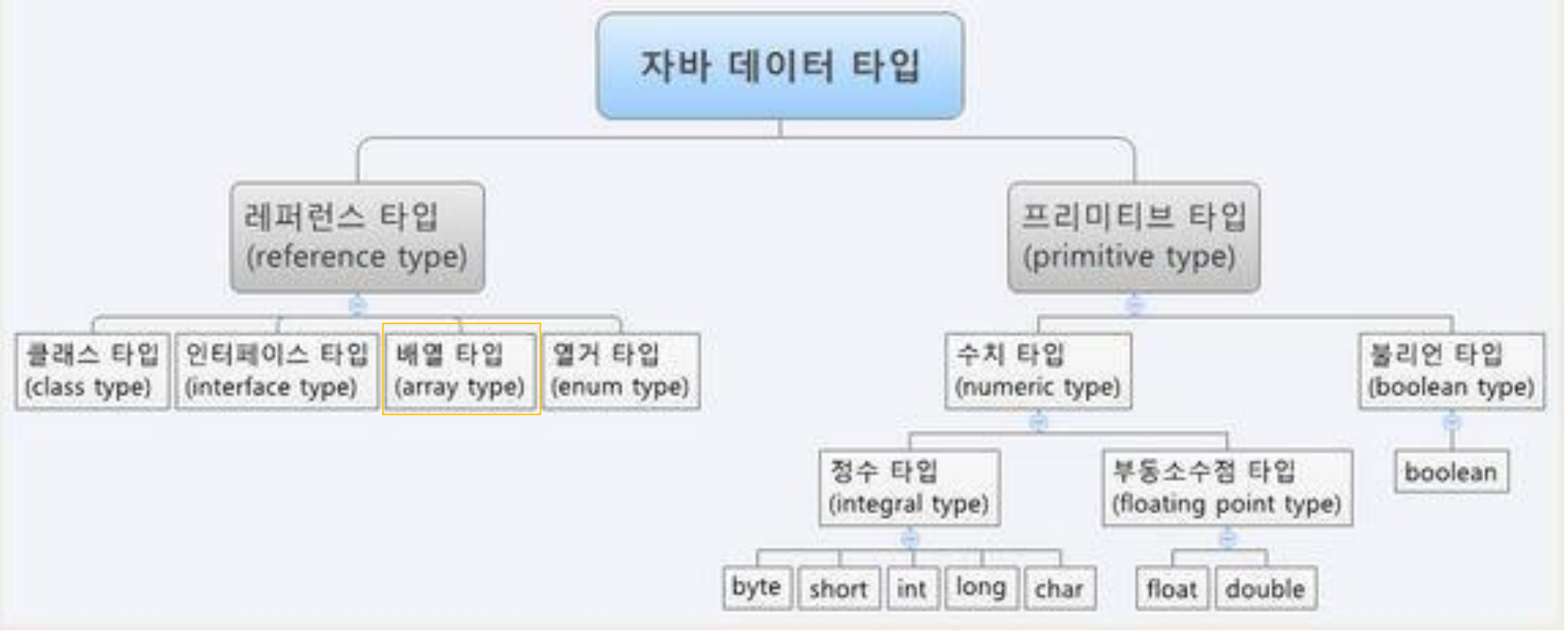
Java 기초

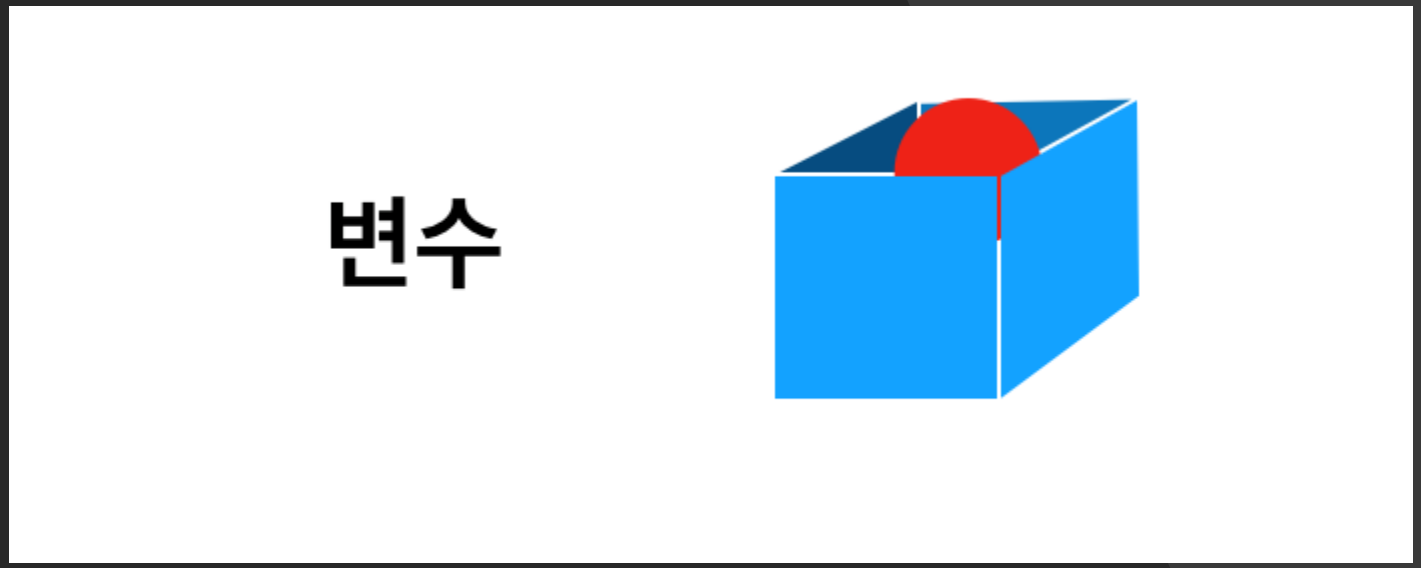
배열

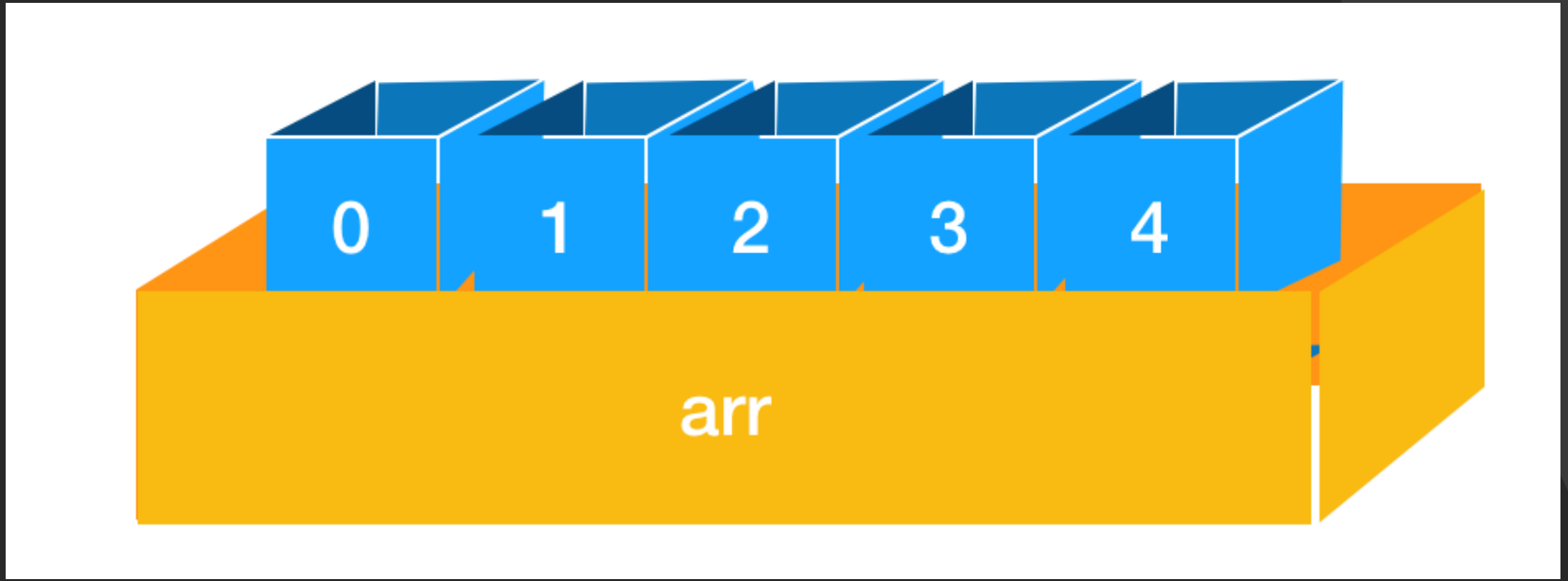
Java 변수 타입


자바 데이터 타입
레퍼런스 타입
(reference type)
프리미티브 타입
(primitive type)
클래스 타입
(class type)
인터페이스 타입
(interface type)
배열 타입
(array type)
열거 타입
(enum type)
수치 타입
(numeric type)
불리언 타입
(boolean type)
정수 타입
(integral type)
부동소수점 타입
(floating point type)
boolean
byte
short
int
long
char
float
double

배열(Array)

- 배열이란?



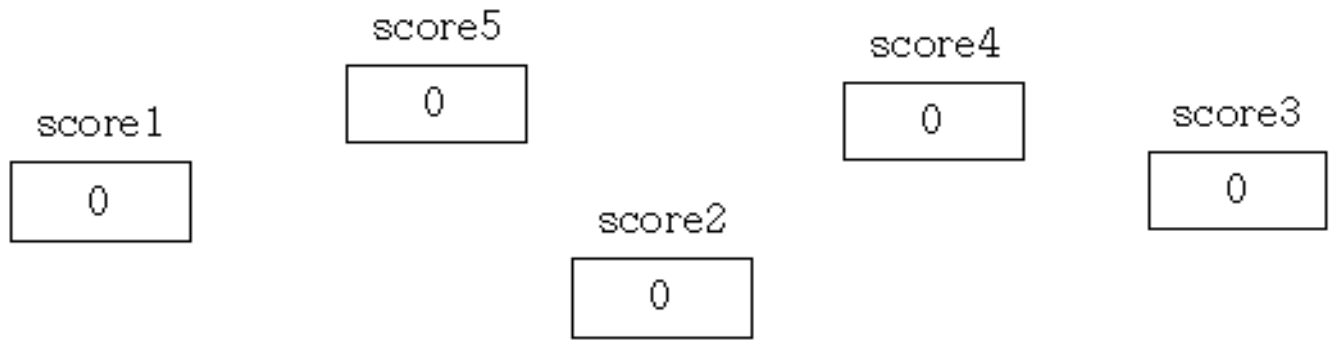


배열(Array)

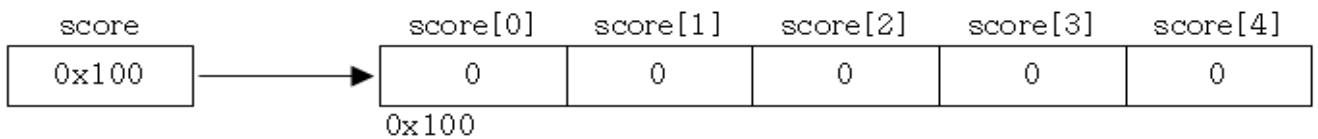
- 배열이란?
  - 하나의 변수 안에 연속된 여러 개의 공간을 갖고있는 변수(과거)
  - 같은 타입의 값 여러 개를 가질 수 있는 변수
  - 같은 타입의 여러 변수를 하나의 묶음으로 다루는 것
  - 많은 양의 같은 타입의 데이터를 다룰 때 유리하다.
  - 배열의 각 요소는 서로 연속적이다.
  - 배열은 여러 개의 변수를 다뤄서 표현할 필요가 없이 하나의 변수만으로 연속된 공간을 선언하여 여러 개의 값을 관리 할 수 있기 때문에 훨씬 가독성이 좋고 편리하다.

배열(Array)

```
int score1=0, score2=0, score3=0, score4=0, score5=0 ;
```




```
int[] score = new int[5]; // 5개의 int 값을 저장할 수 있는 배열을 생성한다.
```



배열(Array)

- 배열 변수 선언 방법
  - 배열 변수 선언 시 반드시 []가 들어간다.
  - 타입 바로 뒤에 와도 상관없고 변수 이름 뒤에 와도 크게 상관없다.

선언방법	선언 예
타입[] 변수이름;	**int[] score;** **String[] name;**
타입 변수이름[];	int score[]; String name[];

[표5-1] 배열의 선언방법과 선언 예

배열(Array)

- 배열 초기화 방법
  - 배열 초기화는 다양한 방법이 있지만 대표적으로는 두 가지 방식이 존재하며 공간만 할당하는 [공간 할당 방식]과 값을 직접 삽입하는 [직접 삽입 방식]이 있다.
  - 이 두가지 방식 중 어떤 방식을 사용하는가에 따라 초기화가 달라진다.
  - 보통 두 방식 골고루 쓰이는데 어떠한 차이점이 있는지 알고 쓰는 것이 더 중요하다.

배열(Array)

- 공간 할당 방식
  - 배열에 몇 개의 공간이 있는지를 선언
  - 값을 나중에 초기화 할 경우 유용하게 사용된다.

```
타입[] 변수명 / 타입 변수명[] = new 타입[선언할 공간 개수];
```

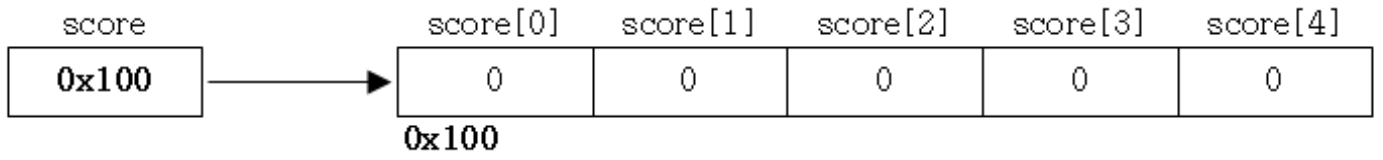
  - 이런식으로 공간만 선언할 경우 각 공간은 타입마다 다른 값으로 초기화 된다.
  - 초기화 값은 다음과 같이 된다.
    - 정수 : 0, 실수 : 0.0, 논리형 : false, 문자형 : ₩u0000, 객체형 : null;

배열(Array)

- 공간 할당 방식

```
int[] score;          // 배열을 선언한다.(생성된 배열을 다루는데 사용될 참조변수 선언)
score = new int[5]; // 배열을 생성한다.(5개의 int값을 저장할 수 있는 공간생성)
```

[참고] 위의 두 문장은int[] score = new int[5];와 같이 한 문장으로 줄여 쓸 수 있다.



배열(Array)

- 공간 할당 방식
  - 배열에 들어있는 값에 접근하려면 다음과 같이 할 수 있다.

```
배열변수명[0부터 배열 길이-1 까지의 자연수]
```

  - 배열은 인덱스란 값을 갖고 있으며 인덱스란 값에 접근할 수 있는 주소를 의미한다.
  - 인덱스는 보통 0부터 [배열 길이-1] 까지의 정수형 주소를 가진다.
  - 해당 인덱스를 통해 접근해야 배열의 특정 영역의 값에 접근할 수 있다.

배열(Array)

공간 할당 방식 예제

```
Run | Debug
public static void main(String[] args) {
    // java 배열 선언 예제
    // 실제 아래와 같은 배열을 선언하게 되면
    // 공간이 다섯개 존재하는 배열이 선언된다.
    int[] a = new int[5];
    // 일반 변수처럼 배열의 변수에 접근할 경우
    // 이상한 값이 출력되게 된다. 배열의 아이템에 접근하기
    // 위해서는 인덱스라는 값을 써야하는데
    // 인덱스는 반드시 0부터 배열 크기-1사이의 자연수가 되어야 한다.
    // 즉 a[인덱스] 를 통해 정상적인 접근이 가능하다.
    System.out.println(a);  // ?
    // 맨 처음 공간 선언 방식으로 선언된 배열에 접근하게 되면
    // a배열 안의 값은 디폴트 값으로 배열이 초기화 된 것을
    // 확인할 수 있다.
    // 디폴트 값은 정수는 0, 실수는 0.0, 캐릭터는 \u0000
    // 논리형은 false, 객체형은 null로 초기화 된다.
    System.out.println(a[0]);  // 0
    // 대입 연산자를 활용하여 배열의 특정 부위에 데이터를 넣고
    // 싶다면 아래와 같이 할 수 있다.
    a[1] = 3;
    System.out.println(a[1]);  // 3
}
```

배열(Array)

- 직접 선언 방식
  - 선언과 즉시 값을 임의로 대입하는 방법
  - 배열에 값을 즉시 삽입할 경우 이 방법을 사용할 수 있다.

```
타입[] 변수명 / 타입 변수명[] = { 나열하고자 하는 데이터 };
```

  - 나열하고자 하는 데이터는 반드시 타입과 맞아야 한다.
  - 데이터와 데이터 간에는 ‘,’로 구분하여 넣는다.
  - 데이터들 양 변에는 브레이스‘{}’를 붙인다.
  - 위의 타입은 변형이 가능하다.

배열(Array)

직접 선언 방식 예제

```
Run | Debug
public static void main(String[] args) {
    // Array 직접 선언 방식 예제
    // 직접 선언 방식은 우측에 값을 놓음으로써
    // 배열에 값을 직접 넣어 선언하는 방식이다.
    // 선언은 아래와 같이 할 수 있다.
    int[] a = {5,4,3,2,1};
    // 배열의 숫자에 접근하게 되면 기존에 초기화
    // 시켰던 데이터가 나온다.
    System.out.println(a[0]);  // 5
    // 이처럼 배열의 초기화를 시작하자마자 시킬
    // 경우 0으로 초기화 된 값이 아닌 선언 시
    // 초기화 시킨 값이 나오게 된다.
    // 직접 선언 방식은 기존의 방식도 있지만
    // 다른 형태로도 선언이 가능하다.
    int[] b = new int[]{1,2,3,4,5};
    System.out.println(b[4]);
    // 만약 배열에 대한 변수만 선언해놓고
    // 값을 나중에 삽입하려는 경우 아래처럼 하지
    // 않으면 에러가 난다.
    int[] c;
    c = new int[]{10,9,8,7,6};
}
```

배열의 활용

- 배열과 for문
  - 배열과 for문은 시너지가 가장 잘 맞는다.
  - 배열의 인덱스는 0부터 순차적으로 이동을 하고 이 접근을 for문의 초기값으로 접근하게 할 수 있다.
  - 이런 패턴은 다른 언어에서도 많이 보이며 배열의 순차적인 접근을 필요로 할 경우 상당히 많이 쓰이는 패턴이기도 하다.

배열의 활용

배열과 for문 예제

```
Run | Debug
public static void main(String[] args) {
    // 배열 for문 조합
    // for문의 초기값 변수로 배열의 인덱스를
    // 대체할 수 있다. 이런 성질을 이용해서 순차적인
    // 출력이 가능하도록 할 수가 있다.
    int[] a = {1,2,3,4,5};
    for (int i = 0; i < 5; i++) {
        System.out.println(a[i]);
    }
    // 주요한건 만약 배열의 길이를 모를 경우이다.
    // 혹은 배열의 길이를 다른곳에서도 사용할 경우
    // 교체해야 될 소스가 많아질 것이고 비효율적이 된다.
    // 이럴때 array의 length라는 속성을 사용할 수 있다.
    // length는 배열의 길이를 출력해주는 속성이며 이 속성
    // 을 통해 for문의 조건식에 length이전까지 반복을
    // 돌리도록 할 수가 있다.
    System.out.println(a.length);  // 5
    for (int i = 0; i < a.length; i++) {
        System.out.println(a[i]);
    }
}
```

배열의 활용

- foreach
  - 자바 1.6 이후에 패치된 내용
  - 배열을 순회하는 로직이 많아지면서 이 로직을 조금 더 수월하게 하기 위한 반복문의 형태
  - 모양은 아래와 같다.

```
for(타입 참조할변수명 : 배열변수){ 반복할 데이터 };
```

  - 배열의 타입은 반드시 배열 변수가 들어있는 타입과 일치해야 한다.

배열의 활용

- foreach 예제

```
Run | Debug
public static void main(String[] args) {
    // forEach 예제
    int[] a = {1,2,3,4,5};
    // a에 들어있는 데이터를 하나하나 var에 삽입해서
    // a에 들어있는 데이터를 처음부터 끝까지 순회할 경우
    // foreach 문을 사용한다.
    for (int var : a) {
        System.out.println(var);
    }
}
```

배열 a = 1,2,3,4,5

1번째 루프

int var <= 배열 a의 첫번째 값 삽입

var 출력 => 1

2번째 루프

int var <= 배열 a의 두번째 값 삽입

var 출력 => 2

3번째 루프

int var <= 배열 a의 세번째 값 삽입

var 출력 => 3

4번째 루프

int var <= 배열 a의 네번째 값 삽입

var 출력 => 4

5번째 루프

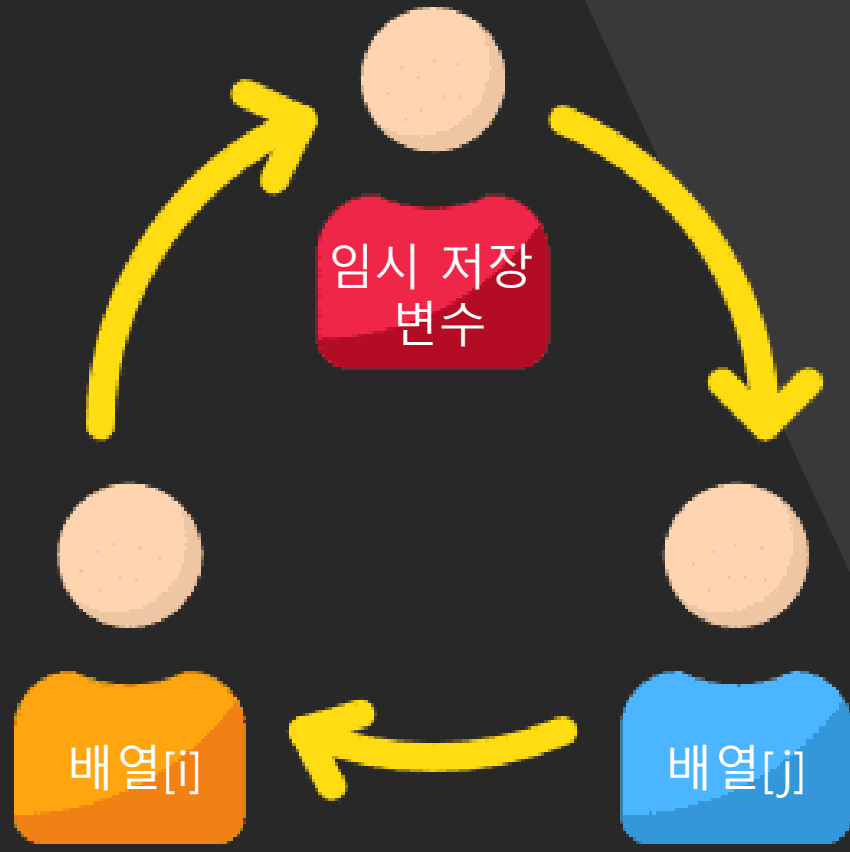
int var <= 배열 a의 다섯번째 값 삽입

var 출력 => 5

루프 종료

배열의 활용

- swap
  - 배열에 들어있는 데이터의 위치를 바꿀 때 보통 이 로직을 쓴다.
  - 이 패턴은 상당히 자주 사용되는 패턴이며 swap을 통해 서로 다른 자리에 있는 데이터의 위치를 바꿀 수 있다.
  - swap을 하기 위해선 반드시 임시 저장시켜야 하는 변수가 존재해야 한다.



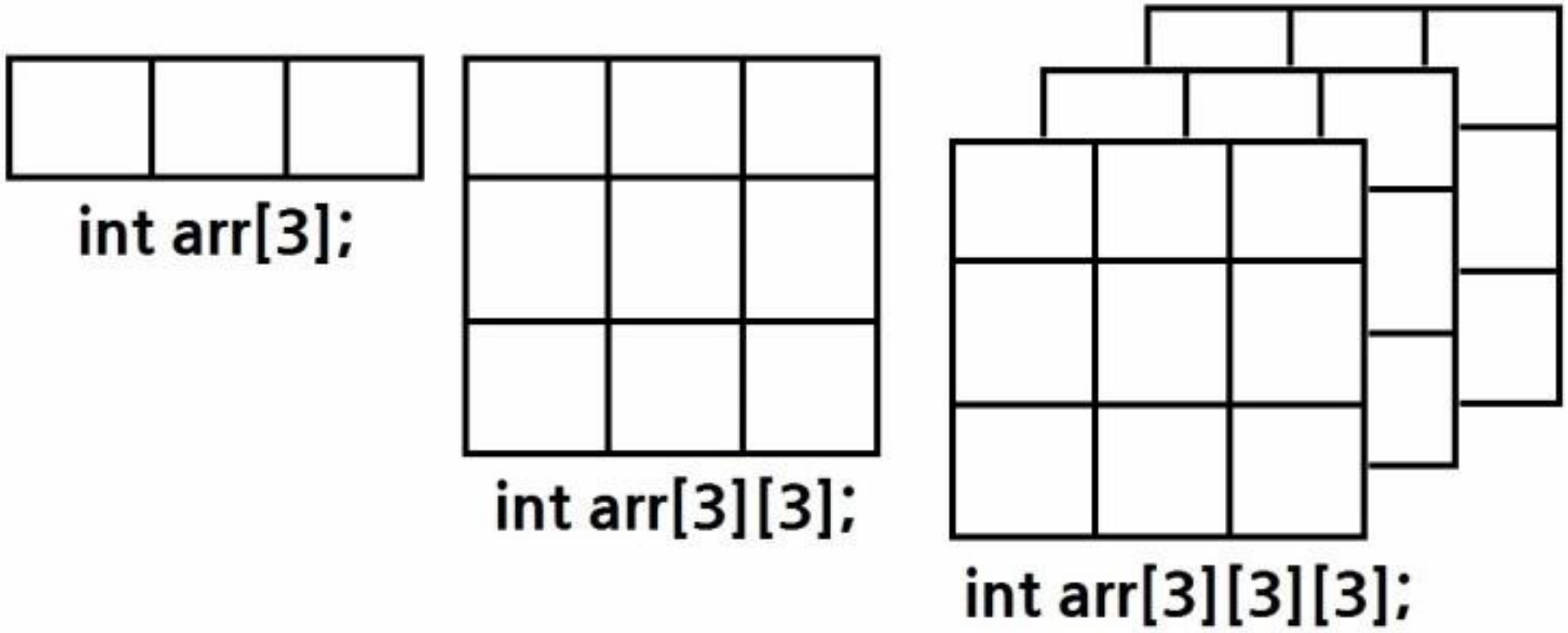
배열의 활용

Swap 예제

```
Run | Debug
public static void main(String[] args) {
    // swap 예제
    // swap은 맨 처음수와 그 다음 수를 바꾸기 위해 쓴다.
    // swap을 하는 순서는 다음과 같다.
    // 1. 두 배열 에서 하나의 배열 인덱스에 들어있는
    //    값을 임시 저장 변수에 넣는다.
    // 2. 임시저장된 인덱스 장소의 값을 바꾸고 싶어하는
    //    값으로 교체한다.
    // 3. 임시 저장된 값을 기존 바꿨던 배열의 인덱스로
    //    교체해서 참조한다.
    // swap시 참조하고자 하는 배열의 인덱스를 벗어나지 않기 위해
    // 배열 순회하려는 전체 length에서 -1을 시킨다.
    // 다음은 처음 숫자를 맨 끝으로 옮기는 로직이다.
    // 인덱스 넣는 란에 변수 뿐만이 아닌 산술 연산도 넣는것이 가능하다.
    int[] a = {1,2,3,4,5};
    int temp = 0;
    for (int i = 0; i < a.length-1; i++) {
        temp = a[i];
        a[i] = a[i+1];
        a[i+1] = temp;
    }
    System.out.println(a[4]);
}
```

다차원 배열

다차원 배열



다차원 배열

- 다차원 배열
  - 배열안에 배열이 들어가 있는 형태
  - 배열안에 배열을 넣어 배열을 한 변수 안에 복수로 참조할 수 있도록 한다.
  - 다차원 배열은 배열안에 배열이 들어가 있는 형태이므로 복잡도가 기하급수적으로 상승한다.
  - 그렇기 때문에 3차 배열 이상이 넘어가게 되면 눈으로 트래킹 할 수 있는 방법이 전무하다.

다차원 배열

- 다차원 배열 선언 방법

선언방법	선언에
타입[][] 변수이름;	int[][] score;
타입 변수이름[][];	int score[][];
타입[] 변수이름[];	int[] score[];

다차원 배열

- 다차원 배열 선언 방법
  - 공간 할당 방식

```
// 다차원 배열 공간 할당 방식
int[][] a = new int[3][3];
```

  - 직접 선언 방식

```
// 다차원 배열 직접 삽입 방식
int[][] b = {{1,2,3},{4,5,6},{7,8,9}};
```

다차원 배열

- 다차원 배열 선언 방법
  - 다차원 배열을 선언할 때 변수는 타입 혹은 변수 명 뒤에 []를 다수개를 놓아 선언한다.
  - 2차원 배열이면 2개 3차원 배열이면 3개 n차원이면 n개의 []를 놓는다.
  - 공간 할당 방식과 직접 삽입 방식은 크게 차이가 있는데 공간 할당 방식은 기존에 배열 선언에서 []를 다차원 개수 만큼 더 추가해서 그 다음 배열에 공간을 몇 개 넣을지 확인한다.
  - 예를 들면 new int[5][3] 이면 5개의 배열 안에 3개의 배열이 추가적으로 있는 형태가 된다.

다차원 배열

- 다차원 배열의 선언 방법
  - 반면 직접 선언 방식은 브레이스({})안에 브레이스를 놓는 형식이다
  - 예를 들면 3x3 배열을 선언하려고 한다면 {}안에 {} 세개를 놓고 각각 하나의 {}에 3개의 데이터를 넣는 형태가 된다.
  - 즉 1부터 9까지 3x3 배열로 넣는다면 {{1,2,3},{4,5,6},{7,8,9}}형태로 넣을 수 있다.

다차원 배열

- 다차원 배열 접근 방법

```
배열변수명[인덱스][인덱스]...[인덱스]
```

  - 다차원 배열 접근 방법은 기존의 방법에서 []가 차원수 만큼 추가된 형태가 된다.
  - 차원수보다 더 많은 []를 통해 참조했을 시 에러가 난다.
  - 차원수보다 더 적은 []를 참조했을 시 남은 []의 배열을 참조한다.

다차원 배열

다차원 배열 접근 방법 예제

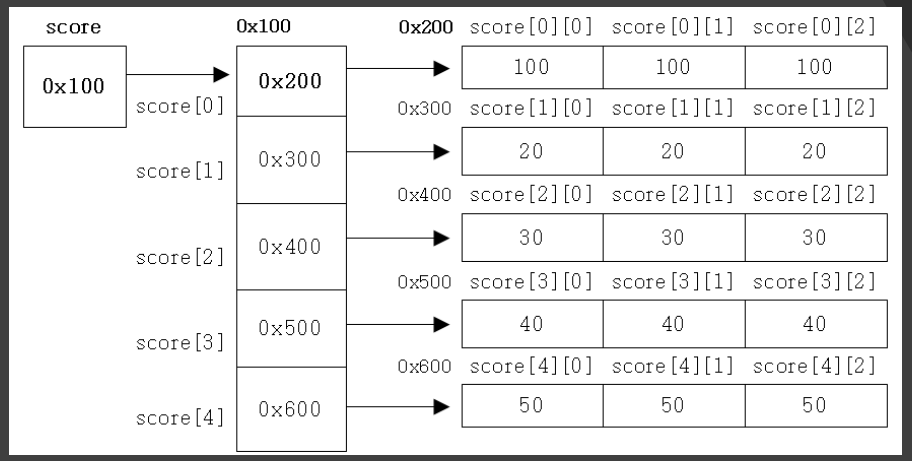
```
Run | Debug
public static void main(String[] args) {
    // 다차원 배열 공간 할당 방식
    int[][] a = new int[3][3];
    // 다차원 배열 접근 방법
    System.out.println(a[0][0]);  // 0

    // 다차원 배열 직접 삽입 방식
    int[][] b = {{1,2,3},{4,5,6},{7,8,9}};
    // 다차원 배열 접근 방법
    System.out.println(b[0][0]);  // 1

    // 배열의 차원수를 넘어서 참조는 불가능하다.
    // System.out.println(b[1][2][3]);

    // 선언된 배열보다 적은 차원을 선언하면
    // 그 인덱스 안에 들어있는 배열이 호출된다.
    int[] c = b[1];
    for (int i : c) {
        System.out.print(i+"\t");  // 4 5 6
    }
    System.out.println();
    // 만약 3차원 배열을 선언할 경우
    // 참조하려는 차원이 1차원일 경우 2차원의 배열이
    // 출력된다.
    int[][][] d = new int[3][3][3];
    d[0][0][0] = 3;
    int[][] e = d[0];
    System.out.println(e[0][0]);  // 3
}
```

다차원 배열

- 다차원 비균등 배열

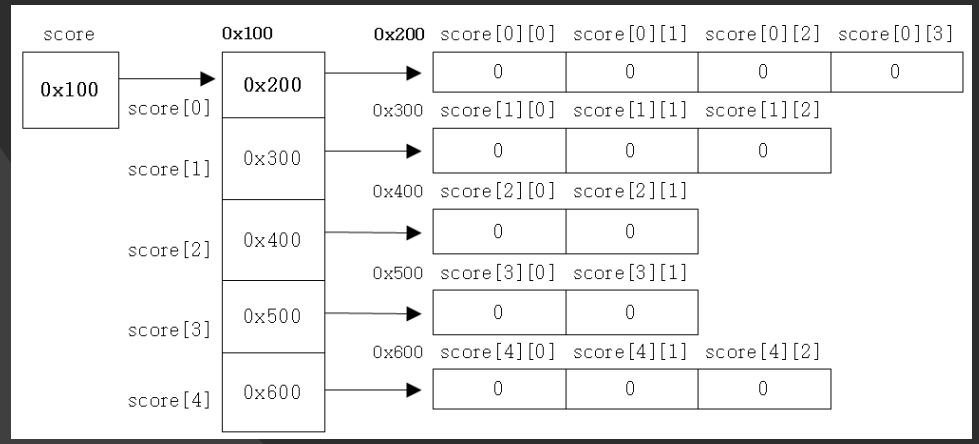
```
int[][] score = new int[5][];
score[0] = new int[3];
score[1] = new int[3];
score[2] = new int[3];
score[3] = new int[3];
score[4] = new int[3];
```

```
int[][] score =
    {
        {100,100,100},
        { 20, 20, 20},
        { 30, 30, 30},
        { 40, 40, 40},
        { 50, 50, 50},
    };
```




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

```
int[][] score =
    {
        {100,100,100,100},
        { 20, 20, 20},
        { 30, 30},
        { 40, 40},
        { 50, 50, 50},
    };
```



다차원 배열

- 다차원 비균등 배열
  - 마지막 차원의 배열이 동일할 수도 있지만 마지막 차원의 배열의 크기가 비균등 할 수도 있다.
  - 보통 이렇게 선언되는 배열을 비균등 배열 이라고 한다.
  - 비균등 배열은 공간 할당방식과 직접 삽입 방식 두가지 형태로 위와같이 선언이 가능하다.
  - 비균등 선언이 차원 개수를 벗어나게 선언할 경우 에러가 발생하며 반드시 차원을 맞춰야 한다.
  - 비균등 선언을 차원 개수와 모자라게 선언할 경우 다음 선언에서 그만큼을 추가로 더 선언해 준다.

다차원 배열

다차원 비균등 배열 예제

```
// 다차원 비균등 배열
// 공간 선언 방식
// 공간 선언방식은 비균등으로 선언할 차원의 크기만 비워두고
// 선언을 하면 된다.
int[][] a = new int[5][];
// 이후 각각의 배열에 해당 차원의 비균등 배열을 선언하면 된다.
a[0] = new int[5];
// 차원수를 넘어갈 경우 에러가 발생한다.
// a[1] = new int[4][3]; // 에러
a[2] = new int[3];
a[3] = new int[2];
a[4] = new int[1];

// 직접 삽입 방식
// 직접 실행 방식은 실제 넣으려고 하는 데이터를 비균등하게
// 삽입하면 된다.
int[][] b = {
    {1,2,3,4,5},
    {6,7,8,9},
    {10,11,12},
    {13,14},
    {15}
};

// 3차원 배열에서의 비균등 배열 선언
int[][][] c = new int[3][][];
c[0] = new int[3][3];
// 만약 배열을 덜 선언했다면 다음 선언에서 추가적으로 더 선언이 가능하다.
c[1] = new int[2][];
c[1][0] = new int[5];
c[1][1] = new int[4];
c[2] = new int[3][3];
```

다차원 배열의 순차 접근

- 다차원 배열의 순차 접근
  - 다차원 배열의 순차 접근을 위해서는 해당 차원 만큼의 반복이 필요하다.
  - 이전시간의 중첩반복문을 사용하면 충분히 이 문제를 해결할 수 있다.
  - 맨 위의 반복문은 첫 배열의 크기 만큼 두번째 반복문은 그 다음 차원의 배열의 크기 만큼 돌린다.
  - 위에서 배웠던 비균등 배열의 경우 length를 사용하여 비균등한 배열의 참조를 할 수 있다.

다차원 배열의 순차 접근

다차원 배열의 순차 접근 예제

```java
Run | Debug
public static void main(String[] args) {
    // 다차원 배열의 순차 접근 방식
    // 균등 배열이라면 처음 반복문은 맨 상위 배열의 크기만큼 돌리고
    // 두번재 차원의 경우 두번째 중첩된 반복문을 해당 차원의 크기만큼
    // 돌린다. 이처럼 각 차원이 증가할 때 마다 for문을 늘려 반복을 돌리면 된다

    // 균등 배열
    int[][] a = {{1,2,3},{4,5,6},{7,8,9}};
    // 비균등 배열
    int[][] b = {{1,2,3,4},{5,6,7},{8,9}};

    // 균등 배열 순차 출력
    // 3x3 이기 때문에 상위의 for문을 3번 하위의 for문을 3번 돌려준다.
    // 안에 참조하려 하는 상위의 인덱스는 상위 for문의 카운트 변수값에 고정시킨다.
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            System.out.print(a[i][j]+"\t");
        }
    }
    System.out.println();

    // 비균등 배열 순차 출력
    // 비균등 배열은 크기가 일정하지 않기 때문에 위와 같은 방식으로 선언이 불가능하다.
    // 그렇기 때문에 array.length를 사용하여 각각의 배열마다의 크기를 불러오는 것이
    // 매우 중요하다.
    for(int i = 0; i < b.length ; i++){
        // 중요한건 안쪽의 length는 배열 파라미터.length가 아니라
        // 배열 파라미터[해당차원 인덱스].length가 된다.
        for (int j = 0; j < b[i].length; j++) {
            System.out.print(b[i][j]+"\t");
        }
    }
}
```

배열의 복사

- 배열의 복사
  - 배열을 일반 기본형 변수처럼 삽입하고 값을 갱신해서 쓰려고 하면 기존에 있던 배열도 영향을 받는 것을 볼 수 있다.
  - 이것은 레퍼런스 타입의 특징이며 같은 값을 참조하려는 다른 두 변수가 존재한다 가정하고 하나의 변수에서 값을 변경하면 다른 변수의 값 또한 변경을 반영하게 된다.
  - 그렇기 때문에 기존의 기본형과 같은 값의 복사는 할 수 없으며 약간 다른 방식을 통해 배열을 복사해서 사용해야 한다.

배열의 복사

잘못된 참조 예제

```
Run | Debug
public static void main(String[] args) {
    // 배열의 복사(잘못된 예)
    // 기존의 기본형 타입 복사처럼 하나의 변수의 배열을
    // 다른 변수에 집어 넣고 해당 배열의 인덱스중 일부를
    // 교체한 뒤 배열의 값을 바꿔보았다.

    // 기본형의 예
    int a = 3;
    int b = a;
    b += 1;
    System.out.println(a+" "+b);  // 3 4

    // 배열의 예
    int[][] c = {{1,2,3,4},{5,6,7},{8,9}};
    int[][] d = c;

    d[0][0] = 20;
    System.out.println(c[0][0]); // 20
    System.out.println(d[0][0]); // 20
}
```

배열의 복사

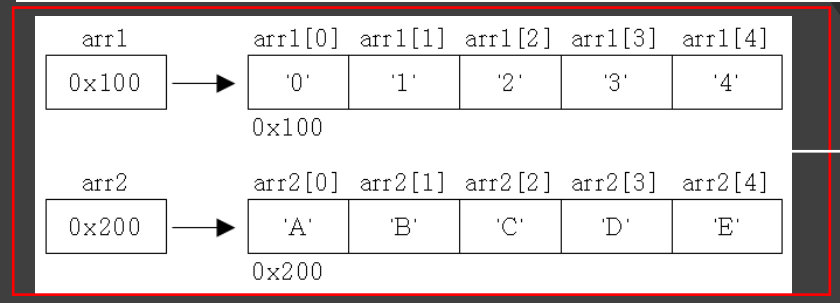
- 배열의 복사

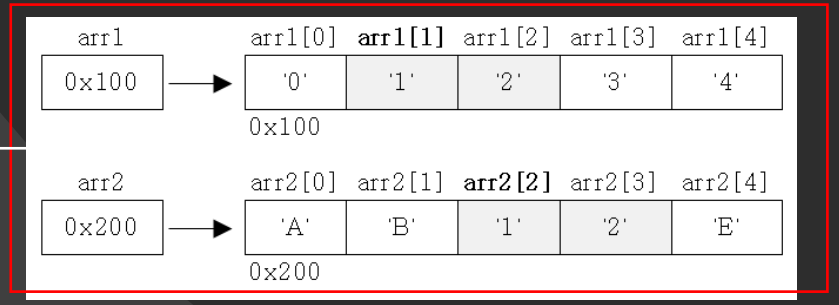
```
int[] number = {1,2,3,4,5};
int[] newNumber = new int[10];

for(int i=0; i<number.length;i++) {
    newNumber[i] = number[i];// 배열 number의 값을 newNumber에 저장한다.
}
```

```
System.arraycopy(arr1, 0, arr2, 0, arr1.length);
```

arr1[0]에서 arr2[0]으로 arr1.length개의 데이터를 복사

```
System.arraycopy(arr1, 1, arr2, 2, 2);
```





배열의 복사

배열의 복사

```
Run | Debug
public static void main(String[] args) {
    // 배열 카피 예제
    int[] a = {1,2,3,4,5};
    int[] b = new int[5];

    // 배열 a의 1번 인덱스 부터 2개의 값을
    // 배열 b의 2번 인덱스에 복사한다.
    System.arraycopy(a, 1, b, 2, 2);

    for (int i : b) {
        System.out.print(i+"\t");
    }
    System.out.println();
}
```