# 기본 문법 및 클래

## 가. 기본 문법

- 데이터 : 값 자체
- 정보: 의미가 있는 데이터
- Java
  Framework: 기존에 있는 틀
  ㄴ 오픈 소스 중 가장 많이 쓰이는 것 Spring -> 전자 정부 프레임워크
- 운영체제: 시스템 하드웨어 관리하고 응용 소프트웨어 실행 위해 하드웨어 추상화 플랫폼과 공통 시스템 서비스를 제공하는 시스템 소프트웨어
- 프로그램: 컴퓨터에서 실행될 때 특정 작업 수행하는 일련의 명령어들의 모음
  ㄴ 명령어: 일 시키는 키워드(약속된 단어)
- JVM: 자바 바이트코드 실행 주체 (java 문법 → 바이트코드 변환 : 컴파일)
  ㄴ 바이트코드: 플랫폼에 독립적
- 비트(0, 1) → 1비트는 2가지 표현 가능 (1칸당 2^n개 표현 가능) , 한 글자 → 128바이트
- 1바이트 → 8비트 (2^8 = 256), 한 글자 → 128비트(7바이트)
  - 문자 : 아스키 코드 예) A → 65
  - 음수(2의 보수 표기법): 0과 1을 뒤집고, 거기에 +1 예) 010 (2) → 110 (-2)
  - 실수: 부동 소수점 표현
- 바이너리(2진수)
- 메인 메소드: public static void main(String[] args) {}로 고정
  -Java 실행 시 가장 먼저 호출, main() 메소드 없으면 실행 아예 x, 애플리케이션의 시작

***<출력문>***

1. print
2. println (line next)
   a. \n: 줄바꿈 기호
3. printf (format) 예) System.out.printf(”%d”, 10); → 10진수

   System.out.printf(”a : % d \n”, a),

   : %d: 정수 %f: 실수 %c: 문자 %s: 문자열 %o: 8진수 %x: 16진수 %n: 줄 바꿈

   %4d: 4칸 확보 후 오른쪽부터 %-4d: 4칸 확보 후 왼쪽부터

   %04d: 4칸 확보 후 오른쪽부터, 빈 공간은 0으로

   %.2f: 실수 (소수점 둘째 자리까지)

***<변수>***

- 정의: 데이터를 저장할 메모리 위치를 나타내는 이름, 메모리상 데이터 보관 공간 확보
  적절한 메모리 공간 확보 위해 변수 타입 등장, '='를 통해 CPU에 연산 작업 의뢰
- 메모리 단위
  bit: 0,1 표현  8bit = 1byte

1. 선언

: 자료형 변수명;  예) int age;  String name;

2. 초기화

: 변수명 = 저장할 값;  예) age = 30; name = "철수";

3. 선언 & 초기화 동시에

: 자료형 변수명 = 저장할 값;  예) int age = 30;

- 대소문자 구분, 공백 x., 숫자 시작 x, '$', '-' 외 특수문자 사용 x
- 예약어 x  예) abstract, default, if 뭐 이런 것들

<자료형> : 기본형 + 참조형

1. 기본 자료형

| 타입 | 세부타입 | 데이터형 | 크기(byte) | 기본값 | 예 |
|---|---|---|---|---|---|
| 논리형 | | boolean | 1 | false | boolean b = true |
| 문자형 | | char | 2 | null(\u0000) | char c = ‘a’, c1 = 65, c2 = ‘\uffff’ |
| 숫자형 | 정수형 | byte | 1 | (byte)0 | byte b = 100; |
| | | short | 2 | (short)0 | short s = 100; |
| | | int | 4 | 0 | int i =100; |
| | | long | 8 | 0L | long l = 100, l2 = 100L |
| | 실수형 | float | 4 | 0.0f | float f = 3.1f, f2=3.1F; |
| | | double | 8 | 0.0d | double d = 3.1 ; |

2. 자료형 크기 비교

- byte < short < int < long  < float < double
- char < int < long  < float < double

3. 데이터 형 변환

   1) 묵시적:  넓음 ← 좁음

   예) byte b = 100;  int i = b;

   2) **명시적: 좁음 ← 넓음**

   예) int i = 100;  byte b = (byte) i;  int c = sa;  short sb = (short) c;

float f = 10;  int g = (int) f; → 같은 크기더라도 컨펌 필요 (명시적)

## 나. 연산자

<3항 연산자>

조건식 ? 수식 1: 수식 2;

- 수식 1: 조건식 결과가 참(true)일 때 수행
- 수식 2: 조건식 결과가 거짓(false)일 때

<산술 연산자>

- +. -. *, a / b (a를 b로 나눈 몫), a % b (a를 b로 나눈 나머지)
- 정수-정수 연산 → 정수, 정수-실수 연산 → 실수
- 버림 연산

  age = age / 10 * 10;
  age = age - (age % 10);
- 반올림 연산

  height = (height + 5) / 10 * 10;

  height = (height + 5) - (age % 10);

cf) 입력 스캐너 → sc.nextInt(); : 숫자  /  sc.next(); : 문자

<증감 연산자>

++a, —a 선행 처리  예) int a = 10  11  9

b++, b— 후행 처리  예) int b = 10  10  10

예) a++ ++ a —a a a— a++

  10  12  11  11 11  10  11

<비교 연산자>  피연산자: 숫자,  결과: boolean

- >, > =, <, < =
- ==  / 같을 때,  ! = / 다를 때
- a instanceof b / 객체 타입 비교

<조건 연산자>  피연산자, 결과 : boolean

- A && B  A와 B 참일 경우 참 (A가 거짓이면, B는 실행 X)

- A || B  A 또는 B가 참일 경우 참 (A가 참이면, B는 실행 X)
- ! A  A가 참이면 거짓, A가 거짓이면 참 반환

<배정 연산자>

- A += B  A = A + B
- A -= B  A = A - B
- A *= B  A = A * B
- A /= B  A = A

## 다. 제어문(조건문)

***<if 문>***

1. **단일**

```
if (조건식)
    실행문장;
else
    실행문장;
```

- 주의사항

-실행 문장이 복수일 때는 { }로 블록 처리

-조건식 자리에는 반드시 참, 거짓 구분

예) if (true or false ← 비교 연산, 조건 연산) {

```
    System.out.println(”1”)
    System.out.println(”1”)
     }
```

2. **다중**

```
if (조건식) {
    실행문장;
} else if (조건식) {
    실행문장;
} else if (조건식) {
    실행문장;
} else {
    실행문장;
}
```

3. **내포된 if**

```
if (조건식) {
  if(조건식)
  ...
  if(조건식)
  ...
  else
  ....
  }
```

***<Switch 문>***

```
switch (수식 or 변수) {
 case 값 1:
   처리 문장들;
    break;
 case 값 2:
   처리 문장들;
    break;
 case 값 n:
   처리 문장들;
    break;
 default:
    묵시적으로 처리해야 하는 문장들;
  }
```

1. 수식

- ~1.4: byte, short, char, int  /  1.5 ~ : enum 클래스 타입  / 1.7 ~ : String 클래스 타입

2. break 문 없이도 사용 가능
3. default ⇒ else 역할과 동일
4. break 문이 없을 경우: break문 찾을 때까지 선택된 case문 아래의 모든 문장 실행

```
예) switch (1) {
      case 1:
      System.out.println (1);
      case 2:
      System.out.println (2);
      default
```

System.out.println (3) ;

}

## 라. 제어문(반복문)

***<for 문>***

for (초기화; 조건식; 변수 증감) {

반복 문장들

}

반복문 빠져나옴

- 조건이 참이면 조건식 → 반복 문장 → 증감 반복
- 조건이 거짓이면 조건식 → 반복문 빠져나옴

예) for (n=1; n<6; n++) {

System.out.print(n+" ");

}

결과: 1 2 3 4 5

***<while 문>***

조건절로 지정된 조건이 참일 동안 while 블록 실행

while (조건절) {

반복 문장들

System.out.println("실행.")

n++; ← 이러한 브레이크 장치 없으면 무한 반복

}

***<do ~ while 문>***

조건을 나중에 평가, while 블록이 적어도 **한 번**은 수행

do {

반복 문장들

} while (조건절);

- do-while: 무조건 한 번 실행, while은 조건 안 맞으면 실행 X

  ⇒ 같은 조건일 경우 do-while만 실행될 수도
- 조건식에서 쓸 변수가 반복 구문 안에서 결정될 때

에) int a = 5, b = 10 ;

do {

```
        System.out.println(”무조건 실행됨”);
    } while (a > b) ⇒ 1번 실행
    int num = 0 ;
        do {
            System.out.print(”숫자 : “);
            num = sc.nextInt();
            } while (num ! = 0) ;
            System.out.println(”끝”)
        }
```

***<break 문>***

1. switch 문 벗어날 때 사용
2. 반복문에서 반복 루프 벗어날 때 사용

예)
```
int i = 1;
    while(i < 100) {
    if (i == 10) break;
    System.out.println(i + “자바의 세계로 오세요!”);
    i++;
```
결과: 1~9 자바의 세계로 오세요!

3. 중첩된 반복문 한 번에 빠져나갈 때

예)
```
int i, j;
for (i = 1; i <= 5; i++) {
for (j = 1; j <= i; j++) {
if (j > 5)
break;
System.out.print("*");
}
System.out.println("");
```
결과

```
*
**
***
****
*****
```

***<continue 문>***

- 반복문 특정 지점에서 제어를 반복문의 처음으로 보냄

예)

```
for (int i = 0; i < 10; i++) {
if (i % 2 == 0)
continue;
System.out.println(i + "자바의 세계로 오세요!");
}
```

결과

1 3 5 7 9 자바의 세계로 오세요!

## 마. 배열

1. 정의

- [ ] 이 배열을 나타냄
- 같은 자료형 데이터(변수)의 모임
  - 배열로 선언된 변수들은 연속된 데이터 공간에 할당됨
- 크기 고정 (한번 생성된 배열은 크기 변경 x)
- 배열을 객체로 취급
- 배열 요소 참조하려면 배열 이름과 index라고 하는 int 유형 정수값 조합하여 사용
- 유형은 기본형, 참조형 모두 가능

장점 (1) 간편하게 같은 타입 많은 변수 생성 (2) 연속된 공간 (3) 반복문과 시너지 효과

2. 용어

- [ ]의 개수가 배열의 차원 수 예) [ ] [ ] → 2차원
- 1차원 배열 선언: 배열 유형 배열 이름 [ ] or 배열 유형 [ ] 베열 이름
  - int → 정수 하나 담을 수 있음
  - int [ ] → int 배열의 주소를 담을 수 있음

예) int [ ] prime, int[] score = new int [78];

- 다차원 배열 선언: 배열 유형 배열 이름 [ ][ ] or 배열 유형 [ ][ ] 베열 이름

예) int [ ][ ] prime

| 타입 | 배열 이름 | 선언 |
|---|---|---|
| int | iArr | int [ ] iArr; |
| char | cArr | char [ ] cArr; |
| boolean | bArr | boolean [ ] bArr; |
| | | |

| String | strArr | String [ ] strArr; |
|---|---|---|
| Date (날짜) | dateArr | Date [ ] dateArr; |

- 배열의 선언
  - 1차원 배열: int [ ] dongList; → 하나의 값 저장할 수 있는 메모리 생성
  - 2차원 배열: int [ ] [ ] → 1차원 배열의 위치를 저장할 수 있는 배열

3. 배열의 생성

- 1차원 배열: 배열의 이름 = new 배열 유형 [배열 크기];

예) prime = new int [10]

  - [ ]에 정수 하나 저장

- 2차원 배열: 배열의 이름 = new 배열 유형 [1차원 배열 개수] [1차원 배열 크기];

  배열의 이름 = new 배열 유형 [1차원 배열 개수] [ ];

  - [ ] 에 1차원 배열 저장
  - 문자열은 new를 안 써도 참조 가능

  예) prime = new int [3] [2];  prime = new int [3] [ ] ;

  - 배열이 생성되면 자동적으로 배열요소는 기본값으로 초기화

  예) int : 0 , boolean : false , char : '\u0000' , 참조형 : null….

  - 멤버 변수, 로컬 변수 모두 포함

4. 초기화

  - 1차원 배열: 배열이름[인덱스] = 값;

  예) prime[0] = 100;

  - 2차원 배열: 배열이름 [인덱스] [인덱스] = 값;

  예) twoArr[0][1] = 100;

  - 배열의 인덱스는 0부터 시작
  - 배열의 크기: 배열 이름.length
  - 마지막 요소 인덱스: 배열 크기 - 1

  ** { }을 활용하는 방식: 배열 선언 시에만 설정 가능

  - 1차원 배열 : 배열 유형 [ ] 배열명 = {값, .. 값};

  예) int [ ] prime = {1, 2, 3};

  - 2차원 배열 : 배열 유형 [ ] [ ] 배열명 = {{값1, 값2} , {값3, 값4}};

  예) int [ ] [ ] twoArr = {{1, 2}, {3, 4}, {5, 6}};

- new 배열타입[ ] {값, ...}

예) int [ ] prime = new int [ ] {1, 2};

5. 배열 관련 API

- System.arraycopy(src, srcPos, dest, destPos, length)

src: 원본 배열  srcPos: 원본 배열 복사 시작 위치(0부터 시작)
dest: 복사할 배열  destPos: 복사받을 시작 위치

length: 복사할 크기

예) String [ ] arr = {"봄", "여름", "가을"}

String [] destArr = new String[arr.length +1];

System.arraycopy(arr, 0, destArr, 0, arr.length}

destArr[3] = "겨울"

for (i=0; i < destArr.length; i++)
system.out.println(destArr[i]);

- Arrays.toString (배열 객체)

: 배열 안 요소를 [요소, 요소, ..] 형태로 출력

- 단순 배열 값 확인할 경우에 이용

예) System.out.println(Arrays.toString(destArr));

배열 선언과 초기화 동시에 → int [ ] points = new int [3];

- int [ ] [ ] arr = new int [3] [ ];

arr[0] = new int [5];

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

***<for-each 문>***

- 가독성이 개선된 반복문, 배열. Collections에서 사용
- index 대신 직접 요소에 접근하는 변수 제공 → 읽기 전용
- naturally ready only (copied vlaue)

예) int intArray[ ] = {1, 3, 5, 7, 9} ;

for (int x: intArray){

System.out.println(x);

}

** 위아래 출력값 동일 **

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

- for (원소: 데이터의 모임)

예)

```
String[] numbers = {"one", "two", "three"};
for (int i=0; i<numbers.length; i++) {
   System.out.println(numbers[i]);
}
```

→ String[] numbers = {"one", "two", "three"};

```
for(String number: numbers) {
   System.out.println(number);
}
```

구조 : for (type var : 루프 돌릴 객체) {

루프

}

cf) **최대값 최소값 찾기** (1)

```
int [ ] int Array = { .... }

int min = 1000;
int max = 0;
for (int num: intArray) {
    if(num > max) {
        max = num ;
    }
    if (num < min) {
        min = num ;
    }
}
System.out.printf(”min:: %d, max: %d%n”, min, max);
```

(2)

```
int [ ] int Array = { .... }

int min = Integer.MAX_VALUE;
int max = Integer.MIN_VALUE;

for (int num: intArray) {
    min = Math.min(min, num);
    max = Math.max(max, num);
}
System.out.printf(”min: %d, max: %d%n”, min, max);
```

**요소의 빈도 카운팅**

```
int [ ] intArray = {...}
int [ ] used = new int [10] ; → 1~10까지라서 10
for (int num: intArray) {
    used[num]++;
}
System.out.println(Arrays.toString(used));
}
```

**2차원 배열 원소 중 3의 배수 개수와 합 출력**

```
int [ ] [ ] grid = {{....}, {...}};
int count = 0;
int sum = 0;
for (int [ ] row: grid) {
     for(int num : row) {
          if (num % 3 == 0) {
               count++;
               sum += num;
           }
       }
    }
    System.out.printf(”개수: %d, 총합: %d%n”, count, sum);
}
```

<2차원 배열 탐색>

a-1, b

a, b-1  a, b  a, b+1

a+1, b

상 하 좌 우

행 -1 1 0 0

열 0 0 -1 1

int[ ] dr = {-1, 1, 0, 0}
int[ ] dc = {0. 0. -1, 1}

for(int d = 0 ; d < 4 < d++){

r += dr[d];

c += dc[d];

## 바. 클래스

- **관련 있는 변수, 함수 묶어 만든 사용자 정의 '자료형'**
- new student ← 객체
- 객체 생성 틀
- 어떤 객체 만들지, 각 객체가 어떤 특징(속성, 동작 / 상태, 기능) 가지고 있을지 결정

1. 객체의 구성

- 속성(Attribute) : 멤버 변수  예) int channel;, int volume;
- 동작(Behavior): 메소드  → void는 리턴값 없을 때 사용

  예)  public void channelUp(), public void volumeDown()

ㄴ 클래스 설계 → 객체 생성 → 클래스에 정의된 속성, 동작을 가지고 동작

class TV { ... → Tv tv = new TV(); → tv.channelDown();

2. 클래스의 선언

public, (default)  final, abstract

protected, private  static, synchronized

[접근 제한자]  [활용 제한자] class 클래스명 {

속성 정의 (멤버 변수)
기능 정의 (메소드)

}

객체(object/ instance)와 인스턴스 변수?

***<메소드>***

1. 기본

- 객체가 할 수 있는 행동 정의
- 이름 소문자로 시작하는 것이 관례

[접근 제한자] [활용 제한자] 반환값 메소드 이름 ([매개 변수들]) {

    행위 기술

}

public static void main (String [ ] a) { }

2. 메소드 선언

- 선언 시 { } 안에 메소드가 해야 할 일 정의

3. 메소드 호출

- 호출한 메소드가 선언되어 있는 클래스 접근

예) class Test {

        public static void call( ) {

        }

    }

- 클래스 객체, 메소드 이름으로 호출

예) Test t = new Test();

    t.call(100);

- static이 메소드에 선언되어 있을 때는 클래스 이름, 메소드 이름으로 호출

4. 여러 특징

- 매개변수: 메소드에서 사용하는 것
- 인자: 호출하는 쪽에서 전달하는 것
- 메소드에서 받은 매개변수는 그 메소드에서 선언한 지역 변수와 똑같이 간주

ㄴ 매개변수는 반드시 해당 유형의 값을 전달

- 메소드로부터 값을 받을 수도 있음, 값이 전달되는 방식 이용

ㄴ 여러 개 인자 전달 가능

- 리턴 유형: 메소드 선언할 때 지정

ㄴ 리턴 유형이 정해져 있으면 반드시 그 유형의 값 리턴

- 여러 개 값 리턴: 배열, Collection 객체 이용
- 리턴값 무조건 사용은 X

## 사. 생성자

: 객체가 실행될 때 처음 한 번 실행되는 함수

1. 클래스 명과 이름 동일 얘) Person() {};
2. 문법적으로 아예 반환 유형 없음
3. 기본 생성자
   - 클래스 내 생성자가 하나도 없을 경우 JVM 자동 비어있는 기본 생성자 제공
   - 형태: 매개변수 X, 클래스 명 () {}

예) Class Dog {
**Dog ( ) { }**

4. 생성자도 함수기 때문에, 필요하다면 매개변수 받을 수 있다

예) Person (String n, int , height h) {
name = n;
int = i;
height = h;
}

5. 메소드 오버로딩
   - 클래스 내 이름이 같고, 매개 변수가 다른 메소드 여러 개 정의

   → 본인 타입에 맞는 함수 실행

예) Class Dog {
**Dog ( ) { }**
Dog (String name) { }
Dog (int age) { }
Dog (String name, int age) { {
}

class Main {
public static void main(String [ ] a) {
Dog d = new Dog ( ) ; → 1이 불림
Dog d2 = new Dog (”쫑”); → 2가 불림
Dog d3 = new Dog(3); → 3이 불림
Dog d4 = new Dog(”메리”, 4); → 4가 불림

```
  }
}
```

5. 객체 생성 시 속성 초기화 담당

```
class Dog {
        String name;
        int age;
        Dog (String n, int a) {
              name = n;
              age = a;
         }

     class Main {
           public static void main(String [ ] a) {
               Dog d = new Dog ( ) ;
               d.name = “쫑”;
               d.age = 3;
               Dog d2 = new Dog (”메리”, 4);
```

6. this의 활용: static 영역에서는 사용 불가능

→ 함수 내 지역변수, 객체 주소 나타냄

- this.멤버 변수
- 지역변수와 멤버변수 이름이 같을 때 멤버변수 이름 지목
- this. ([인자값]) : 생성자 호출
- this 생성자 호출 시 제한사항
  - 생성자 내에서만 호출 가능
  - 생성자 내 첫 번째 구문에 위치해야 함

예)

```
Class Dog {
         String name;
         int age;
         void info () {
              System.out.print(this.name);
              System.out.println(this.age);
              }
```

```
}
```