얼마인가?
- 참조 변수 a가 가리키는 곳으로 가면 2칸의 공간이 있다.
- 즉, a.length = 2인 것이다.
- 반면 a[0].length는 a가 가리키는 곳의 첫 번째 방(a[0])이 가리키는 곳의 방 개수를 의미하므로 a[0].length = 3이 된다.
- 이와 같은 방식으로 a[1].length = 3이 된다는 것을 알 수 있다.

```
System.out.println(a.length);           // 2
System.out.println(a[0].length);        // 3
System.out.println(a[1].length);        // 3
```


■ 2차원 정방 행렬 배열

■ 방법 ② 배열 객체의 행 성분부터 생성하고 열 성분 생성하기

- 두 번째 방법은 2차원 배열의 행 성분만 먼저 생성하고, 각 행에 열 성분을 생성하는 것이다.
- 다소 복잡해 보이지만, 앞서 설명한 2차원 배열의 메모리 구조를 이해하면 쉽게 이해할 수 있을 것이다.
- 작성 방법과 예시는 다음과 같다.

```
자료형[ ][ ] 참조 변수명 = new 자료형[행의 길이]  
참조 변수명[0] = 1차원 배열의 생성;  
참조 변수명[1] = 1차원 배열의 생성;  
...  
참조 변수명[행의 길이 - 1] = 1차원 배열의 생성;
```

예)

```
int[ ][ ] a = new int[2][ ];
```

```
a[0] = new int[3];  
a[0][0] = 1; a[0][1] = 2; a[0][2] = 3;  
a[1] = new int[3];  
a[1][0] = 4; a[1][1] = 5; a[1][2] = 6;
```

```
int[ ][ ] a = new int[2][ ];
```

```
a[0] = new int[ ]{1, 2, 3};  
a[1] = new int[ ]{4, 5, 6};
```

■ 2차원 정방 행렬 배열

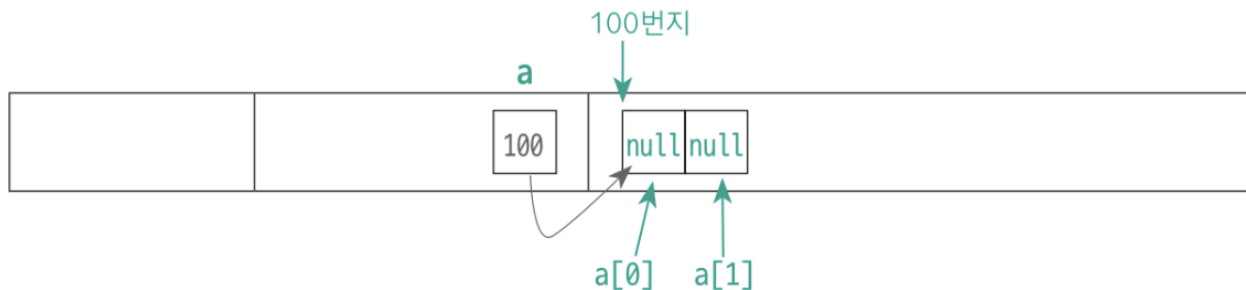
■ 방법 ② 배열 객체의 행 성분부터 생성하고 열 성분 생성하기

- 두 예시는 각 행을 구성하는 1차원 배열을 생성하는 방법에서만 차이가 난다.
- 먼저 2차원 배열을 생성할 때 행의 개수까지만 적고, 열의 개수는 적지 않는다.
- 열의 개수를 나중에 결정한다는 것이다.
- 여기서는 '배열의 두 번째 특징에 따라 생성할 때 크기를 지정해야 하는데, 이렇게 되면 규칙에 어긋나지 않는가?'라고 생각할 수 있다.
- 하지만 그렇지 않다.
- 참조 변수a가 가리키는 곳은 행의 개수만큼 메모리 공간을 차지하는 1차원 배열이기 때문이다.
- 즉, 참조 변수 a를 선언하고 값을 가리키게 하는 데는 행의 길이만 있으면 되는 것이다.

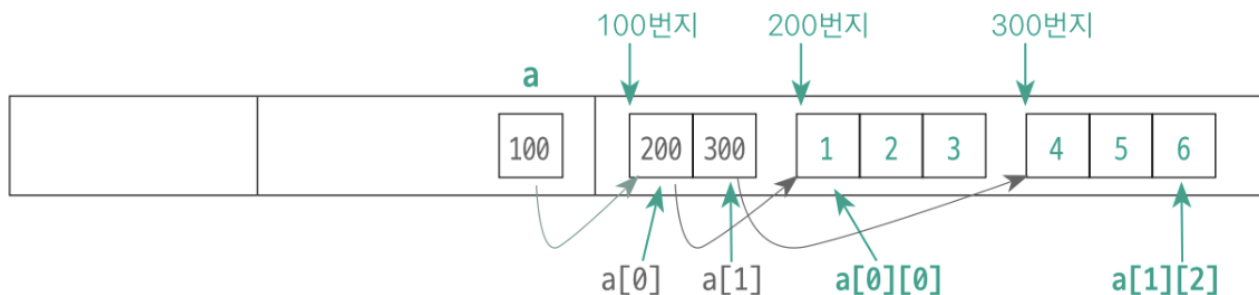
■ 2차원 정방 행렬 배열

■ 방법 ② 배열 객체의 행 성분부터 생성하고 열 성분 생성하기

- 이러한 방법을 이용해 생성되는 메모리의 구조를 살펴보면 다음과 같다.
- 방법 ②를 이용해 2차원 배열 객체의 행 성분을 생성할 때의 메모리 구조



- 방법 ②를 이용해 2차원 배열 객체의 열 성분을 생성할 때의 메모리 구조



■ 2차원 정방 행렬 배열

■ 방법 ② 배열 객체의 행 성분부터 생성하고 열 성분 생성하기

- 우선 행의 개수만 정의해 2차원 배열을 생성하면 첫 번째 메모리의 구조처럼 참조 변수 a가 가리키는 곳으로 갔을 때 2칸의 방이 있다.
- 각 원소는 기본자료형이 아닌 참조 자료형(int[])이고, 힙 메모리는 강제 초기화되는 영역이므로 null이 들어갈 것이다.
- 다음으로 a[0] = new int[3]의 코드를 살펴보자.
- 그대로 풀어쓰면 'int형 데이터 3개를 저장할 수 있는 공간을 만들어 힙 메모리에 넣고(new int[3]) 저장한 위치 정보를 a[0]에 저장하라.'는 의미다.
- 아래쪽 a[1] = new
- int[3]도 이와 동일한 의미다.
- 이후 각각의 2차원 인덱스 위치에 값을 대입한 것이 위의 첫 번째 예제다.
- 두 번째 예제는 각 인덱스 위치에 값을 대입하는 방법만 다를 뿐, 메모리에서 일어나는 모든 과정은 동일하다.
- 방법 ②를 잘 이해하면 뒤에서 배울 2차원 비정방 행렬도 쉽게 이해할 수 있다.
- 배열을 포함한 참조 자료형은 꼭 메모리의 구조와 함께 이해하는 것이 매우 중요하며, 이것이 프로그래밍을 쉽게 이해할 수 있는 유일한 방법이다.

■ 2차원 정방 행렬 배열

■ 방법 ③ 배열의 자료형과 함께 대입할 값 입력하기

- 2차원 정방 행렬의 객체를 생성하는 세 번째 방법은 자료형과 함께 대입할 값을 입력하는 것이다.
- 여기서는 배열의 크기가 대입되는 초깃값의 수에 따라 결정되므로 대괄호 안에는 반드시 크기를 지정하지 말아야한다.

배열의 자료형과 함께 대입할 값 입력하기

```
자료형[][] 참조 변수명 = new 자료형[][] {{값, 값, ..., 값}, ..., {값, 값, ..., 값}} ;
```

이때 배열의 길이는 쓰지 않음.

0번째 행 데이터

마지막 행 데이터

예 `int[][] a = new int[][] {{1, 2, 3}, {4, 5, 6}};`

■ 2차원 정방 행렬 배열

▪ 방법 ③ 배열의 자료형과 함께 대입할 값 입력하기

- 1차원 배열과 동일한 형태이며, 차이점은 초깃값을 구성할 때 중괄호 안에 각각의 중괄호를 넣어 각 행의 데이터를 표현한다는 것이다.
- 최종적으로 메모리에 저장되는 값은 앞의 2가지 방법과 동일하다.
- 또한 초깃값과 함께 자료형을 표현하는 방법 ③은 다음과 같이 선언과 객체의 대입을 분리해 표현할 수 있다.

```
int[ ][ ] a = new int[ ][ ] {{1, 2, 3}, {4, 5, 6}};           // (O)
```

```
int[ ][ ] b;  
b = new int[ ][ ] {{1, 2, 3}, {4, 5, 6}};                   // (O)
```

■ 2차원 정방 행렬 배열

■ 방법 ④ 대입할 값만 입력하기

- 마지막 방법은 2차원 정방 행렬 데이터에 대입할 값만 입력하는 방법이다.
- 이 역시 1차원 배열과 동일한 방식에 중괄호만 이중으로 추가된 형태이므로 쉽게 이해할 수 있을 것이다.

대입할 값만 입력하기

```
자료형[][] 참조 변수명 = {{값, 값, ..., 값}, ..., {값, 값, ..., 값}};
```

0번째 행 데이터

마지막 행 데이터

예 `int[][] a = {{1, 2, 3}, {4, 5, 6}};`

- 역시 메모리에 저장되는 값은 앞의 방법과 동일하다.
- 가장 간단한 형태이지만, 방법 ④는 선언과 값의 대입을 분리할 수 없으며, 선언과 동시에 값을 대입할 때만 사용할 수 있다.

```
int[ ][ ] a = {{1, 2, 3}, {4, 5, 6}};           // (O)
```

```
int[ ][ ] b;  
b = {{1, 2, 3}, {4, 5, 6}};                     // (X)
```

■ 2차원 정방 행렬 배열

- 실습. 2차원 정방 행렬 배열의 4가지 배열 객체 생성 및 원소 값 대입 방법
- `RectangleValueAssignment.java`

```
1 package sec01_array.EX07_RectangleValueAssignment;
2
3 public class RectangleValueAssignment {
4     public static void main(String[] args) {
5         ///#1. 배열객체의 생성 및 원소값 대입 (방법1)
6         int[][] array1 = new int[2][3];
7         array1[0][0]=1;
8         array1[0][1]=2;
9         array1[0][2]=3;
10        array1[1][0]=4;
11        array1[1][1]=5;
12        array1[1][2]=6;
13
14        System.out.println(array1[0][0]+ " "+array1[0][1]+ " "+array1[0][2]+ " ");
15        System.out.println(array1[1][0]+ " "+array1[1][1]+ " "+array1[1][2]+ " ");
```


■ 2차원 정방 행렬 배열

- 실습. 2차원 정방 행렬 배열의 4가지 배열 객체 생성 및 원소 값 대입 방법
- `RectangleValueAssignment.java`

```
16         System.out.println();
17
18         int[][] array2;
19         array2 = new int[2][3];
20         array2[0][0]=1;
21         array2[0][1]=2;
22         array2[0][2]=3;
23         array2[1][0]=4;
24         array2[1][1]=5;
25         array2[1][2]=6;
26
27         System.out.println(array2[0][0]+ " "+array2[0][1]+ " "+array2[0][2]+ " ");
28         System.out.println(array2[1][0]+ " "+array2[1][1]+ " "+array2[1][2]+ " ");
29         System.out.println();
30
```

■ 2차원 정방 행렬 배열

- 실습. 2차원 정방 행렬 배열의 4가지 배열 객체 생성 및 원소 값 대입 방법
- `RectangleValueAssignment.java`

```
31      // #2. 배열객체의 생성 및 원소값 대입 (방법2)
32      int[][] array3 = new int[][] {{1,2,3},{4,5,6}};
33      System.out.println(array3[0][0]+ " "+array3[0][1]+ " "+array3[0][2]+ " ");
34      System.out.println(array3[1][0]+ " "+array3[1][1]+ " "+array3[1][2]+ " ");
35      System.out.println();
36
37      int[][] array4;
38      array4 = new int[][] {{1,2,3},{4,5,6}};
39      System.out.println(array4[0][0]+ " "+array4[0][1]+ " "+array4[0][2]+ " ");
40      System.out.println(array4[1][0]+ " "+array4[1][1]+ " "+array4[1][2]+ " ");
41      System.out.println();
42
43      // #3. 배열객체의 생성 및 원소값 대입 (방법3)
44      int[][] array5 = {{1,2,3},{4,5,6}};
45      System.out.println(array5[0][0]+ " "+array5[0][1]+ " "+array5[0][2]+ " ");
```

■ 2차원 정방 행렬 배열

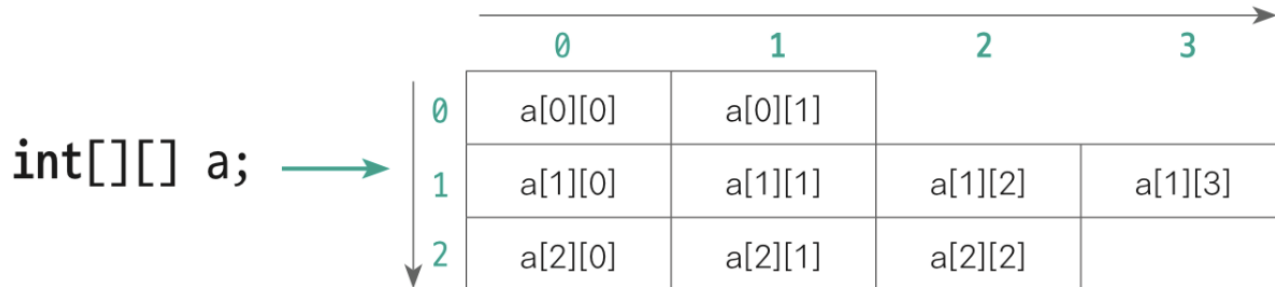
- 실습. 2차원 정방 행렬 배열의 4가지 배열 객체 생성 및 원소 값 대입 방법
- RectangleValueAssignment. java

```
46         System.out.println(array5[1][0]+ " "+array5[1][1]+ " "+array5[1][2]+ " ");
47
48         //int[][] array6;
49         //array6 = {{1,2,3},{4,5,6}}; //불가능
50     }
51 }
```

5. 배열

■ 2차원 비정방 행렬 배열

- 2차원 비정방 행렬은 각 행마다 열의 길이가 다른 2차원 배열을 의미한다.
- 배열의 구조를 보면 각 행별로 들쭉날쭉한 것을 알 수 있다.
- 하지만 기본적인 개념은 2차원 정방 행렬과 완벽하게 동일하다.
- 즉, 1차원 배열을 원소로 포함하고 있는 1차원 배열인 셈이다.
- 원소인 1차원 배열들의 길이가 다양하다는 것에만 차이가 있다.



- 2차원 비정방 행렬 배열의 객체를 생성하는 방법은 3가지다.
- `int[][] a = new int[2][3]`과 같은 정방 행렬 객체 생성의 첫 번째 방법은 항상 정방 행렬만 생성하므로 비정방 행렬 생성 방법으로는 사용할 수 없다.
- 그러면 다음 2차원 비정방 행렬에 관한 배열의 객체를 생성하고, 원소 값을 대입하는 3가지 방법을 살펴보자.

■ 2차원 비정방 행렬 배열

■ 방법 ① 배열 객체의 행 성분부터 생성하고 열 성분 생성하기

- 비정방 행렬 배열을 만드는 첫 번째 방법은 행의 성분만 먼저 생성하고, 각각의 행에 열의 성분을 추가하는 방법이다.
- 각 행마다 길이가 다른 배열을 생성해야 하므로 이렇게 할 수밖에 없다.

배열 객체의 행 성분부터 생성하고 열 성분 생성하기

```
자료형[][] 참조 변수명 = new 자료형[행의 길이][];
```

참조 변수명[0] = 1차원 배열의 생성;

참조 변수명[1] = 1차원 배열의 생성;

...

참조 변수명 [행의 길이-1] = 1차원 배열의 생성;

열의 길이는 표시하지 않음.

예

```
int[][] a = new int[2][];
a[0] = new int[2]; // 첫 번째 행의 열의 개수
a[0][0] = 1; a[0][1] = 2;
a[1] = new int[3]; // 두 번째 행의 열의 개수
a[1][0] = 3; a[1][1] = 4; a[1][2] = 5;
```

= 1차원 배열 생성 방법 1 사용

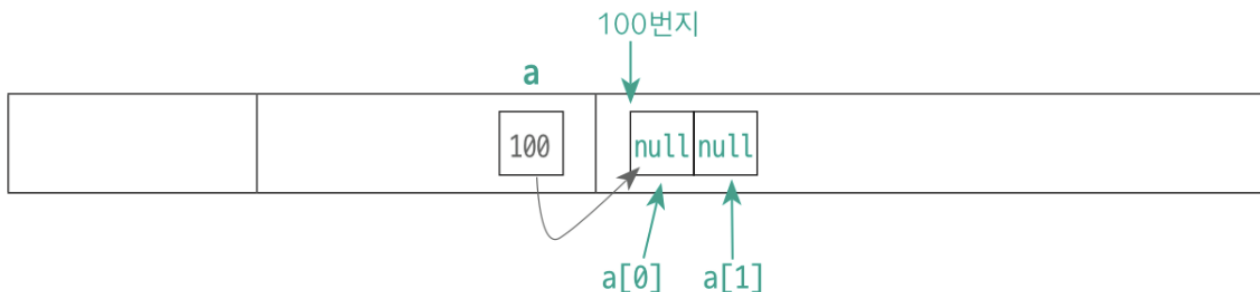
```
int[][] a = new int[2][];
a[0] = new int[]{1, 2};
a[1] = new int[]{3, 4, 5};
```

= 1차원 배열 생성 방법 2 사용

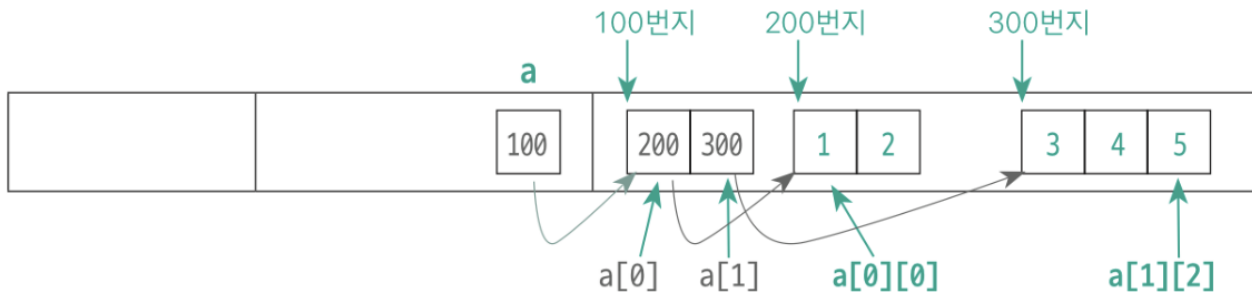
■ 2차원 비정방 행렬 배열

■ 방법 ① 배열 객체의 행 성분부터 생성하고 열 성분 생성하기

- 예제를 살펴보면 각 행마다 크기가 다른 1차원 배열이 할당된다는 점을 제외하고는 정방 행렬과 동일하다.
- 따라서 처음 객체를 생성하면 행 성분의 1차원 배열만 생성되며, 이후 각 행 성분에 1차원 배열을 할당함으로써 최종적인 2차원 배열 객체가 생성된다.
- 방법 ①을 이용해 2차원 비정방 행렬 배열 객체의 행 성분을 생성할 때의 메모리 구조



- 방법 ①을 이용해 2차원 비정방 행렬 배열 객체의 열 성분을 생성할 때의 메모리 구조



■ 2차원 비정방 행렬 배열

■ 방법 ① 배열 객체의 행 성분부터 생성하고 열 성분 생성하기

- 비정방 행렬은 배열의 길이에 주의를 기울여야 한다.
- 각 행마다 배열의 길이가 다르기 때문이다.

자료형과 대입할 값만 입력하기

```
자료형[][] 참조 변수명 = new 자료형[][] {{값, 값, ..., 값}, ..., {값, 값, ..., 값}};
```

배열의 길이는 쓰지 않음.

0번째 행 데이터

마지막 행 데이터

예 `int[][] a = new int[][] {{1, 2}, {3, 4, 5}};`

■ 2차원 비정방 행렬 배열

■ 방법 ② 자료형과 대입할 값만 입력하기

- 방법 ②는 자료형과 함께 이중 중괄호를 사용해 생성할 때 초깃값을 넘겨주는 것이다.
- 대괄호 안에 행렬의 크기를 넣지 않으며, 초깃값에 따라 각 행마다 들어갈 배열의 길이가 결정된다.

자료형과 대입할 값만 입력하기

```
자료형[][] 참조 변수명 = new 자료형[][] {{값, 값, ..., 값}, ..., {값, 값, ..., 값}};
```

배열의 길이는 쓰지 않음.

0번째 행 데이터

마지막 행 데이터

예 `int[][] a = new int[][] {{1, 2}, {3, 4, 5}};`

- 최종적으로 메모리에 저장되는 결과는 앞의 방법과 동일하며, 방법 ②는 다음과 같이 선언과 객체의 대입을 분리해 표현할 수 있다.

배열의 선언과 객체의 대입을 분리해 표현 가능

```
int[][] a = new int[][] {{1, 2}, {3, 4, 5}}; // (○)
```

```
int[][] b;
```

```
b = new int[][] {{1, 2}, {3, 4, 5}}; // (○)
```


■ 2차원 비정방 행렬 배열

▪ 방법 ③ 대입할 값만 입력하기

- 2차원 비정방 행렬 배열의 객체를 생성하는 마지막 방법은 초깃값만 이중 중괄호에 넣어 대입하는 것이다.

대입할 값만 입력하기

자료형[][] 참조 변수명 = {{값, 값, ..., 값}, ..., {값, 값, ..., 값}};

0번째 행 데이터

마지막 행 데이터

예 `int[][] a = {{1, 2}, {3, 4, 5}};`

- 2차원 정방 행렬 때와 마찬가지로 방법 ③은 배열의 선언과 객체 대입을 분리할 수 없다는 제한이 있다.

배열의 선언과 객체의 대입을 분리해 표현 불가능

`int[][] a = {{1, 2}, {3, 4, 5}}; // (○)`

`int[][] b;`

`b = {{1, 2}, {3, 4, 5}}; // (X)`

■ 2차원 비정방 행렬 배열

- 실습. 2차원 비정방 행렬 배열의 3가지 원소 값 대입 방법

NonRectangleArray.java

```
1 package sec01_array.EX08_NonRectangleArray;
2
3 public class NonRectangleArray {
4     public static void main(String[] args) {
5         // #1. 비정방행렬의 선언 및 값 대입 방법1
6         int[][] array1 = new int[2][];
7         array1[0] = new int[2];
8         array1[0][0] = 1;
9         array1[0][1] = 2;
10        array1[1] = new int[3];
11        array1[1][0] = 3;
12        array1[1][1] = 4;
13        array1[1][2] = 5;
14
15        System.out.println(array1[0][0] + " " + array1[0][1]);
```

■ 2차원 비정방 행렬 배열

■ 실습. 2차원 비정방 행렬 배열의 3가지 원소 값 대입 방법

NonRectangleArray.java

```
16      System.out.println(array1[1][0]+ " "+array1[1][1]+ " "+array1[1][2]);
17      System.out.println();
18
19      int[][] array2 = new int[2][];
20      array2[0]=new int[] {1,2};
21      array2[1]=new int[] {3,4,5};
22
23      System.out.println(array2[0][0]+ " "+array2[0][1]);
24      System.out.println(array2[1][0]+ " "+array2[1][1]+ " "+array2[1][2]);
25      System.out.println();
26
27      // #2. 비정방행렬의 선언 및 값 대입 방법2
28      int[][] array3 = new int[][] {{1,2},{3,4,5}};
29      System.out.println(array3[0][0]+ " "+array3[0][1]);
30      System.out.println(array3[1][0]+ " "+array3[1][1]+ " "+array3[1][2]);
31      System.out.println();
```

■ 2차원 비정방 행렬 배열

```
32
33     int[][] array4;
34     array4 = new int[][] {{1,2},{3,4,5}};
35     System.out.println(array4[0][0]+ " "+array4[0][1]);
36     System.out.println(array4[1][0]+ " "+array4[1][1]+ " "+array4[1][2]);
37     System.out.println();
38
39     // #3. 비정방행렬의 선언 및 값 대입 방법3
40     int[][] array5 = {{1,2},{3,4,5}};
41     System.out.println(array5[0][0]+ " "+array5[0][1]);
42     System.out.println(array5[1][0]+ " "+array5[1][1]+ " "+array5[1][2]);
43     System.out.println();
44
45     // int[][] array6;
46     // array6 = {{1,2},{3,4,5}}; //불가능
47 }
48 }
```

■ 2차원 배열의 출력

- 2차원 배열은 가로, 세로 방향으로 데이터가 분포돼 있어 2개의 인덱스를 사용한다.
- 따라서 2차원 배열의 모든 데이터를 출력하기 위해서는 기본적으로 이중 for 문을 사용해야 한다.
- 여기서 중요한 것은 반복 횟수를 지정하는 것이다.
- 앞서 살펴본 2차원 비정방 배열 예제에서는 2개의 행(0행, 1행)에 대해 각각 2회(a[0].length) 및 3회(a[1].length)를 반복해야 한다.
- 따라서 바깥쪽 for 문에는 행의 개수를 나타내는 a.length, 안쪽 for 문에는 각 행별 열의 개수를 나타내는 a[i].length를 사용해야 한다.

이중 for 문을 이용한 2차원 배열 원소 출력

```
int[][] a = {{1, 2}, {3, 4, 5}};

for(int i = 0; i < a.length; i++) {
    for(int j = 0; j < a[i].length; j++) {
        System.out.println(a[i][j]);
    }
}
```

■ 2차원 배열의 출력

- 앞서 1차원 배열에서 살펴본 집합 객체(배열, 컬렉션)의 원소를 1개씩 모두 꺼낼 때까지 반복하는 for-each 문을 사용할 수도 있다.
- 여기서도 이중 for-each 문을 사용해야 한다.
- 2차원 배열에 꺼낸 하나의 원소가 1차원 배열이기 때문이다.
- 이중 for-each 문을 이용한 2차원 배열의 출력 예는 다음과 같다.

이중 for-each 문을 이용한 2차원 배열 원소 출력

```
int[][] a = {{1, 2}, {3, 4, 5}};

for(int[] m: a) {
    for(int n: m) {
        System.out.println(n);
    }
}
```

■ 2차원 배열의 출력

■ 실습. 2차원 배열의 원소 값 출력

ReadArrayData_2D.java

```
1 package sec01_array.EX09_ReadArrayData_2D;
2
3 public class ReadArrayData_2D {
4     public static void main(String[] args) {
5         //#1. 2차원 데이터의 배열의 길이
6         int[][] array1 = new int[2][3];
7         System.out.println(array1.length);           //2
8         System.out.println(array1[0].length);        //3 첫번째 행이 가리키는 1차원 배열의 개수
9         System.out.println(array1[1].length);        //3 두번째 행이 가리키는 1차원 배열의 개수
10        System.out.println();
11
12        int[][] array2 = new int[][] {{1,2},{3,4,5}};
13        System.out.println(array2.length);           //2
14        System.out.println(array2[0].length);        //2 첫번째 행이 가리키는 1차원 배열의 개수
15        System.out.println(array2[1].length);        //3 두번째 행이 가리키는 1차원 배열의 개수
16        System.out.println();
```

■ 2차원 배열의 출력

■ 실습. 2차원 배열의 원소 값 출력

ReadArrayData_2D.java

```
17
18      // #2. 2차원 배열의 출력 방법
19      System.out.print(array2[0][0] + " ");
20      System.out.print(array2[0][1] + " ");
21      System.out.println();
22      System.out.print(array2[1][0] + " ");
23      System.out.print(array2[1][1] + " ");
24      System.out.println(array2[1][2]);
25      System.out.println();
26
27      for(int i=0; i<array2.length; i++) {
28          for(int j=0; j<array2[i].length; j++) {
29              System.out.print(array2[i][j] + " ");           //1,2,3,4,5
30          }
31          System.out.println();
32      }
```


■ 2차원 배열의 출력

■ 실습. 2차원 배열의 원소 값 출력

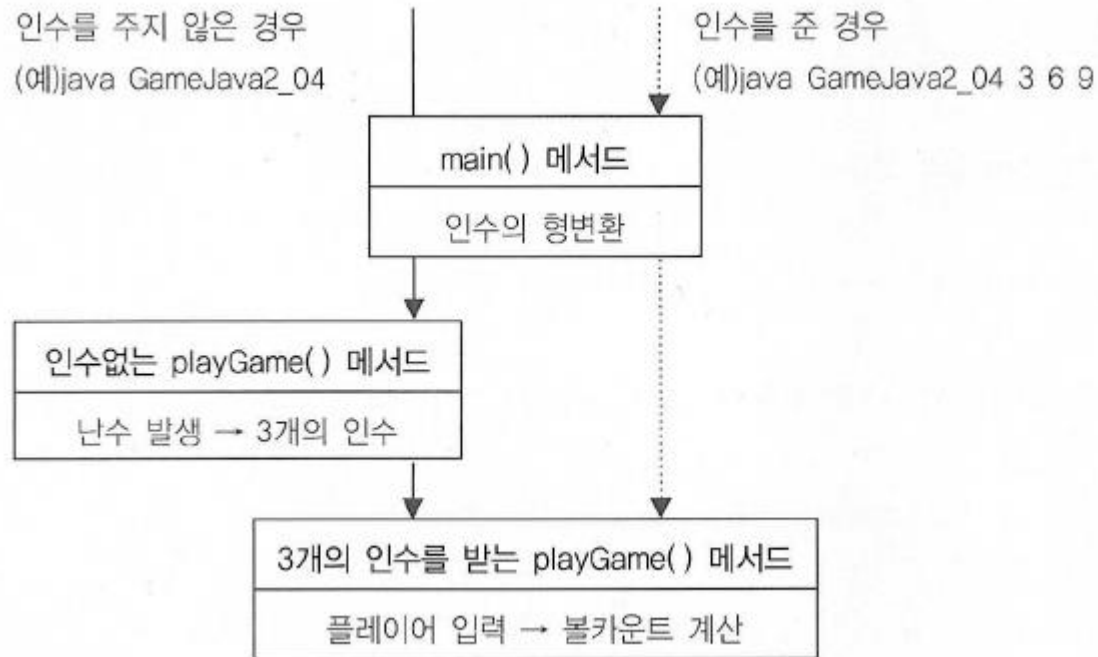
ReadArrayData_2D.java

```
33         System.out.println();
34
35         //for(하나의 원소를 꺼냈을때 저장할 변수:집합객체)
36         for(int[] array: array2) {
37             for(int k:array) {
38                 System.out.print(k+" ");
39             }
40             System.out.println();
41         }
42     }
43 }
```

6. 숫자 야구 게임 만들기

- 숫자 야구 게임은 `main()` 메서드와 두 개의 `playGame()` 메서드로 구성되어 있다.
- `playGame()` 메서드는 3개의 `int`형 변수를 인수로 받는 것과 인수가 없는 두 종류로, 프로그램을 실행시킬 때 3개의 숫자를 인수로 주변 3개의 `int`형 변수를 인수로 받는 `playGame()` 메서드가 호출되고, 인수 없이 프로그램이 실행되면 인수 없는 `playGame()` 메서드가 호출된다.
- 3개의 인수를 받는 `playGame()` 메서드는 주어진 값을 컴퓨터가 숨겨둔 숫자 3개로 간주하고 게임을 실행하고, 인수 없는 `playGame()` 메서드는 난수를 발생해서 3개의 숫자를 정한 후 3 개의 인수를 받는 `playGame()` 메서드에 3개의 숫자를 전달한다.

6. 숫자 야구 게임 만들기



6. 숫자 야구 게임 만들기

- 인수가 없는 playGame() 메서드에서 난수로 숫자 3개를 만들 때는 do-while문을 사용하여 3개의 숫자가 모두 다르도록 조정한다.
- 먼저 1부터 9 사이의 숫자 하나를 난수로 만들어서 x에 할당한다.
- 같은 방법으로 y 값을 구한 후 이미 구한 x와 y의 값을 비교하여 같은 경우에는 다시 y 값을 구하는 일을 반복한다.
- 결국 x와 y 값이 달라질 때까지 난수를 구하는 일을 반복하는 셈이 된다.
- 세 번째 숫자인 x의 경우도 마찬가지이다.
- 다만, z의 경우는 이미 구한 숫자가 x와 y 두 개이기 때문에 x와도 비교하고 y와도 비교해야 한다.
- 이처럼 어떤 일을 일단 한 번 한 후에 조건을 비교해서 반복 여부를 결정할 때는 do-while문이 편리하다.

6. 숫자 야구 게임 만들기

```
int x, y, z;  
Random r = new Random( );  
x = Math.abs(r.nextInt( ) % 9) + 1;  
  
do{  
    y = Math.abs(r.nextInt( ) % 9) + 1;  
}while(y == x);  
  
do{  
    z = Math.abs(r.nextInt( ) % 9) + 1;  
}while((z == x) || (z == y));
```

- 3개의 인수를 받는 playGame() 메서드에서는 주어진 인수를 com 배열에 저장하고, 사용자가 입력한 3개의 수를 입력받아 usr 배열에 저장한다.
- 이때 플레이어(사람)가 입력한 값이 0 또는 9보다 큰 숫자나 같은 숫자가 없도록 앞의 난수 발생 때와 비슷한 방법으로 do-while문을 사용해서 반복하도록 한다.

6. 숫자 야구 게임 만들기

```
do{  
    // 키보드로부터 3개의 숫자를 입력받아 각각 usr[0], usr[1], usr[2]에 저장  
}while((usr[0] == 0) || (usr[1] == 0) || (usr[2] == 0) ||    ← 입력받은 수가 0인 경우  
        (usr[0] > 9) || (usr[1] > 9) || (usr[2] > 9) ||    ← 입력받은 수가 9보다 큰 경우  
        (usr[0] == usr[1]) || (usr[1] == usr[2]) || (usr[0] == usr[2])); ← 입력받은 수가 같은 경우
```

- 무사히 3개의 값을 모두 입력받으면, com 배열의 수와 usr 배열의 수를 비교해서 위치와 값이 같으면 strike 값을 증가시키고 값은 같지만 위치가 다르면 ball 값을 증가시키는 방법으로 볼카운트를 구한다.
- Strike 값이 3개면 게임이 종료되고, 그렇지 않은 경우엔 볼카운트를 보여줘서 플레이어다 다시 한 번 숨겨진 숫자를 추측할 수 있도록 한다.
- 총 11회의 기회를 주고 그 안에 답을 못 맞히면 적절한 메시지를 출력하고 프로그램을 끝낸다.

6. 숫자 야구 게임 만들기

```
1  import java.util.*;
2
3
4  public class GameJava2_05 {
5      public static int playGame() throws {
6          int x, y, z;
7          Random r = new Random();
8          x = Math.abs(r.nextInt() % 9) + 1;
9          do {
10             y = Math.abs(r.nextInt() % 9) + 1;
11             }while(y == x);          // x값과 y값이 같지 않도록(다를 때까지) 반복
12             do {
13                 z = Math.abs(r.nextInt() % 9) + 1;
14             }while((z == x) || (z == y));    // x, y, z 값이 같지 않도록 반복
15             System.out.println(x + ", " + y + ", " + z);
16             return playGame(x, y, z);
17         }
```

6. 숫자 야구 게임 만들기

```
18 public static int playGame(int x, int y, int z) {
19     int count;           // 문제를 푼 횟수
20     int strike, ball;
21     int[] usr = new int[3];    // 사용자가 입력한 숫자 3개
22     int[] com = {x, y, z};    // 컴퓨터가 숨긴 숫자 3개
23
24     System.out.println("숫자 야구 게임");
25
26     count = 0;
27
28     do {
29         count++;
30         do {
31             System.out.println("ㄱ카운트: " + count);
32             Scanner stdIn = new Scanner(System.in));
33             String user;
```


6. 숫자 야구 게임 만들기

```
34         System.out.print("1번째 숫자: ");
35         user = stdIn.nextLine();           // 키보드로 1번째 수 입력
36         usr[0] = new Integer(user).intValue();    // 입력받은 문자를
int형 숫자로 변환
37
38         System.out.print("2번째 숫자: ");
39         user = stdIn.nextLine();           // 키보드로 2번째 수 입력
40         usr[1] = Integer.valueOf(user).intValue();    // 입력받은 문자를
int형 숫자로 변환
41
42         System.out.print("3번째 숫자: ");
43         user = stdIn.nextLine();           // 키보드로 3번째 수 입력
44         usr[2] = new Integer(user).intValue();    // 입력받은 문자를
int형 숫자로 변환
45
```

6. 숫자 야구 게임 만들기

```
46         if((usr[0] == 0) || (usr[1] == 0) || (usr[2] == 0)) {
47             System.out.println("0은 입력하지 마세요. 다시 입력해주세요.");
48         }else if((usr[0] > 9) || (usr[1] > 9) || (usr[2] > 9)) {
49             System.out.println("1부터 9까지의 숫자 중 하나를 입력해주세요. 다시 입력해주세요.");
50         }
51     }while((usr[0] == 0) || (usr[1] == 0) || (usr[2] == 0) ||
52           (usr[0] > 9) || (usr[1] > 9) || (usr[2] > 9) ||
53           (usr[0] == usr[1]) || (usr[1] == usr[2]) || (usr[2] == usr[0]));
54     // 입력받은 답에 이상이 없을 때까지 반복
55
56     strike = ball = 0;           // 볼카운트 초기화
57
58     if(usr[0] == com[0]) strike++;
59     if(usr[1] == com[1]) strike++;
60     if(usr[2] == com[2]) strike++;
```

6. 숫자 야구 게임 만들기

```
61
62         if(usr[0] == com[1]) ball++;
63         if(usr[0] == com[2]) ball++;
64         if(usr[1] == com[0]) ball++;
65         if(usr[1] == com[2]) ball++;
66         if(usr[2] == com[0]) ball++;
67         if(usr[2] == com[1]) ball++;
68
69         System.out.println("Strike: " + strike + " Ball: " + ball); // 볼카운트
출력
70
71         }while((strike < 3) && (count < 11)); // 답을 맞혔거나 10번이상 시
도해서 못맞출 때까지 반복
72
73         return count; // 문제를 맞히려고 시도한 횟수를 반환
74     }
75
```

6. 숫자 야구 게임 만들기

```
76 public static void main(String[] args) {
77     int result;
78
79     if(args.length == 3) {           // 인수가 있는 경우
80         int x = Integer.valueOf(args[0]).intValue();
81         // 인수는 String형이므로 int형으로 형변환
82         int y = Integer.valueOf(args[1]).intValue();
83         int z = Integer.valueOf(args[2]).intValue();
84
85         result = playGame(x, y, z); // 인수를 playGame() 메서드에 전달
86     }else {
87         result = playGame();        // 인수없는 playGame() 메서드 호출
88     }
89
90     System.out.println();
```

6. 숫자 야구 게임 만들기

```
91         if(result <= 2) {           // 문제를 푼 횟수에 따라 칭찬 메시지 출력
92             System.out.println("참 잘했어요!");
93         }else if(result <= 5) {
94             System.out.println("잘했어요!");
95         }else if(result <= 9) {
96             System.out.println("보통이네요!");
97         }else {
98             System.out.println("분발하세요!");
99         }
100     }
101 }
```



Thank You
