변수와 제어문

기본 구조

- 클래스 : 서로 관련 있는 변수나 메소드의 집합
- 메소드 : 어떤 기능의 구현 단위

클래스

```
public class Calculator {

    public int plus(int a, int b) {
        return a + b;
    }

    public int minus(int a, int b) {
        return a - b;
    }

}
```

메소드

메소드

Java

main 메소드 구조

- main 메소드
  - ✓ JVM이 실행하는 기본 메소드
  - ✓ 실행하고자 하는 코드는 main 메소드에 작성해야 함

클래스

```
public class HelloProject {

    public static void main(String[] args) {

        String message = "Hello World";

        System.out.println(message);

    }

}
```

main 메소드

식별자(Identifier)

- 식별자
  - ✓ 클래스, 변수, 메소드 등에 붙이는 이름을 의미함
  - ✓ 대소문자를 구별함(dog와 Dog는 다른 식별자임)
  - ✓ 한글 사용 가능(테스트 코드를 제외하면 한글 사용하면 안 됨)
  - ✓ 숫자 사용 가능(첫 글자로 사용될 수 없음. 숫자로만 구성될 수 없음)
  - ✓ 특수문자 2가지(_, $) 사용 가능($는 거의 안 씀)
  - ✓ 길이 제한 없음
  - ✓ 자바에서 사용 중인 키워드 사용 불가

- 가능한 식별자의 예시
  - ✓ name
  - ✓ section2
  - ✓ super_man

- 불가능한 식별자의 예시
  - ✓ 1block *숫자로 시작*
  - ✓ one% *사용할 수 없는 특수문자(%)*
  - ✓ my home *사용할 수 없는 특수문자(공백)*
  - ✓ true *자바가 사용중인 키워드*

자바 키워드

수업에서 다루지 않는
키워드는 5개 남짓

abstract	continue	for	new	switch
assert	default	if	package	synchronized
boolean	do	goto	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Naming Convension

- Upper Camel Case
  - ✓ 각 단어의 첫 글자는 대문자, 나머지는 소문자 처리
  - ✓ 클래스 이름 규칙
  - ✓ MyHome, MyProject 등

- Lower Camel Case
  - ✓ 각 단어의 첫 글자는 대문자, 나머지는 소문자 처리
  - ✓ 가장 첫 글자는 소문자 처리
  - ✓ 변수, 메소드 이름 규칙
  - ✓ myHome, myProject 등

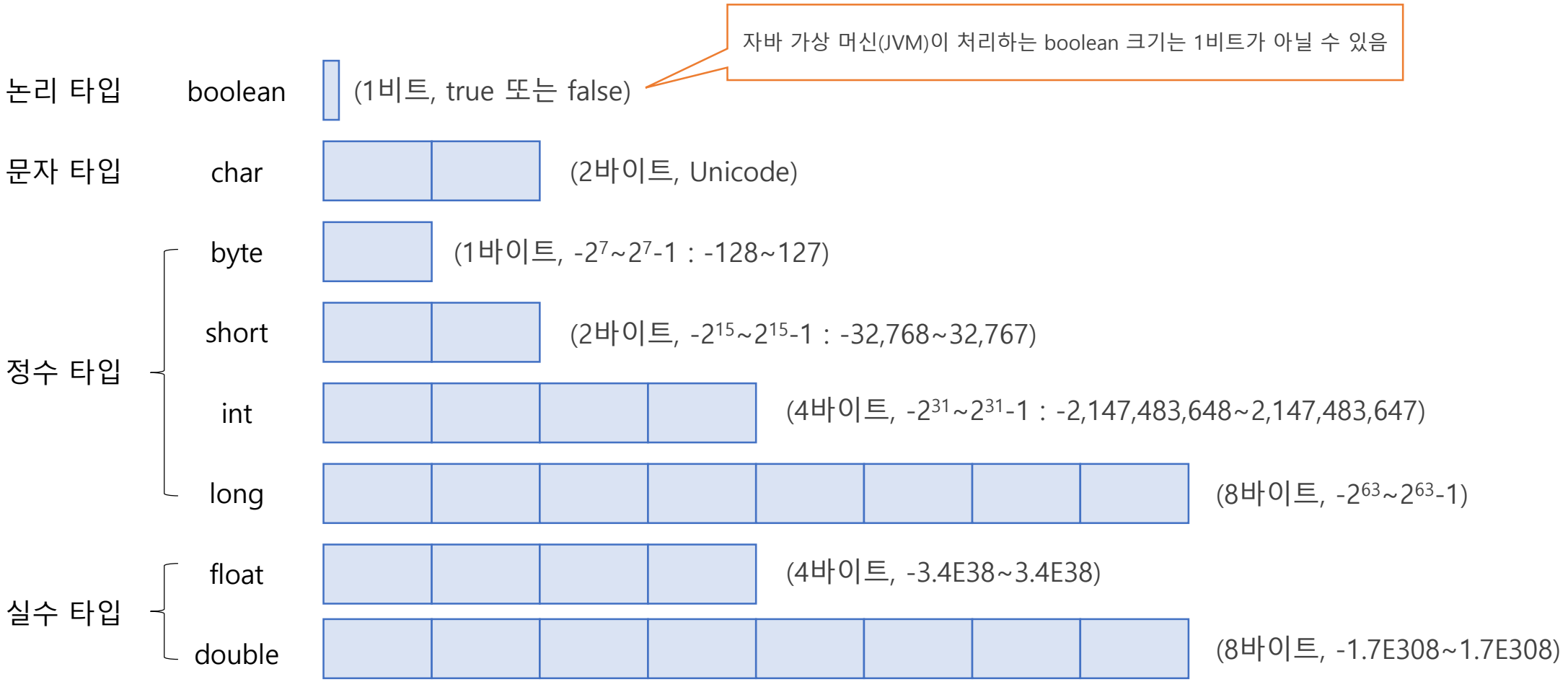
- Snake Case
  - ✓ 각 단어를 밑줄(_)로 연결
  - ✓ 대소문자를 함께 사용할 수 없을 때 사용
  - ✓ 상수 이름 규칙
  - ✓ MY_HOME, MY_PROJECT 등

데이터 타입

- 기본 타입(Primitive Type) 8개 - 데이터를 저장하는 타입
  - ✓ boolean
  - ✓ char
  - ✓ byte
  - ✓ short
  - ✓ int
  - ✓ long
  - ✓ float
  - ✓ double

- 참조 타입(Reference Type) - 데이터의 참조값(주소)를 저장하는 타입
  - ✓ 배열(Array)
  - ✓ 클래스(Class)
  - ✓ 인터페이스(Interface)

기본 타입(Primitive Type)



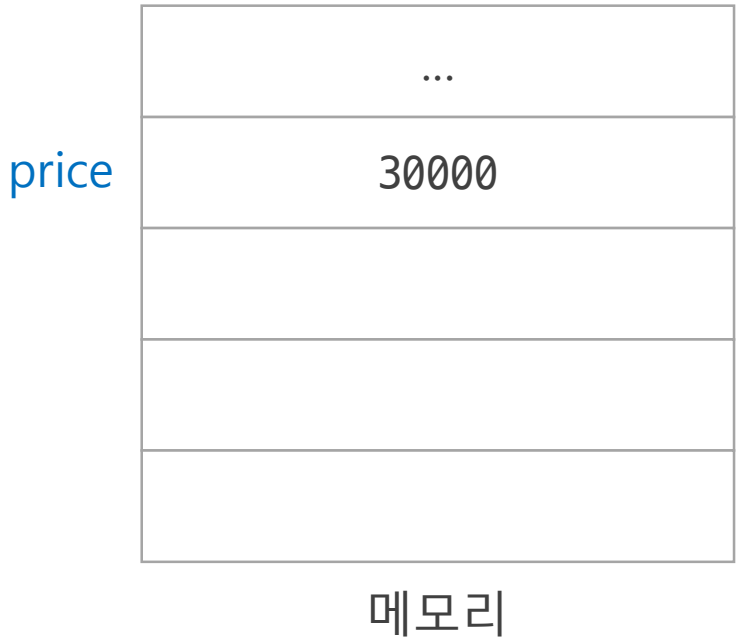
※ 1바이트 = 8비트

변수(Variable)

- 프로그램에서 사용하는 값을 저장하기 위한 공간
- 데이터 타입에서 정한 크기의 메모리 공간을 사용해서 값을 저장
- 데이터 타입 다음에 변수 이름을 작성해서 변수를 선언(declare)한 뒤 사용
- 변수에 저장된 값은 언제든 다른 값으로 바꿀 수 있음

```
int price = 30000;
```

데이터 타입 | 변수 이름 | 값



상수(Constant Variable)

- final 키워드를 추가한 변수
- 변수를 선언할 때 반드시 초기값을 지정해야 함
- 실행 중 저장된 값을 변경하는 것이 불가능
- 변수와 구분하기 위해 대문자를 이용한 Snake Case 방식의 Naming 사용

```
final double PI = 3.141592;
```

상수 선언 | 데이터 타입 | 상수 이름 | 초기값

```
PI = 1.5;
```

오류 발생!

리터럴(Literal)

- 리터럴(Literal)
- 코드 상에서 값을 표현하는 방식

리터럴	의미	예시
25	10진수 25	int n = 25;
025	8진수 25(10진수 21)	int n = 025;
0x25	16진수 25(10진수 37)	int n = 0x25;
0b1010	2진수 1010(10진수 10)	int n = 0b1010;
25L	long 타입의 25	long n = 25L;
3.14	double 타입의 3.14	double n = 3.14;
3.14F	float 타입의 3.14	float n = 3.14F;
true, false	논리 리터럴	boolean isNum = true;
'a'	문자 리터럴	char ch = 'a';
"apple"	문자열 리터럴	String fruit = "apple";
null	널(없음을 의미)	String message = null;

이스케이프(Escape)

- 이스케이프(Escape)
- 백슬래시(₩, \) 뒤에 한 문자가 오는 조합으로 단일 문자로 인식함(문자열 인식도 가능)

이스케이프	의미
'\b', "\b"	백스페이스(backspace)
'\f', "\f"	폼 피드(form feed)
'\n', "\n"	라인 피드(line feed)
'\r', "\r"	캐리지 리턴(carriage return)
'\t', "\t"	탭(tab)
'\'', "\'"	작은 따옴표(single quote)
'\"', "\""	큰 따옴표(double quote)
'\\', "\\"	백슬래시(backslash)

참조 타입(Reference Type)

- 배열(Array)
  - ✓ 배열을 이용해서 여러 개의 데이터를 한 번에 나타냄
  - ✓ 예시 : int[] points = new int[10];

- 클래스(Class)
  - ✓ 각종 클래스를 이용해서 객체를 나타냄
  - ✓ 예시 : String 클래스를 이용해서 문자열을 표현
  - ✓ String message = "안녕하세요";

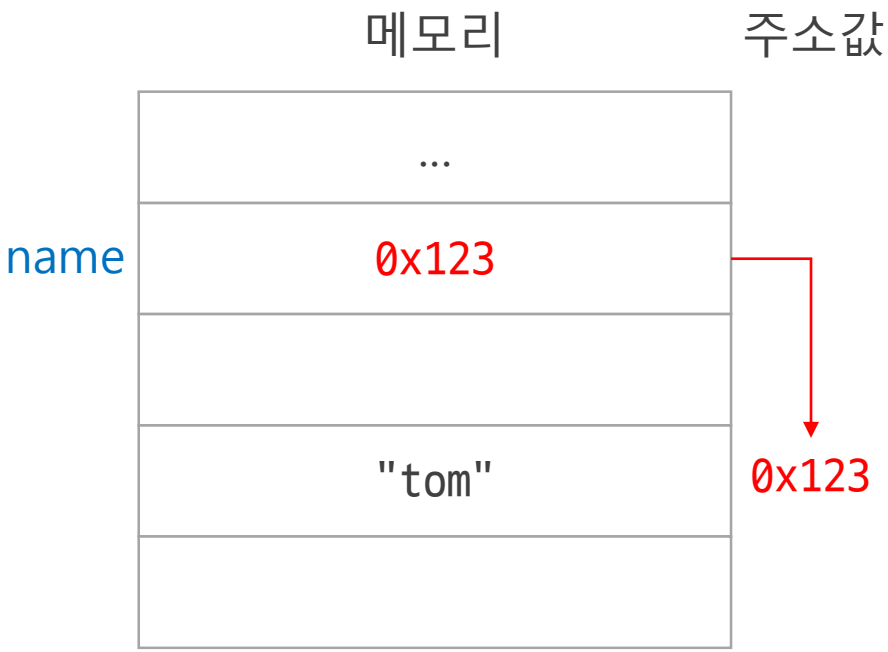
- 인터페이스(Interface)
  - ✓ 각종 인터페이스를 이용해서 구현 클래스를 표현
  - ✓ 예시 : List 인터페이스를 이용해서 ArrayList 클래스를 표현
  - ✓ List<String> hobbies = new ArrayList<>();

참조 타입의 변수

- 메모리는 1바이트마다 고유의 주소값을 가지는데 이것을 참조(Reference)라고 함
- 참조 타입은 값을 저장하기 않고 메모리의 주소값을 저장함(C/C++의 포인터 개념)
- 참조값은 16진수로 구성되어 있음
- 참조 타입의 값은 임의로 수정할 수 없음

```
String name = "tom";
```

데이터 타입 | 변수 이름 | 값

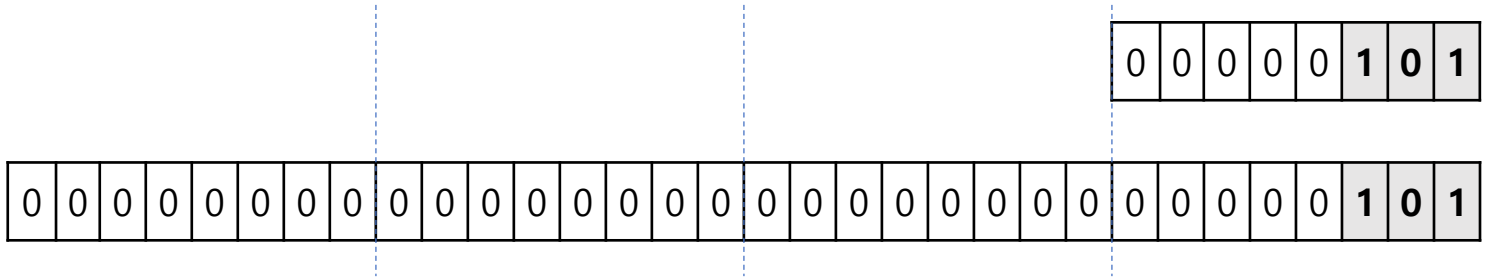


자동 타입 변환(Promotion)

- 작은 타입이 큰 타입으로 변하는 경우 자동 변환됨

byte a = 5; a

int b = a; b



- 서로 다른 타입으로 연산하는 경우 동일한 타입으로 자동 변환됨

double b = 1.5 + 5;

1.5는 8바이트 double 타입이고, 5는 4바이트 int 타입이기 때문에 이대로는 연산이 불가능함.

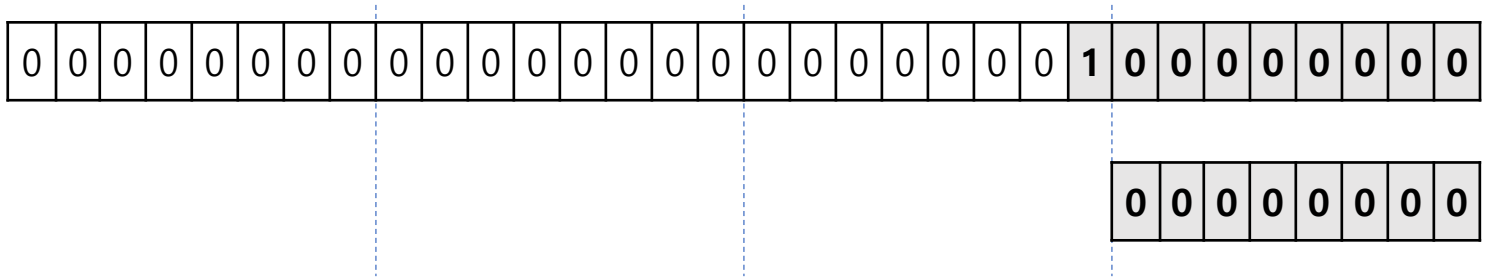
자바는 5를 8바이트 double 타입 5.0으로 자동 변환한 뒤 1.5 + 5.0으로 연산을 진행함.

강제 타입 변환(Casting)

- 큰 타입이 작은 타입으로 변하는 경우 개발자가 강제로 진행해야 함
- 강제로 타입을 ( ) 안에 지정해 줌

int a = 256;

byte b = (byte)a;



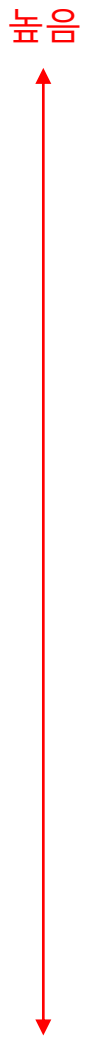
- 실수를 정수로 강제 타입 변환하면 소수점은 모두 손실됨

```
double a = 1.9;
int b = (double)a;
```

변수 b에 저장되는 값은
소수점 이하 0.9가 손실된 1이 저장됨

연산자 우선순위

++(postfix), --(postfix)
+(양수 부호), -(음수 부호), ~, !, ++(prefix), --(prefix)
강제 형 변환(Casting)
*, /, %
+(덧셈), -(뺄셈)
<<, >>, >>>
<, >, <=, >=, instanceof
==, !=
&(비트 AND)
^(비트 XOR)
¦(비트 OR)
&&(논리 AND)
¦¦(논리 OR)
? :
=, +=, -=, *=, /=, %=



- 동일한 우선순위의 연산은 왼쪽에서 오른쪽으로 진행함
- 우선순위를 조정하려면 먼저 처리할 연산을 괄호()로 묶어서 처리하면 됨

산술 연산자

- 정수 연산 : 더하기(+), 빼기(-), 곱하기(*), 몫(/), 나머지(%)
- 실수 연산 : 더하기(+), 빼기(-), 곱하기(*), 나누기(/), 나머지(%)
- 몫과 나머지를 이용한 연산이 많이 사용됨
  - ✓ 25의 앞자리와 뒷자리 분리 : 25를 10으로 나눈 몫은 2, 나머지는 5
  - ✓ 짝수와 홀수 : 2로 나눈 나머지가 0이면 짝수, 1이면 홀수
  - ✓ 3의 배수 : 3으로 나눈 나머지가 0이면 3의 배수
  - ✓ 0~6 사이 값 사용 : 7로 나눈 나머지를 사용

연산자	예시	결과
+	7 + 2	9
-	7 - 2	5
*	7 * 2	14
/	7 / 2	3
%	7 % 2	1

연산자	예시	결과
+	7.0 + 2.0	9.0
-	7.0 - 2.0	5.0
*	7.0 * 2.0	14.0
/	7.0 / 2.0	3.5
%	7.0 % 2.0	1.0

문자열 연산자

- 문자열 연결 연산자 : +
- "문자열" + "문자열" → "문자열문자열"
  - ✓ "안녕" + "하세요" → "안녕하세요"
- 숫자 + "문자열" → "숫자문자열"
  - ✓ 100 + "주년" → "100주년"
- 숫자를 문자열로 변경하는 가장 간단한 방법
  - ✓ 숫자에 빈문자열("")을 더함
  - ✓ 100 + "" -> "100"

- 자주 사용하므로 많이 연습해 둬야 함

대입 연산자

- 대입 연산 : =, 등호(=) 오른쪽의 값을 왼쪽으로 대입함
- 복합 대입 연산 : 더하기(+=), 빼기(-=), 곱하기(*=), 나누기(/=), 나머지(%=) 등

연산자	의미
a = 10	10을 a에 대입
10 = a	a를 10에 대입(불가능)
a += 2	a = a + 2
a =+ 2	a = +2로 인식(주의)
a -= 2	a = a - 2

연산자	의미
a *= 2	a = a * 2
a /= 2	a = a / 2
a %= 2	a = a % 2
s += ".jpg"	s = s + ".jpg"

증감 연산자

- 증가 연산 : ++, 1 증가
- 감소 연산 : --, 1 감소
- 전위 연산(prefix)과 후위 연산(postfix)으로 구분함

```
a = 1;
b = ++a;
```

a를 2로 만든 뒤,
b에 저장한다.
a = 2
b = 2

```
a = 1;
b = a++;
```

a를 b에 저장한 뒤,
a를 2로 만든다.
a = 2
b = 1

```
a = 1;
b = --a;
```

a를 0으로 만든 뒤,
b에 저장한다.
a = 0
b = 0

```
a = 1;
b = a--;
```

a를 b에 저장한 뒤,
a를 0으로 만든다.
a = 0
b = 1

전위 연산

후위 연산

비교 연산자

- 값을 비교하여 boolean 결과(true, false)를 반환함
- 6개 연산이 있음
  - ✓ < : 작으면 true
  - ✓ > : 크면 true
  - ✓ <= : 작거나 같으면 true
  - ✓ >= : 크거나 같으면 true
  - ✓ == : 같으면 true
  - ✓ != : 다르면 true

연산자	예시	결과
<	`7 < 2`	`false`
>	`7 > 2`	`true`
<=	`7 <= 2` `7 <= 7`	`false` `true`
>=	`7 >= 2` `7 >= 7`	`true` `true`
==	`7 == 2`	`false`
!=	`7 != 2`	`true`

논리 연산자

- 논리 AND : &&, 모두 true이면 true/하나라도 false이면 false
- 논리 OR : ||, 하나라도 true이면 true/모두 false이면 false
- 논리 NOT : !, true이면 false/false이면 true
- Short Circuit Evaluation : 논리 AND나 논리 OR의 연산 결과를 이미 알고 있는 경우 더 이상 연산을 진행하지 않는 것을 의미함

연산자	예시	결과
&&	(7 > 2) && (7 > 3) (7 > 2) && (7 > 10)	true false
¦¦	(7 > 2) ¦¦ (7 > 3) (7 > 2) ¦¦ (7 > 10)	true true
!	!(7 > 2) !(7 > 10)	false true

조건 연산자

- 형식

  조건식 ? true인 경우 값 : false인 경우 값

- 3개 항을 사용하기 때문에 삼항 연산자라고 함

```
int age = 15;
String message = "";
message = (age >= 20) ? "성인" : "미성년자";
System.out.println(message);
```

String message는
age가 20 이상이면 "성인",
아니면 "미성년자"를 저장한다.

if문

- 조건문을 만족하는 경우에만 실행
- 실행문은 중괄호{}로 묶어서 표시
- 실행문이 단일 문장인 경우 중괄호{} 생략 가능

```
int age = 15;
String message = "";
if(age >= 20) {
    message = "성인";
}
if(age < 20) {
    message = "미성년자";
}
System.out.println(message);
```

String message는
age가 20 이상이면 "성인",
age가 20 미만이면 "미성년자"
를 저장한다.

if-else문

- 조건문을 만족하는 경우와 만족하지 않는 경우를 동시에 처리
- 실행문은 중괄호{}로 묶어서 표시
- 실행문이 단일 문장인 경우 중괄호{} 생략 가능

```
int age = 15;
String message = "";
if(age >= 20) {
    message = "성인";
}
else {
    message = "미성년자";
}
System.out.println(message);
```

String message는
age가 20 이상이면 "성인",
아니면 "미성년자"를 저장한다.

else if문

- 조건문이 많은 경우에 사용
- 실행문은 중괄호{}로 묶어서 표시
- 실행문이 단일 문장인 경우 중괄호{} 생략 가능

```
int score = 50;
int point = 0;
if(score >= 80) {
    point = 500;
}
else if(score >= 60) {
    point = 300;
}
else {
    point = 0;
}
System.out.println(point);
```

int point는
score가 80 이상이면 500,
score가 60 이상이면 300,
아니면 0을 저장한다.

switch문

- switch문의 표현식 결과와 case문의 값을 비교하여 실행할 문장을 선택
- case문의 값과 일치하는 값이 없으면 default문을 실행(default문은 생략 가능)
- case문의 실행문이 끝나면 break문을 통해서 switch문의 실행을 종료하는 것이 일반적
- 표현식 결과는 double, boolean 타입이 불가능함

```
int schoolYear = 1;
switch(schoolYear) {
case 1:
case 2:
case 3:
    System.out.println("저학년");
    break;
case 4:
case 5:
case 6:
    System.out.println("고학년");
    break;
}
```

int schoolYear가 1,2,3이면 "저학년",
int schoolYear가 4,5,6이면 "고학년" 출력

for문

- 어떤 실행문을 여러 번 반복해서 실행하는 경우에 사용
- 조건문을 만족하면 계속해서 실행
- 실행문은 중괄호{}로 묶어서 표시
- 실행문이 단일 문장인 경우 중괄호{} 생략 가능
- 형식

  for(초기문; 조건문; 증감문) {
  실행문
  }

  ✓ 초기문은 최초 한 번만 실행함
  ✓ 조건문 -> 실행문 -> 증감문 순으로 반복 처리함
  ✓ 배열이나 컬렉션 처리에서 많이 사용

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

1부터 5까지 모든 정수를 출력함

while문

- 어떤 실행문을 여러 번 반복해서 실행하는 경우에 사용
- 조건식을 만족하면 계속해서 실행
- 실행문은 중괄호{}로 묶어서 표시
- 실행문이 단일 문장인 경우 중괄호{} 생략 가능
- 형식

```
while(조건문) {
    실행문
}
```

✓ 조건문을 통해서 종료 시점을 판단
✓ 파일 읽기, 데이터베이스 읽기에서 많이 사용

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

1부터 5까지 모든 정수를 출력함

break문

- 반복문(for, while)을 종료시킬 때 사용
- switch문을 종료시킬 때 사용

```
switch(표현식) {
    …
    break;  // switch문 종료
    …
}
```

```
while(조건문) {
    …
    break;  // while문 종료
    …
}
```

```
for(초기문; 조건문; 증감문) {
    …
    break;  // for문 종료
    …
}
```

continue문

- 반복문(for, while)을 종료시키지 않고, 반복문의 시작 지점으로 이동
- 반복문에서 실행을 제외하고 싶은 경우에 사용

- while문

```
while(조건문) {
    …
    continue;  // while(조건문)으로 이동해서 조건문이 만족하는지 검사
    …
}
```

- for문

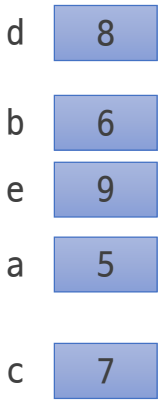
```
for(초기문; 조건문; 증감문) {
    …
    continue;  // 증감문으로 이동한 뒤 조건문이 만족하는지 검사
    …
}
```

배열(Array)

- 한 번에 많은 메모리 공간을 할당해서 여러 개의 값을 저장하는 자료 구조(Data Structure)
- 같은 데이터타입의 자료들을 순차적으로 저장
- 구성
  - ✓ 배열 요소(Element) : 배열에 저장된 값으로 배열이름[인덱스] 형식으로 사용
  - ✓ 인덱스(Index) : 자료가 저장된 위치 정보로 0부터 시작
- 배열의 필요성

(1) 5개의 변수

```
int a, b, c, d, e;
```

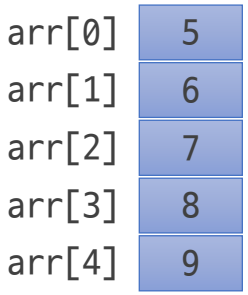


메모리에 순차적으로 저장되지 않아
모든 변수에 개별 접근해야 함

```
total += a;
total += b;
total += c;
total += d;
total += e;
```

(2) 5개를 저장한 배열

```
int[] arr = new int[5];
```



메모리에 순차적으로 저장되어
인덱스를 이용한 순차접근이 가능함

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

Java

1차원배열

- 배열의 선언
  - ✓ 배열의 데이터타입을 선언하는 것
  - ✓ 배열은 참조 타입(Reference Type)으로 관리됨

```
int[] arr;
```

또는

```
int arr[];
```

- 배열의 생성
  - ✓ 배열의 길이를 지정해서 실제 배열을 생성하는 것
  - ✓ 배열의 길이를 변수로 처리해도 됨

```
arr = new int[5];
```

또는

```
int size = 5;  arr = new int[size];
```

- 배열의 초기화
  - ✓ 배열의 선언과 함께 값을 초기화 할 수 있음
  - ✓ 중괄호{}를 이용해 값을 작성

```
int[] arr = new int[] {10, 20, 30, 40, 50};
int[] arr = {10, 20, 30, 40, 50};
```

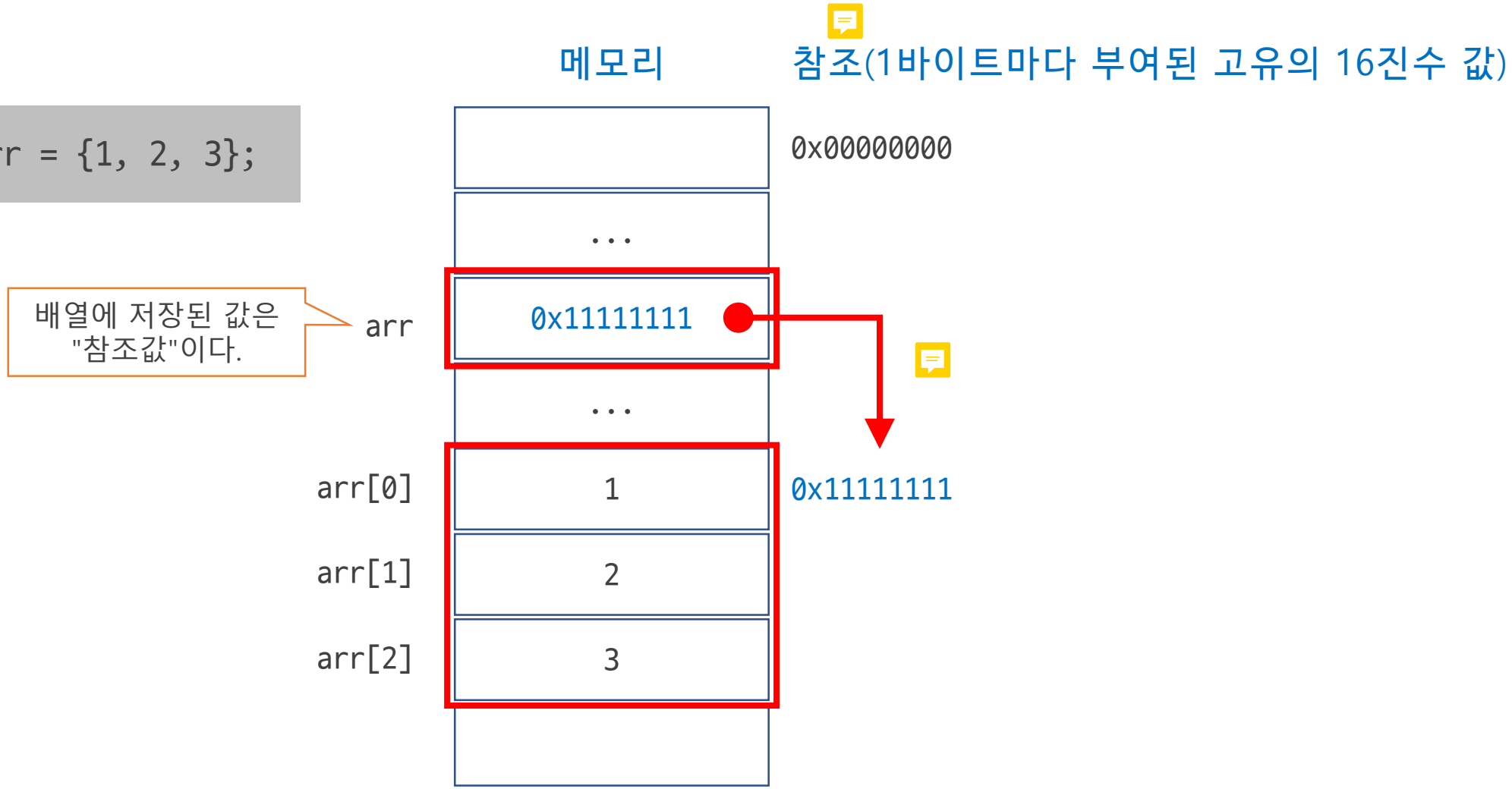
- 배열의 길이
  - ✓ 배열의 길이는 배열이름.length 필드값으로 확인
  - ✓ 한 번 생성된 배열의 길이는 수정 불가

- 배열 요소(Element)
  - ✓ 배열이 관리하는 각각의 변수를 의미
  - ✓ 배열이름[인덱스] 형식으로 호출해서 사용
  - ✓ 일반 변수와 달리 자동 초기화 진행
    (0, 0.0, false, null 등)

Java

배열은 참조 타입(Reference Type)

```
int[] arr = {1, 2, 3};
```



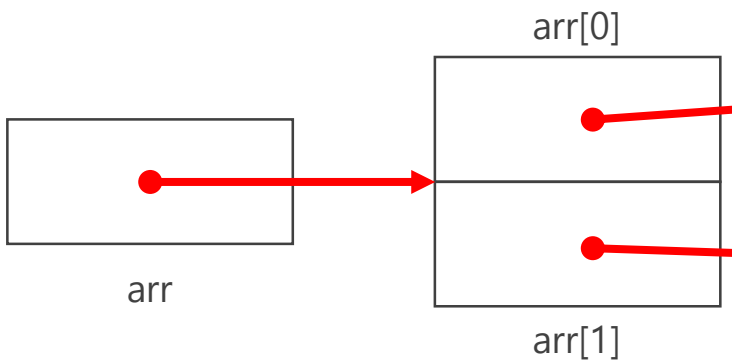
2차원배열

- 행과 열로 구성된 테이블(표) 구조를 가지는 자료 구조
- 실제로는 2개 이상의 1차원배열을 동일한 이름으로 관리하는 자료 구조
- 모든 행의 열 개수가 동일한 정방형 배열과 행마다 열 개수가 다른 비정방형 배열로 구분

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

arr[0][0]	arr[0][1]	arr[0][2]
arr[1][0]	arr[1][1]	arr[1][2]

[테이블 구조로 이해하는 2차원배열]



arr[0][0]	arr[0][1]	arr[0][2]

arr[1][0]	arr[1][1]	arr[1][2]

[2차원배열은 1차원배열 여러 개로 구성됨]

2차원배열

- 정방형 배열의 선언

```
int[][] arr;   또는   int arr[][];
```

- 정방형 배열의 생성
  - ✓ 모든 행의 열의 개수가 동일하기 때문에 최초 한 번만 열의 개수를 정함

```
arr = new int[3][5];
```

- 정방형 배열의 초기화

```
int[][] arr = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10},
    {11, 12, 13, 14, 15}
};
```

- 비정방형 배열의 선언

```
int[][] arr;   또는   int arr[][];
```

- 비정방형 배열의 생성
  - ✓ 각 행마다 열의 개수를 각각 지정함

```
arr = new int[3][];   // 최초 열 개수의 지정이 없음
arr[0] = new int[5];  // 1행은 5열
arr[1] = new int[3];  // 2행은 3열
arr[2] = new int[4];  // 3행은 4열
```

- 비정방형 배열의 초기화

```
int[][] arr = {
    {1, 2, 3, 4, 5},
    {6, 7, 8},
    {9, 10, 11, 12}
};
```

2차원배열의 메모리 구조

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