# <혼자 공부하는 자바>

## Chapter 01. 자바 시작하기

### 01-1 프로그래밍 언어와 자바

컴퓨터가 이해할 수 있는 기계어는 0과 1로 이루어진 이진 코드를 사용합니다. 사람들이 일상에서 일상생활에서 사용하는 언어와는 매우 다릅니다. 사람과 컴퓨터가 대화하기 위해서는 사람의 언어와 기계어의 다리 역할을 하는 프로그래밍 언어가 필요하다.

소스 파일 : 프로그래밍 언어로 작성한 파일, 컴퓨터가 바로 이해할 수 없기 때문에 컴파일이라는 과정을 통해 0과 1로 이루어진 기계어 파일로 번역 후에 컴퓨터에서 사용한다.

**자바 소개**

- 모든 운영체제에서 실행 가능
- 객체 지향 프로그래밍
- 메모리 자동 정리
- 무료 라이브러리 풍부

**자바 개발 도구 설치**

| 구분 | Open JDK | Oracle JDK |
|---|---|---|
| 라이선스 종류 | GNU GPL v2,<br>with the Classpath Exception | Oracle Technology Network License<br>Agreement for Oracle Java SE |
| 사용료 | 개발 및 학습용 : 무료, 상업용 : 유료 | 개발 및 학습용 : 무료, 상업용 : 유료 |

- Java SE 주 버전 . 개선 버전 . 업데이트 버전 . (LTS)장기 지원 서비스 버전
- 주 버전 : 자바 언어에 많은 변화가 있을 경우 증가된다.
- 개선 버전 : 0부터 시작하고 주 버전에서 일부 사항이 개선될 때 증가된다. 보통은 모두 0이다.
- 업데이트 버전 : 1~3개월 주기로 버그가 수정될 때마다 증가한다.
- LTS : 자기 지원 서비스(Long Term Support)를 받을 수 있는 버전이다.

**환경 변수 설정**

- JAVA_HOME 환경 변수 등록

| 윈도우 버전 | [시스템 속성] 대화상자 실행 방법 |
|---|---|
| Windows 7 | 시작 -> 제어판 -> 시스템 및 보안 -> 시스템 -> 고급 시스템 설정 |
| Windows 8 | 화면 오른쪽 아래로 마우스 포인터 옮김 -> 설정 -> 제어판 -> 7과 동일 |
| Windows 10 | 검색 -> '제어판' 입력 후 선택 -> 7과 동일 |

- Path 환경 변수 수정

[환경 변수] 대화상자의 [시스템 변수]에서 Path 환경 변수를 선택하고 [편집] 버튼을 클릭한다. -> [환경 변수 편집] 대화상자가 나타나면 [새로 만들기] 버튼을 클릭하고 추가된 항목에 직접 '%JAVA_HOME%\bin' 을 입력한다. -> 입력을 끝내면 등록된 것을 선택하고 [위로 이동] 버튼을 클릭해서 첫 번째 항목으로 올려준다. -> 모든 대화상자의 [확인] 버튼을 클릭해서 환경 변수를 설정을 마친다.

-> 명령 프롬프트(cmd.exe)를 실행한다. -> javac -version을 입력하고 엔터 키를 누른다. (버전이 출력된다면 제대로 환경 변수가 설정되었다는 의미)

### 01-2 이클립스 개발 환경 구축

통합 개발 환경(IDE, Intergrated Development Environment) : 프로젝트 생성, 자동 코드 완성, 디버깅 등과 같이 개발에 필요한 여러 가지 기능을 통합적으로 제공해주는 툴이다.

이클립스는 자바 프로그램을 개발할 수 있도록 구성되어 있지만, 개발자가 추가적으로 플러그인을 설치하면 웹 애플리케이션 개발, C, C++ 애플리케이션 개발 등 다양한 개발 환경을 구축할 수 있다.

## 01-3 자바 프로그램 개발 과정

자바 프로그램을 개발하기 위해서는 우선 파일 확장명이 .java인 텍스트 파일을 생성하고 자바 언어로 코드를 작성해야 한다. 자바 소스 파일을 컴파일러인 javac 명령어로 컴파일한다. 컴파일이 성공하면 확장명이 .class인 바이트 코드 파일이 생성된다.

바이트 코드 파일은 완전한 기계어가 아니므로 바로 실행할 수 있는 파일이 아니다. 완전한 기계어로 번역해서 실행하려면 java 명령어를 사용해야 한다.

### 바이트 코드 파일과 자바 가상 기계

자바 프로그램은 완전한 기계어가 아닌 바이트 코드 파일(.class)로 구성된다. 파일은 운영체제에서 바로 실행할 수 없고, 자바 가상 기계(JVM, Java Virtual Machine)라는 번역기가 필요하다.

- JVM 사용 이유 : 바이트 코드 파일을 다양한 운영체제에서 수정하지 않고 사용할 수 있도록 하기 위함이다.

### 프로젝트 생성부터 실행까지

- 프로젝트 생성 : [File] - [New] - [Java Project]
- 소스 파일 생성과 작성 : src 폴더를 선택하고 우클릭 - [New] - [Package], 패키지 선택하고 우클릭 - [New] - [Class]
- 바이트 코드 실행 : [Run As] - [Java Application]

### 명령 라인에서 컴파일하고 실행하기

- JDK 8 이전 버전 컴파일 : javac -d[바이트 코드 파일 저장 위치][소스 경로/*.java]
- JDK 11 이후 버전 컴파일 : javac -d[바이트 코드 파일 저장 위치][소스 경로/module-info.java 소스 경로/*.java]
- JDK 8 이전 버전 실행 : java -cp[바이트 코드 파일 저장 위치][패키지이름…클래스이름]
- JDK 11 이후 버전 실행 : java -p[바이트 코드 파일 저장 위치]-m 모듈/패키지이름…클래스이름

### 프로그램 소스 분석

public class Hello {}

- 클래스 : 필드 또는 메소드를 포함하는 블록
- 메소드 : 어떤 일을 처리하는 실행문들을 모아 놓은 블록
- 클래스 선언부 : 중괄호 {} 블록의 앞부부인 public class Hello
- 공개 클래스 : public class
- 클래스 이름 : Hello, 소스 파일명과 동일해야 하며, 대소문자도 일치해야 한다.

```
public static void main(String[] args) {
  System.out.println( "Hello, Java" ) }
```

- 메소드 선언부 : 중괄호 {} 블록의 앞부분인 public static void main(String[] args)
- 메소드 이름 : 괄호 () 바로 앞의 main
- 프로그램 실행 진입점(entry point) : java 명령어로 바이트 코드 파일을 실행하면 제일 먼저 main() 메소드를 찾아 블록 내부를 실행한다. 그래서 main() 메소드

주석 사용하기

- 주석 : 코드에 설명을 붙인 것으로, 주로 실행문에 사용한다.

- 라인 주석(//) : //부터 라인 끝까지 주석으로 처리한다.
- 범위 주석(/**/) : /*와 */ 사이에 있는 내용은 모두 주석으로 처리한다.
- 도큐먼트 주석(/***/) : /**와 */ 사이에 있는 내용은 모두 주석으로 처리한다. 주로 javadoc 명령어로 API 도큐먼트를 생성하는 데 사용한다.
- 주석 기호는 코드 내 어디서든 작성이 가능하지만, 문자열( “” ) 내부에서 작성하면 안된다. 문자열 내부에서 주석 기호는 주석문이 아니라 문자열 데이터로 인식하기 때문이다.

실행문과 세미콜론(;)

- 실행문 : 변수 선언, 값 저장, 메소드 호출에 해당하는 코드
- 컴파일러 : 세미콜론(;)까지 하나의 실행문으로 해석하기 때문에 하나의 실행문을 여러 줄에 걸쳐서 작성하고 맨 마지막에 세미콜론(;)을 붙여도 된다. 구분자로 해서 한 줄에 여러 가지 실행문을 작성할 수도 있다.

# Chapter 02. 변수와 타입

## 02-1 변수

- 변수(variable) : 값을 저장할 수 있는 메모리의 특정 번지에 붙이는 이름으로, 자바의 변수는 다양한 타입의 값을 저장할 수 없다. 정수 타입 변수에는 정수값만, 실수 타입 변수에는 실수값만 저장.

하나의 변수에 동시에 두 가지 값을 저장할 수 없고, 하나의 값만 저장할 수 있다.

### 변수 선언

- 변수 선언 : 변수에 어떤 타입(type)의 데이터를 저장할 것인지 그리고 변수 이름이 무엇인지를 결정한다.
- 변수 이름

| 작성 규칙 | 예 |
|---|---|
| 첫 번째 글자는 문자이거나 '$', '_'이어야 하고 숫자로 시작할 수 없다.(필수) | 가능 : price, $price, _companyName<br>불가능 : 1v, @speed, $#value |
| 영어 대소문자를 구분한다.(필수) | firstname과 firstName은 다른 변수 |
| 첫 문자는 영어 소문자로 시작하되, 다른 단어가 붙을 경우 첫 문자를 대문자로 한다.(관례) | maxSpeed, firstName, carBodyColor |
| 문자 수(길이)의 제한은 없습니다. | |
| 자바 예약어는 사용할 수 없다.(필수) | 다음 표 참조 |

- 예약어 : 이미 해당 프로그래밍 언어에서 의미를 갖고 사용되고 있는 단어로 변수 이름으로 사용할 수 없다.

| 분류 | 예약어 |
|---|---|
| 기본 타입 | boolean, byte, char, short, int, long, float, double |
| 접근 제한자 | private, protected, public |
| 클래스와 관련된 것 | class, abstract, interface, extends, implements, enum |
| 객체와 관련된 것 | new, instanceof, this, super, null |
| 메소드와 관련된 것 | void, return |
| 제어문과 관련된 것 | if, else, switch, case, default, for, do, while, break, continue |
| 논리값 | true, false |
| 예외 처리와 관련된 것 | try, catch, finally, throw, throws |
| 기타 | package, import, synchronized, final, static |

### 값 저장

- 대입 연산자(=) 사용
- 변수 초기화 : 변수 선언은 저장되는 값의 종류와 이름만 언급한 것이다. 변수에 최초로 값이 저장될 때 변수가 생성된다.
- 초기값 : 변수 초기화에서 사용된 값

### 변수 사용

- 변수 : 출력문이나 연산식 내부에서 변수에 저장된 값을 출력하거나 연산할 때 사용한다.
- println() 메소드의 매개값에 변수를 사용하면 변수에 저장된 값을 사용해서 출력한다.

## 02-2 기본 타입

- 정수 타입 : byte, char, short, int, long
- 실수 타입 : float, double
- 논리 타입 : boolean

### 정수 타입

| 타입 | 메모리 사용 크기 |
|---|---|

| byte | 1byte | 8bit |
|---|---|---|
| short | 2byte | 16bit |
| char | 2yte | 16bit |
| int | 4byte | 32bit |
| long | 8byte | 64bit |

- 리터럴(literal) : 소스 코드에서 프로그래머에 의해 직접 입력된 값

2진수 : 0b 또는 0B로 시작하고 0과 1로 구성된다.

8진수 : 0으로 시작하고 0~7 숫자로 구성된다.

10진수 : 소수점이 없는 0~9 숫자로 구성된다.

16진수 : 0x 또는 0X로 시작하고 0~9 숫자와 A, B, C, D, E, F 또는 a, b, c, d, e, f로 구성된다.

- char 타입 : 하나의 문자를 작은 따옴표( '' )로 감싼 것을 문자 리터럴이라고 한다. 유니코드로 변환되어 저장된다.

- String 타입 : 작은 따옴표( '' )로 감싼 문자는 char 타입 변수에 저장되어 유니코드로 저장되지만, 큰따옴표( "" )로 감싼 문자 또는 여러 개의 문자들은 유니코드로 변환되지 않는다.

| 이스케이프 문자 | 출력 용도 |
|---|---|
| \t | 탭만큼 띄움 |
| \n | 줄 바꿈(라인 피드) |
| \r | 캐리지리턴 |
| \" | "출력 |
| \' | '출력 |
| \\ | \출력 |
| \u16진수 | 16진수 유니코드에 해당하는 문자 출력 |

### 실수 타입

| 타입 | 메모리 사용 크기 | |
|---|---|---|
| float | 4byte | 32bit |
| double | 8byte | 64bit |

- 실수 리터럴 : 소스 코드에서 소수점이 있는 숫자 리터럴은 10진수 실수로 인식한다. 또한 알파벳 소문자 e 또는 대문자 E가 포함되어 있는 숫자 리터럴은 지수와 가수로 표현된 소수점이 있는 10진수 실수로 인식한다.

- 메모리에 여유가 있고 특별한 이유가 없는 한 실수 리터럴을 저장할 때에는 double 타입을 사용하는 것이 좋습니다.

### 논리 타입

- 참과 거짓을 의미, true와 false를 사용한다.

- boolean 타입 변수 : 두 가지 상태값에 따라 조건문과 제어문의 실행 흐름을 변경하는 데 사용한다.

## 02-3 타입 변환

### 자동 타입 변환

- 말 그대로 자동으로 타입 변환이 일어나는 것을 의미한다.

- 값의 허용 범위가 작은 타입이 허용 범위가 큰 타입으로 저장될 때 발생한다.

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

- char 타입보다 허용 범위가 작은 byte 타입은 char 차입으로 자동 타입 변환될 수 없다. 왜냐하면 char 타입의 허용 범위는 음수를 포함하지 않는데, byte 타입은 음수를 포함하기 때문이다.

### 강제 타입 변환

- 큰 허용 범위 타입을 작은 허용 범위 타입으로 강제로 나눠서 저장하는 것을 의미한다.

### 정수 연산에서의 자동 타입 변환

- 정수 타입 변수가 산술 연산식에서 피연산자로 사용되면 int 타입보다 작은 byte, short 타입의 변수는 int 타입으로 자동 타입 변환되어 연산을 수행한다.

### 실수 연산에서의 자동 타입 변환

- 실수 타입 변수가 산술 연산식에서 피연산자로 사용될 경우 두 피연산자가 동일한 타입이라면 해당 타입으로 연산되지만, 피연산자 중 하나가 double 타입이라면 다른 피연산자도 double 타입으로 자동 타입 변환되어 연산을 수행한다.
- 만약, 꼭 int 타입으로 연산을 해야 한다면 double 타입을 int 타입으로 강제 변환하고 덧셈 연산을 수행하면 된다.

### + 연산에서의 문자열 자동 타입 변환

- 피연산자가 모두 숫자일 경우에는 덧셈 연산을 수행하지만, 피연산자 중 하나가 문자열일 경우에는 나머지 피연산자도 문자열로 자동 변환되어 문자열 결합 연산을 수행한다.
- + 연산자가 연이어 나오면 앞에서부터 순차적으로 + 연산을 수행한다. 먼저 수행된 연산이 덧셈 연산이라면 덧셈 결과를 가지고 그다음 + 연산을 수행한다. 만약 먼저 수행된 연산이 결합 연산이라면 이후 + 연산은 모두 결합 연산이 된다.
- 특정 부분을 우선 연산하고 싶을 경우 : 해당 부분을 괄호()로 감싸줍니다. 괄호는 최우선으로 연산을 수행한다.

문자열을 기본 타입으로 강제 타입 변환

- 문자열이 숫자가 아닌 알파벳이나 특수 문자, 한글 등을 포함하고 있을 경우 숫자 타입으로 변환을 시도하면 숫자 형식 예외가 발생한다.
- 기본 타입(byte, short, char, int, long, float, double, boolean)의 값을 문자열로 변경하는 경우도 있는데, 이 경우는 간단히 String.valueOf() 메소드를 이용하면 된다.
- String str = String.valueOf(기본타입값);

## 02-4 변수와 시스템 입출력

모니터로 변수값 출력하기

| 메소드 | 의미 |
|---|---|
| println(내용); | 괄호 안의 내용을 출력하고 행을 바꿔라 |
| print(내용); | 괄호 안의 내용을 출력만 해라 |
| printf("형식문자열", 값1, 값2, ...) | 괄호 안의 첫 번째 문자열 형식대로 내용을 출력해라 |

| 형식화된 문자열 | | 설명 | 출력 형태 |
|---|---|---|---|
| 정수 | %d | 정수 | 123 |
| | %6d | 6자리 정수. 왼쪽 빈 자리 공백 | ___123 |
| | %-6d | 6자리 정수, 오른쪽 빈 자리 공백 | 123___ |
| | %06d | 6자리 정수, 왼쪽 빈 자리 0 채움 | 000123 |
| 실수 | %10.2f | 소수점 이상 7자리, 소수점 이하 2자리, 왼쪽 빈 자리 공백 | ____123.45 |
| | %-10.2f | 소수점 이상 7자리, 소수점 이하 2자리, 오른쪽 빈 자리 공백 | 123.45____ |
| | %010.2f | 소수점 이상 7자리, 소수점 이하 2자리, 왼쪽 빈 자리 0 채움 | 0000123.45 |
| 문자열 | %s | 문자열 | abc |
| | %6s | 6자리 문자열, 왼쪽 빈 자리 공백 | ___abc |
| | %-6s | 6자리 문자열, 오른쪽 빈 자리 공백 | abc___ |

| | | | |
|---|---|---|---|
| 특 수 문자 | \t | 탭(tab) | |
| | \n | 줄 바꿈 | |
| | %% | % | % |

**키보드에서 입력된 내용을 변수에 저장하기**

| 숫자 | 알파벳 | | | | 기능키 | 방향키 |
|---|---|---|---|---|---|---|
| | A = 65 | N = 78 | a = 97 | n = 110 | | |
| | B = 66 | O = 79 | b = 98 | m =111 | | |
| 0 = 48 | C = 67 | P = 80 | c = 99 | p = 112 | BACK SPACE = 8 | |
| 1 = 49 | D = 68 | Q = 81 | d = 100 | q = 113 | TAB = 9 | |
| 2 = 50 | E = 69 | R = 82 | e = 101 | r = 114 | ENTER = [CR=13, LF=10] | |
| 3 = 51 | F = 70 | S = 83 | f = 102 | s = 115 | SHIFT = 16 | ← = 37 |
| 4 = 52 | G = 71 | T = 84 | g = 103 | t = 116 | CONTROL = 17 | ↑ = 38 |
| 5 = 53 | H = 72 | U = 85 | h =104 | u = 117 | ALT = 18 | → = 39 |
| 6 = 54 | I = 73 | V = 86 | i = 105 | v = 118 | ESC = 27 | ↓ = 40 |
| 7 = 55 | J = 74 | W = 87 | j = 106 | w = 119 | SPACE = 32 | |
| 8 = 56 | K = 75 | X = 88 | k = 107 | x = 120 | PAGEUP = 33 | |
| 9 = 57 | L = 76 | Y = 89 | l = 108 | y = 121 | PAGEDN = 34 | |
| | M = 77 | Z = 90 | m = 109 | z = 122 | | |

- System.in.read() : 키보드에서 입력된 키코드를 읽습니다.

- Scanner : System.in.read()는 키코드를 하나씩 읽기 때문에 2개 이상의 키가 조합된 한글을 읽을 수 없다. 키보드로부터 입력된 내용을 통 문자열로 읽기 위해 사용할 수 있다.

# Chapter 03. 연산자

## 03-1 연산자와 연산식

연산자(operator) : 연산에 사용되는 표시나 기호

피연산자(operand) : 연산자와 함께 연산되는 데이터

연산식(expression) : 연산자와 피연산자를 이용해 연산의 과정을 기술한 것

### 연산자의 종류

| 연산자 종류 | 연산자 | 피연산자 수 | 산출값 | 기능 |
|---|---|---|---|---|
| 산술 | +, -, *, /, % | 이항 | 숫자 | 사칙연산 및 나머지 계산 |
| 부호 | +, - | 단항 | 숫자 | 음수와 양수의 부호 |
| 문자열 | + | 이항 | 문자열 | 두 문자열을 연결 |
| 대입 | =, +=, -=, *=, /=, %= | 이항 | 다양 | 우변의 값을 좌변의 변수에 대입 |
| 증감 | ++, -- | 단항 | 숫자 | 1만큼 증가/감소 |
| 비교 | ==, !=, >, <, >=, <= ,intanceof | 이항 | boolean | 값의 비교 |
| 논리 | !, &, \|, &&, \|\| | 단항<br>이항 | boolean | 논리 부정, 논리곱, 논리합 |
| 조건 | (조건식)? A:B | 삼항 | 다양 | 조건식에 따라 A 또는 B 중 하나를 선택 |

- 연산식은 반드시 하나의 값을 산출하며, 값 대신에 연산식을 사용할 수 있다.

### 연산의 방향과 우선순위

1. 단항, 이항, 삼항 연산자 순으로 우선순위
2. 산술, 비교, 논리, 대입 연산자 순으로 우선순위
3. 단항, 부호, 대입 연산자를 제외한 모든 연산의 방향은 왼쪽에서 오른쪽
4. 복잡한 연산식에는 괄호 ()를 사용해서 우선순위

## 03-2 연산자의 종류

### 단항 연산자

- 부호 연산자(+, -)

| 연산식 | | 설명 |
|---|---|---|
| + | 피연산자 | 피연산자의 부호 유지 |
| - | 피연산자 | 피연산자의 부호 변경 |

- 증감 연산자(++, --)

| 연산식 | | 설명 |
|---|---|---|
| ++ | 피연산자 | 다른 연산을 수행하기 전에 피연산자의 값을 1 증가시킴 |
| -- | 피연산자 | 다른 연산을 수행하기 전에 피연산자의 값을 1 감소시킴 |
| 피연산자 | ++ | 다른 연산을 수행한 후에 피연산자의 값을 1 증가시킴 |
| 피연산자 | -- | 다른 연산을 수행한 후에 피연산자의 값을 1 감소시킴 |

- 논리 부정 연산자(!)

| 연산식 | | 설명 |
|---|---|---|
| ! | 피연산자 | 피연산자가 true이면 false 값을 산출<br>피연산자가 false이면 true 값을 산출 |

### 이항 연산자

- 산술 연산자(+, -, *, /, %)

| 연산식 | | | 설명 |
|---|---|---|---|
| 피연산자 | + | 피연산자 | 덧셈 연산 |
| 피연산자 | - | 피연산자 | 뺄셈 연산 |
| 피연산자 | * | 피연산자 | 곱셈 연산 |
| 피연산자 | / | 피연산자 | 왼쪽 피연산자를 오른쪽 피연산자로 나눗셈 연산 |
| 피연산자 | % | 피연산자 | 왼쪽 피연산자를 오른쪽 피연산자로 나눈 나머지를 구하는 연산 |

- 문자 결합 연산자(+) : 산술 연산자, 부호 연산자인 동시에 문자열 결합 연산자이기도 하다. 피연산자 중 한쪽이 문자열이면 + 연산자는 문자열 결합 연산자로 사용되어 다른 피연산자로 문자열로 변환하고 서로 결합한다.

- 비교 연산자(<, <=, >, >=, ==, !=)

| 구분 | 연산식 | | | 설명 |
|---|---|---|---|---|
| 동등 비교 | 피연산자1 | == | 피연산자2 | 두 피연산자의 값이 같은지를 검사 |
| | 피연산자1 | != | 피연산자2 | 두 피연산자의 값이 다른지를 검사 |
| 크기 비교 | 피연산자1 | > | 피연산자2 | 피연산자1이 큰지를 검사 |
| | 피연산자1 | >= | 피연산자2 | 피연산자1이 크거나 같은지를 검사 |
| | 피연산자1 | < | 피연산자2 | 피연산자1이 작은지를 검사 |
| | 피연산자1 | <= | 피연산자2 | 피연산자1이 작거나 같은지를 검사 |

- 논리 연산자(&&, ||, &, |, ^, !)

| 구분 | 연산식 | | | 결과 | 설명 |
|---|---|---|---|---|---|
| AND (논리곱) | true | && 또는 & | true | true | 피연산자가 모두 true일 경우에만 연산 결과가 true |
| | true | | false | false | |
| | false | | true | false | |
| | false | | false | false | |
| OR (논리합) | true | \|\| 또는 \| | true | true | 피연산자 중 하나만 true이면 연산 결과는 true |
| | true | | false | true | |
| | false | | true | true | |
| | false | | false | false | |
| XOR (배타적 논리합) | true | ^ | true | false | 피연산자가 하나는 true이고 다른 하나가 false일 경우에만 연산 결과가 true |
| | true | | false | true | |
| | false | | true | true | |
| | false | | false | false | |
| NOT (논리 부정) | | ! | true | false | 피연산자의 논리값을 바꿈 |
| | | | false | true | |

- 대입 연산자(=, +=, -=, *=, /=, %=)

| 구분 | 연산식 | | | 설명 |
|---|---|---|---|---|
| 단순 대입 연산자 | 변수 | = | 피연산자 | 오른쪽의 피연산자의 값을 왼쪽 변수에 저장 |
| 복합 대입 연산자 | 변수 | += | 피연산자 | 변수=변수+피연산자와 동일 |
| | 변수 | -= | 피연산자 | 변수=변수-피연산자와 동일 |
| | 변수 | *= | 피연산자 | 변수=변수*피연산자와 동일 |
| | 변수 | /= | 피연산자 | 변수=변수/피연산자와 동일 |
| | 변수 | %= | 피연산자 | 변수=변수%피연산자와 동일 |
| | 변수 | &= | 피연산자 | 변수=변수&피연산자와 동일 |
| | 변수 | \|= | 피연산자 | 변수=변수\|피연산자와 동일 |
| | 변수 | ^= | 피연산자 | 변수=변수^피연산자와 동일 |

**삼항 연산자**

- 3개의 피연산자를 필요로 하는 연산자

- 조건식(피연산자1) ? 값 또는 연산식(피연산자2) : 값 또는 연산식(피연산자3)

# Chapter 04. 조건문과 반복문

## 04-1 조건문: if문, switch문

### if문

- 조건식의 결과에 따라 블록 실행 여부가 결정된다.
- true 또는 false 값을 산출할 수 있는 연산식이나. boolean 타입 변수가 올 수 있다.
- 조건식이 true이면 블록을 실행하고, false이면 블록을 실행하지 않는다.

### if-else문

- if문의 조건식이 true이면 if문의 블록이 실행되고, 조건식이 false이면 else 블록이 실행된다.

### if-else if-else문

- 처음 if문의 조건식의 false일 경우 다른 조건식의 결과에 따라 실행 블록을 선택할 수 있는데, if 블록의 끝에 else if문을 붙이면 된다.
- else if문의 수는 제한이 없으며, 여러 개의 조건식 중 true가 되는 블록만 실행되고 전체 if문을 벗어나게 된다.
- else if 블록의 마지막에는 else 블록을 추가할 수 있는데, 모든 조건식이 false일 경우 else 블록을 실행하고 if문을 벗어나게 된다.

switch문

- if문처럼 조건식이 true일 경우에 블록 내부의 실행문을 실행하는 것이 아니라, 변수가 어떤 값을 갖느냐에 따라 실행문이 선택된다.
- 변수의 값에 따라서 실행문이 결정되기 때문에 같은 기능의 if문보다 코드가 간결하다.
- 괄호 안의 변수 값과 동일한 값을 갖는 case로 가서 실행문을 실행한다. 만약 괄호 안의 변수 값과 동일한 값을 갖는 case가 없으면 default로 가서 실행문을 실행한다. default는 생략 가능하다.
- case 끝에 break가 붙어 있는 이유는 case를 실행하지 않고 switch문을 빠져나가기 위해서이다. break가 없다면 다음 case가 연달아 실행되는데, 이때는 case 값과 상관없이 실행된다.

## 04-2 반복문: for문, while문, do-while문

### for문

- 주어진 횟수만큼 실행문을 반복 실행할 때 적합한 반복 제어문이다.
- 주의할 점 : 초기화식에서 루프 카운터 변수를 선언할 때 부동 소수점을 쓰는 float 타입을 사용하지 말아야 한다는 것이다.
- 중첩 for문 : for문은 또 다른 for문을 내포하고 있따. 바깥쪽 for문이 한 번 실행할 때마다 중첩된 for문은 지정된 횟수만큼 반복해서 돌다가 다시 바깥쪽 for문으로 돌아간다.

### while문

- 조건식이 true일 경우에 계속해서 반복한다.
- 조건식에는 비교 또는 논리 연산식이 주로 오는데, 조건식이 false가 되면 반복 행위를 멈추고 while문을 종료한다.
- 조건식에 true를 사용하면 무한루프를 돌게 된다. 무한히 반복해서 실행하기 때문에 언젠가는 while문을 빠져나가기 위한 코드가 필요하다.
- while문을 종료시키기 위해서는 변수의 값을 fasle로 만들거나, break문을 이용하는 방법이 있다.

### do-while문

- 조건식에 의해 반복 실행한다는 점에서 while문과 동일하다.
- 블록 내부의 실행문을 우선 실행하고 실행 결과에 따라서 반복 실행을 계속할지 결정한다.

break문

- for문, while문, do-while문의 실행을 중지할 때 사용된다.
- 대게 if문과 같이 사용되어 if문의 조건식에 따라 for문과 while문을 종료할 때 사용한다.
- 만약 반복문이 중첩되어 있을 경우 break문은 가장 가까운 반복문만 종료하고 바깥쪽 반복문은 종료하지 않는다. 중첩된 반복문에서 바깥쪽 반복문까지 종료시키려느 바깥쪽 반복문에 이름(라벨)을 붙이고, break 이름;을 사용하면 된다.

continue문

- 반복문인 for문, while문, do-while문에서만 사용되는데, 블록 내부에서 continue문이 실행되면 for문의 증감식 또는 while문, do-while문의 조건식으로 이동한다.
- 반복문을 종료하지 않고 계속 반복을 수행한다는 점이 break문과 다르다.
- 대게 if문과 같이 사용되는데, 특정 조건을 만족하는 경우에 continue문을 실행해서 그 이후의 문장을 실행하지 않고 다음 반복으로 넘어간다.

# Chapter 05. 참조 타입

## 05-1 참조 타입과 참조 변수

### 기본 타입과 참조 타입

- 기본 타입 : 정수, 실수, 문자, 논리 리터럴을 저장하는 타입을 말한다.
- 참조 타입 : 객체의 번지를 참조하는 타입으로 배열, 열거, 클래스, 인터페이스를 말한다.
- 기본 타입 변수와 참조 타입 변수의 차이점은 저장되는 값이 무엇인지에 따라 결정된다.

### 메모리 사용 영역

- 메소드 영역 : JVM이 시작할 때 생성되고 모든 스레드가 공유하는 영역이다.
- 코드에서 사용되는 클래스들은 클래스 로더로 읽어 클래스별로 정적 필드와 상수, 메소드 코드, 생성자 코드 등을 분류해서 저장한다.
- 힙 영역 : 객체와 배열이 생성되는 영역이다. 여기에 생성된 객체와 배열은 JVM 스택 영역의 변수나 다른 객체의 필드에서 참조한다. 만일 참조하는 변수나 필드가 없다면 의미 없는 객체가 되기 때문에 JVM이 이것을 쓰레기로 취급하고 쓰레기 수집기를 실행시켜 자동으로 제거한다.
- JVM 스택 영역 : 메소드를 호출할 때마다 프레임을 추가하고 메소드가 종료되면 해당 프레임을 제거하는 동작을 수행한다.

### 참조 변수의 ==, != 연산

- 변수의 값이 같은지, 아닌지를 조사하지만 참조 타입 변수들 간의 ==, != 연산은 동일한 객체를 참조하는지, 다른 객체를 참조하는지 알아볼 때 사용된다.
- 동일한 번지 값을 갖고 있다는 것은 동일한 객체를 참조한다는 의미이다. 동일한 객체를 참조하고 있을 경우 == 연산의 결과는 true이고 != 연산 결과는 false이다.

### null과 NullPointerException

- 참조 타입 변수는 힙 영역의 객체를 참조하지 않다는 뜻으로 null(널) 값을 가질 수 있다.
- null 값도 초기값으로 사용할 수 있기 때문에 null로 초기화된 참조 변수는 스택 영역에 생성된다.
- 참조 변수가 null을 가지고 있을 경우에는, 참조 객체가 없으므로 변수를 통해 객체를 사용할 수 없다. 만약 null 상태에서 있지도 않은 객체의 데이터(필드)나 메소드를 사용하는 코드를 실행하면 NullPointerException이 발생한다.

### String 타입

- 문자열은 String 객체로 생성되고 변수는 String 객체를 참조하기 때문이다. 하지만 일반적으로 String 변수에 저장한다고 표현한다.
- 변수에 문자열을 저장할 경우에는 문자열 리터럴을 사용하지만, new 연산자를 사용해서 직접 String 객체를 생성시킬 수도 있다.
- new 연산자 : 힙 영역에 새로운 객체를 만들 때 사용하는 연산자, 객체 생성 연산자라고 한다.

## 05-2 배열

### 배열이란?

- 같은 타입의 데이터를 연속된 공간에 나열하고 각 데이터에 인덱스를 부여해놓은 자료구조이다.
- int 배열은 int 값만 저장 가능하고, String 배열은 문자열만 저장한다.
- 한 번 생성된 배열은 길이를 늘리거나 줄일 수 없다.

### 배열 선언

- 타입[] 변수;
- 타입 변수[];
- 대괄호 []는 배열 변수를 선언하는 기호로 사용되는데, 타입 뒤에 붙을 수도 있고 변수 뒤에 붙을 수도 있다. 타입은 배열에 저장될 데이터의 타입을 말한다.
- 배열 변수는 참조 변수에 속한다. 배열도 객체이므로 힙 영역에 생성되고 배열 변수는 힙 영역의 배열 객체를 참조하게 된다. 만일 참조할 배열 객체가 없다면 배열 변수는 null 값으로 초기화될 수 있다.
- 타입[] 변수 = null;

### 배열 생성

- 값 목록으로 배열 생성 : 타입[] 변수 = {값0, 값1, 값2, … };
- 중괄호 {}는 주어진 값들을 항목으로 가지는 배열 객체를 힙에 생성하고, 배열 객체의 번지를 리턴한다.
- 배열 변수는 리턴된 번지를 저장함으로써 참조가 이뤄진다.
- 배열 변수를 이미 선언한 후에는 다른 실행문에서 중괄호를 사용한 배열 생성이 허용되지 않는다. new 연산자를 사용해서 값 목록을 지정해주면 된다. new 연산자 바로 뒤에는 배열 변수 선언에서 사용한 "타입[]"를 붙여주고 중괄호 {}에는 값들을 나열해준다.
- new 연산자로 배열 생성 : 값의 목록을 가지고 있지 않지만, 향후 값들을 저장할 배열을 미리 만들고 싶다면 new 연산자로 배열 객체를 생성할 수 있다. 타입[] 변수 = new 타입[길이];
- 길이는 배열이 저장할 수 있는 값의 개수를 말한다. 이미 배열 변수가 선언된 경우에도 new 연산자로 배열을 생성할 수 있다
- new 연산자로 배열을 처음 생성할 경우 배열은 자동적으로 기본값으로 초기화된다.

| 분류 | 타입 | 초기값 |
|---|---|---|
| 기본 타입(정수) | byte[] | 0 |
| | char[] | '\u0000' |
| | short[] | 0 |
| | int[] | 0 |
| | long[] | 0L |
| 기본 타입(실수) | float[] | 0.0F |
| | double[] | 0.0 |
| 기본 타입(논리) | boolean[] | false |
| 참조 타입 | 클래스[] | null |
| | 인터페이스[] | null |

- 배열이 생성되고 나서 특정 인덱스 위치에 새로운 값을 저장하려면 : 변수[인덱스] = 값;

### 배열 길이

- 배열에 저장할 수 있는 전체 항목의 개수
- 배열의 길이를 얻으려면 배열 객체의 length 필드를 읽는다. 참고로 필드는 객체 내부의 데이터를 말한다.
- 배열 변수.length;

### 명령 라인 입력

- 명령 라인(명령 프롬프트)에서 java 명령어로 실행하면 JVM은 길이가 0인 String 배열을 먼저 생성하고 main() 메소드를 호출할 때 매개값으로 전달한다.
- [JDK 11 이후 버전] java -p . -m 모듈명/패키지.클래스 문자열0 문자열1 문자열2 … 문자열n-1
[JDK 8 이전 버전] java 패키지.클래스 문자열0 문자열1 문자열2 … 문자열n-1

- main() 메소드 : String[]args 매개 변수를 통해서 명령 라인에서 입력된 데이터의 수(배열의 길이)와 입력된 데이터(배열의 항목 값)를 알 수 있게 된다.

### 다차원 배열

- 2차원 배열 : 수학의 행렬 같은 모양으로, 가로 세로 인덱스를 사용한다.
- 배열의 생성 원리는 수학 행렬과 근본적으로 다르지만 사용 방식은 행렬과 동일하다. 1차원 배열이 서로 연결된 구조로 다차원 배열을 구현하기 때문에 수학 행렬 구조가 아닌 계단식 구조를 가질 수 있다.

객체를 참조하는 배열

- 기본 타입(byte, char, short, int, long, float, double, boolean) 배열은 각 항목에 직접 값을 갖고 있지만, 참조 타입(클래스, 인터페이스) 배열은 각 항목에 객체의 번지를 갖고 있다.

### 배열 복사

- 배열은 한 번 생성하면 크기를 변경할 수 없기 때문에 더 많은 저장 공간이 필요하다면 더 큰 배열을 새로 만들고 이전 배열로부터 항목 값들을 복사해야 한다.
- 배열 간의 항목 값들을 복사하려면 for문을 사용하거나 System.arraycopy() 메소드를 사용한다.

### 향상된 for문

- 자바는 배열이나 컬렉션을 좀 더 쉽게 처리하기 위해 향상된 for문을 제공한다.
- 반복 실행을 하기 위해 루프 카운터 변수와 증감식을 사용하지 않는다.
- for문의 괄호 ()에는 배열에서 꺼낸 항목을 저장할 변수 선언과 콜론(:) 그리고 배열을 나란히 작성한다. 배열 및 컬렉션 항목의 개수만큼 반복하고, 자동적으로 for문을 빠져나간다.

# Chapter 06. 클래스

## 06-1 객체 지향 프로그래밍

- 객체(object) : 물리적으로 존재하거나 추상적으로 생각할 수 있는 것 중에서 자신의 속성을 가지고 있으면서 식별 가능한 것을 말한다.

### 객체의 상호작용

- 객체들은 독립적으로 존재하고, 다른 객체와 서로 상호작용 하면서 동작한다. 객체들 사이의 상호작용 수단은 메소드이다. 이때 객체가 다른 객체의 기능을 이용하는 것이 바로 메소드 호출이다.

### 객체 간의 관계

- 객체 : 개별적으로 사용될 수 있지만, 대부분 다른 객체와 관계를 맺고 있다.
- 집합 관계 : 객체는 하나의 부품이고 하나는 완성품에 해당한다.
- 사용 관계 : 객체 간의 상호작용을 말한다. 객체는 다른 객체의 메소드를 호출해 원하는 결과를 얻어낸다.
- 상속 관계 : 상위(부모) 객체를 기반으로 하위(자식) 객체를 생성하는 관계를 말한다. 상위 객체는 종류를 의미하고, 하위 객체는 구체적인 사물에 해당한다.

### 객체와 클래스

- 설계도는 클래스, 클래스로 만든 객체는 인스턴스
- 객체 지향 프로그래밍 개발 : 클래스 설계한다. -> 설계된 클래스를 가지고 사용할 객체를 생성한다. -> 생성된 객체를 이용한다.

### 클래스 선언

- 하나 이상의 문자로 이루어야 한다.
- 첫 글자에는 숫자가 올 수 없다.
- '$', '_' 외의 특수 문자는 사용할 수 없다.
- 자바 키워드는 사용할 수 없다.
- public 접근 제한자 : 파일 이름과 동일한 이름의 클래스 선언에만 붙일 수 있다. 만약 파일 이름과 일치하지 않는 클래스 선언에 public 접근 제한자를 붙이면 컴파일 에러가 발생한다.

### 객체 생성과 클래스 변수

- new 클래스();
- new 연산자 뒤에 생성자가 오는데, 생성자는 클래스() 형태를 가지고 있다.
- new 연산자로 생성된 객체는 메모리 힙 영역에 생성된다.

### 클래스의 구성 멤버

- 필드(Field) : 객체의 데이터가 저장되는 곳, 생성자와 메소드 내에서만 사용되고 생성자와 메소드가 실행 종료되면 자동 소멸된다.
- 생성자(Constructor) : 객체 생성 시 초기화 역할 담당, 필드를 초기화 하거나 메소드를 호출해서 객체를 사용할 준비를 한다.
- 메소드(Method) : 객체의 동작에 해당하는 실행 블록, 메소드를 호출하게 되면 중괄호 블록에 있는 모든 코드들이 일괄적으로 실행된다.

## 06-2 필드

- 필드 : 객체의 고유 데이터, 객체가 가져야 할 부품, 객체의 현재 상태 데이터를 저장하는 곳이다.

### 필드 선언

- 생성자 선언과 메소드 선언의 앞과 뒤 어떤 곳에서도 필드 선언이 가능하다.
- 생성자와 메소드 중괄호 {} 블록 내부에는 선언될 수 없다.
- 타입 필드 [ = 초기값];

### 필드 사용

- 클래스 내부의 생성자나 메소드에서 사용할 경우 단순히 필드 이름으로 읽고 변경하면 되지만, 클래스 외부에서 사용할 경우 우선적으로 클래스로부터 객체를 생성한 뒤 필드를 사용해야 한다.

## 06-3 생성자

- 생성자 : new 연산자로 클래스로부터 객체를 생성할 때 호출되어 객체의 초기화를 담당한다.
- 객체 초기화 : 필드를 초기화하거나 메소드를 호출해서 객체를 사용할 준비를 하는 것을 말한다.
- 생성자가 성공적으로 실행되지 않고 예외(에러)가 발생했다면 객체는 생성되지 않는다.

### 기본 생성자

- [public] 클래스() { }
- 클래스가 public class로 선언되면 기본 생성자에도 public이 붙지만, 클래스가 public 없이 class로만 선언되면 기본 생성자에도 public이 붙지 않는다.

### 생성자 선언

- 메소드와 비슷한 모양을 가지고 있으나, 리턴 타입이 없고 클래스 이름과 동일하다.
- 생성자 블록 내부에는 객체 초기화 코드가 작성되는데, 일반적으로 필드에 초기값을 저장하거나 메소드를 호출해 객체 사용 전에 필요한 준비를 한다.

### 필드 초기화

- 클래스로부터 객체가 생성될 때 필드는 기본 초기값으로 자동 설정된다.
- 필드를 선언할 때 초기값을 주는 방법, 생성자에서 초기값을 주는 방법

### 생성자 오버로딩

- 생성자 오버로딩 : 매개 변수를 달리하는 생성자를 여러 개 선언하는 것을 말한다.
- 오버로딩 되어 있을 경우, new 연산자로 생성자를 호출할 때 제공되는 매개값의타입과 수에 의해 호출될 생성자가 결정된다.

다른 생성자 호출: this()

- 생성자 오버로딩이 많아질 경우 생성자 간의 중복된 코드가 발생할 수 있다. 매개 변수의 수만 달리하고 필드 초기화 내용이 비슷한 생성자에서 이러한 현상을 많이 볼 수 있다.
- 필드 초기화 내용은 한 생성자에만 집중적으로 작성하고 나머지 생성자는 초기화 내용을 가지고 있는 생성자를 호출하는 방법으로 개선할 수 있다.
- this()는 자신의 다른 생성자를 호출하는 코드로 반드시 생성자의 첫 줄에서만 허용된다.

## 06-4 메소드

- 메소드 시그니처 : 메소드 선언부
- 리턴 타입 : 메소드가 리턴하는 결과의 타입을 표시한다.
- 메소드 이름 : 메소드의 기능이 드러나도록 식별자 규칙에 맞게 이름을 지어준다.
- 매개 변수 선언 : 메소드를 실행할 때 필요한 데이터를 받기 위한 변수를 선언한다.

- 메소드 실행 블록 : 실행할 코드를 작성한다.

**메소드 선언**

- 리턴 타입 : 리턴값의 타입을 말한다.
- 리턴값 : 메소드를 실행한 후의 결과값을 말한다.
- 메소드는 리턴값이 있을 수도 있고 없을 수도 있으나 리턴값이 있을 경우 리턴 타입이 선언부에 명시되어야 한다.
- 메소드 이름 : 자바 식별자 규칙에 맞게 작성하면 된다.
- 매개 변수 선언 : 매개 변수는 메소드가 실행할 때 필요한 데이터를 외부로부터 받기 위해 사용된다. 메소드에서 매개 변수가 필요한 경우가 있고 필요 없는 경우가 있다.
- 매개 변수의 개수를 모를 경우 : 갯수가 이미 정해져 있는 것이 일반적이지만, 어떤 상황에서는 메소드를 선언할 때 매개 변수의 개수를 알 수 없는 경우도 있다.

**리턴(return)문**

- 리턴값이 있는 메소드 : 리턴문을 사용해서 리턴값을 지정해야 한다. 만약 return문이 없다면 컴파일 에러가 발생하고, return문이 실행되면 메소드는 즉시 종료된다.
- return 리턴값;
- return문의 리턴값은 리턴 타입이거나 리턴 타입으로 변환될 수 있어야 한다.
- 리턴값이 없는 메소드: void : 선언된 메소드에서도 return문을 사용할 수 있다. 리턴값을 지정하는 것이 아니라 메소드 실행을 강제 종료시키는 역할을 한다.

**메소드 호출**

- 클래스 내외부의 호출에 의해 실행된다.
- 클래스 내부의 다른 메소드에서 호출할 경우에는 단순한 메소드 이름으로 호출하면 되지만, 클래스 외부에서 호출할 경우에는 우선 클래스로부터 객체를 생성한 뒤 참조 변수를 이용해서 메소드를 호출해야 한다.
- 객체 내부에서 호출 : 메소드가 매개 변수를 가지고 있을 때에는 매개 변수의 타입과 수에 맞게 매개값을 제공한다. 변수 타입은 메소드 리턴 타입과 동일하거나, 자동 타입 변환이 될 수 있어야 한다는 점에서 주의해야 한다.
- 객체 외부에서 호출 : 객체가 생성되었다면 참조 변수와 함께 도트(.) 연산자를 사용해서 메소드를 호출할 수 있다. 도트(.) 연산자는 객체 접근 연산자로 객체가 가지고 있는 필드나 메소드에 접근할 때 사용된다.

**메소드 오버로딩**

- 오버로딩 : 클래스 내에 같은 이름의 메소드를 여러 개 선언하는 것
- 하나의 메소드 이름으로 여러 기능을 담는다 하여 붙여진 이름이라 생각하면 된다.
- 매개값을 다양하게 받아 처리할 수 있도록 하기 위해서이다.

## 06-5 인스턴스 멤버와 정적 멤버

- 인스턴스 멤버는 객체마다 가지고 있는 멤버를 말하고, 정적 멤버는 클래스에 위치시키고 객체들이 공유하는 멤버를 말한다.

**인스턴스 멤버와 this**

- 인스턴스 멤버 : 객체(인스턴스)를 생성한 후 사용할 수 있는 필드와 메소드를 말하는데, 이들을 각각 인스턴스 필드, 인스턴스 메소드라 한다.
- 인스턴스 멤버 선언 : 메소드는 코드 블록이므로 객체마다 동일한 코드 블록을 가지고 있을 필요가 없기 때문이

다. 인스턴스 필드가 사용되면 메소드 역시 객체 없이는 실행할 수 없다.

- this : 객체 외부에서 인스턴스 멤버에 접근하기 위해 참조 변수를 사용하는 것과 마찬가지로 내부에서도 인스턴스 멤버에 접근하기 위해 this를 사용할 수 있다. 객체는 자신을 this라고 한다.
- this는 주로 생성자와 메소드의 매개 변수 이름이 필드와 동일한 경우, 인스턴스 멤버인 필드임을 명시하고자 할 때 사용된다.

### 정적 멤버와 static

- 정적 멤버 : 클래스에 고정된 멤버로서 객체를 생성하지 않고 사용할 수 있는 필드와 메소드를 말한다.
- 정적 멤버 선언 : 정적 필드와 정적 메소드를 선언하려면 필드와 메소드 선언 시 static 키워드를 추가적으로 붙이면 된다.

0 정적 필드와 정적 메소드는 클래스에 고정된 멤버이므로 클래스 로더가 클래스(바이트 코드)를 로딩해서 메소드 메모리 영역에 적재할 때 클래스별로 관리된다. 클래스의 로딩이 끝나면 바로 사용할 수 있다.

- 정적 멤버 사용 : 클래스가 메모리로 로딩되면 정적 멤버로 바로 사용할 수 있는데, 클래스 이름과 함께 도트(.) 연산자로 접근한다.
- 정적 메소드 선언 시 주의할 점 : 객체가 없어도 실행된다는 특징 때문에 정적 메소드를 선언할 때는 이들 내부에 인스턴스 필드나 인스턴스 메소드를 사용할 수 없다. 또한 객체 자신의 참조인 this 키워드도 사용이 불가능하다. 정적 메소드에서 인스턴스 멤버를 사용하고 싶다면 객체를 먼저 생성하고 참조 변수로 접근해야 한다.

### 싱글톤

- 가끔 전체 프로그램에서 단 하나의 객체만 만들도록 보장해야 하는 경우가 있다. 단 하나만 생성된다고 해서 이 객체를 싱글톤이라 한다.
- 클래스 외부에서 new 연산자로 생성자를 호출할 수 없도록 막아야 한다. 생성자를 호출할 만큼 객체가 생성되기 때문이다. 생성자를 외부에서 호출할 수 없도록 하려면 생성자 앞에 private 접근 제한자를 붙여주면 된다.

### final 필드와 상수

- final 필드 : 초기값이 저장되면 이것이 최종적인 값이 되어서 프로그램 실행 도중에 수정할 수 없다는 것이다.
- final 타입 필드 [= 초기값];
- 필드 선언 시에 주는 방법, 생성자에서 주는 방법
- 상수 : final 필드는 한 번 초기화하면 수정할 수 없는 필드라고 했다. 하지만 상수라고 부르진 않는다. 불변의 값은 객체마다 저장할 필요가 없는 공용성을 띠고 있으며, 여러 가지 값으로 초기화될 수 없다.
- final 필드는 객체마다 저장되고, 생성자의 매개값을 통해서 여러 가지 값을 가질 수 있어 상수가 될 수 없다.
- 상수 이름은 모두 대문자로 작성하는 것이 관례이다. 서로 다른 단어가 혼합된 이름이라면 언더바(_)로 단어들을 연결해준다.

## 06-6 패키지와 접근 제한자

- 상위패키지.하위패키지.클래스

### 패키지 선언

- 클래스를 작성할 때 해당 클래스가 어떤 패키지에 속할 것인지를 선언하는 것
- 숫자로 시작해서는 안되고 _, $를 제외한 특수 문자를 사용해서는 안 된다.

- java로 시작하는 패키지는 자바 표준 API에서만 사용하므로 사용해서는 안된다.
- 모두 소문자로 작성하는 것이 관례이다.
- 이클립스에서 패키지 생성과 클래스 생성 : 프로젝트의 src 폴더 우클릭-[File]-[New]-[Package] / 해당 패키지 우클릭-[File]-[New]-[Class]
- 상하위 패키지를 계층적으로 보고 싶다면 Package Explorer 뷰의 오른쪽 상단에서 역삼각형 버튼을 클릭하고 [Package Presentation]-[Hierarchical]
- import문 : 패키지 선언과 클래스 선언 사이에 작성한다. 사용하고자 하는 클래스들이 동일한 패키지 소속이라면 개별 import문을 작성하는 것보다는 *를 이용해서 해당 패키지에 소속된 클래스들을 사용할 것임을 알려주는 것도 좋은 방법이다.
- [Window]-[Preferences] 메뉴를 선택한 후 [Preferences] 대화상자에서 [Java]-[CODE Style]-[Organize imports]를 선택한다. Number of imports needed for .*의 99를 1로 변경하고 [OK] 버튼을 클릭한다. Ctrl + Shift + O를 누른다.

### 접근 제한자

- 클래스 및 인터페이스 그리고 이들이 가지고 있는 멤버의 접근을 제한한다.
- public 접근 제한자 : 단어 뜻 그대로 외부 클래스가 자유롭게 사용할 수 있도록 한다.
- protected 접근 제한자 : 같은 패키지 또는 자식 클래스에서 사용할 수 있도록 한다.
- privated 접근 제한자 : 단어 뜻 그대로 개인적인 것이라 외부에서 사용될 수 없도록 한다.
- default 접근 제한자 : 위 세 가지 접근 제한자가 적용되지 않으면 접근 제한을 갖는다. 같은 패키지에 소속된 클래스에서만 사용할 수 있도록 한다.

### 클래스의 접근 제한

- default 접근 제한 : 클래스를 선언할 때 public을 생략했다면 클래스는 default 접근 제한을 갖는다. 클래스가 default 접근 제한을 가지면 같은 패키지에서는 아무런 제한 없이 사용할 수 있지만 다른 패키지에서는 사용할 수 없도록 제한된다.
- public 접근 제한 : 클래스를 선언할 때 public 접근 제한자를 붙였다면 클래스는 public 접근 제한을 가진다. 클래스가 public 접근 제한을 가지면, 같은 패키지뿐만 아니라 다른 패키지에서도 아무런 제한 없이 사용할 수 있다.

### 생성자의 접근 제한

- 객체를 생성하기 위해서는 new 연산자로 생성자를 호출한다. 하지만 생성자를 어디에서나 호출할 수 있는 것은 아니다. 생성자가 어떤 접근 제한을 갖느냐에 따라 호출 가능 여부가 결정된다.
- public 접근 제한 : 모든 패키지에서 아무런 제한 없이 생성자를 호출할 수 있도록 한다.
- protected 접근 제한 : default 접근 제한과 마찬가지로 같은 패키지에 속하는 클래스에서 생성자를 호출할 수 있도록 한다. 다른 패키지에 속한 클래스가 해당 클래스의 자식 클래스라면 생성자를 호출할 수 있다.
- default 접근 제한 : 같은 패키지에서는 아무런 제한 없이 생성자를 호출할 수 있으나, 다른 패키지에서는 생성자를 호출할 수 없도록 한다.
- private 접근 제한 : 동일한 패키지이건 다른 패키지이건 상관없이 생성자를 호출하지 못하도록 제한한다. 오로지 클래스 내부에서만 생성자를 호출할 수 있고 객체를 만들 수 있다.

### 필드와 메소드의 접근 제한

- public 접근 제한 : 모든 패키지에서 아무런 제한 없이 필드와 메소드를 사용할 수 있도록 해준다.
- protected 접근 제한 : default 접근 제한과 마찬가지로 같은 패키지에 속하는 클래스에서 필드와 메소드

를 사용할 수 있도록 한다. 다른 패키지에 속한 클래스가 해당 클래스의 자식 클래스라면 필드와 메소드를 사용할 수 있다.

- default 접근 제한 : 필드와 메소드를 선언할 때 접근 제한자를 생략하면 default 접근 제한을 가진다. 같은 패키지에서는 아무런 제한 없이 필드와 메소드를 사용할 수 있으나 다른 패키지에서는 필드와 메소드를 사용할 수 없도록 한다.
- private 접근 제한 : 동일한 패키지이건 다른 패키지이건 상관없이 필드와 메소드를 사용하지 못하도록 제한한다. 오로지 클래스 내부에서만 사용할 수 있다.

### Getter와 Setter 메소드

- 외부에서 마음대로 변경할 경우 객체의 무결성이 깨질 수 있다.
- Setter : 필드는 외부에서 접근할 수 없도록 막고 메소드는 공개해서 외부에서 메소드를 통해 필드에 접근하도록 유도한다. 메소드는 매개값을 검증해서 유효한 값만 객체의 필드로 저장할 수 있다.
- Getter : 외부에서 객체의 데이터를 읽을 때도 메소드를 사용하는 것이 좋다. 필드값을 직접 사용하면 부적절한 경우도 있다. 메소드로 필드값을 가공한 후 외부로 전달하면 된다.
- 자동 생성 : [Source]-[Generate Getters and Setters]

# Chapter 07. 상속

## 07-1 상속

- 상속은 이미 잘 개발된 클래스를 재사용해서 새로운 클래스를 만들기 때문에 중복되는 코드를 줄여준다.
- 상속을 이용하면 부모 클래스의 수정으로 모든 자식 클래스들도 수정되는 효과를 가져오기 때문에 유지 보수 시간을 최소화할 수도 있다.

### 클래스 상속

- 여러 개의 부모 클래스를 상속할 수 없다. 그러므로 다음과 같이 *extends* 뒤에는 단 하나의 부모 클래스만 와야 한다.
- 부모 클래스에서 *private* 접근 제한을 갖는 필드와 메소드는 상속 대상에서 제외된다. 그리고 부모 클래스와 자식 클래스가 다른 패키지에 존재한다면 *default* 접근 제한을 갖는 필드와 메소드도 상속 대상에서 제외한다.
- 자바는 다중 상속을 허용하지 않아 여러 개의 부모 클래스를 상속할 수 없다.

### 부모 생성자 호출

- 모든 객체는 클래스의 생성자를 호출해야만 생성되며, 부모 생성자는 자식 생성자의 맨 첫 줄에서 호출된다.

메소드 재정의

- 자식 클래스에서 부모 클래스의 메소드를 다시 정의하는 것을 말한다.
- 부모의 메소드와 동일한 시그니처(리턴 타입, 메소드 이름, 매개 변수 목록)를 가져야 한다.
- 접근 제한을 더 강하게 재정의할 수 없다.
- 새로운 예외(*Exception*)를 *throws*할 수 없다.
- 더 강하게 재정의할 수 없다는 것은 부모 메소드가 *public* 접근 제한을 가지고 있을 경우 재정의하는 자식 메소드는 *default*나 *private* 접근 제한으로 수정할 수 없다는 뜻이다. 하지만 반대는 가능하다.
- 자동 생성 : *[Source]-[Override/Implement Methods]*
- 부모 메소드 호출 : 자식 클래스에서 부모 클래스의 메소드를 재정의하게 되면, 부모 클래스의 메소드는 숨겨지고 재정의된 자식 메소드만 사용된다. 자식 클래스 내부에서 재정의된 부모 클래스의 메소드를 호출해야 하는 상황이 발생한다면 명시적으로 *super* 키워드를 붙여서 부모 메소드를 호출할 수 있다.

### *final* 클래스와 *final* 메소드

- 상속할 수 없는 *final* 클래스 : 클래스를 선언할 때 *final* 키워드를 *class* 앞에 붙이면 이 클래스는 최종적인 클래스이므로 상속할 수 없는 클래스가 된다. 즉, *final* 클래스는 부모 클래스가 될 수 없어 자식 클래스를 만들 수 없다는 것이다.
- 재정의할 수 없는 *final* 메소드 : 메소드를 선언할 때 *final* 키워드를 붙이면 이 메소드는 최종적인 메소드이므로 재정의할 수 없는 메소드가 된다. 즉, 부모 클래스를 상속해서 자식 클래스를 선언할 때 부모 클래스에 선언된 *final* 메소드는 자식 클래스에서 재정의할 수 없다는 것이다.

## 07-2 타입 변환과 다형성

- 다형성 : 사용 방법은 동일하지만 다양한 객체를 이용해서 다양한 실행 결과가 나오도록 하는 성질이다.

### 자동 타입 변환

- 프로그램 실행 도중에 자동적으로 타입 변환이 일어나는 것을 말한다. 자식의 부모의 특징과 기능을 상속받기 때문에 부모와 동일하게 취급될 수 있다는 것이다.
- 부모 타입으로 자동 타입 변환될 이후에는 부모 클래스에 선언된 필드와 메소드만 접근이 가능하다. 변수는 자식 객체를 참조하지만 변수로 접근 가능한 멤버는 부모 클래스 멤버로만 한정된다.

### 필드의 다형성

- 필드의 타입을 부모 타입으로 선언하면 다양한 자식 객체들이 저장될 수 있기 때문에 필드 사용 결과가 달라질 수 있다.
- 부모 클래스를 상속하는 자식 클래스는 부모가 가지고 있는 필드와 메소드를 가지고 있으니 사용 방법이 동일할 것이다.
- 자식 클래스는 부모의 메소드를 재정의해서 메소드의 실행 내용을 변경함으로써 더 우수한 실행결과가 나오게 할 수도 있다.
- 자식 타입을 부모 타입으로 변환할 수 있다.

### 매개 변수의 다형성

- 자동 타입 변환은 필드의 값을 대입할 때에도 발생하지만, 주로 메소드를 호출할 때 많이 발생한다.
- 메소드를 호출할 때에는 매개 변수의 타입과 동일한 매개값을 지정하는 것이 정석이지만, 매개값을 다양화하기 위해 매개 변수에 자식 객체를 지정할 수도 있다.
- 매개 변수의 다형성은 매개값으로 어떤 자식 객체가 제공되느냐에 따라 메소드의 실행 결과가 다양해질 수 있다는 것이다.

### 강제 타입 변환

- 부모 타입을 자식 타입으로 변환하는 것을 말한다. 모든 부모 타입을 자식 타입으로 강제 변환할 수 있는 것은 아니다.
- 자식 타입이 부모 타입으로 자동 타입 변환한 후 다시 자식 타입으로 변환할 때 강제 타입 변환을 사용할 수 있다.

### 객체 타입 변환

- 자식 타입이 부모 타입으로 변환되어 상태에서만 가능하기 때문에 처음부터 부모 타입으로 생성된 객체는 자식 타입으로 변환할 수 없다.
- *instanceof* 연산자 : 어떤 객체가 어떤 클래스의 인스턴스인지 확인하기 위해 사용한다. 주로 매개값의 타입을 조사할 때 사용된다.
- 메소드 내에서 강제 타입 변환이 필요할 경우 반드시 매개값이 어떤 객체인지 *instanceof* 연산자로 확인하고 안전하게 강제 타입 변환을 해야 한다.

## 07-3 추상 클래스

### 추상 클래스의 용도

- 공통된 필드와 메소드의 이름을 통일할 목적
- 실체 클래스를 작성할 때 시간 절약

### 추상 클래스 선언

- 추상 클래스를 선언할 때에는 클래스 선언에 *abstract* 키워드를 붙여야 한다. *abstract*를 붙이면 *new* 연산자를 이용해서 객체를 만들지 못하고 상속을 통해 자식 클래스만 만들 수 있다.
- 실체 클래스 생성 : 추상 클래스는 실체 클래스의 공통되는 필드와 메소드를 추출해서 만들었기 때문에 객체를 직접 생성해서 사용할 수 없다. 추상 클래스는 새로운 실체 클래스를 만들기 위해 부모 클래스로만 사용된다. 코드로 설명하면 추상 클래스는 *extends* 뒤에만 올 수 있는 클래스이다.

### 추상 메소드와 재정의

- 추상 클래스는 실체 클래스가 공통적으로 가져야 할 필드와 메소드들을 정의해놓은 추상적인 클래스로, 실체 클

래스의 멤버(필드, 메소드)를 통일하는 데 목적이 있다.

- 모든 실체들이 가지고 있는 메소드의 실행 내용이 동일하다면 추상 클래스에 메소드를 작성하는 것이 좋을 것이다. 하지만 메소드의 선언만 통일하고 실행 내용은 실체 클래스마다 달라야 하는 경우가 있다.
- [public | protected] abstract 리턴타입 메소드이름(매개변수, …);
- 추상 클래스 설계 시 하위 클래스가 반드시 실행 내용을 채우도록 강제하고 싶은 메소드가 있을 경우 해당 메소드를 추상 메소드로 선언한다.
- 자식 클래스는 반드시 추상 메소드를 재정의해서 실행 내용을 작성해야 하는데, 그렇지 않으면 컴파일 에러가 발생한다.

# Chapter 08. 인터페이스

## 08-1 인터페이스

- 인터페이스 : 개발 코드와 객체가 서로 통신하는 접점 역할을 한다. 개발 코드가 인터페이스의 메소드를 호출하면 인터페이스는 객체의 메소드를 호출시킨다.

### 인터페이스 선언

- '~java' 형태의 소스 파일로 작성되고 컴파일러(javac)를 통해 '~.class' 형태로 컴파일되기 때문에 물리적 형태는 클래스와 동일하나 소스를 작성할 때 선언하는 방법이 다르다.
- 인터페이스 선언은 interface 키워드를 사용한다. [public] interface 인터페이스이름{ … }
- 패키지 우클릭-[File]-[New]-[Interface]

상수 필드 선언

- 인터페이스는 객체 사용 방법을 정의한 것이므로 실행 시 데이터를 저장할 수 있는 인스턴스 또는 정적 필드를 선언할 수 없다. 그러나 상수 필드는 선언이 가능하다.
- 단, 상수는 인터페이스에 고정된 값으로 실행 시에 데이터를 바꿀 수 없다.
- [public static final] 타입 상수이름 = 값;
- 인터페이스는 데이터를 저장할 수 없기 때문에 상수 필드만 선언할 수 있다.

추상 메소드 선언

- 인터페이스를 통해 호출된 메소드는 최종적으로 객체에서 실행된다. 그렇기 때문에 인터페이스의 메소드는 실행 블록이 필요 없는 추상 메소드로 선언한다.
- [public abstract] 리턴타입 메소드이름(매개변수, …);

### 인터페이스 구현

- 객체는 인터페이스에서 정의된 추상 메소드와 동일한 메소드 이름, 매개 타입, 리턴 타입을 가진 실체 메소드를 가지고 있어야 한다. 이러한 객체를 인터페이스의 구현 객체라고 하고, 구현 객체를 생성하는 클래스를 구현 클래스라고 한다.

구현 클래스

- 보통의 클래스와 동일한데, 인터페이스 타입으로 사용할 수 있음을 알려주기 위해 클래스 선언부에 implements 키워드를 추가하고 인터페이스 이름을 명시해야 한다. 그리고 인터페이스에 선언된 추상 메소드의 실체 메소드를 선언해야 한다.
- public class 구현클래스이름 implements 인터페이스 이름{ }

다중 인터페이스 구현 클래스

- 인터페이스A와 인터페이스B가 객체의 메소드를 호출할 수 있으려면 객체는 이 두 인터페이스를 모두 구현해야 한다.
- public class 구현클래스이름 implements 인터페이스A, 인터페이스B { }
- 구현 클래스는 모든 인터페이스의 추상 메소드에 대해 실체 메소드를 작성해야 한다.

### 인터페이스 사용

- 클래스를 선언할 때 인터페이스는 필드, 생성자 또는 메소드의 매개 변수, 생성자 또는 메소드의 로컬 변수로 선언될 수 있다.
- 인터페이스가 필드 타입으로 사용될 경우, 필드에 구현 객체를 대입할 수 있다.
- 인터페이스가 생성자의 매개 변수 타입으로 사용될 경우, new 연산자로 객체를 생성할 때 구현 객체를 생성자의 매개값으로 대입할 수 있다.

- 인터페이스가 로컬 변수 타입으로 사용될 경우, 변수에 구현 객체를 대입할 수 있다.
- 인터페이스가 메소드의 매개 변수 타입으로 사용될 경우, 메소드 호출 시 구현 객체를 매개값으로 대입할 수 있다.

## 08-2 타입 변환과 다형성

- 인터페이스의 다형성 : 프로그램 소스 코드는 변함이 없는데, 구현 객체를 교체함으로써 프로그램의 실행 결과가 다양해진다.

### 자동 타입 변환

- 구현 객체가 인터페이스 타입으로 변환되는 것을 말한다.
- 인터페이스 구현 클래스를 상속해서 자식 클래스를 만들었다면 자식 객체 역시 인터페이스 타입으로 자동 타입 변환할 수 있다.
- 필드의 다형성과 매개 변수의 다형성을 구현할 수 있다.

### 필드의 다형성

- 메소드를 수정하지 않아도 다양한 메소드의 실행 결과를 얻을 수 있게 되는 것이다.

### 매개 변수의 다형성

- 매개 변수를 인터페이스 타입으로 선언하고 호출할 때에는 구현 객체를 대입한다.
- 매개 변수의 타입이 인터페이스일 경우 어떠한 구현 객체도 매개값으로 사용할 수 있고, 어떤 구현 객체가 제공되느냐에 따라 메소드의 실행 결과는 다양해질 수 있다.

### 강제 타입 변환

- 구현 객체가 인터페이스 타입으로 자동 타입 변환하면, 인터페이스에 선언된 메소드만 사용 가능하다는 제약 사항이 따른다.
- 구현 클래스에 선언된 필드와 메소드를 사용해야 할 경우도 발생한다. 이때 강제 타입 변환을 해서 다시 구현 클래스 타입으로 변환한 다음, 구현 클래스의 필드와 메소드를 사용할 수 있다.

객체 타입 확인

- 구현 객체가 인터페이스 타입으로 변환되어 있는 상태에서 가능하다. 그러나 어떤 구현 객체가 변환되어 있는지 알 수 없는 상태에서 무작정 강제 타입 변환할 경우 *ClassCastException*이 발생할 수도 있다.
- 인터페이스 타입으로 자동 타입 변환된 매개값을 메소드 내에서 다시 구현 클래스 타입으로 강제 타입 변환해야 한다면 반드시 매개값이 어떤 객체인지 *instanceof* 연산자로 확인하고 안전하게 강제 타입 변환을 해야 한다.

인터페이스 상속

- 인터페이스도 다른 인터페이스를 상속할 수 있다. 클래스와는 달리 다중 상속을 허용한다.
- *public interface* 하위인터페이스 *extends* 상위인터페이스1, 상위인터페이스2 { … }
- 하위 인터페이스를 구현하는 클래스는 하위 인터페이스의 메소드뿐만 아니라 상위 인터페이스의 모든 추상 메소드에 대한 실체 메소드를 가지고 있어야 한다. 구현 클래스로부터 객체를 생성한 후에 하위 및 상위 인터페이스 타입으로 변환이 가능하다.

# Chapter 09. 중첩 클래스와 중첩 인터페이스

## 09-1 중첩 클래스와 중첩 인터페이스 소개

- 중첩클래스 : 클래스 내부에 선언한 클래스

### 중첩 클래스

- 멤버 클래스 : 클래스 멤버로서 선언되는 중첩 클래스, 클래스나 객체가 사용 중이라면 언제든지 재사용이 가능하다.
- 로컬 클래스 : 생성자 또는 메소드 내부에서 선언되는 중첩 클래스, 메소드를 실행할 때만 사용되고 메소드가 종료되면 없어진다.

| 선언 위치에 따른 분류 | | 선언 위치 | 설명 |
|---|---|---|---|
| 멤버 클래스 | 인스턴스 멤버 클래스 | class A {<br>  class B { ... }<br>} | A 객체를 생성해야만 사용할 수 있는 B 클래스 |
| | 정적 멤버 클래스 | class A {<br>  static class B { ... }<br>} | A 클래스로 바로 접근할 수 있는 B 클래스 |
| 로컬 클래스 | | class A {<br>  void method() {<br>  class B { ... }<br>  }<br>} | method()가 실행할 때만 사용할 수 있는 B 클래스 |

인스턴스 멤버 클래스

- static 키워드 없이 중첩 선언된 클래스, 인스턴스 멤버 클래스는 인스턴스 필드와 메소드만 선언이 가능하고 정적 필드와 메소드는 선언할 수 없다.

정적 멤버 클래스

- static 키워드로 선언된 클래스, 모든 종류의 필드와 메소드를 선언할 수 있다.

로컬 클래스

- 중첩 클래스는 메소드 내에서도 선언할 수 있는데 로컬 클래스라고 한다.
- 접근 제한자(public, private) 및 static을 붙일 수 없다. 메소드 내부에서만 사용되므로 접근을 제한할 필요가 없기 때문이다.

### 중첩 클래스의 접근 제한

- 바깥 필드와 메소드에서 사용 제한
- 멤버 클래스에서 사용 제한 : 멤버 클래스가 인스턴스 또는 정적으로 선언됨에 따라 멤버 클래스 내부에서 바깥 클래스의필드와 메소드에 접근할 때에도 제한이 따른다.
- 로컬 클래스에서 사용 제한 : 로컬 클래스의 객체는 메소드 실행이 종료되면 없어지는 것이 일반적이지만, 메소드가 종료되어도 계속 실행 상태로 존재할 수 있다.
- 중첩 클래스에서 바깥 클래스 참조 얻기 : 중첩 클래스에서 this 키워드를 사용하면 바깥 클래스의 객체 참조가 아니라, 중첩 클래스의 객체 참조가 된다. 중첩 클래스 내부에서 'this.필드, this.메소드()'로 호출하면 중첩 클래스의 필드와 메소드가 사용된다.

중첩 인터페이스

- 중첩 인터페이스는 클래스의 멤버로 선언된 인터페이스이다.
- 해당 클래스와 긴밀한 관계를 맺는 구현 클래스를 만들기 위해서 클래스 내부에 선언한다.

- 인스턴스 멤버 인터페이스와 정적 멤버 인터페이스 모두 가능하다. 인스턴스 멤버 인터페이스는 바깥 클래스의 객체가 있어야 사용 가능하고, 정적 멤버 인터페이스는 바깥 클래스의 객체 없이 바깥 클래스만으로 바로 접근할 수 있다.

## 09-2 익명 객체

- 익명 객체는 이름이 없는 개체. 어떤 클래스를 상속하거나 인터페이스를 구현해야만 익명 객체를 만들 수 있다.

### 익명 자식 객체 생성

- 부모 타입의 필드 또는 변수를 선언하고 자식 객체를 초기값으로 대입하는 경우, 우선 부모 클래스를 상속해서 자식 클래스를 선언한다. 그리고 new 연산자를 이용해서 자식 객체를 생성한 후 부모 타입의 필드 또는 변수에 대입하는 것이 일반적이다.
- 자식 클래스를 명시적으로 선언하는 이유는 어디서건 이미 선언된 자식 클래스로 간단히 객체를 생성해서 사용할 수 있기 때문에 재사용성이 높다고 한다.
- 부모클래스 [필드|변수] = new 부모 클래스(매개값, ...) { };

### 익명 구현 객체 생성

- 인터페이스 타입의 필드 또는 변수를 선언하고, 구현 객체를 초기값으로 대입하는 경우를 생각해보면, 우선 구현 클래스를 선언한다. 그리고 new 연산자를 이용해서 구현 객체를 생성한 후 인터페이스 타입의 필드 또는 로컬 변수에 대입하는 것이 일반적이다.
- 구현 클래스가 재사용 되지 않고 오로지 특정 위치에서 사용할 경우라면 구현 클래스를 명시적으로 선언하는 것은 귀찮은 작업이 된다.
- 인터페이스 [필드|변수] = new 인터페이스() { };

### 익명 객체의 로컬 변수 사용

- 익명 객체는 메소드 실행이 종료되면 없어지는 것이 일반적이지만, 메소드가 종료되어도 계속 실행 상태로 존재할 수 있다.
- 매개 변수나 로컬 변수는 메소드 실행이 끝나면 스택 메모리에서 사라지기 때문에 익명 객체에서 지속적으로 사용할 수 없다.

# Chapter 10. 예외 처리

## 10-1 예외 클래스

- 예외(exception) : 사용자의 잘못된 조작 또는 개발자의 잘못된 코딩으로 인해 발생하는 프로그램 오류를 말한다. 예외 처리를 통해 프로그램을 종료하지 않고 정상 실행 상태가 유지되도록 할 수 있다.
- 예외가 발생할 가능성이 높은 코드를 컴파일할 때 예외 처리 유무를 확인한다.

### 예외와 예외 클래스

- 일반 예외 : 컴파일러 체크 예외, 프로그램 실행 시 예외가 발생할 가능성이 높기 때문에 자바 소스를 컴파일 하는 과정에서 해당 예외 처리 코드가 있는지 검사한다.
- 실행 예외 : 컴파일러 넌 체크 예외, 실행 시 예측할 수 없이 갑자기 발생하기 때문에 컴파일하는 과정에서 예외 처리 코드가 있는지 검사하지 않는다.
- JVM은 프로그램을 실행하는 도중에 예외가 발생하면 해당 예외 클래스로 객체를 생성한다. 그리고나서 예외 처리 코드에서 예외 객체를 이용할 수 있도록 해준다.
- 모든 클래스는 java.lang.Exception 클래스를 상속 받는다.
- 일반 예외와 실행 예외 클래스는 RuntimeException 클래스를 기준으로 구별한다. RuntimeExceptioin의 하위 클래스가 아니면 일반 예외 클래스이고, 하위 클래스이면 실행 예외 클래스이다.
- 클래스 상속 관계에서 부모(조상)에 RuntimeException이 있다면 실행 예외 클래스이다.

### 실행 예외

- NullPointerException : 객체 참조가 없는 상태, 즉 null 값을 갖는 참조 변수로 객체 접근 연산자인 도트(.)를 사용했을 때 발생한다. 객체가 없는 상태에서 객체를 사용하려 했으니 예외가 발생하는 것이다.
- ArrayIndexOutOfBoundsException : 배열에서 인덱스 범위를 초과할 경우 실행 예외이다.
- NumberFormatException : 프로그램을 개발하다 보면 문자열로 되어 있는 데이터를 숫자로 변경하는 경우가 자주 발생한다.

| 리턴 타입 | 메소드 이름(매개 변수) | 설명 |
|---|---|---|
| int | Integer.parseInt(String s) | 주어진 문자열을 정수로 변환해서 리턴 |
| double | Double.parseDouble(String s) | 주어진 문자열을 실수로 변환해서 리턴 |

- ClassCastException: 타입 변환은 상위 클래스와 하위 클래스 간에 발생하고 구현 클래스와 인터페이스 간에도 발샌한다.

## 10-2 예외 처리

- 자바 컴파일러는 소스 파일을 컴파일할 때 일반 예외가 발생할 가능성이 있는 코드를 발견하면 컴파일 에러를 발생시켜 개발자가 강제적으로 예외 처리 코드를 작성하도록 요구하나 실행 예외는 컴파일러가 체크해 주지 않기 때문에 개발자의 경험을 바탕으로 예외 처리 코드를 작성해야 한다.

### 예외 처리 코드

- try-catch-finally 블록 : 생성자 내부와 메소드 내부에서 작성되어 일반 예외와 실행 예외가 발생할 경우 예외 처리를 할 수 있도록 해준다
- try 블록에는 예외 발생 가능 코드가 위치한다.
- try 블록의 코드가 예외 발생 없이 정상 실행되면 catch 블록의 코드는 실행되지 않고 finally 블록의 코드를 실행한다. 만약 try 블록의 코드에서 예외가 발생하면 즉시 실행을 멈추고 catch 블록으로 이동해 예외 처리 코드를 실행한다. 그리고 finally 블록의 코드를 실행한다.

- finally 블록은 생략 가능하다. 예외 발생 여부와 상관없이 항상 실행할 내용이 있을 경우에만 finally 블록을 작성해주면 된다. 심지어 try 블록과 catch 블록에서 return문을 사용하더라도 finally 블록은 항상 실행된다.

**예외 종류에 따른 처리 코드**

- 다중 catch : catch 블록의 예외 클래스 타입은 try 블록에서 발생된 예외의 종류를 말하는데, try 블록에서 해당 타입의 예외가 발생하면 catch 블록을 실행하도록 되어 있다.
- catch 순서 : 상위 예외 클래스가 하위 예외 클래스보다 아래쪽에 위치해야 한다는 것이다. try 블록에서 예외가 발생했을 때, 예외를 처리해줄 catch 블록은 위에서부터 차례대로 검색된다.

**예외 떠넘기기**

- throws 키워드는 메소드 선언부 끝에 작성되어 메소드에서 처리하지 않은 예외를 호출한 곳으로 떠넘기는 역할을 한다. 키워드 뒤에는 떠넘길 예외 클래스를 쉼표로 구분해서 나열해주면 된다.
- main() 메소드에서 throws Exception을 붙이는 것은 좋지 못한 예외 처리 방법이다. 프로그램 사용자는 프로그램이 알 수 없는 예외 내용을 출력하고 종료되는 것을 좋아하지 않는다. 따라서 main()에서 try-catch 블록으로 예외를 최종 처리하는 것이 바람직하다.

# Chapter 11. 기본 API 클래스

## 11-1 java.lang 패키지

| 클래스 | | 용도 |
|---|---|---|
| Object | | - 자바 클래스의 최상위 클래스로 사용 |
| System | | - 표준 입력 장치(키보드)로부터 데이터를 입력받을 때 사용<br>- 표준 출력 장치(모니터)로 출력하기 위해 사용<br>- 자바 가상 기계를 종료할 때 사용<br>- 쓰레기 수집기를 실행 요청할 때 사용 |
| Class | | - 클래스를 메모리로 로딩할 때 사용 |
| String | | - 문자열을 저장하고 여러 가지 정보를 얻을 때 사용 |
| Wrapper | Byte, Short, Character<br>Integer, Float, Double<br>Boolean, Long | - 기본 타입의 데이터를 갖는 객체를 만들 때 사용<br>- 문자열을 기본 타입으로 변환할 때 사용<br>- 입력값 검사에 사용 |
| Math | | - 수학 함수를 이용할 때 사용 |

### 자바 API 도큐먼트

- API는 라이브러리라고 부르기도 하는데, 프로그램 개발에 자주 사용되는 클래스 및 인터페이스 모음을 말한다.
- 방대한 자바 표준API 중에서 우리가 원하는 API를 쉽게 찾아 이용할 수 있도록 도와주는 API 도큐먼트가 있다. API는 도큐먼트는 HTML 페이지로 작성되어 있고, https://docs.oracle.com/en/java/javase/index.html 방문하면 버전별로 볼 수 있다.

| JDK 8 API 도큐먼트 | JDK 11 이후 버전 API 도큐먼트 |
|---|---|
| 1. 왼쪽 상단 Packages 몯록에서 java.lang 패키지 링크를 찾아 클릭<br>2. 왼쪽 하단에 java.lang 패키지의 내용이 나옴<br>3. Classes 목록에서 String 클래스 링크를 찾아 클릭<br>4. 오른쪽에 Class String 페이지가 나옴 | 방법1<br>1. All Modules 목록에서 java.base 링크를 찾아 클릭<br>2. java.base 모듈 페이지의 Packages 목록에서 java.lang 패키지 링크를 찾아 클릭<br>3. java.lang 패키지 페이지의 Class Summary 목록에서 String 클래스 링크를 찾아 클릭<br><br>방법2<br>1. 오른쪽 상단의 Search 검색란에 “String”을 입력<br>2. 드롭다운 목록에서 java.lang.String 항목을 선택 |

### API 도큐먼트에서 클래스 페이지 읽는 방법

- 최상단의 SUMMARY: NESTED | FIELD | CONSTR | METHOD를 본다.
- 클래스의 선언부를 본다. 클래스가 final 또는 abstract 키워드가 있는지 확인한다.
- 클래스에 선언된 필드 목록을 본다.
- 클래스에 선언된 생성자 목록을 본다.
- 클래스에 선언된 메소드 목록을 본다.

### Object 클래스

- 클래스를 선언할 때 extends 키워드로 다른 클래스를 상속하지 않더라도 암시적으로 java.lang.Object 클래스를 상속하게 된다. 자바의 모든 클래스는 Object 클래스의 자식이거나 자손 클래스이다
- Obejct는 자바의 최상위 부모 클래스에 해당한다.
- 객체 비교(equals()) : 메소드의 매개 타입은 Object인데, 모든 객체가 매개값으로 대입될 수 있음을 말한다. 모든 객체는 Obejct 타입으로 자동 타입 변환도리 수 있다.

- equals() 메소드를 재정의할 때에는 매개값(비교 객체)이 기준 객체와 동일한 타입의 객체인지 먼저 확인해야 한다. Object 타입의 매개 변수는 모든 객체가 매개값으로 제공될 수 있기에 instanceof 연산자로 기준 객체와 동일한 타입인지 제일 먼저 확인해야 한다.

- 객체 해시코드(hashCode()) : 객체를 식별하는 하나의 정수값을 말한다. Object 클래스의 hashCode() 메소드는 객체의 메모리 번지를 이용해서 해시코드를 만들어 리턴하기 때문에 객체마다 다른 값을 가지고 있다.

- 객체 문자 정보(toString()) : 객체 문자 정보를 리턴하고, 객체를 문자열로 표현한 값이다. 메소드는 '클래스 이름@16진수해시코드' 로 구성된 문자 정보를 리턴한다.

### System 클래스

- 프로그램 종료(exit()) : exit() 메소드는 현재 실행하고 있는 프로세스를 강제 종료시키는 역할을 한다. int 매개값을 지정하도록 되어 있는데 이 값을 종료 상태값이라고 한다.

- 현재 시각 읽기(currentTimeMillis(), nanoTime()) : 컴퓨터의 시계로부터 현재 시간을 읽어서 밀리세컨드(1/1000초) 단위와 나노세컨드(1/10의 9승초) 단위의 long 값을 리턴한다.

### Class 클래스

- Class 객체 얻기(getClass(), forName()) : 1번 Class clazz = 클래스이름.class / 2번 Class clazz = Class.forName( "패키지..클래스이름" ) / 3번 Class clazz = 참조변수.getClaa();

1, 2번 방법은 객체 없이 클래스 이름만 가지고 Class 객체를 얻는 방법이고, 3번 방법은 클래스로부터 객체가 이미 생성되어 있을 경우에 사용하는 방법이다.

- 클래스 경로를 활용해서 리소스 절대 경로 얻기 : Class 객체는 해당 클래스의 파일 경로 정보를 가지고 있기 때문에 이 경로를 활용해서 다른 리소스 파일의 경로를 얻을 수 있다.

### String 클래스

- String 생성자 : 소스상에서 문자열 리터럴은 String 객체로 자동 생성되지만, String 클래스의 다양한 생성자를 이용해서 직접 String 객체를 생성할 수도 있다. 어떤 생성자를 이용해서 String 객체를 생성할지는 제공되는 매개값의 타입에 달려 있다.

- String 메소드

| 리턴 타입 | 메소드 이름(매개 변수) | 설명 |
|---|---|---|
| char | charAt(int index) | 특정 위치의 문자를 리턴한다. |
| boolean | equals(Object anObject) | 두 문자열을 비교한다. |
| byte[] | getBytes() | byte[]로 리턴한다. |
| | getBytes(Charset charset) | 주어진 문자셋으로 인코딩한 byte[]로 리턴한다. |
| int | indexOf(String str) | 문자열 내에서 주어진 문자열의 위치를 리턴한다. |
| | length() | 총 문자의 수를 리턴한다. |
| String | replace(CharSequence target, CharSequence replacement) | target 부분을 replacement로 대치한 새로운 문자열을 리턴한다. |
| | substring(int beginIndex) | beginIndex 위치에서 끝까지 잘라낸 새로운 문자열을 리턴한다. |
| | substring(int beginIndex, int endIndex) | beginIndex 위치에서 endIndex 전까지 잘라낸 새로운 문자열을 리턴한다. |
| | toLowerCase() | 알파벳 소문자로 변환한 새로운 문자열을 리턴한다. |
| | toUpperCase() | 알파벳 대문자로 변환한 새로운 문자열을 리턴한다. |
| | trim() | 앞뒤 공백을 제거한 새로운 문자열을 리턴한다. |
| | valueOf(int i)<br>valueOf(double d) | 기본 타입 값을 문자열로 리턴한다. |

- 문자열 추출(charAt())

- 문자열 비교(equals())
- 바이트 배열로 변환(getBytes())
- 문자열 찾기(indexOf())
- 문자열 길이(length())
- 문자열 대치(replace())
- 문자열 잘라내기(substring())
- 알파벳 소, 대문자 변경(toLowerCase(), toUpperCase())
- 문자열 앞뒤 공백 잘라내기(trim())
- 문자열 변환(valueOf())

Wrapper(포장) 클래스

- 박싱(Boxing)과 언박싱(Unboxing)

| 기본 타입의 값을 줄 경우 | 문자열을 줄 경우 |
|---|---|
| Byte obj = new Byte(10); | Byte obj = new Byte("10"); |
| Character obj = new Character('가'); | 없음 |
| Short obj = new Short(100); | Short obj = new Short("100"); |
| Integer obj = new Integer(1000); | Integer obj = new Integer("1000"); |
| Long obj = new Long(10000); | Long obj = new Long("10000"); |
| Float obj = new Float(2.5F); | Float obj = new Float("2.5F"); |
| Double obj = new Double(3.5); | Double obj = new Double("3.5"); |
| Boolean obj = new Boolean(true); | Boolean obj = new Boolean("true"); |

- 자동 박싱과 언박싱 : 기본 타입 값을 직접 박싱, 언박싱 하지 않아도 자동적으로 박싱과 언박싱이 일어나는 경우가 있다. 자동 박싱은 포장 클래스 타입에 기본값이 대입될 경우에 발생한다. 자동 언박싱은 기본 타입에 포장 객체가 대입되는 경우와 연산에서 발생한다.
- 문자열을 기본 타입 값으로 변환 : 포장 클래스의 주요 용도는 기본 타입의 값을 박싱해서 포장 객체로 만드는 것이지만, 문자열을 기본 타입 값으로 변환할 때에도 많이 사용된다.
- 포장 값 비교 : 포장 객체는 내부의 값을 비교하기 위해 ==와 != 연산자를 사용하지 않는 것이 좋다. 이 연산자는 내부의 값을 비교하는 것이 아니라 포장 객체의 참조를 비교하기 때문이다.

| 타입 | 값의 범위 |
|---|---|
| boolean | true, false |
| char | \u0000 ~ \u007f |
| byte, short, int | -128 ~ 127 |

Math 클래스

| 메소드 | 설명 |
|---|---|
| int abs(int a)<br>double abs(double a) | 절대값 |
| double ceil(double a) | 올림값 |
| double floor(double a) | 버림값 |
| int max(int a, int b)<br>double max(double a, double b) | 최대값 |
| int min(int a, int b)<br>double min(double a, double b) | 최소값 |
| double random() | 랜덤값 |
| double rint(double a) | 가까운 정수의 실수값 |
| long round(double a) | 반올림값 |

## 11-2 java.util 패키지

| 클래스 | 용도 |
|---|---|
| Date | 날짜와 시간 정보를 저장하는 클래스 |
| Calendar | 운영체제의 날짜와 시간을 얻을 때 사용 |

- Date 클래스 : 특정 시점의 날짜를 표현하는 클래스, Date 객체 안에는 특정 시점의 연도, 월, 일, 시간 정보가 저장된다.

- Calendar 클래스 : 달력을 표현한 클래스, 해당 운영체제의 Calendar 객체를 얻으면, 연도, 월, 일, 요일, 오전/오후, 시간 등의 정보를 얻을 수 있다.

### Date 클래스

- 객체 간에 날짜 정보를 주고받을 때 매개 변수나 리턴 타입으로 주로 사용된다.

- Date now = new Date();

- 객체의 toString() 메소드는 영문으로 된 날짜를 리턴하기 때문에 원하는 날짜 형식의 문자열을 원하고 싶다면 java.text 패키지의 SimpleDateFormat 클래스와 함께 사용하는 것이 좋다.

- SimpleDateFormat sdf = new SimpleDateFormat( "yyyy년 MM월 dd일 hh시 mm분 ss초" );

- SimpleDateFormat 객체를 얻었다면, format() 메소드를 호출해서 원하는 형식의 날짜 정보를 얻을 수 있다. format() 메소드의 매개값은 Date 객체이다.

- String strNow = sdf.format(now);

Calendar 클래스

- 추상 클래스이므로 new연산자를 사용해서 인스턴스를 생성할 수 없다.

- Calendar 클래스의 정적 메소드인 getInstance() 메소드를 이용하면 현재 운영체제에 설정되어 있는 시간대를 기준으로 한 Calendar 하위 객체를 얻을 수 있다.

- Calendar now = Calendar.getInstance();

| get() 메소드 | 설명 |
|---|---|
| int year = now.get(Calendar.YEAR); | 연도를 리턴 |
| int month = new.get(Calendar.MONTH) + 1; | 월을 리턴 |
| int day = now.get(Calendar.DAT_OF_MONTH); | 일을 리턴 |
| String week = now.get(Calendar.DAY_OF_WEEK); | 요일을 리턴 |
| String amPm = now.get(Calendar.AM_PM); | 오전/오후를 리턴 |
| int hour = now.get(Calendar.HOUR); | 시를 리턴 |
| int minute = now.get(Calendar.MINUTE); | 분을 리턴 |
| int second = now.get(Calendar.SECOND); | 초를 리턴 |

# Chapter 12. 스레드

## 12-1 멀티 스레드

- 프로세스 : 운영체제에서 실행 중인 하나의 애플리케이션, 사용자가 애플리케이션을 실행하면 운영체제로부터 실행에 필요한 메모리를 할당받아 애플리케이션의 코드를 실행하는 것

### 스레드

- 두 가지 이상의 작업을 동시에 처리하는 멀티 태스킹을 할 수 있도록 CPU 및 메모리 자원을 프로세스마다 적절히 할당해주고, 병렬로 실행시킨다.
- 멀티 프로스세는 자신의 메모리를 가지고 실행하므로 서로 독립적이지만, 멀티 스레드는 하나의 프로세스 내부에 생성되므로 스레드 하나가 예외를 발생시키면 다른 스레드도 영향을 받는다.
- 스레드는 한 가지 작업을 실행하기 위해 순차적으로 실행할 코드를 실처럼 이어놓았다고 해서 유래된 이름이다.

### 메인 스레드

- main() 메소드의 첫 코드부터 아래로 순차적으로 실행하고, main() 메소드의 마지막 코드를 실행하거나 return문을 만나면 실행이 종료된다.

```
public static void main(String[] args) {
        String data = null;
        if(…) {
        }
        while(…) {
        }
        System.out.println( "…" );
}
```

- 필요에 따라 작업 스레드들을 만들어서 병렬로 코드를 실행할 수 있다. 즉, 멀티 스레드를 생성해서 멀티 태스킹을 수행한다.
- 싱글 스레드 애플리케이션에서는 메인 스레드가 조료하면 프로세스도 종료되지만 멀티 스레드 애플리케이션에서는 실행 중인 스레드가 하나라도 있다면, 프로세스는 종료되지 않는다.

### 작업 스레드 생성과 실행

- 어떤 자바 애플레케이션이건 메인 스레드는 반드시 존재하기 때문에 메인 작업 이외에 추가적인 병렬 작업의 수만큼 스레드를 생성하면 된다.
- Thread 클래스로부터 직접 생성 : Thread thread = new Thread(Runnable target);
- Runnable은 작업 스레드가 실행할 수 있는 코드를 가지고 있는 객체라고 해서 붙여진 이름이다. 인터페이스 타입이기 때문에 구현 객체를 만들어 대입해야 한다.
- Thread 하위 클래스로부터 생성 : Thread 클래스를 상속한 후 run() 메소드를 재정의해서 스레드가 실행할 코드를 작성하면 된다. 작업 스레드 클래스로부터 작업 스레드 객체를 생성하는 방법은 일반적인 객체를 생성하는 방법과 동일하다.
- 스레드 이름 : 자신의 이름을 가지고 있다. 큰 역할을 하는 것은 아니지만, 디버깅할 때 어떤 스레드가 어떤 작업을 하는지 조사할 목적으로 가끔 사용된다.

메인 스레드는 'main' 이라는 이름을 가지고 있고, 우리가 직접 생성한 스레드는 자동적으로 'Thread-n' 이라는 이름으로 설정된다. n은 스레드의 번호를 말하는데, Thread-n 대신 다른 이름으로 설정하고 싶다면 Thread

클래스의 setName() 메소드로 변경하면 된다. 반대로 스레드 이름을 알고 싶을 경우에는 getName() 메소드를 호출하면 된다.

### 동기화 메소드

- 공유 객체를 사용할 때의 주의할 점 : 멀티 스레드 프로그램에서 스레드들이 객체를 공유해서 작업해야 하는 경우, 스레드 A가 사용하던 객체를 스레드 B가 상태를 변경할 수 있기 때문에 스레드 A가 의도했던 것과는 다른 결과를 산출할 수도 있다.
- 동기화 메소드 : 스레드가 사용 중인 객체를 다른 스레드가 변경할 수 없게 하려면 스레드 작업이 끝날 때까지 객체에 잠금을 걸어서 다른 스레드가 사용할 수 없도록 해야 한다. 멀티 스레드 프로그램에서 단 하나의 스레드만 실행할 수 있는 코드 영역을 임계 영역이라고 한다.

자바는 임계 영역을 지정하기 위해 동기화 메소드를 제공한다. 스레드가 객체 내부의 동기화 메소드를 실행하면 즉시 객체에 잠금을 걸어 다른 스레드가 동기화 메소드를 실행하지 못하도록 한다.

## 12-2 스레드 제어

- 스레드 객체를 생성하고 start() 메소드를 호출하면 바로 실행되는 것이 아니라 실행 대기 상태가 된다.
- 실행 대기 상태 : 언제든지 실행할 준비가 되어 있는 상태, run() 메소드는 모두 실행하기 전에 다시 실행 대기 상태로 돌아갈 수 있으며, 실행 대기 상태에 있는 다른 스레드가 선택되어 실행 상태가 되기도 한다.
- 실행 상태에서 run() 메소드의 내용이 실행되면 스레드의 실행이 멈추고 종료 상태가 된다.
- 스레드 객체 상태(NEW) - start() -> 실행 대기(RUNNABLE) <- 반복 -> 실행 -> 종료(TERMINATED)

### 스레드 상태

- 스레드 객체를 생성하고 start() 메소드를 호출하면 곧바로 스레드 실행되는 것처럼 보이지만 사실은 실행 대기 상태가 된다. 실행 상태에 있는 스레드 중에서 운영체제는 하나의 스레드를 선택하고 CPU(코어)가 run() 메소드를 실행하도록 한다. 이때를 실행 상태라고 한다.
- 스레드는 실행 대기 상태와 실행 상태를 번갈아가면서 자신의 run() 메소드를 조금씩 실행한다. 실행 상태에서 run() 메소드가 종료되면, 더 이상 실행할 코드가 없기 때문에 스레드의 실행은 멈추게 된다. 이 상태를 종료 상태라 한다.
- 실행대기(RUNNABLE) -> 실행 -> 일시 정지 -> 실행대기(RUNNABLE)

### 스레드 상태 제어

- 주어진 시간 동안 일시 정지 시키는 sleep() 메소드와 스레드를 안전하게 종료시키는 stop 플래그, interrupt() 메소드를 사용한다.
- interrupt() : 일시 정지 상태의 스레드에서 InterruptedException을 발생시켜, 예외 처리 코드(catch)에서 실행 대기 상태로 가거나 종료 상태로 갈 수 있도록 한다.
- sleep(long milis) : 주어진 시간 동안 스레드를 일시 정지 상태로 만든다. 주어진 시간이 지나면 자동적으로 실행 대기 상태가 된다.
- 주어진 시간 동안 일시 정지 : 실행 중인 스레드를 일정 시간 멈추게 하고 싶다면 Thread 클래스의 정적 메소드인 sleep()을 사용하면 된다. Thread.sleep() 메소드를 호출한 스레드는 주어진 시간 동안 일시 정지 상태가 되고, 다시 실행 대기 상태로 돌아간다.
- 스레드의 안전한 종료 : 스레드는 자신의 run() 메소드가 모두 실행되면 자동적으로 종료된다. 하지만 경우에 따라서는 실행 중인 스레드를 즉시 종료해야 할 때가 있다.

Thread는 스레드를 즉시 종료하기 위해서 stop() 메소드를 제공하고 있는데, 이 메소드는 deprecated(중요도가

떨어져 이제 사용되지 않음) 되었습니다. stop() 메소드로 스레드를 갑자기 종료하게 되면 스레드가 사용 중이던 자원들이 불안전한 상태로 남겨지기 때문이다.

- stop 플래그를 이용하는 방법 : 스레드는 run() 메소드가 끝나면 자동적으로 종료되므로, run() 메소드가 정상적으로 종료되도록 유도하는 것이 중요하다.

- interrupt() 메소드를 이용하는 방법 : interrupt() 메소드는 스레드가 일시 정지 상태에 있을 때 InterruptedException을 발생시키는 역할을 한다. 이를 이용하면 run() 메소드를 정상 종료할 수 있다.

- 주목할 점은 스레드가 실행 대기 또는 실행 상태에 있을 때 interrupt() 메소드가 실행되면 즉시 InterruptedException이 발생하지 않고, 스레드가 미래에 일시 정지 상태가 되면 InterruptedException이 발생한다.

- 일시 정지를 만들지 않고도 interrupt()의 호출 여부를 알 수 있는 방법이 있다. interrupt() 메소드가 호출되었다면 스레드의 interrupted()와 isInterrupted() 메소드는 true를 리턴한다.

* 데몬 스레드 : 주 스레드의 작업을 돕는 보조적인 역할을 수행하는 스레드이다. 주 스레드가 종료되면 데몬 스레드는 강제적으로 자동 종료되는데, 그 이유는 주 스레드의 보조 역할을 수행하므로 주 스레드가 종료되면 데몬 스레드의 존재 의미가 사라지기 때문이다.

# Chapter 13. 컬렉션 프레임워크

## 13-1 컬렉션 프레임워크

- 컬렉션 프레임워크 : 자료구조를 사용해서 객체들을 효율적으로 추가, 삭제, 검색할 수 있도록 인터페이스와 구현 클래스를 java.util 패키지에서 제공한다. 컬렉션은 객체의 저장을 뜻하고, 프레임워크란 사용 방법을 정해놓은 라이브러리를 말한다.

### List 컬렉션

- 배열과 비슷하게 객체를 인덱스로 관리한다. 배열과의 차이점은 저장 용량이 자동으로 증가하며, 객체를 저장할 때 자동 인덱스가 부여된다는 것
- 객체 자체를 저장하는 것이 아니라 객체의 번지를 참조한다. 그렇기 때문에 동일한 객체를 중복 저장할 수 있는데, 이 경우 동일한 번지가 참조된다.
- null도 저장이 가능하며, 이 경우 해당 인덱스는 객체를 참조하지 않는다.

| 기능 | 메소드 | 설명 |
|---|---|---|
| 객체 추가 | boolean add(E e) | 주어진 객체를 맨 끝에 추가한다. |
| | void add(int index, E element) | 주어진 인덱스에 객체를 추가한다. |
| | E set(int index, E element) | 주어진 인덱스에 저장된 객체를 주어진 객체로 바꾼다. |
| 객체 검색 | boolean contains(Object o) | 주어진 객체가 저장되어 있는지 조사한다, |
| | E get(int index) | 주어진 인덱스에 저장된 객체를 리턴한다. |
| | boolean isEmpty() | 컬렉션이 비어 있는지 조사한다. |
| | int size() | 저장되어 있는 전체 객체 수를 리턴한다. |
| 객체 삭제 | void clear() | 저장된 모든 객체를 삭제한다. |
| | E remove(int index) | 주어진 인덱스에 저장된 객체를 삭제한다. |
| | boolean remove(Object o) | 주어진 객체를 삭제한다. |

- ArrayList : 빈번한 객체 삭제와 삽입이 일어나는 곳에서는 ArrayList를 사용하는 것이 바람직하지 않는다.
- Vector : ArrayList와 동일한 내부 구조를 가지고 있다. 생성하기 위해서는 저장할 객체 타입을 타입 파라미터로 표기하고 기본 생성자를 호출하면 된다. 다른 점은 Vector는 동기화된 메소드로 구성되어 있기 때문에 멀티 스레드가 동시에 Vector의 메소드들을 실행할 수 없고, 하나의 스레드가 메소드를 실행을 완료해야만 다른 스레드가 메소드를 실행할 수 있다는 것이다.
- LinkedList : List 구현 클래스이므로 ArrayList와 사용 방법은 똑같은데, 내부 구조는 완전 다르다. 내부 배열에 객체를 저장해서 관리하지만, LinkedList는 인접 참조를 링크해서 체인처럼 관리한다.

| 구분 | 순차적으로 추가/삭제 | 중간에 추가/삭제 | 검색 |
|---|---|---|---|
| ArrayList | 빠르다 | 느리다 | 빠르다 |
| LinkedList | 느리다 | 빠르다 | 느리다 |

### Set 컬렉션

- List 컬렉션은 객체의 저장 순서를 유지하지만, Set 컬레션은 저장 순서가 유지되지 않는다. 객체를 중복해서 저장할 수 없고, 하나의 null만 저장할 수 있다.

| 기능 | 메소드 | 설명 |
|---|---|---|
| 객체 추가 | boolean add(E e) | 주어진 객체를 저장한다. 객체가 성공적으로 저장되면 true를 리턴하고 중복 객체면 false를 리턴한다. |
| 객체 검색 | boolean contains(Object o) | 주어진 객체가 저장되어 있는지 조사한다. |
| | boolean isEmpty() | 컬렉션이 비어 있는지 조사한다. |
| | Iterator<E> iterator() | 저장된 객체를 한 번씩 가져오는 반복자를 리턴한다. |
| | int size() | 저장되어 있는 전체 객체 수를 리턴한다. |

| 객체 삭제 | void clear() | 저장된 모든 객체를 삭제한다. |
|---|---|---|
| | boolean remove(Object o) | 주어진 객체를 삭제한다. |

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| boolean | hasNext() | 가져올 객체가 있으면 true를 리턴하고 없으면 false를 리턴한다. |
| E | next() | 컬렉션에서 하나의 객체를 가져온다. |
| void | remove() | Set 컬렉션에서 객체를 제거한다. |

- HashSet : Set 인터페이스의 구현 클래스, 생성하기 위해서는 기본 생성자를 호출하면 된다.

```
Set<E> set = new HashSet<E>();
```

### Map 컬렉션

- 키와 값으로 구성된 Map.Entry 객체를 저장하는 구조를 가지고 있다. Entry는 Map 인터페이스 내부에 선언된 중첩 인터페이스이다.

| 기능 | 메소드 | 설명 |
|---|---|---|
| 객체 추가 | V put(K key, V value) | 주어진 키로 값을 저장한다. 새로운 키일 경우 null을 리턴하고 동일한 키가 있을 경우 값을 대체하고 이전 값을 리턴한다. |
| 객체 검색 | boolean containsKey(Object key) | 주어진 키가 있는지 여부를 확인한다. |
| | boolean containsValue(Object value) | 주어진 값이 있는지 여부를 확인한다. |
| | Set<Map.Entry<K.V>> entrySet() | 키와 값의 쌍으로 구성된 모든 Map.Entry 객체를 Set에 담아서 리턴한다. |
| | V get(Object key) | 주어진 키가 있는 값을 리턴한다. |
| | boolean isEmpty() | 컬렉션이 비어 있는지 여부를 확인한다. |
| | Set<K> keySet() | 모든 키를 Set 객체에 담아서 리턴한다. |
| | int size() | 저장된 키의 총 수를 리턴한다. |
| | Collection<V> values() | 저장된 모든 값을 Collection에 담아서 리턴한다. |
| 객체 삭제 | void clear() | 모든 Map.Entry(키와 값)를 삭제한다. |
| | V remove(Object key) | 주어진 키와 일치하는 Map.Entry를 삭제하고 값을 리턴한다. |

- HashMap : Map 인터페이스를 구현한 대표적인 Map 컬렉션이다. 키로 사용할 객체는 hashCode()와 equals() 메소드를 재정의해서 동등 객체가 될 조건을 정해야 한다.

- Hashtable : HashMap과 동일한 내부 구조를 가지고 있다 Hashtable도 키로 사용할 객체는 hashCode()와 equals() 메소드를 재정의해서 동등 객체가 될 조건을 정해야 한다. HashMap과 차이점은 Hashtable은 동기화된 메소드로 구성되어 있기 때문에 멀티 스레드가 동시에 Hashtable의 메소드들을 실행할 수 없고, 하나의 스레드가 실행을 완료해야만 다른 스레드를 실행할 수 있다.

## 13-2 LIFO와 FIFO 컬렉션

- 후입선출(LIFO) : 나중에 넣은 객체가 먼저 빠져나가는 자료구조
- 선출(FIFO) : 먼저 넣은 객체가 먼저 빠져나가는 자료구조

### Stack

- LIFO 자료구조를 구현한 클래스

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| E | push(E item) | 주어진 객체를 스택에 넣는다. |
| E | peek() | 스택의 맨 위 객체를 가져온다. 객체를 스택에서 제거하지 않는다. |
| E | pop() | 스택의 맨 위 객체를 가져온다. 객체를 스택에서 제거한다. |

- 객체를 생성하려면 저장할 객체 타입을 E 타입 파라미터 자리에 표기하고 기본 생성자를 호출하면 된다.

*Queue*

- *FIFO* 자료구조에서 사용되는 메소드를 정의한다.

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| boolean | offer(E e) | 주어진 객체를 넣는다. |
| E | peek() | 객체 하나를 가져온다. 객체를 큐에서 제거하지 않는다. |
| E | poll() | 객체 하나를 가져온다. 객체를 큐에서 제거한다. |

- *LinkedList*는 대표적인 구현 클래스로, *List* 인터페이스를 구현했기 때문에 *List* 컬렉션이기도 하다.

# Chapter 14. 입출력 스트림

## 14-1 입출력 스트림

- 자바에서 데이터는 스트림을 통해 입출력된다. 스트림은 단일 방향으로 연속적으로 흘러가는 것을 말한다.
- 출발지(키보드, 파일, 프로그램) -> 입력스트림 -> 프로그램 -> 출력 스트림 -> 도착지(모니터, 파일, 프로그램)

### 입출력 스트림의 종류

- *java.io* 패키지에는 여러 가지 종류의 스트림 클래스를 제공하고 있다.
- 바이트 기반 스트림 : 그림, 멀티미디어 등의 바이너리 데이터를 읽고 출력할 때 사용
- 문자 기반 스트림 : 문자 데이터를 읽고 출력할 때 사용

| 구분 | 바이트 기반 스트림 | | 문자 기반 스트림 | |
|---|---|---|---|---|
| | 입력 스트림 | 출력 스트림 | 입력 스트림 | 출력 스트림 |
| 최상위 클래스 | InputStream | OutputStream | Reader | Writer |
| 하위 클래스<br>(예) | XXXInputStream<br>(FileInputStream) | XXXOutputStream(FileOutputStream) | XXXReader<br>(FileReader) | XXXWriter<br>(FileWriter) |

### 바이트 출력 스트림 : *OutputStream*

- 바이트 기반 출력 스트림의 최상위 클래스로 추상 클래스
- 모든 바이트 기반 출력 스트림 클래스는 *OutputStream* 클래스를 상속받아서 만들어진다.
- *FileOutputStream, PrintStream, BufferedOutputStream, DataOutputStream*

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| void | write(int b) | 1byte를 출력한다. |
| void | writer(byte[] b) | 매개값으로 주어진 배열 b의 모든 바이트를 출력한다. |
| void | write(byte[] b, int off, int len) | 매개값으로 주어진 배열 b[off]부터 len개까지의 바이트를 출력한다. |
| void | flush() | 출력 버퍼에 잔류하는 모든 바이트를 출력한다. |
| void | close() | 출력 스트림을 닫는다. |

### 바이트 입력 스트림 : *InputStream*

- 바이트 기반 입력 스트림의 최상위 클래스로 추상 클래스
- 모든 바이트 기반 입력 스트림은 *InputStream* 클래스를 상속받아서 만들어진다.
- *FileInputStream, BufferedInputStream, DataInputStream*

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| int | read() | 1byte를 읽고 읽은 바이트를 리턴한다. |
| int | read(byte[] b) | 읽은 바이트를 매개값으로 주어진 배열에 저장하고 읽은 바이트 수를 리턴한다. |
| int | read(byte[] b, int off, int len) | len개의 바이트를 읽고 매개값으로 주어진 배열에서 b[off]부터 len개까지 저장한다. 그리고 읽은 바이트 수를 리턴한다. |
| void | close() | 입력 스트림을 닫는다. |

### 문자 출력 스트림 : *Writer*

- 문자 기반 출력 스트림의 최상위 클래스로 추상 클래스
- 모든 문자 기반 출력 스트림 클래스는 *Writer* 클래스를 상속 받아서 만들어진다.
- *FileWriter, BufferedWriter, PrintWriter, OurputStream*

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|

| | | |
|---|---|---|
| void | write(int c) | 매개값으로 주어진 한 문자를 보낸다. |
| void | write(char[] cbuf) | 매개값으로 주어진 배열의 모든 문자를 보낸다. |
| void | write(char[] cbuf, int off, int len) | 매개값으로 주어진 배열에서 cbuf[off]부터 len개까지의 문자를 보낸다. |
| void | write(String str) | 매개값으로 주어진 문자열을 보낸다. |
| void | write(String str, int off, int len) | 매개값으로 주어진 문자열에서 off 순번부터 len개까지의 문자를 보낸다. |
| void | flush() | 버퍼에 잔류하는 모든 문자를 출력한다. |
| void | close() | 출력 스트림을 닫는다. |

### 문자 입력 스트림 : Reader

- 문자 입력 스트림의 최상위 클래스로 추상 클래스
- 모든 문자 기반 입력 스트림은 Reader 클래스를 상속받아서 만들어진다.
- FileReader, BufferedReader, InputStreamReader

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| int | read() | 1개의 문자를 읽고 리턴한다. |
| int | read(char[] cbuf) | 읽은 문자들을 매개값으로 주어진 문자 배열에 저장하고 읽은 문자수를 리턴한다. |
| int | read(char[] cbuf, int off, int len) | len개의 문자를 읽고 매개값으로 주어진 문자 배열에서 cbuf[off]부터 len개까지 저장한다. 그리고 읽은 문자 수를 리턴한다. |
| void | close() | 입력 스트림을 닫는다. |

## 14-2 보조 스트림

- 보조 스트림 : 다른 스트림과 연결이 되어 여러 가지 편리한 기능을 제공해주는 스트림

-> 입력 스트림(보조 스트림) -> 프로그램 -> (보조스트림)출력 스트림 ->

### 보조 스트림 연결하기

- 보조스트림 변수 = new 보조스트림(연결스트림)

### 문자 변환 보조 스트림

- OutputStreamWriter : 바이트 기반 출력 스트림에 연결되어 문자 출력 스트림인 Writer로 변환하는 보조 스트림

Writer writer = new OutputStreamWriter(바이트 기반 출력 스트림);

- InputStreamReader : 바이트 기반 입력 스트림에 연결되어 문자 입력 스트림인 Reader로 변환하는 보조 스트림

Reader reader = new InputStreamReader(바이트 기반 입력 스트림);

성능 향상 보조 스트림

- 프로그램이 입출력 소스와 직접 작업하지 않고 중간에 메모리 버퍼와 작업함으로써 실행 성능을 향상시킬 수 있다.

- BufferedOutputStream과 BufferedWriter : BufferedOutputStream은 바이트 기반 출력 스트림에 연결되어 버퍼를 제공해주는 보조 스트림이고, BufferedWriter은 문자 기반 출력 스트림에 연결되어 버퍼를 제공해주는 보조 스트림이다.

BufferedOutputStream bos = new BufferedOutputStream(바이트 기반 출력 스트림);

BufferedWriter bw = new BufferedWriter(문자 기반 출력 스트림);

- BufferedInputStream과 BufferedReader : BufferedInputStream은 바이트 기반 입력 스트림에 버퍼를 제공해주는 보조 스트림이고, BufferedReader는 문자 기반 입력 스트림에 연결되어 버퍼를 제공해주는 보조 스트림이다.

BufferedInputStream bis = new BufferedInputStream(바이트 기반 입력 스트림);

BufferedReader br = new BufferedReader(문자 기반 입력 스트림);

### 기본 타입 입출력 보조 스트림

-> 바이트 -> InputStream(DataInputStream) -> 프로그램(기본 타입 : int, double) -> (DataOutputStream)OutputStream -> 바이트

DataInputStream dis = new DataInputStream(바이트 기반 입력 스트림);

DataOutputStream dos = new DataOutputStream(바이트 기반 출력 스트림);

| DataInputStream | | DataOutputStream | |
|---|---|---|---|
| boolean | readBoolean() | void | writeBoolean(boolean v) |
| byte | readByte() | void | writeByte(int v) |
| char | readChar() | void | writeChar(int v) |
| double | readDouble() | void | writeDouble(double v) |
| float | readFloat() | void | writeFloat(float v) |
| int | readInt() | void | writeInt(int v) |
| long | readLong() | void | writeLong(long v) |
| short | readShort() | void | writeShort(int v) |
| String | readUTF() | void | writeUTF(String str) |

### 프린터 보조 스트림

-> 프로그램(기본 타입 또는 문자열 : print(), println(), printf()) -> PrintStream / PrintWriter -> OutputStream/Writer -> 문자 코드로 출력

PrintStream ps = new PrintStream(바이트 기반 출력 스트림);

PrintStream pw = new PrintWriter(문자 기반 출력 스트림);

| PrintStream / PrintWriter | | | |
|---|---|---|---|
| void | print(boolean b) | void | println(boolean b) |
| void | print(char c) | void | println(char c) |
| void | print(double d) | void | println(double d) |
| void | print(float f) | void | println(float f) |
| void | print(int i) | void | println(int i) |
| void | print(long l) | void | println(long l) |
| void | print(Object obj) | void | println(Object obj) |
| void | print(String s) | void | println(String s) |
| | | void | println() |

### 객체 입출력 보조 스트림

-> 바이트 -> InputStream(ObjectInputStream 역직렬화) -> 프로그램(객체) -> ObjectOutputStream 직렬화(OutputStream) -> 바이트

ObjectInputStream ois = new ObjectInputStream(바이트 기반 입력 스트림);

ObjectOutputStream oos = new ObjectOutputStream(바이트 기반 출력 스트림);

oos.writeObject(객체);

객체 타입 변수 = (객체타입) ois.readObject();

public class XXX implements Serializable { … }

## 14-3 입출력 관련 API

- 콘솔(console) : 시스템을 사용하기 위해 키보드로 입력을 받고 모니커로 출력하는 소프트웨어를 말한다. 유닉스나 리눅스 운영체제는 터미널에 해당하고, 윈도우 운영체제는 명령 프롬프트에 해당한다.

### System.in 필드

- 자바는 콘솔에서 키보드의 데이터를 입력 받을 수 있도록 System 클래스의 in 정적 필드를 제공한다. System.in은 InputStream 타입의 필드이다.

- 주로 InputStreamReader 보조 스트림과 BufferedReader 보조 스트림을 연결해서 사용하거나, Scanner를 이용해서 입력된 문자열을 읽는다.

```
InputStream is = System.in
```

### System.out 필드

- 콘솔에서 키보드로 입력된 데이터를 System.in으로 읽었다면, 반대로 콘솔에서 모니터로 데이터를 출력하기 위해서는 System 클래스의 out 정적 필드를 사용한다. System.out은 PrintStream 타입의 필드이다.

- PrintStream이 제공하는 print(), println(), printf()와 같은 메소드를 이용해서 모니터로 출력할 수 있다.

### Scanner 클래스

- Scanner 클래스는 입출력 스트림도 아니고, 보조 스트림도 아니다. Scanner는 문자 파일이나, 바이트 기반 입출력 스트림에서 라인 단위 문자열을 쉽게 읽도록 하기 위해 java.util 패키지에서 제공하는 클래스이다.

```
Scanner scanner = new Scanner(System.in);
String inputData = scanner.nextLine();
```

### File 클래스

- java.io 패키지에서 제공하는 File 클래스는 파일 및 폴더(디렉토리) 정보를 제공해주는 역할을 한다.

```
File file = new File("C:/Temp/file.txt");
File file = new File("C:\\Temp\\file.txt");
```

- 경로 구분자는 운영체제마다 다르다. 윈도우에서는 \ 또는 /를 둘 다 사용할 수 있고, 유닉스나 리눅스에서는 /를 사용한다. 만약 윈도우에서 \를 경로 구분자로 사용하고 싶다면 이스케이프 문자(\\)로 기술해야 한다.

```
boolean isExist = file.exist();
```

exist() 메소드의 리턴값이 false일 때 파일 또는 폴더를 생성할 수 있다.

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| boolean | createNewFile() | 새로운 파일을 생성한다. |
| boolean | mkdir() | 새로운 폴더를 생성한다. |
| boolean | mkdirs() | 경로상에 없는 모든 폴더를 생성한다. |

exist() 메소드의 리턴값이 true라면 메소드를 사용할 수 있다.

| 리턴 타입 | 메소드 | 설명 |
|---|---|---|
| boolean | delete() | 파일 또는 폴더를 삭제한다. |
| boolean | canExecute() | 실행할 수 있는 파일인지 여부를 확인한다. |
| boolean | canRead() | 읽을 수 있는 파일인지 여부를 확인한다. |
| boolean | canWrite() | 수정 및 저장할 수 있는 파일인지 여부를 확인한다. |
| String | getName() | 파일의 이름을 리턴한다. |
| String | getParent() | 부모 폴더를 리턴한다. |
| File | getPatentFile() | 부모 폴더를 File 객체로 생성 후 리턴한다. |
| String | getPath() | 전체 경로를 리턴한다. |
| boolean | isDirectory() | 폴더인지 여부를 확인한다. |

| | | |
|---|---|---|
| boolean | isFile() | 파일인지 여부를 확인한다. |
| boolean | isHidden() | 숨김 파일인지 여부를 확인한다. |
| long | lastModified() | 마지막 수정 날짜 및 시간을 리턴한다. |
| long | length() | 파일의 크기를 리턴한다. |
| String[] | list() | 폴더에 포함된 파일 및 서브 폴더 목록 전부를 String 배열로 리턴한다. |
| String[] | list(FilenameFilter filter) | 폴더에 포함된 파일 및 서브 폴더 목록 중에 FilenameFilter에 맞는 것만 String 배열로 리턴한다. |
| File[] | listFiles() | 폴더에 포함된 파일 및 서브 폴더 목록 전부를 File 배열로 리턴한다. |
| File[] | listFiles(FilenameFilter filter) | 폴더에 포함된 파일 및 서브 폴더 목록 중에 FilenameFilter에 맞는 것만 File 배열로 리턴한다. |

※ 한빛미디어 '혼자 공부하는 자바' 의 내용을 바탕으로 정리한 내용입니다.