

0. 목차

#1.인강/0.자바/1.자바-입문

- 인프런 강의
- 저자: 김영한

전체 목차

- 1. Hello World
- 2. 변수
- 3. 연산자
- 4. 조건문
- 5. 반복문
- 6. 스코프, 형변환
- 7. 훈련
- 8. 배열
- 9. 메서드
- 10. 다음으로

버전 수정 이력

2023-12-13

- 중복 오타 제거(JiWon님 도움)

2023-11-28

- 릴리즈
- char 바이트 용량 수정(쌤수님 도움)

1. Hello World

#1.인강/0.자바/1.자바-입문

- /개발 환경 설정
- /다운로드 소스 코드 실행 방법
- /자바 프로그램 실행
- /주석(comment)
- /자바란?

개발 환경 설정

IDE - 인텔리제이 vs 이클립스

- 자바 프로그램을 개발할 때는 인텔리제이(IntelliJ) 또는 이클립스(Eclipse)라는 툴을 많이 사용한다. 과거에는 이 클립스를 많이 사용했지만 최근에는 빠른 속도와 사용의 편의성 때문에 인텔리제이를 주로 사용한다.
- 자바로 개발하는 대부분의 메이저 회사들도 최근에는 인텔리제이를 주로 사용하므로 우리도 인텔리제이로 학습

OS - 윈도우 vs Mac

- 자바로 개발하는 대부분의 메이저 회사들은 Mac 사용
- 윈도우를 사용해도 무방함
- 강의는 Mac을 사용하지만 윈도우 사용자들을 최대한 배려해서 진행
 - 윈도우 화면 스크린샷, 윈도우용 단축키

참고: 자바를 별도로 설치하지 않아도 됩니다. 인텔리제이 안에서 자바 설치도 함께 진행합니다.

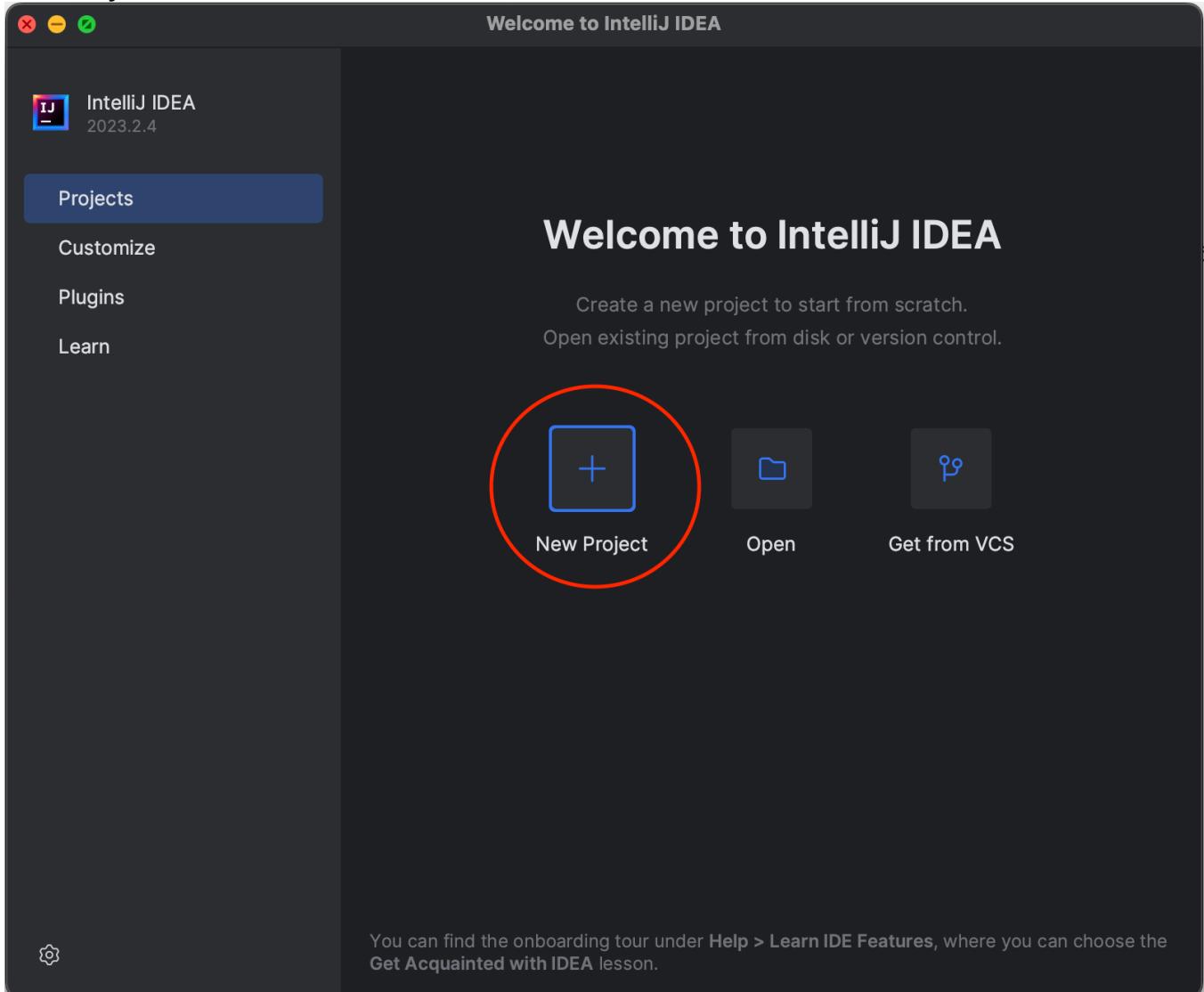
인텔리제이(IntelliJ) 설치하기

- 다운로드 링크: <https://www.jetbrains.com/ko-kr/idea/download>
- IntelliJ IDEA Community Edition (무료 버전)
 - OS 선택: Windows, macOS, Linux
 - ◆ Windows: .exe 선택
 - ◆ macOS: M1, M2: Apple Silicon 선택, 나머지: Intel 선택

참고: 인텔리제이는 무료 버전인 Community Edition과 유료 버전인 IntelliJ IDEA Ultimate가 있습니다. 제가 진행하는 모든 강의는 무료 버전인 Community Edition으로 충분합니다. 특히 자바 언어를 학습하는 단계에서는 유료 버전과 무료 버전의 차이가 없습니다.

인텔리제이 실행하기

New Project



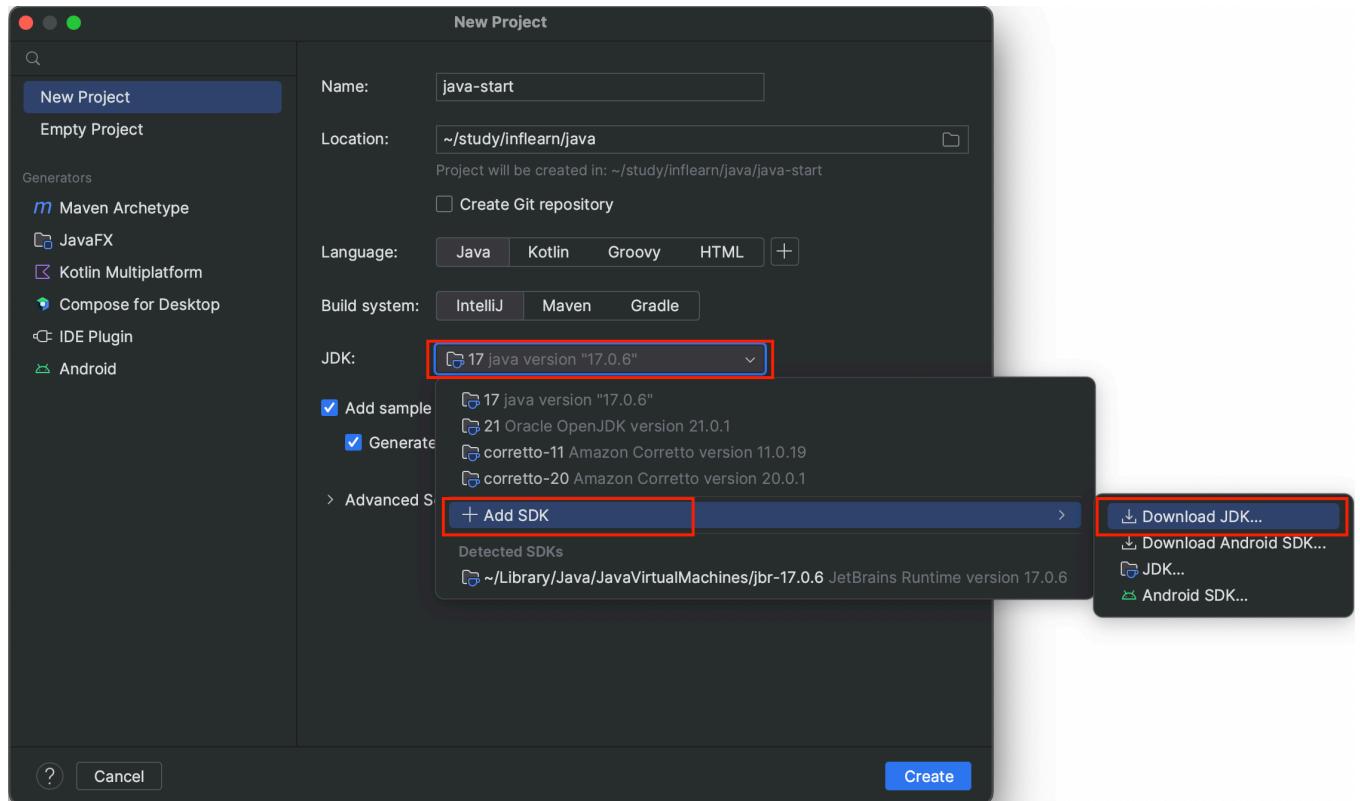
- New Project를 선택해서 새로운 프로젝트를 만들자

New Project 화면

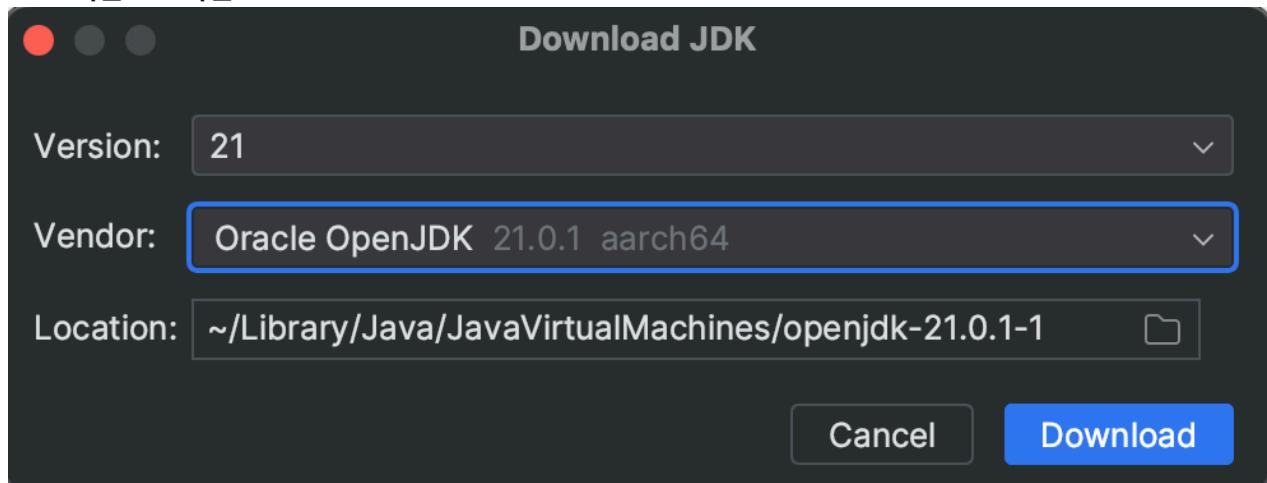
- Name: java-start
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: 자바 버전 17 이상
- Add sample code 선택

JDK 다운로드 화면 이동 방법

자바로 개발하기 위해서는 JDK가 필요하다. JDK는 자바 프로그래머를 위한 도구 + 자바 실행 프로그램의 묶음이다.



JDK 다운로드 화면



- Version: 21을 선택하자.
- Vendor: Oracle OpenJDK를 선택하자. 다른 것을 선택해도 된다.
 - aarch64: 애플 M1, M2, M3 CPU 사용시 선택, 나머지는 뒤에 이런 코드가 붙지 않은 JDK를 선택하면 된다.
- Location: JDK 설치 위치, 기본값을 사용하자.

Download 버튼을 통해서 다운로드 JDK를 다운로드 받는다.

다운로드가 완료 되고 이전 화면으로 돌아가면 Create 버튼 선택하자. 그러면 다음 IntelliJ 메인 화면으로 넘어간다.

IntelliJ 메인 화면

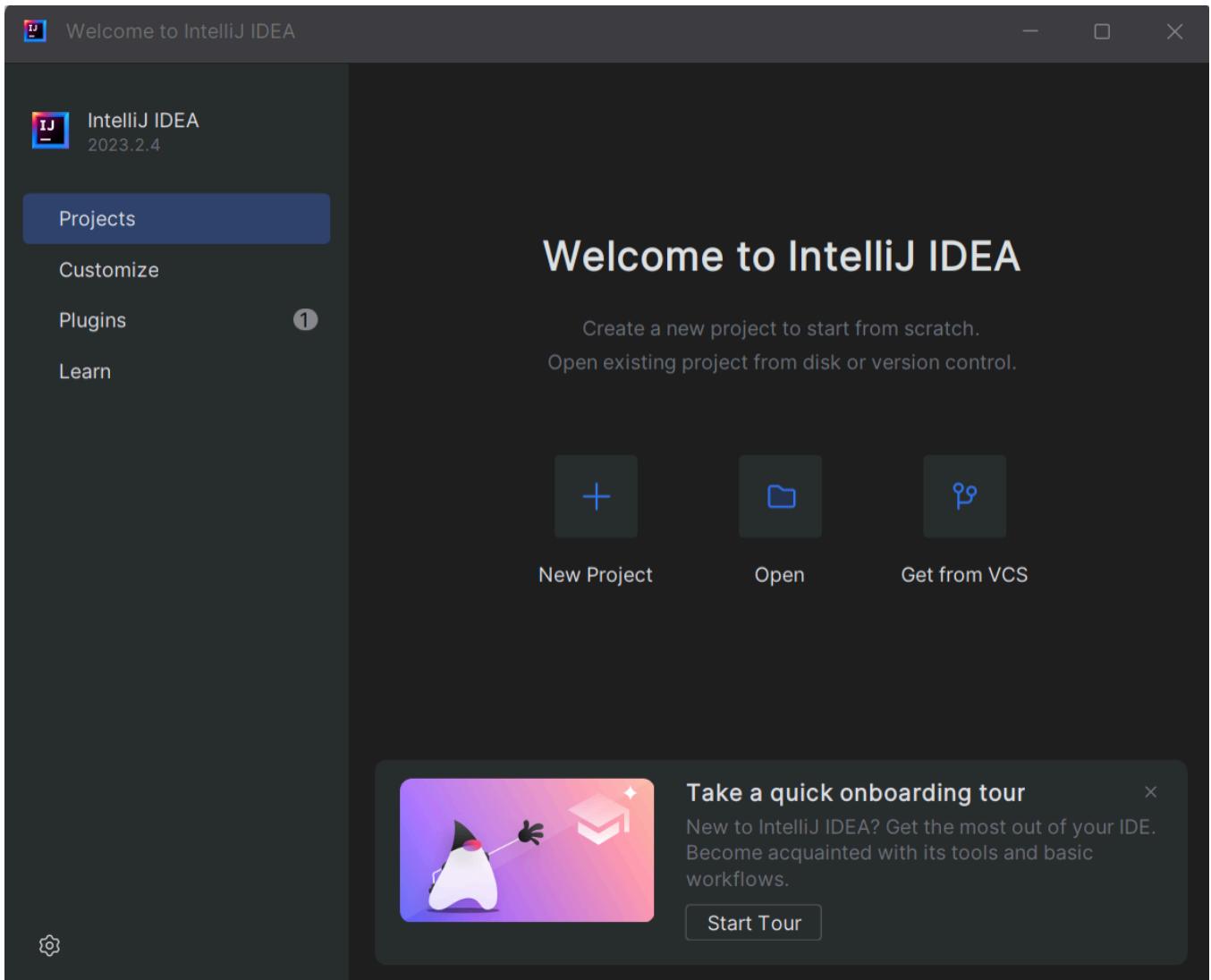
```
// Press Shift twice to open the Search Everywhere dialog and type ⌘S
// then press Enter. You can now see whitespace characters in your code
public class Main {
    public static void main(String[] args) {
        // Press Opt+Enter with your caret at the highlighted text to s
        // IntelliJ IDEA suggests fixing it.
        System.out.printf("Hello and welcome!");

        // Press Ctrl+R or click the green arrow button in the gutter t
        for (int i = 1; i <= 5; i++) {
            // Press Ctrl+D to start debugging your code. We have set c
            // for you, but you can always add more by pressing Cmd+F8.
            System.out.println("i = " + i);
        }
    }
}
```

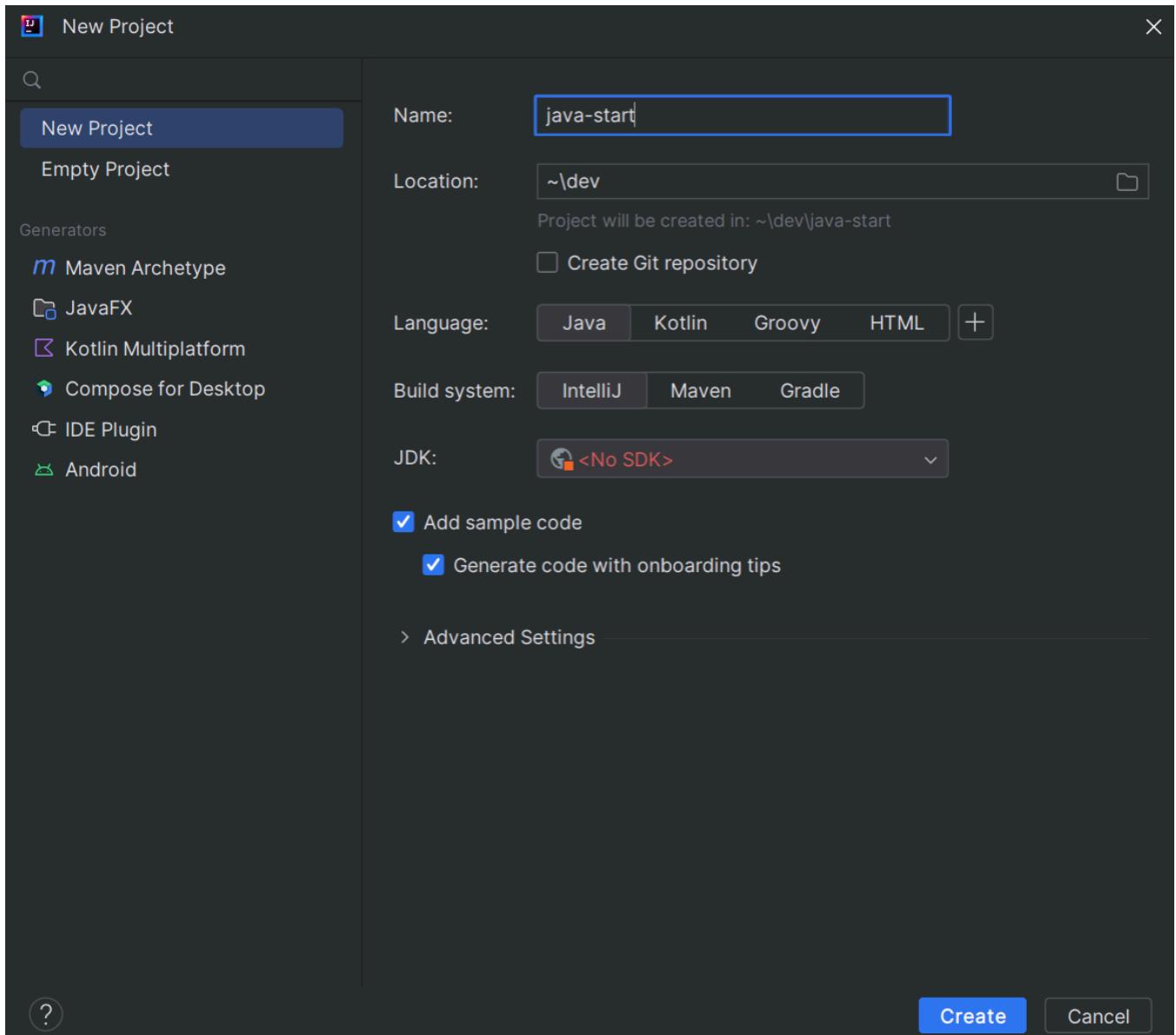
- 앞서 Add sample code 선택해서 샘플 코드가 만들어져 있다.
- 위쪽에 빨간색으로 강조한 초록색 화살표 버튼을 선택하고 Run 'Main.main()' 버튼을 선택하면 프로그램이 실행된다.

윈도우 사용자 추가 설명서

윈도우 사용자도 Mac용 IntelliJ와 대부분 같은 화면이다. 일부 다른 화면 위주로 설명하겠다.

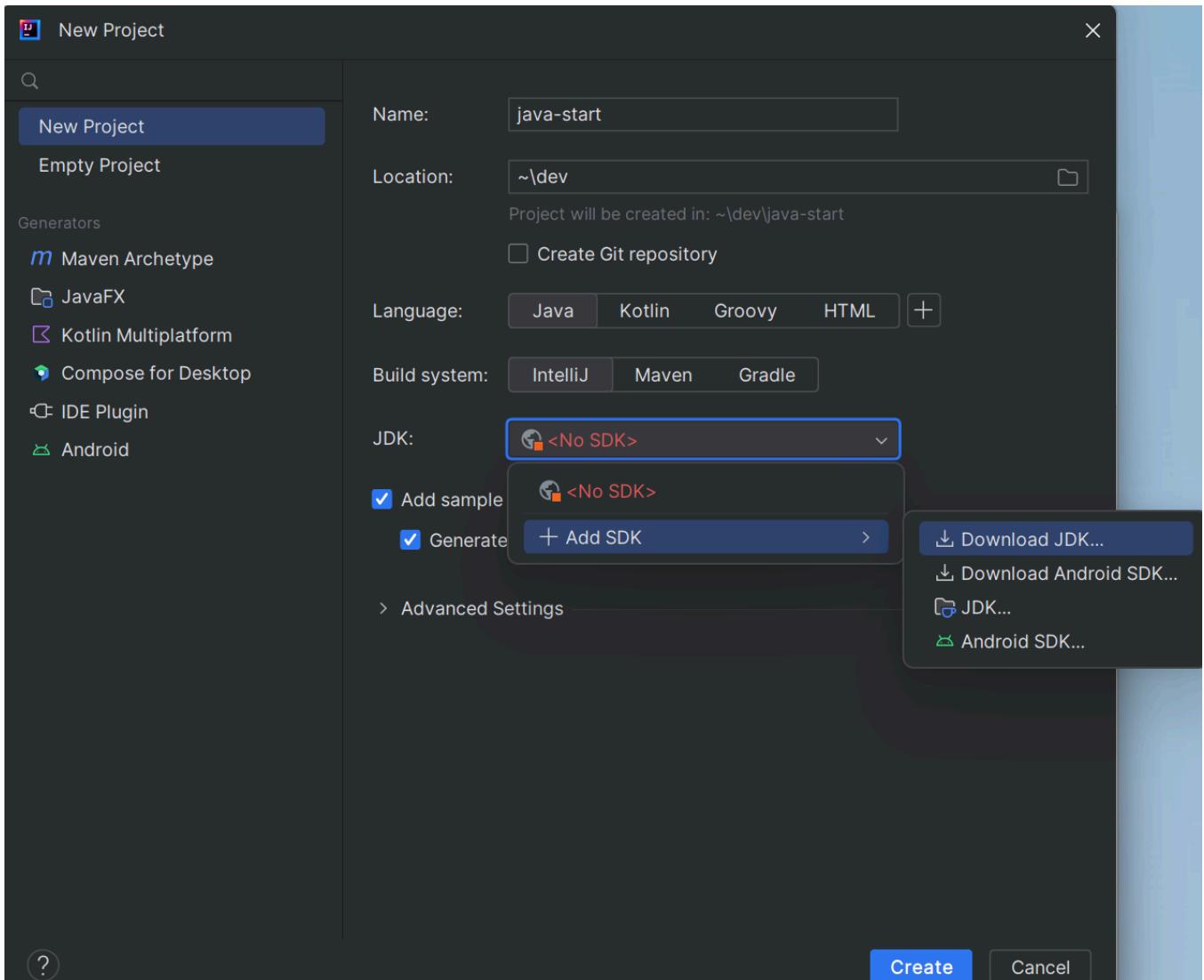


- 프로그램 시작 화면
- New Project 선택

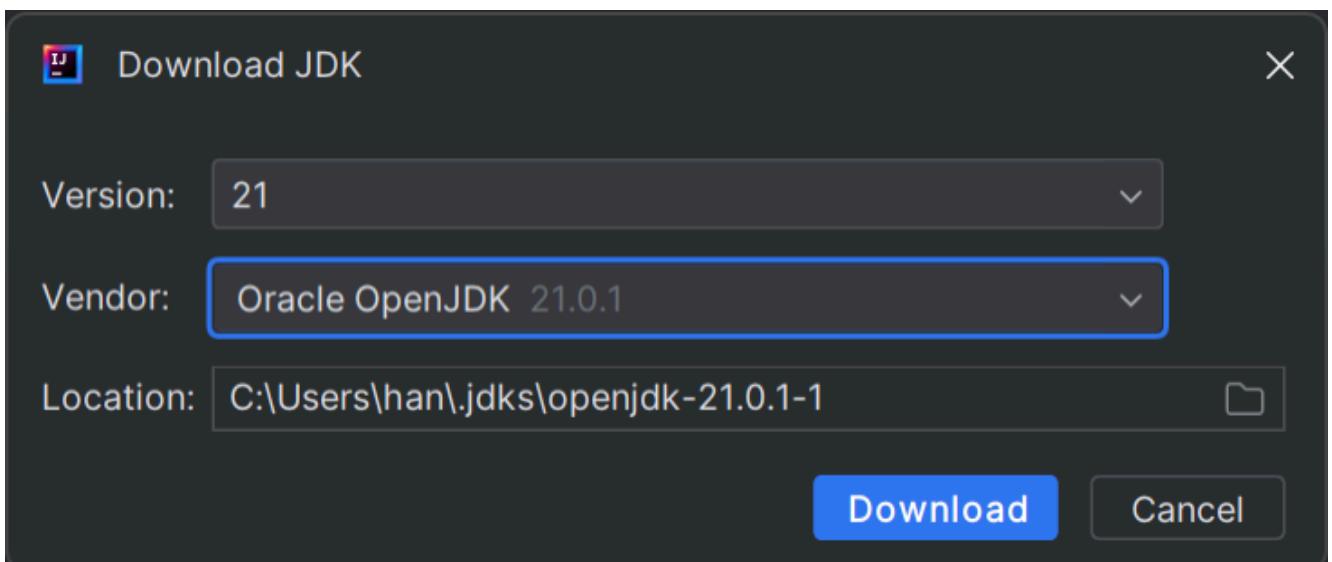


New Project 화면

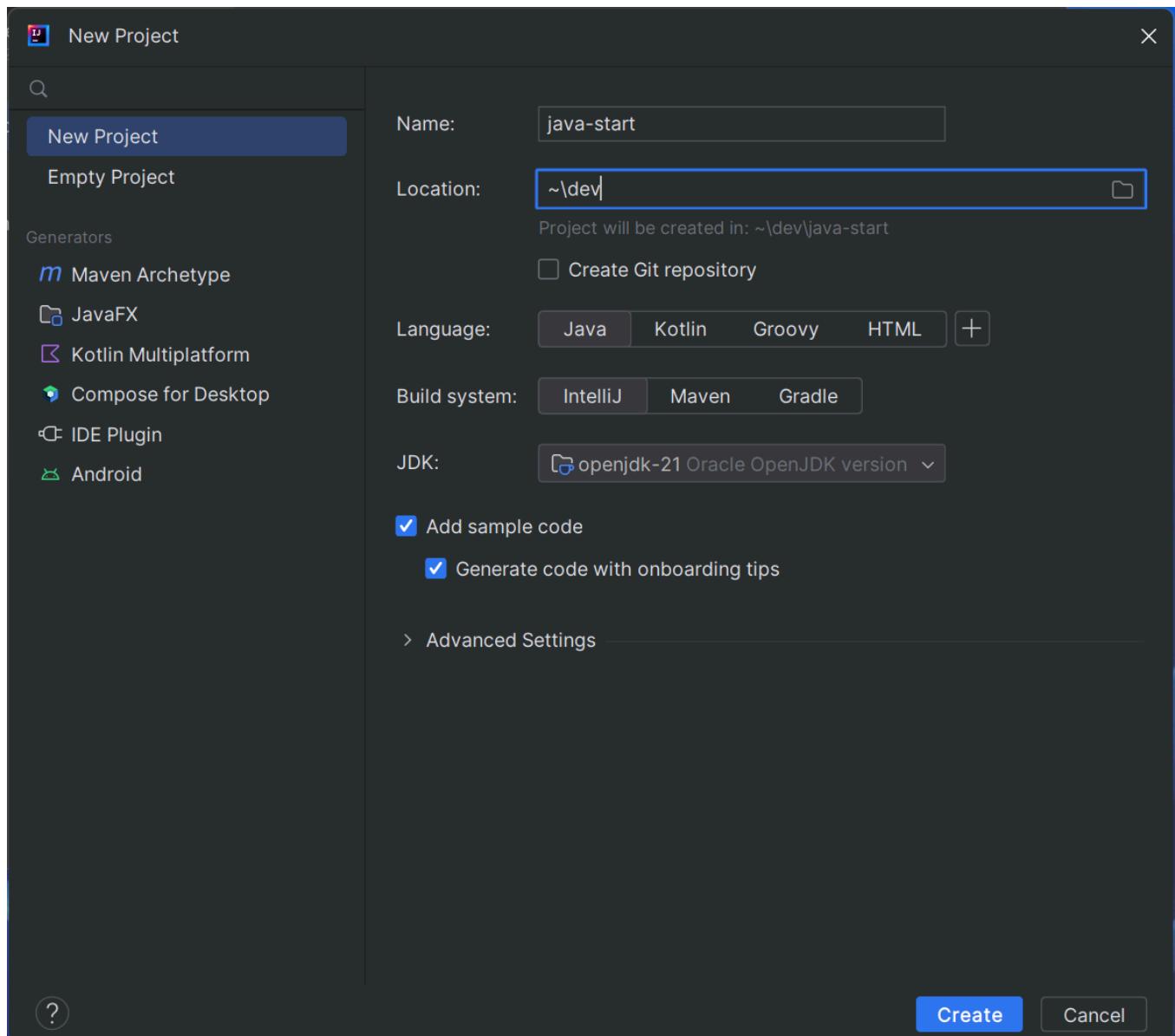
- Name: java-start
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: 자바 버전 17 이상
- Add sample code 선택



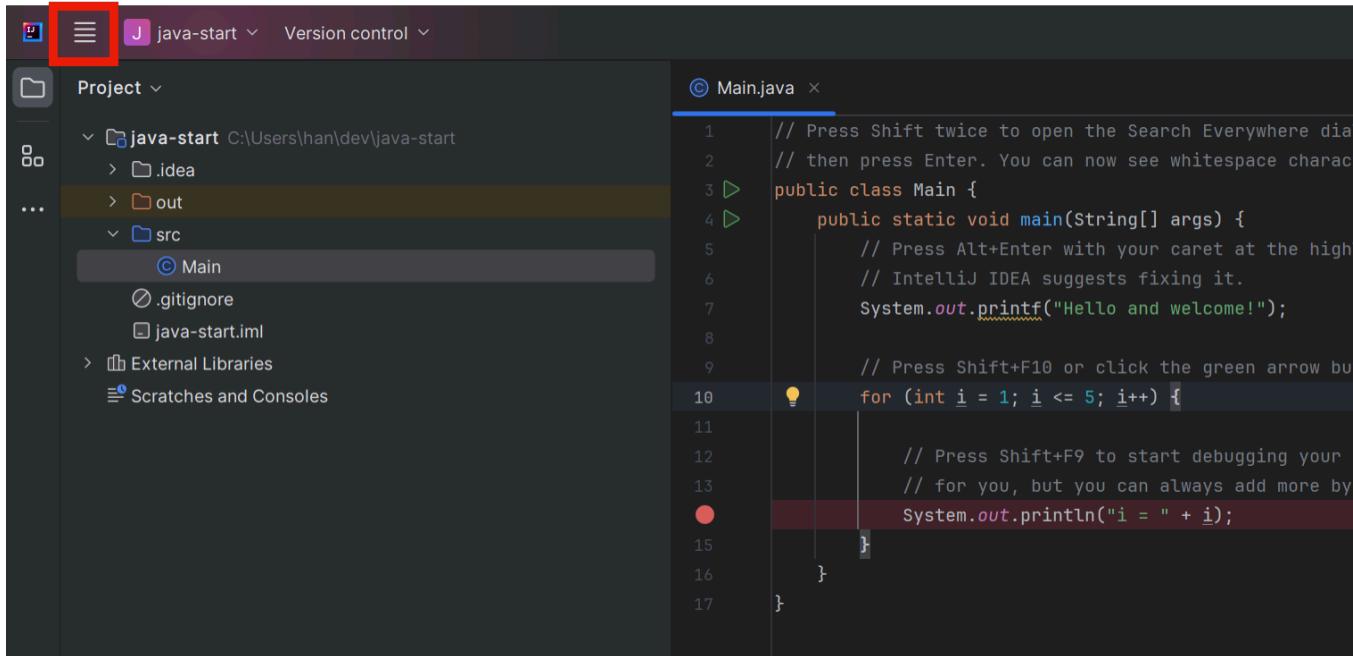
JDK 설치는 Mac과 동일하다.



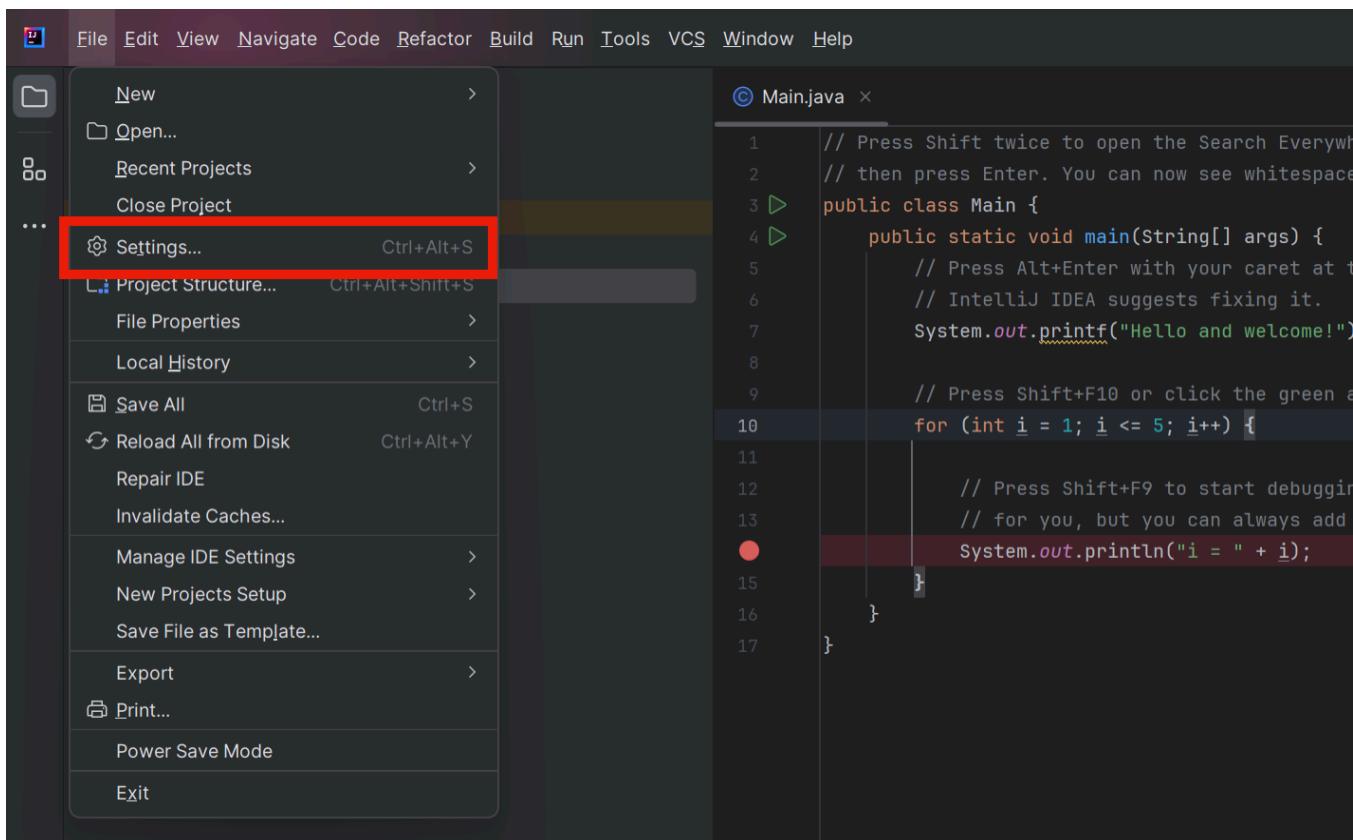
- Version: 21
- Vendor: Oracle OpenJDK
- Location은 가급적 변경하지 말자.



- New Project 완료 화면



- 원도우는 메뉴를 확인하려면 왼쪽 위의 빨간색 박스 부분을 선택해야 한다.



- Mac과 다르게 **Settings...** 메뉴가 File에 있다. 이 부분이 Mac과 다르므로 유의하자.

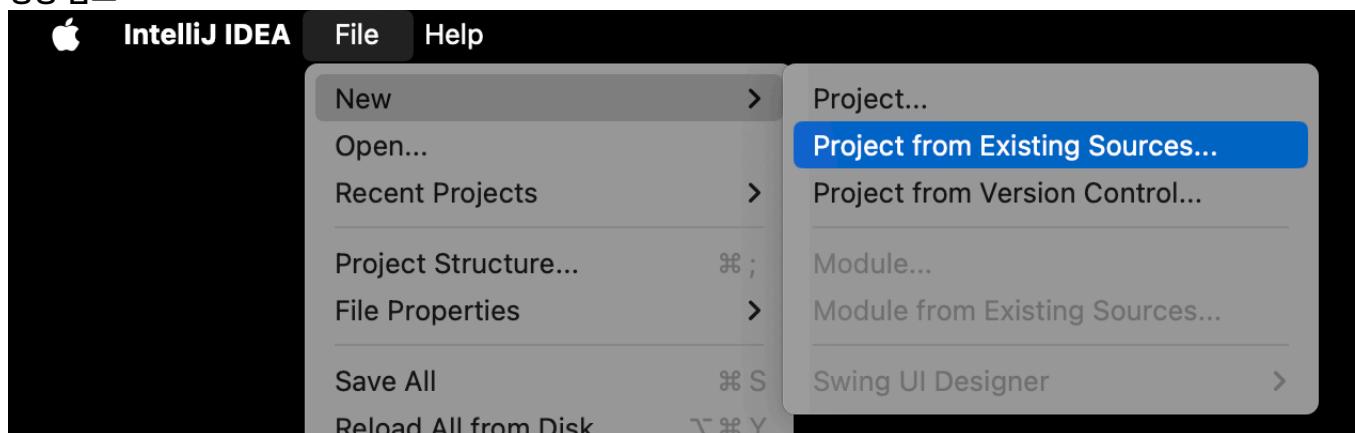
한글 언어팩 → 영어로 변경

- IntelliJ는 가급적 한글 버전 대신, 영문 버전을 사용하자. 개발하면서 필요한 기능들을 검색하게 되는데, 영문으로 된 자료가 많다. 이번 강의도 영문을 기준으로 진행한다.

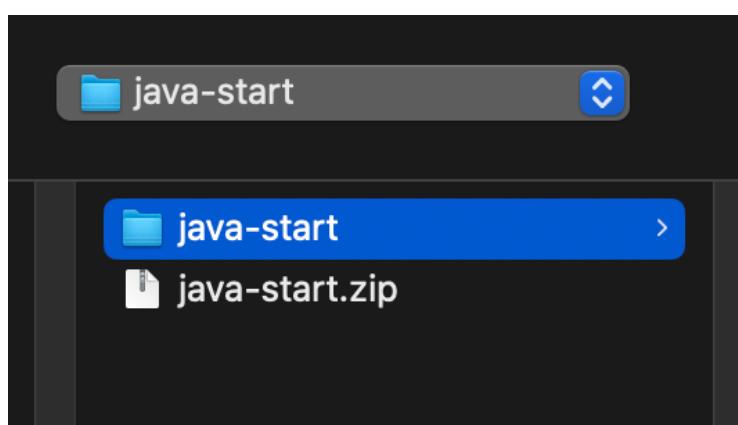
- 만약 한글로 나온다면 다음과 같이 영문으로 변경하자.
- Mac:** IntelliJ IDEA(메뉴) → Settings... → Plugins → Installed
- 윈도우:** File → Settings... → Plugins → Installed
 - Korean Language Pack 체크 해제
 - OK 선택후 IntelliJ 다시 시작

다운로드 소스 코드 실행 방법

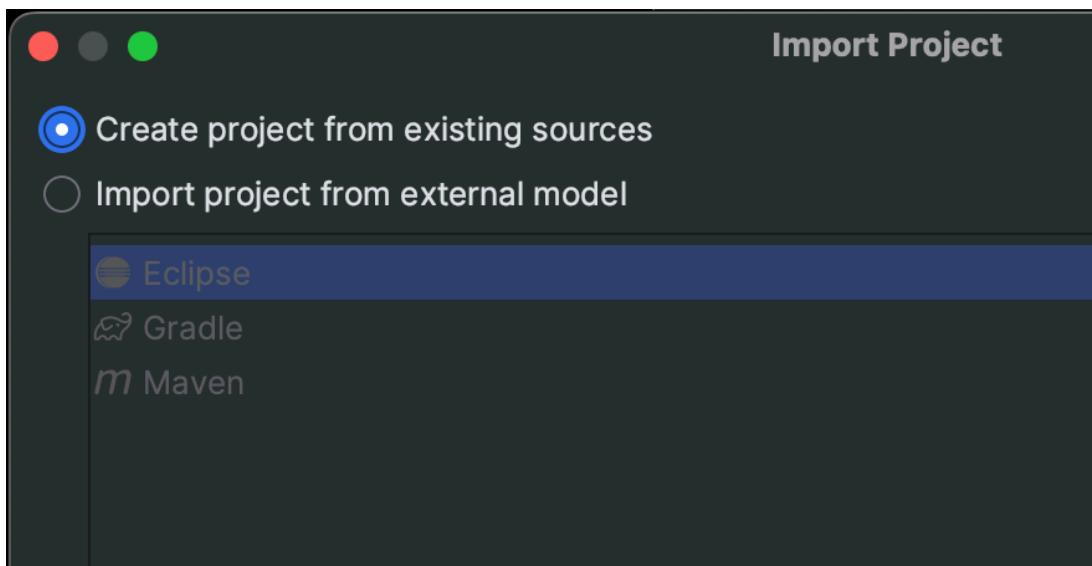
영상 참고



File -> New -> Project from Existing Sources... 선택

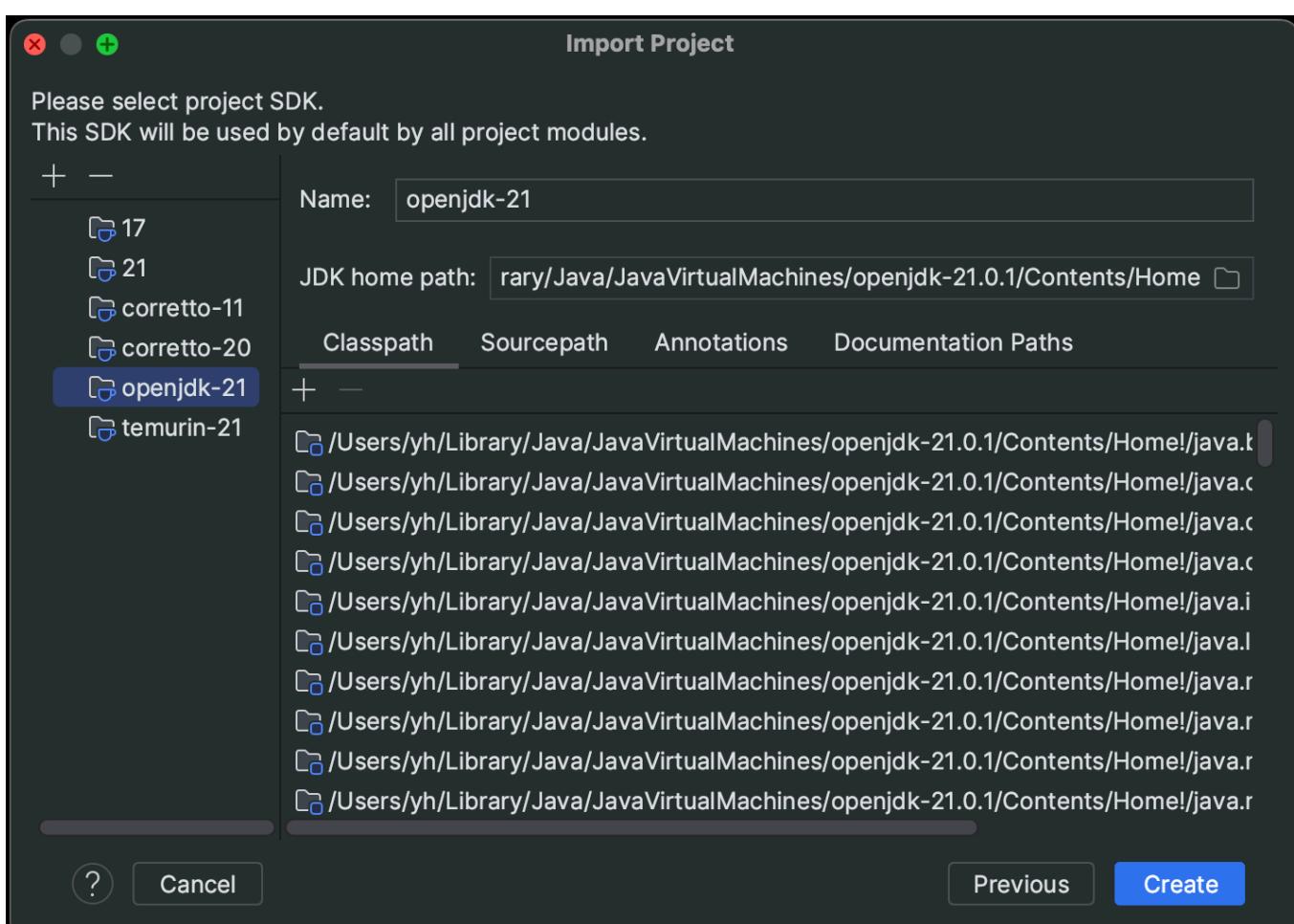


압축을 푼 프로젝트 폴더 선택

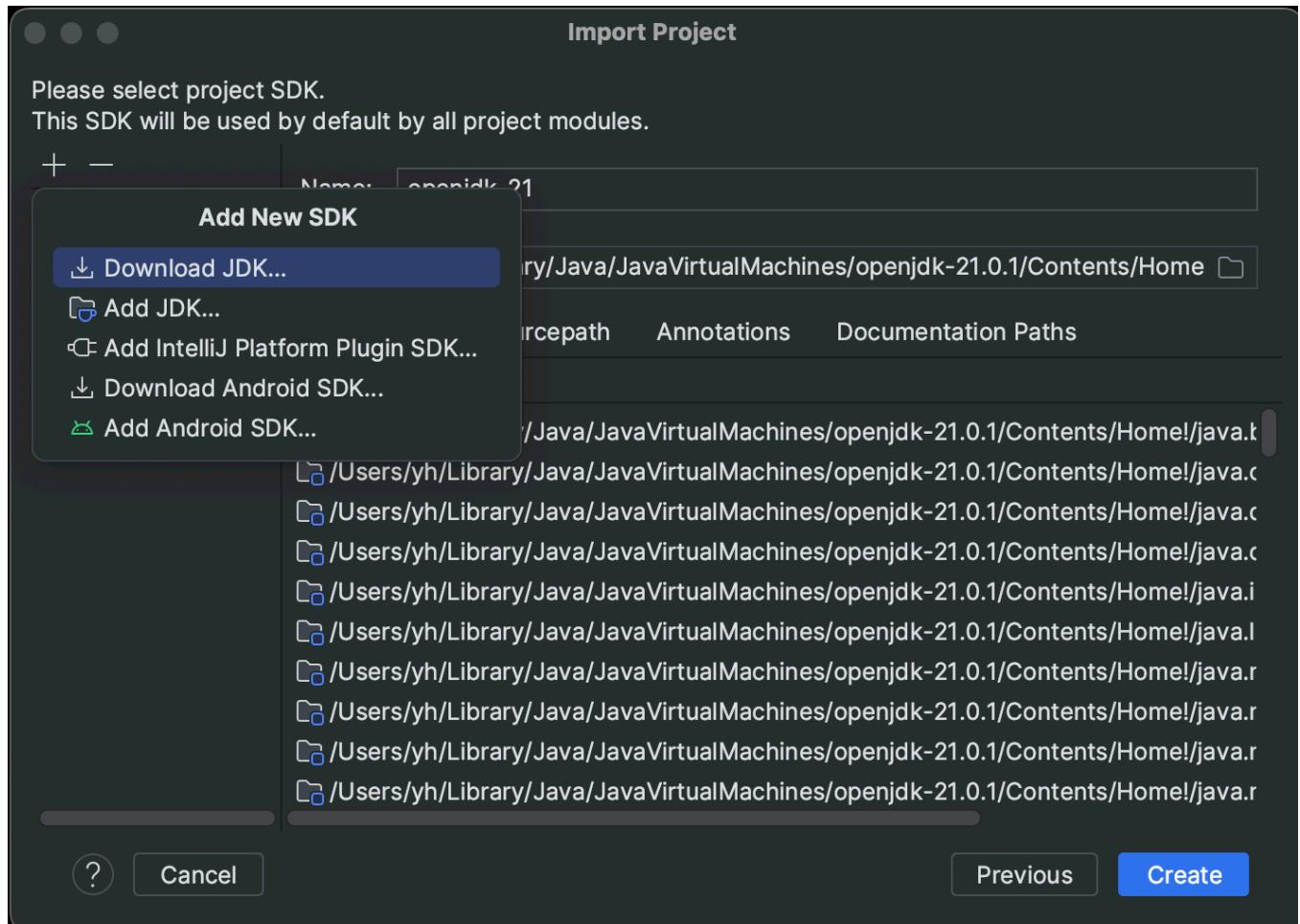


Create project from existing sources 선택

이후 계속 Next 선택



openjdk-21 선택



만약 JDK가 없다면 왼쪽 상단의 + 버튼을 눌러서 openjdk 21 다운로드 후 선택

이후 Create 버튼 선택

자바 프로그램 실행

HelloJava

```
public class HelloJava {  
  
    public static void main(String[] args) {  
        System.out.println("hello java");  
    }  
}
```

영상을 따라서 해당 코드를 만들고 실행해보자.

| 주의!

자바 언어는 대소문자를 구분한다. 대소문자가 다르면 오류가 발생할 수 있다.

실행 결과

```
hello java
```

코드를 분석해보자. 지금 단계에서는 이 코드의 모든 내용을 이해할 수 없다. 앞으로 차근차근 하나씩 알아가보자

```
public class HelloJava
```

- `HelloJava`를 클래스라 한다. 클래스(class)의 개념을 학습해야 이해할 수 있다. 클래스는 뒤에서 학습한다.
- 지금은 단순히 `HelloJava.java`라는 파일을 만들었다고 이해하면 된다.
- 파일명과 클래스 이름이 같아야 한다.
- `{}` 블록을 사용해서 클래스의 시작과 끝을 나타낸다.

```
public static void main(String[] args)
```

- `main` 메서드라 한다. 함수, 메서드의 개념을 학습해야 이해할 수 있다. 함수, 메서드는 뒤에서 학습한다.
- 자바는 `main(String[] args)` 메서드를 찾아서 프로그램을 시작한다.
- 지금은 단순히 `main`은 프로그램의 시작점이라고 이해하면 된다.
- `{}` 블록을 사용해서 메서드의 시작과 끝을 나타낸다.

```
System.out.println("hello java");
```

- `System.out.println()` : 값을 콘솔에 출력하는 기능이다.
- `"hello java"` : 자바는 문자열을 사용할 때 " (쌍따옴표)를 사용한다. 쌍따옴표 사이에 원하는 문자열을 감싸면 된다.
- `;` : 자바는 세미콜론으로 문장을 구분한다. 문장이 끝나면 세미콜론을 필수로 넣어야 한다.

참고: 괄호

- 소괄호 ()
- 중괄호 {}
- 대괄호 []

실행 과정

1. `HelloJava` 프로그램을 실행한다.
2. 자바는 시작점인 `main()` 메서드를 실행한다.
3. `System.out.println("hello java")`을 만나고, 문자열 `hello java`을 출력한다.
4. `main()` 메서드의 `{}` 블록이 끝나면 프로그램은 종료된다.

블록(block) 예시

```
public class HelloJava { //HelloJava 클래스의 범위 시작

    public static void main(String[] args) { //main() 메서드의 범위 시작
        System.out.println("hello java");
    } //main() 메서드의 범위 끝

} //HelloJava 클래스의 범위 끝
```

- 블록("{}")이 시작되고 끝날 때마다 들여쓰기가 적용되어 있는 것을 확인할 수 있다. 이것은 코드를 쉽게 구분하고 이해하도록 도와주는 좋은 관례이다. 블록이 중첩될 때마다 들여쓰기의 깊이가 추가된다.
- 들여쓰기는 보통 스페이스 4번을 사용한다. 참고로 IntelliJ IDE를 사용하면 키보드 Tab 을 한번 누르면 자동으로 스페이스 4번을 적용한다.
- 참고로 들여쓰기를 하지 않아도 프로그램은 작동한다. 하지만 코드를 읽기에 좋지 않다.

추가 예제

프로그램 코드에 익숙해지도록 다음 코드를 작성하고 실행해보자.

HelloJava2

```
public class HelloJava2 {

    public static void main(String[] args) {
        System.out.println("hello java1");
        System.out.println("hello java2");
        System.out.println("hello java3");
    }
}
```

실행 결과

```
hello java1
hello java2
hello java3
```

프로그램은 main() 을 시작으로 위에서 아래로 한 줄 씩 실행된다.

주석(comment)

소스 코드가 복잡하다면 소스 코드에 대한 이해를 돋기 위해 설명을 적어두고 싶을 수 있다.

또는 특정 코드를 지우지 않고, 잠시 실행을 막아두고 싶을 때도 있다.

이럴 때 주석을 사용하면 된다. 자바는 주석이 있는 곳을 무시한다.

주석의 종류

- 한 줄 주석 (single line comment)
 - // 기호로 시작한다. 이 기호 이후의 모든 텍스트는 주석으로 처리된다.
- 여러 줄 주석(multi line comment)
 - /* 로 시작하고 */ 로 끝난다. 이 사이의 모든 텍스트는 주석으로 처리된다.

CommentJava

```
public class CommentJava {  
  
    /*  
     * 주석을 설명하는 부분입니다.  
     */  
    public static void main(String[] args) {  
        System.out.println("hello java1"); //hello java1을 출력합니다. (한 줄 주석 - 부  
분 적용)  
        //System.out.println("hello java2"); 한 줄 주석 - 라인 전체 적용  
  
        /* 여러 줄 주석  
        System.out.println("hello java3");  
        System.out.println("hello java4");  
        */  
    }  
}
```

실행 결과

```
hello java1
```

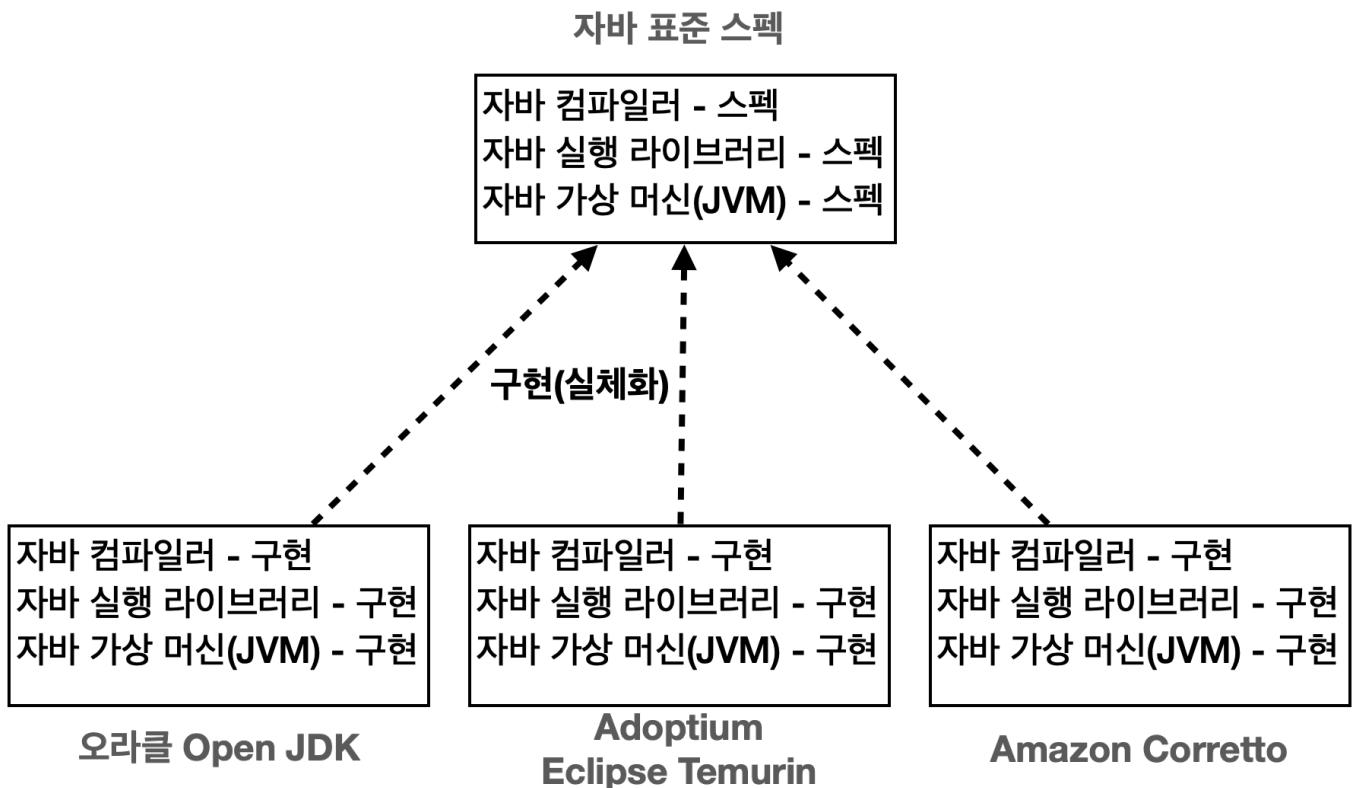
주석으로 처리한 코드가 실행되지 않은 것을 확인할 수 있다.

주석은 쉽게 이야기해서 자바 프로그램이 읽지 않고 무시하는 부분이다. 사람이 읽기 위해서 사용한다.

자바란?

자바 표준 스펙

자바 표준 스펙과 구현



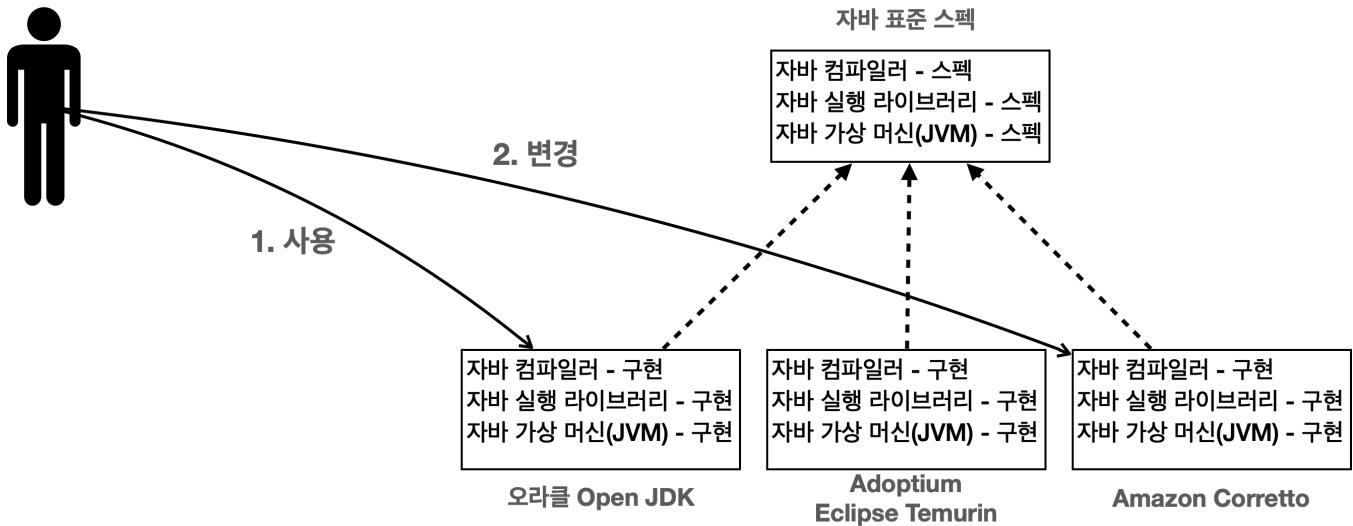
자바는 표준 스펙과 구현으로 나눌 수 있다.

- 자바 표준 스펙
 - 자바는 이렇게 만들어야 한다는 설계도이며, 문서이다.
 - 이 표준 스펙을 기반으로 여러 회사에서 실제 작동하는 자바를 만든다.
 - 자바 표준 스펙은 자바 커뮤니티 프로세스(JCP)를 통해 관리된다.
- 다양한 자바 구현
 - 여러 회사에서 자바 표준 스펙에 맞추어 실제 작동하는 자바 프로그램을 개발한다.
 - 각각 장단점이 있다. 예를 들어 Amazon Corretto는 AWS에 최적화 되어 있다.
 - 각 회사들은 대부분 윈도우, MAC, 리눅스 같이 다양한 OS에서 작동하는 버전의 자바도 함께 제공한다.

참고: 다양한 자바 구현에 대해서는 다음 사이트를 참고하자.

<https://whichjdk.com/ko>

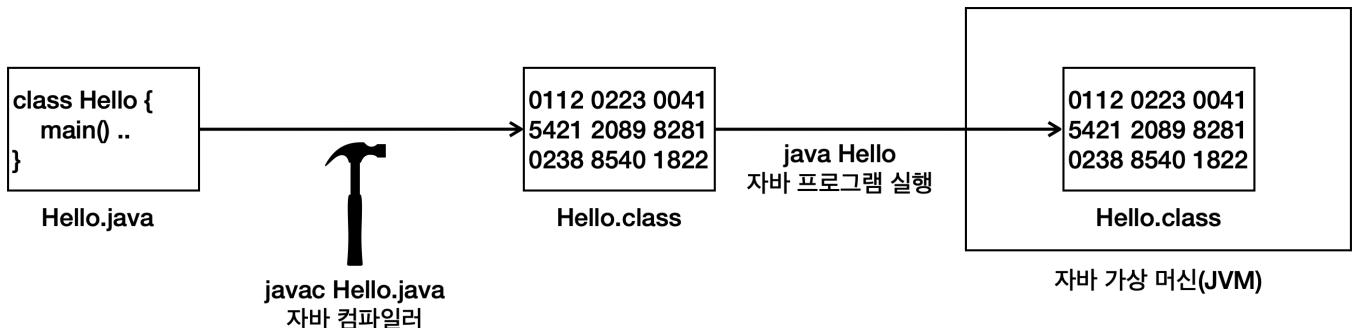
변경의 용이



- 자바 구현들은 모두 표준 스페에 맞도록 개발되어 있다. 따라서 오라클 Open JDK를 사용하다가 Amazon Corretto 자바로 변경해도 대부분 문제 없이 동작한다.

| 참고: 학습 단계에서는 어떤 자바를 사용하든 크게 상관이 없다.

컴파일과 실행

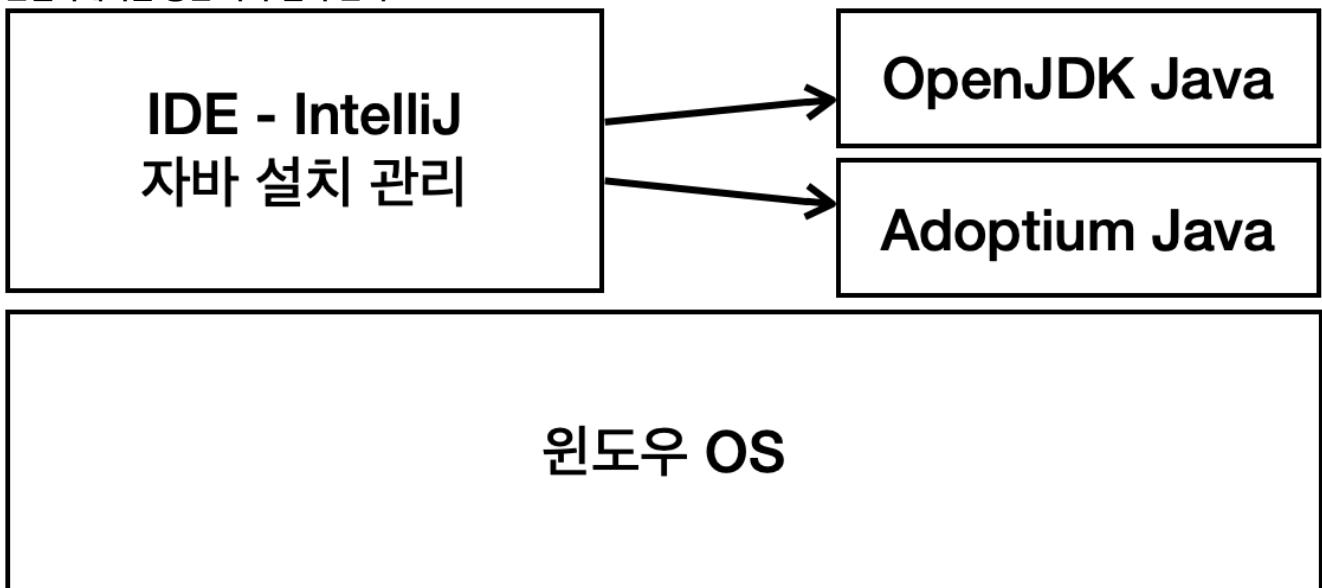


자바 프로그램은 컴파일과 실행 단계를 거친다.

- Hello.java 와 같은 자바 소스 코드를 개발자가 작성한다.
- 자바 컴파일러를 사용해서 소스 코드를 컴파일 한다.
 - 자바가 제공하는 javac라는 프로그램을 사용한다.
 - .java → .class 파일이 생성된다.
 - 자바 소스 코드를 바이트코드로 변환하며 자바 가상 머신에서 더 빠르게 실행될 수 있게 최적화하고 문법 오류도 검출한다.
- 자바 프로그램을 실행한다.
 - 자바가 제공하는 java라는 프로그램을 사용한다.
 - 자바 가상 머신(JVM)이 실행되면서 프로그램이 작동한다.

IDE와 자바

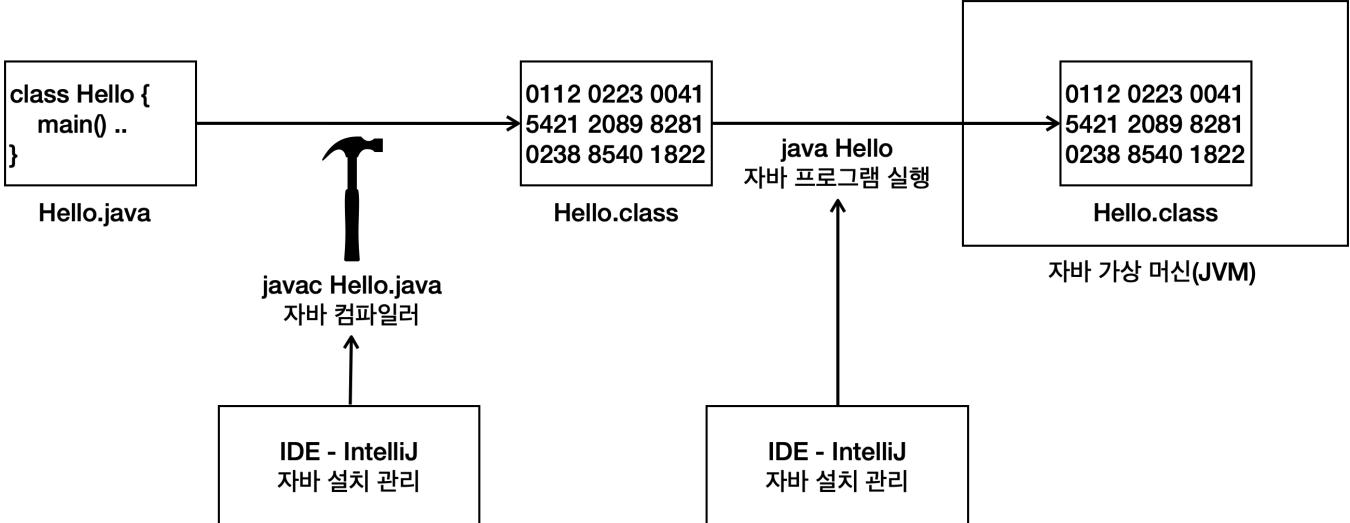
인텔리제이를 통한 자바 설치 관리



- 인텔리제이는 내부에 자바를 편리하게 설치하고 관리할 수 있는 기능을 제공한다.
- 이 기능을 사용하면 인텔리제이를 통해 자바를 편리하게 다운로드 받고 실행할 수 있다.

참고: 자바를 OS에 직접 설치해도 되지만, 처음 프로그래밍을 시작하는 사람에게 이 과정은 매우 번거롭다. 자바를 직접 설치하는 경우 환경 설정이 복잡하다. 그래서 자바를 설치하다가 잘 안되어서 시작도 하기 전에 포기하는 경우가 많다. 자바 언어를 배우는 단계라면 인텔리제이를 통해 자바를 설치하는 정도면 충분하다. 자바를 직접 설치하고 실행하는 내용은 별도로 다룬다.

인텔리제이를 통한 자바 컴파일, 실행 과정



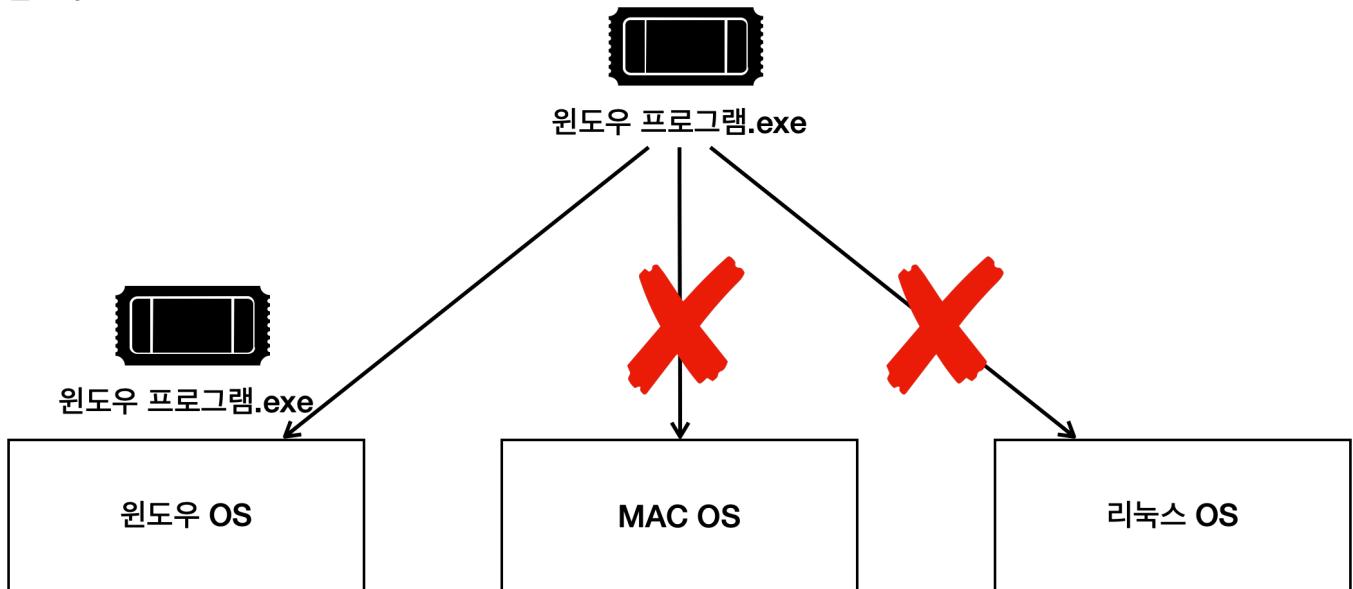
- **컴파일**

- 자바 코드를 컴파일 하려면 `javac`라는 프로그램을 직접 사용해야 하는데, 인텔리제이는 자바 코드를 실행 할 때 이 과정을 자동으로 처리해준다.

- ◆ 예) `javac Hello.java`
- 인텔리제이 화면에서 프로젝트에 있는 `out` 폴더에 가보면 컴파일된 `.class` 파일이 있는 것을 확인할 수 있다.
- 실행
 - 자바를 실행하려면 `java`라는 프로그램을 사용해야 한다. 이때 컴파일된 `.class` 파일을 지정해주면 된다.
 - 예) `java Hello`, 참고로 확장자는 제외한다.
- 인텔리제이에서 자바 코드를 실행하면 컴파일과 실행을 모두 한번에 처리한다.
- 인텔리제이 덕분에 매우 편리하게 자바 프로그램을 개발하고, 학습할 수 있다.

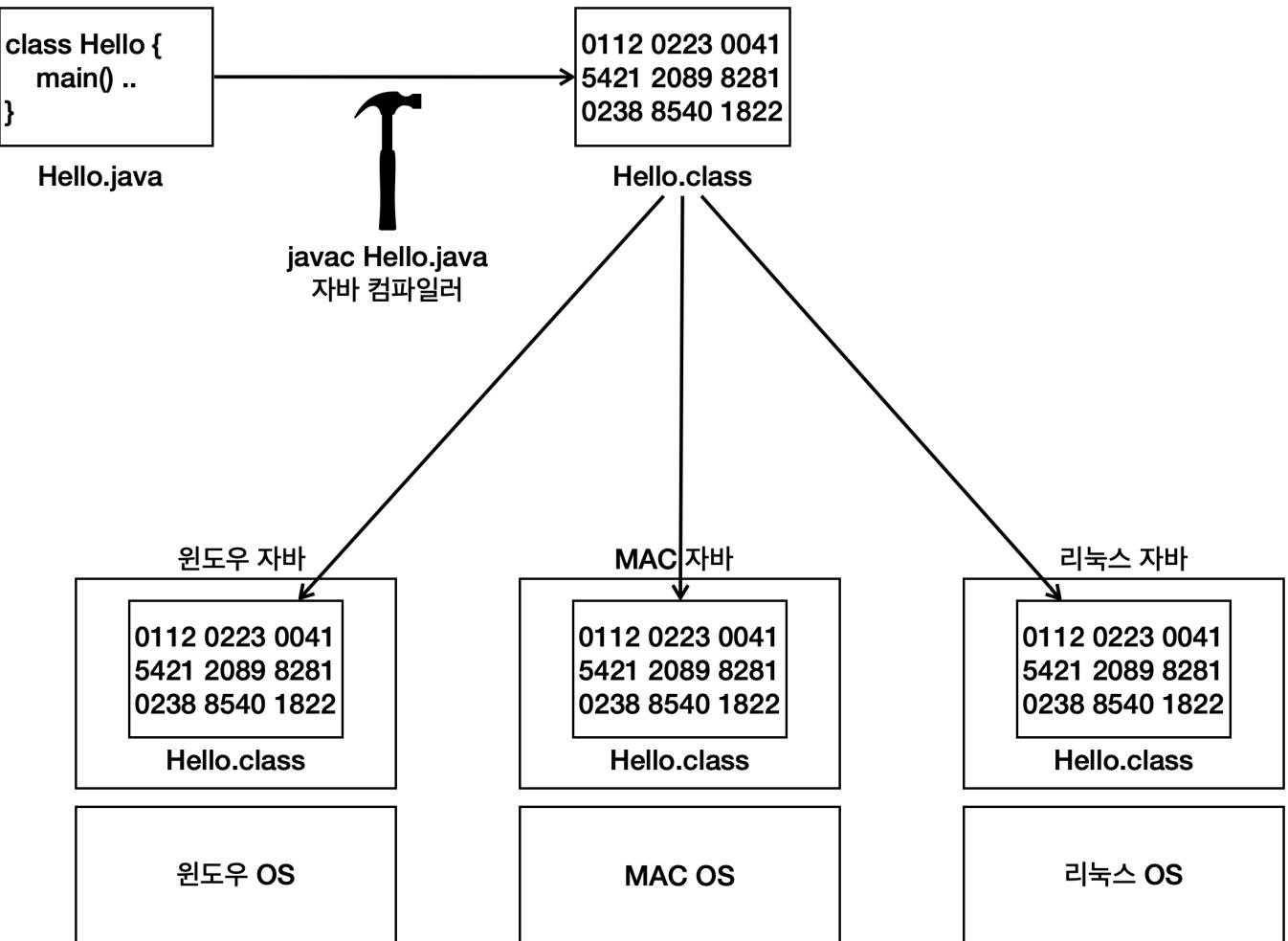
자바와 운영체제 독립성

일반적인 프로그램



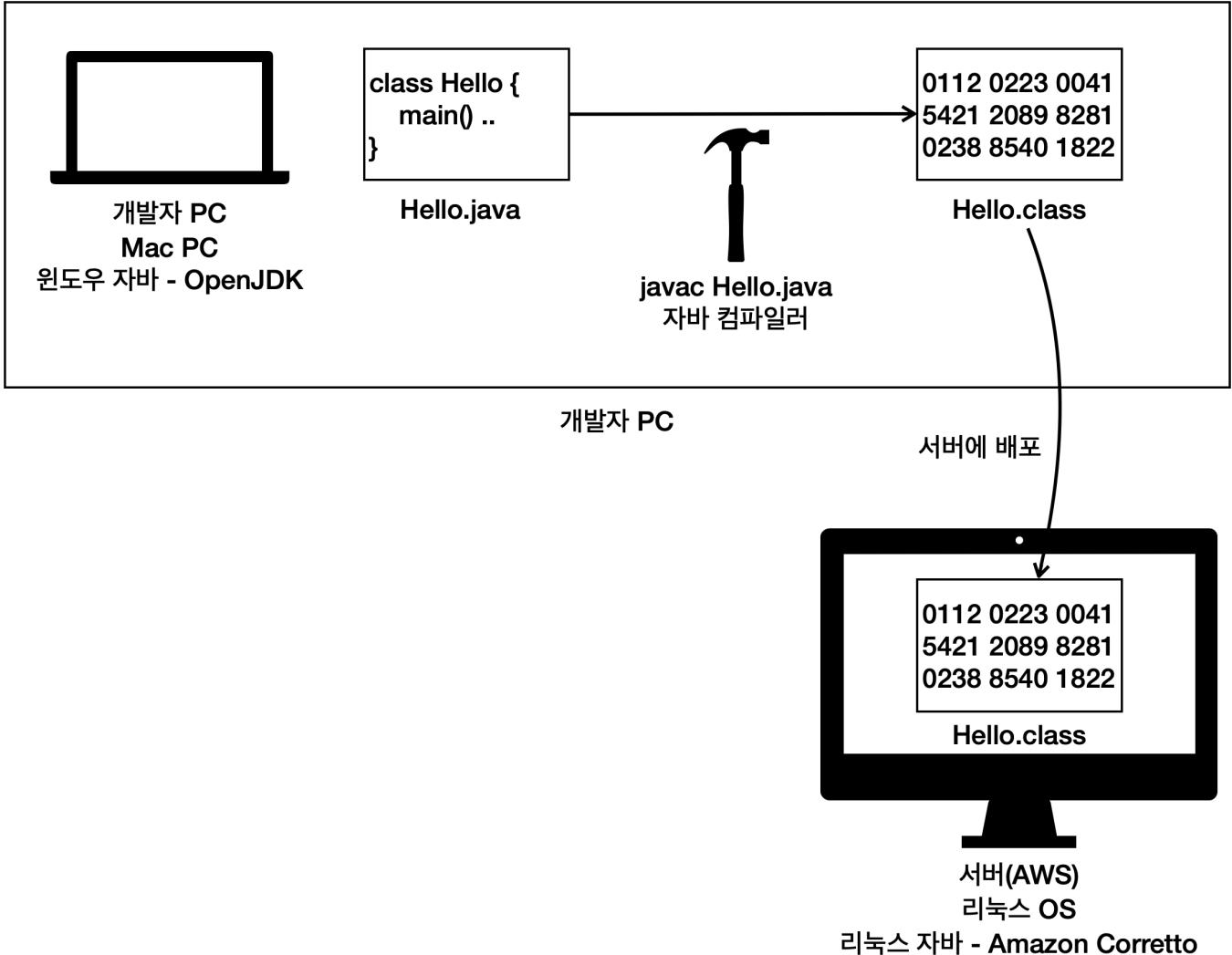
- 일반적인 프로그램은 다른 운영체제에서 실행할 수 없다.
- 예를 들어서 윈도우 프로그램은 MAC이나 리눅스에서 작동하지 않는다.
- 왜냐하면 윈도우 프로그램은 윈도우 OS가 사용하는 명령어들로 구성되어 있기 때문이다. 해당 명령어는 다른 OS와는 호환되지 않는다.

자바 프로그램



- 자바 프로그램은 자바가 설치된 모든 OS에서 실행할 수 있다.
- 자바 개발자는 특정 OS에 맞추어 개발을 하지 않아도 된다. 자바 개발자는 자바에 맞추어 개발하면 된다. OS 호환성 문제는 자바가 해결한다. `Hello.class`와 같이 컴파일된 자바 파일은 모든 자바 환경에서 실행할 수 있다.
- 윈도우 자바는 윈도우 OS가 사용하는 명령어들로 구성되어 있다. MAC이나 리눅스 자바도 본인의 OS가 사용하는 명령어들로 구성되어 있다. 개발자는 각 OS에 맞도록 자바를 설치하기만 하면 된다.

자바 개발과 운영 환경



- 개발할 때 자바와 서버에서 실행할 때 다른 자바를 사용할 수 있다.
- 개발자들은 개발의 편의를 위해서 윈도우나 MAC OS를 주로 사용한다.
- 서버는 주로 리눅스를 사용한다. 만약 AWS를 사용한다면 Amazon Corretto 자바를 AWS 리눅스 서버에 설치하면 된다.
- 자바의 운영체제 독립성 덕분에 각각의 환경에 맞추어 자바를 설치하는 것이 가능하다.

2. 변수

#1.인강/0.자바/1.자바-입문

- /변수 시작
- /변수 값 변경
- /변수 선언과 초기화
- /변수 타입1
- /변수 타입2
- /변수 명명 규칙
- /문제와 풀이
- /정리

변수 시작

변수에 대해서 본격적으로 알아보기 전에 다음 코드를 작성하고 실행해보자.

Var1

```
package variable;

public class Var1 {

    public static void main(String[] args) {
        System.out.println(10);
        System.out.println(10);
        System.out.println(10);
    }
}
```

패키지(package)

- 이번에는 처음으로 패키지를 만든다.
- 패키지는 지금 단계에서는 자바 파일을 구분하기 위한 폴더로 이해하면 된다.
- `variable`라는 패키지를 만들었다면, 해당 패키지에 들어가는 자바 파일 첫줄에 `package variable;`와 같이 소속된 패키지를 선언해주어야 한다.
- 자바 파일이 위치하는 패키지와 `package variable` 선언 위치가 같아야 한다.

실행 결과

```
10  
10  
10
```

단순히 숫자 10을 3번 출력하는 코드이다. 그런데 여기서 숫자 10을 3번 출력하는 대신에 숫자 20을 3번 출력하도록 코드를 변경해보자. 어떻게 해야할까?

Var1

```
package variable;  
  
public class Var1 {  
  
    public static void main(String[] args) {  
        System.out.println(20); //변경 10 -> 20  
        System.out.println(20); //변경 10 -> 20  
        System.out.println(20); //변경 10 -> 20  
    }  
}
```

숫자 10이라고 적혀 있는 곳을 모두 찾아서 숫자 20으로 변경해야 한다. 여기서는 총 3번의 코드 변경이 발생했다. 단순한 예제여서 코드를 3번만 변경했지만, 만약 숫자 10을 출력하는 부분이 100개라면 100개의 코드를 모두 변경해야 한다.

더 나아가서 사용자가 숫자를 입력하고, 사용자가 입력한 숫자를 출력하고 싶다면 어떻게 해야할까? 사용자가 입력한 값은 항상 변한다. 누군가는 100을 입력하고 누군가는 200을 입력할 수도 있다. (사용자 입력은 뒤에서 다룬다) 결국 어딘가에 값을 보관해두고 필요할 때 값을 꺼내서 읽을 수 있는 저장소가 필요하다. 쉽게 비유하자면 데이터를 담을 수 있는 그릇이 필요하다.

모든 프로그래밍 언어는 이런 문제를 해결하기 위해 **변수(variable)**라는 기능을 제공한다. 변수는 이름 그대로 변할 수 있다는 뜻이다.

다음 코드를 작성해보자.

Var2

```
package variable;  
  
public class Var2 {  
  
    public static void main(String[] args) {  
        int a; //변수 선언
```

```
a = 10; //변수 초기화
System.out.println(a);
System.out.println(a);
System.out.println(a);
}
}
```

a = 10 실행 결과

```
10
10
10
```

이번에는 a = 20 으로 변경해서 실행해보자

Var2

```
package variable;

public class Var2 {

    public static void main(String[] args) {
        int a; //변수 선언
        a = 20; //10 -> 20으로 변경
        System.out.println(a);
        System.out.println(a);
        System.out.println(a);
    }
}
```

a = 20 실행 결과

```
20
20
20
```

a의 값을 변경하면 출력결과가 모두 함께 변경되는 것을 확인할 수 있다.

코드를 간단히 분석해보자.

변수 선언

숫자 정수 보관소

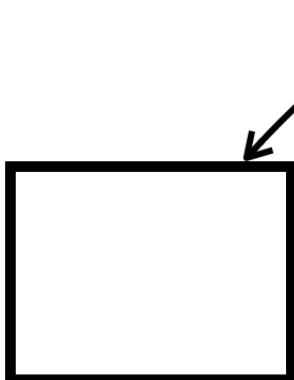


int a

int a

- 숫자 정수(integer)를 보관할 수 있는 이름이 a라는 데이터 저장소를 만든다. 이것을 변수라 한다.
- 이렇게 변수를 만드는 것을 변수 선언이라 한다.
- 이제 변수 a에는 숫자 정수를 보관할 수 있다.
- 숫자 정수 뿐만 아니라 문자, 소수와 같이 다양한 종류 값을 저장할 수 있는 변수들이 있다. 우선은 숫자 정수를 저장하는 int를 알아두자

변수에 값 대입



int a



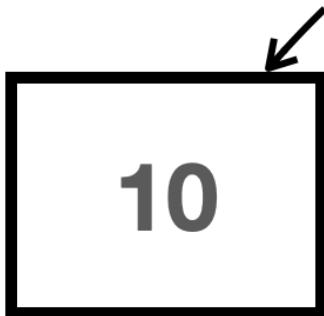
int a

a = 10

- 자바에서 =은 오른쪽에 있는 값을 왼쪽에 저장한다는 뜻이다. 수학에서 이야기하는 두 값이 같다>equals)와는 다른 뜻이다!
- 숫자를 보관할 수 있는 데이터 저장소인 변수 a에 값 10을 저장한다.
- 이처럼 선언한 변수에 처음으로 값을 대입해서 저장하는 것을 변수 초기화라 한다.

변수 값 읽기

a 읽기



int a

```
System.out.println(a)
```

- 변수에 저장되어 있는 값을 읽어서 사용하는 방법은 간단하다. 변수 이름을 적어주기만 하면 된다.
- 변수 a에 10이 들어가 있다면 자바는 실행 시점에 변수의 값을 읽어서 사용한다. 따라서 다음과 같이 해석된다.
 - System.out.println(a) //변수 a의 값을 읽음
 - System.out.println(10) //a의 값인 10으로 변경, 숫자 10 출력
- 참고로 변수의 값을 반복해서 읽을 수 있다. 변수의 값을 읽는다고 값이 없어지는 것이 아니다.

변수 값 변경

변수는 이름 그대로 변할 수 있는 수이다. 쉽게 이야기해서 변수 a에 저장된 값을 언제든지 바꿀 수 있다는 뜻이다.

이번에는 중간에 변수의 값을 변경해보자.

Var3

```
package variable;

public class Var3 {

    public static void main(String[] args) {
        int a; //변수 선언
        a = 10; //변수 초기화: a(10)
        System.out.println(a); //10 출력
        a = 50; //변수 값 변경: a(10 -> 50)
        System.out.println(a); //50 출력
    }
}
```

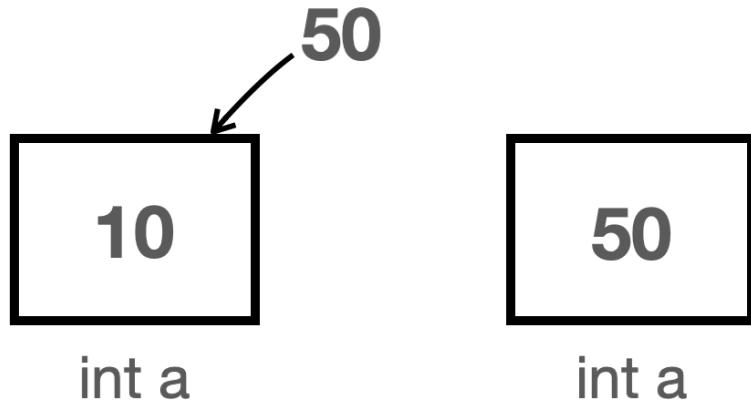
실행 결과

```
10
```

```
50
```

변수의 값이 변경된 이후에는 10 대신에 50이 출력된 것을 확인할 수 있다.

변수 값 변경



프로그램은 한 줄씩 순서대로 실행된다. 어떻게 실행된 것인지 자세히 확인해보자.

```
a = 10; //변수 초기화: a(10) //1. 변수 a에 10을 저장한다.  
System.out.println(a); //2. 변수 a의 값을 읽는다. a에는 10이 들어있다. 10을 출력한다.  
a = 50; //변수 값 변경: //3. 변수 a의 값을 50으로 변경한다. a(10 -> 50)  
System.out.println(a); //4. 변수 a의 값을 읽는다. a에는 50이 들어있다. 50을 출력한다.
```

참고로 변수의 값을 변경하면 변수에 들어있던 기존 값은 삭제된다.

변수 선언과 초기화

변수 선언

변수를 선언하면 컴퓨터의 메모리 공간을 확보해서 그곳에 데이터를 저장할 수 있다. 그리고 변수의 이름을 통해서 해당 메모리 공간에 접근할 수 있다. 쉽게 이야기해서 데이터를 보관할 수 있는 공간을 만들고, 그곳에 이름을 부여한다.

Var4

```
package variable;
```

```
public class Var4 {
```

```
public static void main(String[] args) {  
    int a;  
    int b;  
  
    int c, d;  
}  
}
```

변수 선언



int a



int b



int c



int d

변수는 다음과 같이 하나씩 선언할 수도 있고

```
int a;  
int b;
```

다음과 같이 한번에 여러 변수를 선언할 수도 있다.

```
int c, d;
```

변수 초기화

변수를 선언하고, 선언한 변수에 처음으로 값을 저장하는 것을 변수 초기화라 한다.

Var5

```
package variable;  
  
public class Var5 {  
  
    public static void main(String[] args) {  
        //1. 변수 선언, 초기화 각각 따로  
        int a;  
        a = 1;  
        System.out.println(a);  
  
        int b = 2; //2. 변수 선언과 초기화를 한번에  
        System.out.println(b);  
    }  
}
```

```

        int c = 3, d = 4; //3. 여러 변수 선언과 초기화를 한번에
        System.out.println(c);
        System.out.println(d);
    }
}

```

1. 변수의 선언과 초기화를 각각 따로 할 수 있다.
2. 변수를 선언하면서 동시에 초기화 할 수 있다.

`int b` 를 사용해서 변수 `b` 를 만들고 그 다음에 바로 `b = 2` 를 사용해서 변수 `b` 에 값 2를 저장한다.

3. 여러 변수를 선언하면서 초기화도 동시에 진행할 수 있다.

변수는 초기화 해야한다

만약 변수를 초기화 하지 않고 사용하면 어떻게 될까?

Var6

```

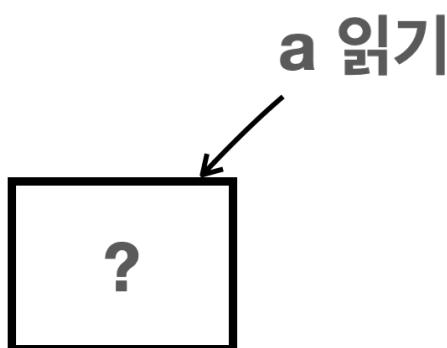
package variable;

public class Var6 {

    public static void main(String[] args) {
        int a;
        System.out.println(a); //주석을 풀면 컴파일 에러 발생
    }
}

```

초기화 하지 않은 변수 읽기



`int a`

다음과 같은 컴파일 에러가 발생한다.

```
java: variable a might not have been initialized
```

해석해보면 변수가 초기화되지 않았다는 오류이다.

왜 이런 오류가 발생할까? 컴퓨터에서 메모리는 여러 시스템이 함께 사용하는 공간이다. 그래서 어떠한 값들이 계속 저

장된다.

변수를 선언하면 메모리상의 어떤 공간을 차지하고 사용한다. 그런데 그 공간에 기존에 어떤 값이 있었는지는 아무도 모른다. 따라서 초기화를 하지 않으면 이상한 값이 출력될 수 있다. 이런 문제를 예방하기 위해 자바는 변수를 초기화 하도록 강제한다.

참고: 지금 학습하는 변수는 지역 변수(Local Variable)라고 하는데, 지역 변수는 개발자가 직접 초기화를 해주어야 한다. 나중에 배울 클래스 변수와 인스턴스 변수는 자바가 자동으로 초기화를 진행해준다.

참고: 컴파일 에러는 자바 문법에 맞지 않았을 때 발생하는 에러이다. 컴파일 에러는 오류를 빨리, 그리고 명확하게 찾을 수 있기 때문에 사실은 좋은 에러이다. 덕분에 빠르게 버그를 찾아서 고칠 수 있다.

에러를 확인하고 나면 꼭 다음과 같이 해당 라인 전체에 주석을 적용하자. 그렇지 않으면 다른 예제를 실행할 때도 이 부분에 컴파일 에러가 발생할 수 있다.

```
//System.out.println(a); //주석을 풀면 컴파일 오류 발생
```

변수 타입1

변수는 데이터를 다루는 종류에 따라 다양한 형식이 존재한다.

다음 코드를 실행해보자.

Var7

```
package variable;

public class Var7 {

    public static void main(String[] args) {
        int a = 100; //정수
        double b = 10.5; //실수
        boolean c = true; //불리언(boolean) true, false 입력 가능
        char d = 'A'; //문자 하나
        String e = "Hello Java"; //문자열, 문자열을 다루기 위한 특별한 타입

        System.out.println(a);
        System.out.println(b);
        System.out.println(c);
        System.out.println(d);
```

```
        System.out.println(e);  
    }  
}
```

실행 결과

```
100  
10.5  
true  
A  
Hello Java
```

변수는 데이터를 다루는 종류에 따라 다양한 형식이 존재한다. 이러한 형식을 영어로는 **타입** (type)이라 하고, 우리말로는 **형식** 또는 **형**이라 한다. 예를 들어서 **int** 타입, **int** 형식, **int** 형 등으로 부른다. 특별히 구분하지 않고 섞어서 부르기 때문에 모두 같은 말로 이해하면 된다.

변수 타입의 예

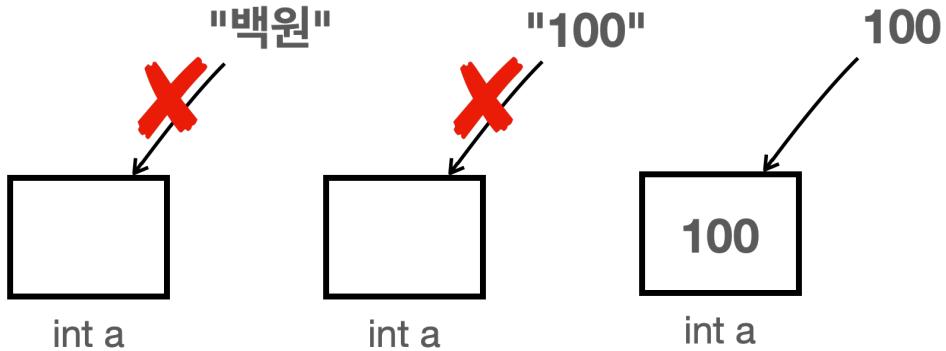
100	10.5	true	'A'	"Hello Java"
int a	double b	boolean c	char d	String e

- **int**: 정수를 다룬다. 예) 1, 100, 1000
- **double**: 실수를 다룬다. 예) 0.2, 1.5, 100.121
- **boolean**: 불리언 타입이라 한다. **true**, **false** 값만 사용할 수 있다. 주로 참과 거짓을 판단하는 곳에서 사용 한다.
- **char**: 문자 하나를 다룰 때 사용한다. 작은따옴표(')를 사용해서 감싸야 한다. 예) 'A', '가'
- **String**: 문자열을 다룬다. 큰따옴표를 사용해야 한다. 예) "hello java"

참고: **String**은 첫 글자가 대문자로 시작하는 특별한 타입이다. 이 부분은 뒤에 클래스를 배워야 자세히 이해할 수 있다. 지금은 문자열을 다루는 특별한 타입이라고 이해하면 된다. **String**에 대한 자세한 내용은 별도로 다룬다.

자신의 타입에 맞는 데이터 사용

각 변수는 지정한 타입에 맞는 값을 사용해야 한다. 예를 들어서 다음의 앞의 두 코드는 컴파일 오류가 발생한다



- `int a = "백원"` : 정수 타입에 문자열(X)
- `int a = "100"` : 정수 타입에 문자열(X), 이것은 숫자 100이 아니라 문자열 `"100"` 이다. 문자를 나타내는 쌍따옴표(")로 감싸져 있다.
- `int a = 100` : 정수 타입에 정수 100(O)

리터럴

코드에서 개발자가 직접 적은 `100`, `10.5`, `true`, `'A'`, `"Hello Java"` 와 같은 고정된 값을 프로그래밍 용어로 리터럴(literal)이라 한다.

```
int a = 100; //정수 리터럴
double b = 10.5; //실수 리터럴
boolean c = true; //불리언 리터럴
char d = 'A'; //문자 하나 리터럴
String e = "Hello Java"; //문자열 리터럴
```

변수의 값은 변할 수 있지만 리터럴은 개발자가 직접 입력한 고정된 값이다. 따라서 리터럴 자체는 변하지 않는다.

| 참고: 리터럴(literal)이라는 단어의 어원이 문자 또는 글자를 의미한다.

변수 타입2

숫자 타입

이번에는 다양한 숫자 타입을 자세히 알아보자.

바로 다음 코드를 작성해보자.

Var8

```
package variable;

public class Var8 {
```

```

public static void main(String[] args) {
    //정수
    byte b = 127; // -128 ~ 127
    short s = 32767; // -32,768 ~ 32,767
    int i = 2147483647; // -2,147,483,648 ~ 2,147,483,647 (약 20억)

    // -9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807
    long l = 9223372036854775807L;

    //실수
    float f = 10.0f;
    double d = 10.0;
}
}

```

메모리를 적게 사용하면 작은 숫자를 표현할 수 있고, 메모리를 많이 사용하면 큰 숫자를 표현할 수 있다.
변수를 선언하면 표현 범위에 따라 메모리 공간을 차지한다. 그래서 필요에 맞도록 다양한 타입을 제공한다.

변수와 메모리 공간 크기

2^8

127

byte b

2^{16}

32,767

short s

2^{32}

2,147,483,647

int i

2^{64}

9,223,372,036,854,775,807

long l

표현할 수 있는 숫자의 범위와 차지하는 메모리 공간은 다음과 같다. 기타 타입도 함께 정리해두었다.

- 정수형

- byte : -128 ~ 127 (1byte, 2^8)
- short : -32,768 ~ 32,767 (2byte, 2^{16})
- int : -2,147,483,648 ~ 2,147,483,647 (약 20억) (4byte, 2^{32})
- long : -9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807 (8byte, 2^{64})
- 실수형
 - float : 대략 -3.4E38 ~ 3.4E38, 7자리 정밀도 (4byte, 2^{32})
 - double : 대략 -1.7E308 ~ 1.7E308, 15자리 정밀도 (8byte, 2^{64})
- 기타
 - boolean : true, false (1byte)
 - char : 문자 하나(2byte)
 - String : 문자열을 표현한다. 메모리 사용량은 문자 길이에 따라 동적으로 달라진다. (특별한 타입이다. 자세한 내용은 뒤에서 학습한다)

리터럴 타입 지정

- 정수 리터럴은 int를 기본으로 사용한다. 따라서 int 범위까지 표현할 수 있다. 숫자가 int 범위인 약 20억을 넘어가면 L을 붙여서 정수 리터럴을 long 으로 변경해야 한다. (대문자 L, 소문자 l 모두 가능하다 그런데 소문자 l은 숫자 1과 착각할 수 있어서 권장하지 않는다.)
- 실수 리터럴은 기본이 double 형을 사용한다. float 형을 사용하려면 f를 붙여서 float 형으로 지정해야 한다.

변수 타입 정리

이렇게 많은 변수들을 실제로 다 외우고 사용해야 할까?

다음 타입은 실무에서 거의 사용하지 않는다.

- byte : 표현 길이가 너무 작다. 또 자바는 기본으로 4byte(int)를 효율적으로 계산하도록 설계되어 있다. int 를 사용하자.
 - byte 타입을 직접 선언하고 여기에 숫자 값을 대입해서 계산하는 일은 거의 없다.
 - 대신에 파일을 바이트 단위로 다루기 때문에 파일 전송, 파일 복사 등에 주로 사용된다.
- short : 표현 길이가 너무 작다. 또 자바는 기본으로 4byte(int)를 효율적으로 계산하도록 설계되어 있다. int 를 사용하자
- float : 표현 길이와 정밀도가 낮다. 실수형은 double 을 사용하자.
- char : 문자 하나를 표현하는 일은 거의 없다. 문자 하나를 표현할 때도 문자열을 사용할 수 있다.
 - 예를 들어 String a = "b" 와 같이 사용하면 된다.

참고: 메모리 용량은 매우 저렴하다. 따라서 메모리 용량을 약간 절약하기 보다는 개발 속도나 효율에 초점을 맞추는 것이 더 효과적이다.

자주 사용하는 타입

실무에서 자주 사용하는 타입은 다음과 같다.

- 정수 - `int`, `long`: 자바는 정수에 기본으로 `int`를 사용한다. 만약 20억이 넘을 것 같으면 `long`을 쓰면 된다.
 - 파일을 다룰 때는 `byte`를 사용한다.
- 실수 - `double`: 실수는 고민하지 말고 `double`을 쓰면 된다.
- 불린형 - `boolean`: `true`, `false` 참 거짓을 표현한다. 이후 조건문에서 자주 사용된다.
- 문자열 - `String`: 문자를 다룰 때는 문자 하나든 문자열이든 모두 `String`을 사용하는 것이 편리하다.

자주 사용하는 타입을 제외하고 실무에서 나머지를 사용하는 경우는 거의 없다. 그나마 파일 전송시에 `byte`를 사용하는 것 정도이다.

따라서 자주 사용하는 타입만 이해하고 나머지는 이런게 있구나 하고 넘어가도 충분하다.

변수 명명 규칙

자바에서 변수의 이름을 짓는데는 규칙과 관례가 있다.

규칙은 필수이다. 규칙을 지키지 않으면 컴파일 오류가 발생한다.

관례는 필수는 아니지만 전세계 개발자가 해당 관례를 따르기 때문에 사실상 규칙이라고 생각해도 된다.

규칙

- 변수 이름은 숫자로 시작할 수 없다. (예: `1num`, `1st`)
 - 그러나 숫자를 이름에 포함하는 것은 가능하다 (예: `myVar1`, `num1`).
- 이름에는 공백이 들어갈 수 없다.
- 자바의 예약어를 변수 이름으로 사용할 수 없다. (예: `int`, `class`, `public`)
- 변수 이름에는 영문자(`a-z`, `A-Z`), 숫자(`0-9`), 달러 기호(\$) 또는 밑줄(_)만 사용할 수 있다.

관례

- 소문자로 시작하는 낙타 표기법
 - 변수 이름은 소문자로 시작하는 것이 일반적이다. 여러 단어로 이루어진 변수 이름의 경우, 첫 번째 단어는 소문자로 시작하고 그 이후의 각 단어는 대문자로 시작하는 낙타 표기법(camel case)를 사용한다. (예: `orderDetail`, `myAccount`)

낙타표기법

낙타표기법(Camel Case)은 프로그래밍에서 변수, 함수, 클래스 등의 이름을 지을 때 많이 사용하는 표기법 중 하나이다. 이 표기법의 이름은 작성한 이름이 여러 단어로 구성되어 있을 때, 각 단어의 첫 글자가 대문자로 시작하고, 이 대문자들이 낙타의 등봉처럼 보이는 것에서 유래했다. 낙타표기법을 사용하면 이름에 공백을 넣지 않고도 여러 단어를 쉽게 구분할 수 있으므로, 변수의 이름을 이해하기 쉽게 만들어준다. 또한, 대부분의 프로그래밍 언어에서는 이름에 공백을 포함할 수 없기 때문에, 낙타표기법은 이런 제한을 우회하는 좋은 방법이다.

자바 언어의 관례 한번에 정리

클래스는 대문자로 시작, 나머지는 소문자로 시작

- 자바에서 클래스 이름의 첫 글자는 대문자로 시작한다. 그리고 나머지는 모두 첫 글자를 소문자로 시작한다. 여기에 낙타 표기법을 적용하면 된다. 이렇게 하면 모든 자바 관례를 다 외울 수 있다!
- 예시: 클래스는 첫 글자 대문자, 나머지는 모두 첫 글자 소문자로 시작 + 낙타 표기법
 - 클래스: Person, OrderDetail
 - 변수를 포함한 나머지: firstName, userAccount
- 여기에 예외가 딱 2개 있다.
 - 상수는 모두 대문자를 사용하고 언더바로 구분한다. (상수는 뒤에서 학습)
 - ◆ USER_LIMIT
 - 패키지는 모두 소문자를 사용한다. (패키지는 뒤에서 학습)
 - ◆ org.springframework.boot

참고: 변수 이름은 의미있고, 그 용도를 명확하게 설명해야 한다.

- a, b : 이런 변수는 용도를 설명하지 않는다. 단순한 예제에서만 사용하는 것이 좋다.
- studentCount, maxScore, userAccount, orderCount : 용도를 명확하게 설명한다.

문제와 풀이

- 문제를 스스로 풀어보세요.

백문이 불여일타!

프로그래밍은 수영이나 자전거 타기와 비슷하다. 아무리 책을 보고 강의 영상을 봐도 직접 물속에 들어가거나 자

전거를 타봐야 이해할 수 있다. 프로그래밍도 마찬가지이다. 이해하고 외우는 것도 중요하지만 무엇보다 직접 코딩을 해보는 것이 더욱 중요하다. 읽어서 이해가 잘 안되더라도 직접 코딩을 해보면 자연스럽게 이해가 되는 경우도 많다. 백번 읽는 것 보다 한번 직접 코딩해서 결과를 보는 것이 좋은 개발자로 빠르게 성장할 수 있는 지름길이다.

코딩이 처음이라면 필독!

프로그래밍이 처음이라면 아직 코딩 자체가 익숙하지 않기 때문에 문제와 풀이에 상당히 많은 시간을 쓰게 될 수 있다. 강의를 들을 때는 다 이해가 되는 것 같았는데, 막상 혼자 생각해서 코딩을 하려니 잘 안되는 것이다. 이것은 아직 코딩이 익숙하지 않기 때문인데, 처음 코딩을 하는 사람이라면 누구나 겪는 자연스러운 현상이다. 문제를 스스로 풀기 어려운 경우, 너무 고민하기보다는 먼저 **강의 영상의 문제 풀이 과정을 코드로 따라하면서 이해하자. 반드시 코드로 따라해야 한다.** 그래야 코딩하는 것에 조금씩 익숙해질 수 있다. 그런 다음에 정답을 지우고 스스로 문제를 풀어보면 된다. 참고로 강의를 듣는 시간만큼 문제와 풀이에도 많은 시간을 들어야 제대로 성장 할 수 있다!

문제1

다음 코드에 반복해서 나오는 숫자 4, 3을 다른 숫자로 한번에 변경할 수 있도록 다음을 변수 num1, num2를 사용하도록 변경해보세요.

VarEx1

```
package variable.ex;

public class VarEx1Question {
    public static void main(String[] args) {
        System.out.println(4 + 3);
        System.out.println(4 - 3);
        System.out.println(4 * 3);
    }
}
```

문제1 - 정답

VarEx1

```
package variable.ex;

public class VarEx1 {
    public static void main(String[] args) {
        int num1 = 4;
        int num2 = 3;
```

```
        System.out.println(num1 + num2);
        System.out.println(num1 - num2);
        System.out.println(num1 * num2);
    }
}
```

실행 결과

```
7
1
12
```

문제2

다음과 같은 작업을 수행하는 프로그램을 작성하세요.

클래스 이름은 VarEx2 라고 적어주세요.

1. 변수 num1 을 선언하고, 이에 10을 할당하세요.
2. 변수 num2 를 선언하고, 이에 20을 할당하세요.
3. 두 변수의 합을 구하고, 그 결과를 새로운 변수 sum에 저장하세요.
4. sum 변수의 값을 출력하세요.

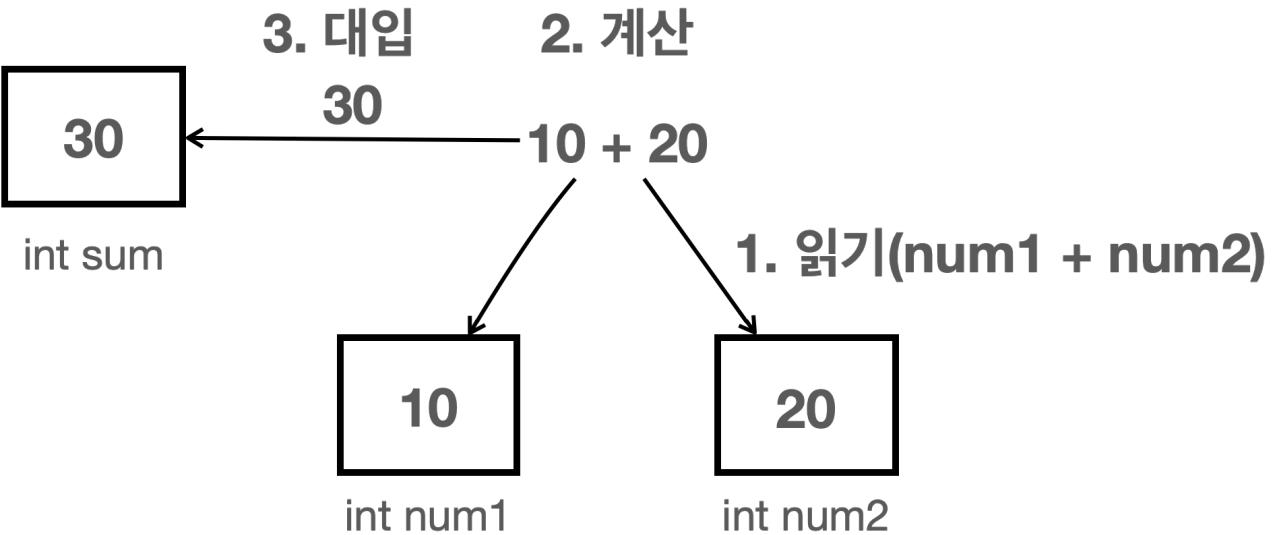
문제2 - 정답

VarEx2

```
package variable.ex;

public class VarEx2 {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 20;
        int sum = num1 + num2;
        System.out.println(sum);
    }
}
```

int sum = num1 + num2 계산 과정



문제3 - long, boolean 데이터 타입

클래스 이름: VarEx3

`long` 타입의 변수를 선언하고, 그 변수를 10000000000(백억)으로 초기화한 후 출력하는 프로그램을 작성하세요.
`boolean` 타입의 변수를 선언하고, 그 변수를 `true`로 초기화한 후 출력하는 프로그램을 작성하세요.

정답 - long, boolean 데이터 타입

```

package variable.ex;

public class VarEx3 {
    public static void main(String[] args) {
        long longVar = 10000000000L;
        System.out.println(longVar);

        boolean booleanVar = true;
        System.out.println(booleanVar);
    }
}
    
```

실행 결과

10000000000

true

정리

3. 연산자

#1.인강/0.자바/1.자바-입문

- /산술 연산자
- /문자열 더하기
- /연산자 우선순위
- /증감 연산자
- /비교 연산자
- /논리 연산자
- /대입 연산자
- /문제와 풀이
- /정리

산술 연산자

연산자 시작

+, -, *, / 와 같이 계산을 수행하는 기호를 연산자라 한다. 자바에는 다음과 같은 다양한 연산자가 있다. 참고로 더 많은 연산자가 있지만, 여기서는 실무에서 주로 다루는 연산자 위주로 설명하겠다.

연산자 종류

- 산술 연산자: +, -, *, /, %(나머지 연산자)
- 증감(증가 및 감소) 연산자: ++, --
- 비교 연산자: ==, !=, >, <, >=, <=
- 논리 연산자: && (AND), || (OR), ! (NOT)
- 대입 연산자: =, +=, -=, *=, /=, %=
- 삼항 연산자: ? :

연산자와 피연산자

3 + 4
a + b

- 연산자(operator): 연산 기호 - 예) +, -
- 피연산자(operand): 연산 대상 - 예) 3, 4, a, b

산술 연산자

산술 연산자는 주로 숫자를 계산하는 데 사용된다. 우리가 이미 잘 알고 있는 수학 연산을 수행한다.

- + (더하기)
- - (빼기)
- * (곱하기)
- / (나누기)
- % (나머지)

다음 코드를 작성해보자.

Operator1

```
package operator;

public class Operator1 {

    public static void main(String[] args) {
        // 변수 초기화
        int a = 5;
        int b = 2;

        // 덧셈
        int sum = a + b;
        System.out.println("a + b = " + sum);    // 출력: a + b = 7

        // 뺄셈
        int diff = a - b;
        System.out.println("a - b = " + diff);    // 출력: a - b = 3

        // 곱셈
        int multi = a * b;
        System.out.println("a * b = " + multi);    // 출력: a * b = 10

        // 나눗셈
        int div = a / b;
        System.out.println("a / b = " + div);    // 출력: a / b = 2

        // 나머지
        int mod = a % b;
        System.out.println("a % b = " + mod);    // 출력: a % b = 1
    }
}
```

```
}
```

int sum = a + b 계산 과정

```
int sum = a + b //1. 변수 값 읽기  
int sum = 5 + 2 //2. 변수 값 계산  
int sum = 7 //3. 계산 결과를 sum에 대입  
sum = 7 //최종 결과
```

실행 결과

```
a + b = 7  
a - b = 3  
a * b = 10  
a / b = 2  
a % b = 1
```

- 5 / 2의 결과는 2.5가 되어야 하지만 결과는 소수점이 제거된 2가 나왔다.
 - 자바에서 같은 int 형끼리 계산하면 계산 결과도 같은 int 형을 사용한다. int 형은 정수이기 때문에 소수점 이하를 포함할 수 없다.
 - 이 부분에 대한 자세한 내용과 해결 방안은 뒤의 형변환에서 다룬다.
- 나머지 연산자(%)
 - 이를 그대로 나머지를 구하는 연산자이다. 5 / 2는 몫이 2 나머지가 1이다. 따라서 나머지 연산자 5 % 2의 결과는 1이 된다.
 - 나머지 연산자는 실무와 알고리즘 모두 종종 사용되므로 잘 기억해두자

주의! 0으로 나누기

10 / 0과 같이 숫자는 0으로 나눌 수 없다. (수학에서 허용하지 않음)

방금 예제에서 변수 b = 0을 대입하면 5 / 0이 된다. 이 경우 프로그램에 오류가 발생한다.

실행하면 다음과 같은 예외를 확인할 수 있다.

```
Exception in thread "main" java.lang.ArithmaticException: / by zero
```

예외가 발생하면 해당 시점 이후의 결과가 출력되지 않고 프로그램이 종료된다. 예외에 대한 자세한 부분은 예외 처리에서 다룬다.

문자열 더하기

자바는 특별하게 문자열에도 + 연산자를 사용할 수 있다. 문자열에 + 연산자를 사용하면 두 문자를 연결할 수 있다.

다음 코드를 작성해보자.

```
package operator;

public class Operator2 {

    public static void main(String[] args) {

        //문자열과 문자열 더하기1
        String result1 = "hello " + "world";
        System.out.println(result1);

        //문자열과 문자열 더하기2
        String s1 = "string1";
        String s2 = "string2";
        String result2 = s1 + s2;
        System.out.println(result2);

        //문자열과 숫자 더하기1
        String result3 = "a + b = " + 10;
        System.out.println(result3);

        //문자열과 숫자 더하기2
        int num = 20;
        String str = "a + b = ";
        String result4 = str + num;
        System.out.println(result4);
    }

}
```

실행 결과

```
hello world
string1string2
a + b = 10
a + b = 20
```

문자열과 문자열 더하기1

```
String result1 = "hello " + "world"
```

- "hello " 문자열과 "world" 문자열을 더해서 "hello world" 문자열을 만든다.

- 결과를 `result1`에 저장한다.

문자열과 문자열 더하기2

```
String result2 = s1 + s2
```

- `s1`과 `s2` 변수에 있는 문자열을 읽는다.
- `"string1" + "string2"` 연산을 수행해서 `"string1string2"` 문자열을 만든다.
- 결과를 `result2`에 저장한다.

문자열과 숫자 더하기1

다음 식은 문자열과 숫자를 더한다. 자바에서 문자와 숫자를 더하면 숫자를 문자열로 변경한 다음에 서로 더한다.

```
"a + b = " + 10
```

- 문자: `"a + b = "`
- 숫자: `10`

계산 과정

```
"a + b = "(String) + 10(int) //문자열과 숫자 더하기
"a + b = "(String) + "10"(int -> String) //숫자를 문자열로 변경
"a + b = " + "10" //문자열과 문자열 더하기
"a + b = 10" //결과
```

문자열과 숫자 더하기2

변수에 담겨 있어도 문자와 숫자를 더하면 문자가 된다. 계산 과정을 확인해보자.

계산 과정

```
str(String) + num(int)
"a + b = "(String) + num(int) //str 변수에서 값 조회
"a + b = "(String) + 20(int) //num 변수에서 값 조회
"a + b = "(String) + "20"(int -> String) //숫자를 문자열로 변경
"a + b = " + "20" //문자열과 문자열 더하기
"a + b = 20" //결과
```

자바는 문자열인 `String` 타입에 다른 타입을 더하는 경우 대상 타입을 문자열로 변경한다. 쉽게 이야기해서 문자열에 더하는 것은 다 문자열이 된다.

연산자 우선순위

수학에서 $1 + 2 * 3$ 의 연산 결과는 무엇일까? 덧셈보다 곱셈이 우선순위가 더 높다. 따라서 다음과 같이 계산한다.

```
1 + (2 * 3) //곱셈(*)이 연산자 우선순위가 높다. 따라서 먼저 계산한다.  
1 + 6  
7 //결과
```

자바도 마찬가지이다. 코드로 연산자 우선순위를 확인해보자.

```
package operator;  
  
public class Operator3 {  
  
    public static void main(String[] args) {  
        int sum1 = 1 + 2 * 3; //1 + (2 * 3)과 같다.  
        int sum2 = (1 + 2) * 3;  
        System.out.println("sum1 = " + sum1); //sum1 = 7  
        System.out.println("sum2 = " + sum2); //sum2 = 9  
    }  
  
}
```

실행 결과

```
sum1 = 7  
sum2 = 9
```

- 출력 결과를 보면 `sum1 = 7`이 나왔다. 연산자 우선순위에 의해 곱셈이 먼저 계산된 것이다.
- 연산자 우선순위를 변경하려면 수학과 마찬가지로 괄호 `()`를 사용하면 된다. `()`를 사용한 곳이 먼저 계산된다.
- `sum2`는 괄호를 사용해서 덧셈이 먼저 처리되도록 했다.

sum2 계산 순서

```
(1 + 2) * 3  
3 * 3  
9
```

이번에는 조금 더 복잡한 예제를 만들어보자.

```
package operator;  
  
public class Operator4 {  
  
    public static void main(String[] args) {  
        int sum3 = 2 * 2 + 3 * 3; //(2 * 2) + (3 * 3)  
        int sum4 = (2 * 2) + (3 * 3); //sum3과 같다.  
        System.out.println("sum3 = " + sum3); //sum3 = 13
```

```
        System.out.println("sum4 = " + sum4); //sum4 = 13
    }

}
```

실행 결과

```
sum3 = 13
sum4 = 13
```

sum3, sum4에 저장하는 두 연산은 같은 연산이다. 그런데 괄호가 없는 $2 * 2 + 3 * 3$ 연산은 평소 수학을 잘 하는 분들은 금방 풀겠지만 보통은 이 연산을 보고 잠깐 연산자 우선순위를 생각을 해야한다.

```
2 * 2 + 3 * 3
(2 * 2) + (3 * 3) //곱셈이 우선순위가 높다
4 + 9
13
```

이렇게 복잡한 경우 sum4의 $(2 * 2) + (3 * 3)$ 와 같이 괄호를 명시적으로 사용하는 것이 더 명확하고 이해하기 쉽다.

코드를 몇자 줄여서 모호하거나 복잡해지는 것 보다는 코드가 더 많더라도 명확하고 단순한 것이 더 유지보수 하기 좋다.

연산자 우선순위가 애매하거나 조금이라도 복잡하다면 언제나 괄호를 고려하자!

연산자 우선순위 암기법

자바는 다음과 같은 연산자 우선순위가 있다. 높은 것에서 낮은 순으로 적었다. 처음에 나오는 괄호 () 가 우선순위가 가장 높고, 마지막의 대입 연산자(=)가 우선순위가 가장 낮다.

1. 괄호()
2. 단항 연산자(예: ++, --, !, ~, new, (type))
3. 산술 연산자(*, /, % 우선, 그 다음에 +, -)
4. Shift 연산자(<<, >>, >>>)
5. 비교 연산자(<, <=, >, >=, instanceof)
6. 등식 연산자(==, !=)
7. 비트 연산자(&, ^, |)
8. 논리 연산자(&&, ||)
9. 삼항 연산자(?:)

10. 대입 연산자 (=, +=, -=, *=, /=, %= 등등)

그러면 이 많은 우선순위를 어떻게 외워야 할까? 사실 대부분의 실무 개발자들은 연산자 우선순위를 외우지 않는다.

연산자 우선순위는 딱 2가지만 기억하면 된다.

1. 상식선에서 우선순위를 사용하자

우선순위는 상식선에서 생각하면 대부분 문제가 없다.

다음 예를 보자

```
int sum = 1 + 2 * 3
```

당연히 + 보다 * 이 우선순위가 높다.

다음으로 산술 연산자(+)와 대입연산자(=)를 비교하는 예를 보자.

```
int sum = 1 + 2
```

```
int sum = 1 + 2
```

```
int sum = 3 //산술 연산자가 먼저 처리된다.
```

```
sum = 3 //대입 연산자가 마지막에 처리된다.
```

- 1 + 2 를 먼저 처리한 다음에 그 결과 값을 변수 sum에 넣어야 한다. 대입 연산자인 = 이 먼저 수행된다고 생각하기가 사실 더 어렵다.

2. 애매하면 괄호()를 사용하자

코드를 딱 보았을 때 연산자 우선순위를 고민해야 할 것 같으면, 그러니까 뭔가 복잡해보이면 나 뿐만 아니라 모든 사람이 그렇게 느낀다. 이때는 다음과 같이 괄호를 사용해서 우선순위를 명시적으로 지정하면 된다.

```
((2 * 2) + (3 * 3)) / (3 + 2)
```

정리

- 연산자 우선순위는 상식선에서 생각하고, 애매하면 괄호를 사용하자
- 누구나 코드를 보고 쉽고 명확하게 이해할 수 있어야 한다. 개발자들이 연산자 우선순위를 외우고 개발하는 것이 아니다! 복잡하면 명확하게 괄호를 넣어라!
- 개발에서 가장 중요한 것은 단순함과 명확함이다! 애매하거나 복잡하면 안된다.

증감 연산자

증가 및 감소 연산자를 줄여서 증감 연산자라 한다.

증감 연산자는 `++` 와 `--` 로 표현되며, 이들은 변수의 값을 1만큼 증가시키거나 감소시킨다.

프로그래밍에서는 값을 1씩 증가하거나 1씩 감소할 때가 아주 많기 때문에 이런 편의 기능을 제공한다.

OperatorAdd1

```
package operator;

public class OperatorAdd1 {

    public static void main(String[] args) {
        int a = 0;

        a = a + 1;
        System.out.println("a = " + a); //1
        a = a + 1;
        System.out.println("a = " + a); //2

        //증감 연산자
        ++a; //a = a + 1
        System.out.println("a = " + a); //3
        ++a; //a = a + 1
        System.out.println("a = " + a); //4
    }

}
```

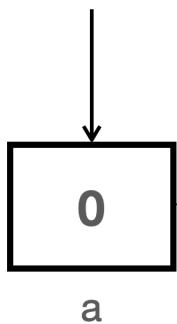
실행 결과

```
a = 1
a = 2
a = 3
a = 4
```

변수 `a`의 값을 하나 증가하려면 `a = a + 1` 연산을 수행해야 한다. 자기 자신에 `1`을 더하고 그 결과를 자신에게 다시 저장해야 한다.

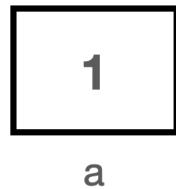
코드는 다음과 같이 수행된다.

1. 읽기($a + 1$) 2. 연산($0+1 \rightarrow 1$



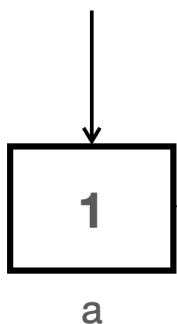
3. 대입($a = 1$)

4. 결과



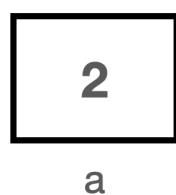
```
//a = 0  
a = a + 1  
a = 0 + 1 //변수 a의 값 확인(0)  
a = 1
```

1. 읽기($a + 1$) 2. 연산($1+1 \rightarrow 2$



3. 대입($a = 2$)

4. 결과



```
//a = 1  
a = a + 1  
a = 1 + 1 //변수 a의 값 확인(1)  
a = 2
```

$a = a + 1$ 을 $++a$ 로 간단히 표현할 수 있는 것이 바로 증감 연산자이다.

정리하면 해당 변수에 들어있는 숫자 값을 하나 증가하는 것이다.

$++$ (증가), $--$ (감소)

값을 하나 감소할 때는 $--a$ 와 같이 표현하면 된다. 이것은 $a = a - 1$ 이 된다.

전위, 후위 증감연산자

증감 연산자는 피연산자 앞에 두거나 뒤에 둘 수 있으며, 연산자의 위치에 따라 연산이 수행되는 시점이 달라진다.

- $++a$: 증감 연산자를 피연산자 앞에 둘 수 있다. 이것을 앞에 있다고 해서 전위(Prefix) 증감 연산자라 한다.
- $a++$: 증감 연산자를 피연산자 뒤에 둘 수 있다. 이것을 뒤에 있다고 해서 후위(Postfix) 증감 연산자라 한다.

코드로 확인해보자.

OperatorAdd2

```
package operator;

public class OperatorAdd2 {

    public static void main(String[] args) {
        // 전위 증감 연산자 사용 예
        int a = 1;
        int b = 0;
        b = ++a; // a의 값을 먼저 증가시키고, 그 결과를 b에 대입
        System.out.println("a = " + a + ", b = " + b); // 결과: a = 2, b = 2

        // 후위 증감 연산자 사용 예
        a = 1; // a 값을 다시 1로 지정
        b = 0; // b 값을 다시 0으로 지정
        b = a++; // a의 현재 값을 b에 먼저 대입하고, 그 후 a 값을 증가시킴
        System.out.println("a = " + a + ", b = " + b); // 결과: a = 2, b = 1
    }

}
```

실행 결과

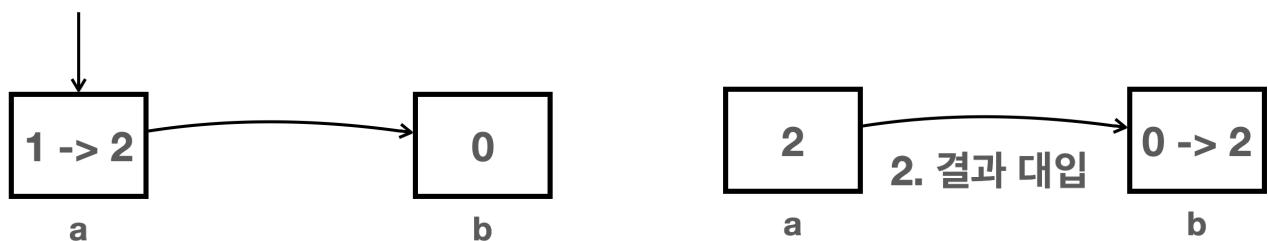
```
a = 2, b = 2
a = 2, b = 1
```

증감 연산자가 변수 앞에 오는 경우를 **전위 증감 연산자**라고 하며, 이 경우에는 증감 연산이 먼저 수행된 후 나머지 연산이 수행된다.

예) $++a$ 전위 증감 연산자

1. $++a$

a의 값을 먼저 증가



```
a = 1, b = 0
b = ++a //전위 증감 연산자
```

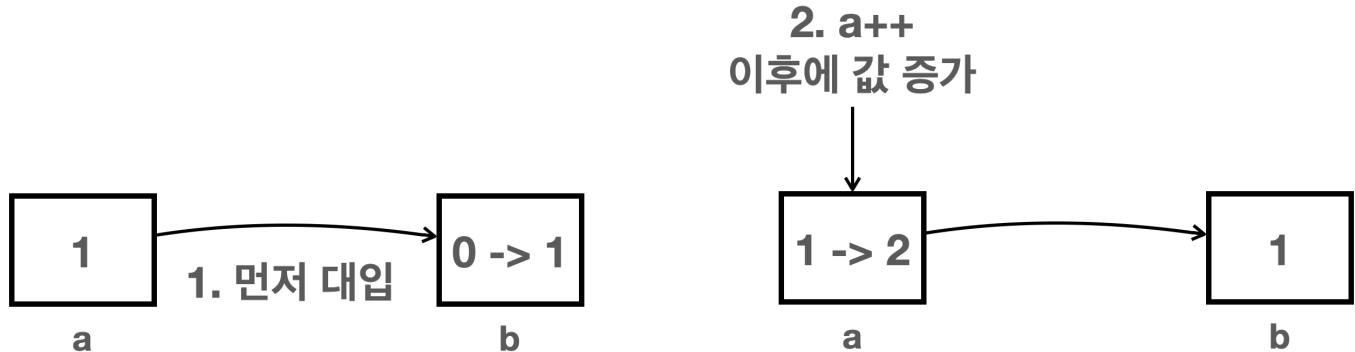
```
a = a + 1 //a의 증감 연산이 먼저 진행, a = 2
```

```
b = a //이후에 a를 대입 b = 2
```

결과: a = 2, b = 2

증감 연산자가 변수 뒤에 오는 경우를 **후위 증감 연산자**라고 하며, 이 경우에는 다른 연산이 먼저 수행된 후 증감 연산이 수행된다.

예) a++ 후위 증감 연산자



```
a = 1, b = 0
```

```
b = a++ //후위 증감 연산자
```

```
b = a; //a의 값을 먼저 b에 대입 b = 1
```

```
a = a + 1; //이후에 a의 값을 증가 a = 2
```

결과: a = 2, b = 1

참고로 다음과 같이 증감 연산자를 단독으로 사용하는 경우에는 다른 연산이 없기 때문에, 본인의 값만 증가한다. 따라서 전위이든 후위이든 둘다 결과가 같다.

```
++a;
```

```
a++;
```

비교 연산자

비교 연산자는 두 값을 비교하는 데 사용한다. 비교 연산자는 주로 뒤에서 설명하는 조건문과 함께 사용한다.

비교 연산자

- == : 동등성 (equal to)
- != : 불일치 (not equal to)

- > : 크다 (greater than)
- < : 작다 (less than)
- >= : 크거나 같다 (greater than or equal to)
- <= : 작거나 같다 (less than or equal to)

비교 연산자를 사용하면 참 (`true`) 또는 거짓 (`false`)이라는 결과가 나온다. 참 거짓은 `boolean` 형을 사용한다.

여기서 주의할 점은 `=` 와 `==` (`= x2`)이 다르다는 점이다.

- `=` : 대입 연산자, 변수에 값을 대입한다.
- `==` : 동등한지 확인하는 비교 연산자

불일치 연산자는 `!=` 를 사용한다. `!` 는 반대라는 뜻이다.

비교 연산자를 예제 코드로 확인해보자.

Comp1

```
package operator;

public class Comp1 {
    public static void main(String[] args) {
        int a = 2;
        int b = 3;

        System.out.println(a == b); // false, a와 b는 같지 않다
        System.out.println(a != b); // true, a와 b는 다르다
        System.out.println(a > b); // false, a는 b보다 크지 않다
        System.out.println(a < b); // true, a는 b보다 작다
        System.out.println(a >= b); // false, a는 b보다 크거나 같지 않다
        System.out.println(a <= b); // true, a는 b보다 작거나 같다

        // 결과를 boolean 변수에 담기
        boolean result = a == b; // a == b: false
        System.out.println(result); // false
    }
}
```

문자열 비교

문자열이 같은지 비교할 때는 `==` 이 아니라 `.equals()` 메서드를 사용해야 한다.

`==` 를 사용하면 성공할 때도 있지만 실패할 때도 있다. 지금은 이 부분을 이해하기 어려우므로 지금은 단순히 문자열의

비교는 `.equals()` 메서드를 사용해야 한다 정도로 알고 있자, 자세한 내용은 별도로 다룬다.

Comp2 - 문자열 비교 예시

```
package operator;

public class Comp2 {
    public static void main(String[] args) {
        String str1 = "문자열1";
        String str2 = "문자열2";

        boolean result1 = "hello".equals("hello"); //리터럴 비교
        boolean result2 = str1.equals("문자열1");//문자열 변수, 리터럴 비교
        boolean result3 = str1.equals(str2);//문자열 변수 비교

        System.out.println("result1 = " + result1);
        System.out.println("result2 = " + result2);
        System.out.println("result3 = " + result3);

    }
}
```

실행 결과

```
result1 = true
result2 = true
result3 = false
```

논리 연산자

논리 연산자는 `boolean` 형의 `true`, `false`를 비교하는데 사용한다.

논리 연산자

- `&&` (그리고) : 두 피연산자가 모두 참이면 참을 반환, 둘중 하나라도 거짓이면 거짓을 반환
- `||` (또는) : 두 피연산자 중 하나라도 참이면 참을 반환, 둘다 거짓이면 거짓을 반환
- `!` (부정) : 피연산자의 논리적 부정을 반환. 즉, 참이면 거짓을, 거짓이면 참을 반환

코드로 간단히 확인해보자

Logical1

```
package operator;

public class Logical1 {

    public static void main(String[] args) {
        System.out.println("&&: AND 연산");
        System.out.println(true && true); //true
        System.out.println(true && false); //false
        System.out.println(false && false); //false

        System.out.println("||: OR 연산");
        System.out.println(true || true); //true
        System.out.println(true || false); //true
        System.out.println(false || false); //false

        System.out.println("!: 연산");
        System.out.println(!true); //false
        System.out.println(!false); //true

        System.out.println("변수 활용");
        boolean a = true;
        boolean b = false;
        System.out.println(a && b); // false
        System.out.println(a || b); // true
        System.out.println(!a); // false
        System.out.println(!b); // true
    }
}
```

- `&&`: 두 피연산자가 모두 참이어야 `true`를 반환한다. 둘중 하나라도 거짓이면 `false`를 반환한다.
 - `||`: 두 피연산자 중 하나라도 참이면 `true`를 반환한다. 둘다 모두 거짓이면 `false`를 반환한다.
 - `!`: 피연산자의 논리적 부정을 반환한다. 참이면 거짓을, 거짓이면 참을 반환한다.
-
- `a && b`는 `false`를 반환한다. 왜냐하면 둘 중 하나인 `b`가 거짓이기 때문이다.
 - `a || b`는 `true`를 반환한다. 왜냐하면 둘 중 하나인 `a`가 참이기 때문이다.
 - `!a`와 `!b`는 각각의 논리적 부정을 반환한다

논리 연산자 활용

논리 연산자를 활용하는 다음 코드를 만들어보자.

변수 `a` 가 10보다 크고 20보다 작은지 논리 연산자를 사용해서 확인해보자.

Logical2

```
package operator;

public class Logical2 {

    public static void main(String[] args) {
        int a = 15;
        //a는 10보다 크고 20보다 작다
        boolean result = a > 10 && a < 20; // (a > 10) && (a < 20)
        System.out.println("result = " + result);
    }
}
```

실행 결과

```
result = true
```

참고로 다음과 같이 변수의 위치를 변경해도 결과는 같다.

범위를 나타내는 경우 이렇게 작성하면 코드를 조금 더 읽기 좋다.

```
boolean result = 10 < a && a < 20;
```

대입 연산자

대입 연산자

대입 연산자(=)는 값을 변수에 할당하는 연산자다. 이 연산자를 사용하면 변수에 값을 할당할 수 있다.

예를 들어, `int a = 1` 는 `a`라는 변수에 1이라는 값을 할당한다.

축약(복합) 대입 연산자

산술 연산자와 대입 연산자를 한번에 축약해서 사용할 수 있는데, 이것을 축약(복합) 대입 연산자라 한다.

연산자 종류: `+=`, `-=`, `*=`, `/=`, `%=`

이 연산자는 연산과 대입을 한번에 축약해서 처리한다. 다음 왼쪽과 오른쪽의 결과는 같다.

```
i = i + 3 → i += 3
```

```
i = i * 4 → i *= 4
```

예제 코드를 통해 확인해보자

Assign1

```
package operator;

public class Assign1 {

    public static void main(String[] args) {
        int a = 5; // 5
        a += 3; // 8 (5 + 3): a = a + 3
        a -= 2; // 6 (8 - 2): a = a - 2
        a *= 4; // 24 (6 * 4): a = a * 4
        a /= 3; // 8 (24 / 3): a = a / 3
        a %= 5; // 3 (8 % 5) : a = a % 5
        System.out.println(a);
    }
}
```

실행 결과

```
3
```

문제와 풀이

코딩이 처음이라면 필독!

프로그래밍이 처음이라면 아직 코딩 자체가 익숙하지 않기 때문에 문제와 풀이에 상당히 많은 시간을 쓰게 될 수 있다. 강의를 들을 때는 다 이해가 되는 것 같았는데, 막상 혼자 생각해서 코딩을 하려니 잘 안되는 것이다. 이것은 아직 코딩이 익숙하지 않기 때문인데, 처음 코딩을 하는 사람이라면 누구나 겪는 자연스러운 현상이다.

문제를 스스로 풀기 어려운 경우, 너무 고민하기보다는 먼저 **강의 영상의 문제 풀이 과정을 코드로 따라하면서 이해하자. 반드시 코드로 따라해야 한다.** 그래야 코딩하는 것에 조금씩 익숙해질 수 있다. 그런 다음에 정답을 지우고 스스로 문제를 풀어보면 된다. 참고로 강의를 듣는 시간만큼 문제와 풀이에도 많은 시간을 들어야 제대로 성장 할 수 있다!

문제1 - int와 평균

다음과 같은 작업을 수행하는 프로그램을 작성하세요

클래스 이름은 OperationEx1 라고 적어주세요.

1. num1, num2, num3라는 이름의 세 개의 int 변수를 선언하고, 각각 10, 20, 30으로 초기화하세요.
2. 세 변수의 합을 계산하고, 그 결과를 sum이라는 이름의 int 변수에 저장하세요.
3. 세 변수의 평균을 계산하고, 그 결과를 average라는 이름의 int 변수에 저장하세요. 평균 계산 시 소수점 이하의 결과는 버립니다.
4. sum과 average 변수의 값을 출력하세요.

참고

자바에서 int 끼리의 나눗셈은 자동으로 소수점 이하를 버린다.

문제1 - 정답

OperationEx1

```
package variable.ex;

public class OperationEx1 {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 20;
        int num3 = 30;

        int sum = num1 + num2 + num3;
        int average = sum / 3; //int 끼리의 나눗셈은 자동으로 소수점 이하를 버림

        System.out.println(sum);
        System.out.println(average);
    }
}
```

실행 결과

```
60
20
```

문제2 - double과 평균

클래스 이름: OperationEx2

```
// 다음 double 변수들을 선언하고 그 합과 평균을 출력하는 프로그램을 작성하세요.  
double val1 = 1.5;  
double val2 = 2.5;  
double val3 = 3.5;
```

정답 - double 데이터 타입

```
package operator.ex;  
  
public class OperationEx2 {  
    public static void main(String[] args) {  
        double val1 = 1.5;  
        double val2 = 2.5;  
        double val3 = 3.5;  
  
        double sum = val1 + val2 + val3;  
        double avg = sum / 3;  
  
        System.out.println(sum);  
        System.out.println(avg);  
    }  
}
```

실행 결과

```
7.5  
2.5
```

문제3 - 합격 범위

클래스 이름: OperationEx3

- int 형 변수 score를 선언하세요.
- score가 80점 이상이고, 100점 이하이면 true를 출력하고, 아니면 false를 출력하세요.

정답 - OperationEx3

```
package operator.ex;  
  
public class OperationEx3 {  
    public static void main(String[] args) {  
        int score = 80;
```

```
    boolean result = score >= 80 && score <= 100;
    System.out.println(result);
}
}
```

실행 결과

```
true
```

정리

자주 사용하는 연산자

- 산술 연산자: +, -, *, /, % (나머지)
- 증가 및 감소 연산자: ++, --
- 비교 연산자: ==, !=, >, <, >=, <=
- 논리 연산자: && (AND), || (OR), ! (NOT)
- 대입 연산자: =, +=, -=, *=, /=, %=

다음 연산자들도 자주 사용하는데, 뒷 부분에서 학습한다

- 삼항 연산자: ? :
- instanceof 연산자: 객체 타입을 확인한다.
- 그외: new, [] (배열 인덱스), . (객체 멤버 접근), () (메소드 호출)

비트 연산자는 실무에서 거의 사용할 일이 없다. 필요할 때 찾아보자.

- 비트 연산자: &, |, ^, ~, <<, >>, >>>

4. 조건문

#1.인강/0.자바/1.자바-입문

- /if문1 - if, else
- /if문2 - else if
- /if문3 - if문과 else if문
- /switch문
- /삼항 연산자
- /문제와 풀이1
- /문제와 풀이2
- /정리

if문1 - if, else

조건문 시작

지금까지 살펴본 프로그램은 단순히 위에서 아래로 순서대로 한 줄씩 실행되었다.

특정 조건에 따라서 다른 코드를 실행하려면 어떻게 해야할까? 예를 들어서 만약 18살 이상이면 "성인입니다"를 출력하고, 만약 18살 미만이라면 "미성년자입니다."를 출력해야 한다.

아마도 다음과 같이 코딩을 해야 할 것 같다.

```
만약 (나이 >= 18)면 "성인입니다"  
만약 (나이 < 18)면 "미성년자입니다"
```

영어로 하면 다음과 같다.

```
if (age >= 18) "성인입니다"  
if (age < 18) "미성년자입니다"
```

이렇게 특정 조건에 따라서 다른 코드를 실행하는 것을 조건문이라 한다.

조건문에는 `if` 문, `switch` 문이 있다. 둘다 특정 조건에 따라서 다른 코드를 실행하는 것이라 생각하면 된다.

먼저 `if` 문부터 알아보자.

if문

`if` 문은 특정 조건이 참인지 확인하고, 그 조건이 참(`true`)일 경우 특정 코드 블록을 실행한다.

```
if (condition) {  
    // 조건이 참일 때 실행되는 코드  
}
```

코드 블록: {} (중괄호) 사이에 있는 코드

If1

```
package cond;  
  
public class If1 {  
    public static void main(String[] args) {  
        int age = 20; // 사용자 나이  
  
        if (age >= 18) {  
            System.out.println("성인입니다.");  
        }  
  
        if (age < 18) {  
            System.out.println("미성년자입니다.");  
        }  
    }  
}
```

실행 결과

성인입니다.

age = 20 값을 주면 처음 if 문에서 참이 된다.

if (age >= 18) 분석

```
//age = 20  
if (age >= 18) {"성인입니다."  
if (20 >= 18) {"성인입니다."} //age의 값은 20이다.  
if (true) {"성인입니다."} //조건이 참으로 판명된다.  
{ "성인입니다."} //if문에 있는 코드 블록이 실행된다.
```

조건이 참이므로 "성인입니다." 가 화면에 출력된다.

이후에 다음 코드가 실행된다.

if (age < 18) 분석

```
if (age < 18) {"미성년자입니다."  
if (20 < 18) {"미성년자입니다."} //age의 값은 20이다.  
if (false) {"미성년자입니다."} //조건이 거짓으로 판명된다.
```

```
// 해당 코드 블록은 실행되지 않는다.
```

조건이 거짓이므로 `if` 문 블록을 실행하지 않고, 빠져나온다. 따라서 "미성년자입니다"는 화면에 출력되지 않는다.

```
int age = 20의 값을 15로 변경하면 미성년자입니다. 가 출력되는 것을 확인할 수 있다.
```

else문

`else` 문은 `if` 문에서 만족하는 조건이 없을 때 실행하는 코드를 제공한다.

```
if (condition) {  
    // 조건이 참일 때 실행되는 코드  
} else {  
    // 만족하는 조건이 없을 때 실행되는 코드  
}
```

`else` 문을 사용하면 앞서 진행했던 프로그램을 다음과 같이 더 간략하게 바꿀 수도 있다.

- 기준: 만약 18살 이상이면 "성인입니다"를 출력하고, 만약 18살 미만이라면 "미성년자입니다."를 출력해야 한다.
- 변경: 만약 18살 이상이면 "성인입니다"를 출력하고, 그렇지 않으면 "미성년자입니다."를 출력해야 한다.

쉽게 이야기해서 18살이 넘으면 성인이고, 그렇지 않으면 모두 미성년자이다.

아마도 다음과 같이 코딩을 해야 할 것 같다.

```
만약 (나이 >= 18) 면 "성인입니다"  
그렇지 않으면 "미성년자입니다"
```

영어로 하면 다음과 같다.

```
if (age >= 18) "성인입니다"  
else "미성년자입니다"
```

If2

```
package cond;  
  
public class If2 {  
    public static void main(String[] args) {  
        int age = 20; // 사용자의 나이  
  
        if (age >= 18) {  
            System.out.println("성인입니다."); // 참일 때 실행  
        } else {  
            System.out.println("미성년자입니다."); // 만족하는 조건이 없을 때 실행  
        }  
    }  
}
```

```
    }
}
}
```

실행 결과

성인입니다.

`int age = 20`의 값을 15로 변경하면 미성년자입니다. 가 출력되는 것을 확인할 수 있다.

if문2 - else if

다음 문제를 코드로 풀어보자.

문제

당신은 연령에 따라 다른 메시지를 출력하는 프로그램을 작성해야 한다.

이 프로그램은 `int age`라는 변수를 사용해야 하며, 연령에 따라 다음의 출력을 해야 한다.

- 7세 이하일 경우: "미취학"
- 8세 이상 13세 이하일 경우: "초등학생"
- 14세 이상 16세 이하일 경우: "중학생"
- 17세 이상 19세 이하일 경우: "고등학생"
- 20세 이상일 경우: "성인"

`if` 문을 사용해서 코드를 작성해보자.

If3

```
package cond;

public class If3 {
    public static void main(String[] args) {
        int age = 14;

        if(age <= 7) { //~7: 미취학
            System.out.println("미취학");
        }
    }
}
```

```

if(age >= 8 && age <= 13) { //8~13: 초등학생
    System.out.println("초등학생");
}
if(age >= 14 && age <= 16) { //14~16: 중학생
    System.out.println("중학생");
}
if(age >= 17 && age <= 19) { //17~19: 고등학생
    System.out.println("고등학생");
}
if(age >= 20) { //20~: 성인
    System.out.println("성인");
}
}
}

```

이 코드는 다음과 같은 단점이 있다.

- 불필요한 조건 검사: 이미 조건을 만족해도 불필요한 다음 조건을 계속 검사한다. 예를 들어서 나이가 5살이라면 미취학이 이미 출력이 된다. 그런데 나머지 `if` 문을 통한 조건 검사도 모두 실행해야 한다.
- 코드 효율성: 예를 들어서 나이가 8살인 초등학생이라면 미취학을 체크하는 조건인 `age <= 7`을 통해 나이가 이미 8살이 넘는다는 사실을 알 수 있다. 그런데 바로 다음에 있는 초등학생을 체크하는 조건에서 `age >= 8 && age <= 13`라는 2가지 조건을 모두 수행한다. 여기서 `age >= 8`이라는 조건은 이미 앞의 `age <= 7`이라는 조건과 관련이 있다. 결과적으로 조건을 중복 체크한 것이다.

이런 코드에 `else if`를 사용하면 불필요한 조건 검사를 피하고 코드의 효율성을 향상시킬 수 있다.

else if

`else if` 문은 앞선 `if` 문의 조건이 거짓일 때 다음 조건을 검사한다. 만약 앞선 `if` 문이 참이라면 `else if`를 실행하지 않는다.

if-else 코드

```

if (condition1) {
    // 조건1이 참일 때 실행되는 코드
} else if (condition2) {
    // 조건1이 거짓이고, 조건2가 참일 때 실행되는 코드
} else if (condition3) {
    // 조건2이 거짓이고, 조건3이 참일 때 실행되는 코드
} else {
    // 모든 조건이 거짓일 때 실행되는 코드
}

```

쉽게 이야기해서 이렇게 전체 `if` 문을 하나로 묶는다고 보면 된다. 이렇게 하면 특정 조건이 만족하면 해당 코드를 실행

하고 `if` 문 전체를 빠져나온다. 특정 조건을 만족하지 않으면 다음 조건을 검사한다. 여기서 핵심은 순서대로 맞는 조건을 찾아보고, 맞는 조건이 있으면 딱 1개만 실행이 되는 것이다.

참고로 `else` 는 생략할 수 있다.

`else` 생략 코드

```
if (condition1) {  
    // 조건1이 참일 때 실행되는 코드  
} else if (condition2) {  
    // 조건1이 거짓이고, 조건2가 참일 때 실행되는 코드  
}
```

이제 앞서 만든 코드를 `else if` 를 사용해서 완성해보자.

If4

```
package cond;  
  
public class If4 {  
    public static void main(String[] args) {  
        int age = 14;  
  
        if(age <= 7) { //~7: 미취학  
            System.out.println("미취학");  
        } else if(age <= 13) { //8~13: 초등학생  
            System.out.println("초등학생");  
        } else if(age <= 16) { //14~16: 중학생  
            System.out.println("중학생");  
        } else if(age <= 19) { //17~19: 고등학생  
            System.out.println("고등학생");  
        } else { //20~: 성인  
            System.out.println("성인");  
        }  
    }  
}
```

`age = 7`인 경우

`if(age <= 7)` 의 조건이 참이다. "미취학"을 출력하고 전체 `if` 문 밖으로 나간다.

`age = 13`인 경우

`if(age <= 7)` 의 조건이 거짓이다. 다음 조건으로 넘어간다.

`else if(age <= 13)`의 조건이 참이다. "초등학생"을 출력하고 전체 `if` 문 밖으로 나간다.

age = 50인 경우

`if(age <= 7)`의 조건이 거짓이다. 다음 조건으로 넘어간다.

`else if(age <= 13)`의 조건이 거짓이다. 다음 조건으로 넘어간다.

`else if(age <= 16)`의 조건이 거짓이다. 다음 조건으로 넘어간다.

`else if(age <= 19)`의 조건이 거짓이다. 다음 조건으로 넘어간다.

`else` 만족하는 조건 없이 `else`까지 왔다. `else`에 있는 "성인"을 출력하고 전체 `if` 문 밖으로 나간다.

if문3 - if문과 else if문

`if` 문에 `else if`를 함께 사용하는 것은 서로 연관된 조건일 때 사용한다. 그런데 서로 관련이 없는 독립 조건이면 `else if`를 사용하지 않고 `if` 문을 각각 따로 사용해야 한다.

예시

```
// 예시1. if-else 사용: 서로 연관된 조건이어서, 하나로 묶을 때
```

```
if (condition1) {  
    // 작업1 수행  
} else if (condition2) {  
    // 작업2 수행  
}
```

```
// 예시2. if 각각 사용: 독립 조건일 때
```

```
if (condition1) {  
    // 작업1 수행  
}  
if (condition2) {  
    // 작업2 수행  
}
```

예시 1은 작업1, 작업2 둘 중 하나만 수행된다. 그런데 예시 2는 조건만 맞다면 둘다 수행될 수 있다.

`if` 문에 여러 조건이 있다고 항상 `if-else`로 묶어서 사용할 수 있는 것은 아니다. 조건이 서로 영향을 주지 않고 각각 수행해야 하는 경우에는 `else if` 문을 사용하면 안되고, 대신에 여러 `if` 문을 분리해서 사용해야 한다.

여러 독립적인 조건을 검사해야 하는 경우가 그런 상황의 대표적인 예시이다. 즉, 각 조건이 다른 조건과 연관되지 않고, 각각의 조건에 대해 별도의 작업을 수행해야 할 때 이런 상황이 발생한다.

예제를 통해 자세히 이해해보자.

문제

온라인 쇼핑몰의 할인 시스템을 개발해야 한다. 한 사용자가 어떤 상품을 구매할 때, 다양한 할인 조건에 따라 총 할인 금액이 달라질 수 있다.

각각의 할인 조건은 다음과 같다.

- 아이템 가격이 10000원 이상일 때, 1000원 할인
- 나이가 10살 이하일 때 1000원 할인

이 할인 시스템의 핵심은 **한 사용자가 동시에 여러 할인을 받을 수 있다는 점**이다.

예를 들어, 10000원짜리 아이템을 구매할 때 1000원 할인을 받고, 동시에 나이가 10살 이하이면 추가로 1000원 더 할인을 받는다. 그래서 총 2000원 까지 할인을 받을 수 있다.

If5

```
package cond;

public class If5 {
    public static void main(String[] args) {
        int price = 10000; // 아이템 가격
        int age = 10; // 나이
        int discount = 0;

        if (price >= 10000) {
            discount = discount + 1000;
            System.out.println("10000원 이상 구매, 1000원 할인");
        }

        if (age <= 10) {
            discount = discount + 1000;
            System.out.println("어린이 1000원 할인");
        }

        System.out.println("총 할인 금액: " + discount + "원");
    }
}
```

실행 결과

```
//price = 10000, age = 10
10000원 이상 구매, 1000원 할인
```

```
어린이 1000원 할인  
총 할인 금액: 2000원
```

- 이 코드에서는 각각 독립된 `if` 문이 있다. 따라서 해당하는 모든 할인을 적용한다.
- 만약 `else if`를 쓰면, 첫 번째로 충족하는 조건만 할인이 적용되고 나머지는 무시된다. 따라서 사용자는 나머지 할인을 놓칠 수 있다.

`if` 문을 사용해야 하는 곳에 `else if`를 사용해서 어떤 문제가 발생하는지 확인해보자.

If6 - else if문 적용

```
package cond;  
  
public class If6 {  
    public static void main(String[] args) {  
        int price = 10000; // 아이템 가격  
        int age = 10; // 나이  
        int discount = 0;  
  
        if (price >= 10000) {  
            discount = discount + 1000;  
            System.out.println("10000원 이상 구매, 1000원 할인");  
        } else if (age <= 10) {  
            discount = discount + 1000;  
            System.out.println("어린이 1000원 할인");  
        } else {  
            System.out.println("할인 없음");  
        }  
  
        System.out.println("총 할인 금액: " + discount + "원");  
    }  
}
```

실행 결과

```
//price = 10000, age = 10  
10000원 이상 구매, 1000원 할인  
총 할인 금액: 1000원
```

- 첫 번째로 충족되는 조건인 1000원 할인만 적용되고, `if` 문을 빠져나온다. 따라서 사용자는 나머지 할인을 놓치게 된다.

정리

`if` 문을 각각 사용할지, `if`와 `else if`를 함께 묶어서 사용할지는 요구사항에 따라 다르다. 둘의 차이를 이해하고

적절하게 사용하면 된다.

참고 - if문 {} 중괄호 생략

다음과 같이 `if` 문 다음에 실행할 명령이 하나만 있을 경우에는 {} 중괄호를 생략할 수 있다. `else if`, `else`도 마찬가지이다.

```
if (true)
    System.out.println("if문에서 실행됨");
```

다음과 같은 경우에는 두번째 문장은 `if` 문과 무관하다. 만약 둘다 `if` 문 안에 포함하려면 {}를 사용해야 한다.

```
if (true)
    System.out.println("if문에서 실행됨");
    System.out.println("if문에서 실행 안됨");
```

만약 둘다 `if` 문 안에 포함하려면 다음과 같이 {}를 사용해야 한다.

```
if (true) {
    System.out.println("if문에서 실행됨");
    System.out.println("if문에서 실행 안됨");
}
```

프로그래밍 스타일에 따라 다르겠지만, 일반적으로는 `if` 문의 명령이 한개만 있을 경우에도 다음과 같은 이유로 중괄호를 사용하는 것이 좋다.

- **가독성**: 중괄호를 사용하면 코드를 더 읽기 쉽게 만들어 준다. 조건문의 범위가 명확하게 표시되므로 코드의 흐름을 더 쉽게 이해할 수 있다.
- **유지보수성**: 중괄호를 사용하면 나중에 코드를 수정할 때 오류를 덜 발생시킬 수 있다. 예를 들어, `if` 문에 또 다른 코드를 추가하려고 할 때, 중괄호가 없으면 이 코드가 `if` 문의 일부라는 것이 명확하지 않을 수 있다.

switch문

다음 문제를 코드로 풀어보자

당신은 회원 등급에 따라 다른 쿠폰을 발급하는 프로그램을 작성해야 한다.

이 프로그램은 `int grade`라는 변수를 사용하며, 회원 등급(`grade`)에 따라 다음의 쿠폰을 발급해야 한다.

- 1등급: 쿠폰 1000

- 2등급: 쿠폰 2000
- 3등급: 쿠폰 3000
- 위의 등급이 아닐 경우: 쿠폰 500

각 쿠폰이 할당된 후에는 "발급받은 쿠폰 " + 쿠폰값을 출력해야 한다.

2등급 사용자 출력 예)

발급받은 쿠폰 2000

`if` 문을 사용해서 코드를 작성해보자.

Switch1

```
package cond;

public class Switch1 {

    public static void main(String[] args) {
        //grade 1:1000, 2:2000, 3:3000, 나머지: 500
        int grade = 2;

        int coupon;
        if (grade == 1) {
            coupon = 1000;
        } else if (grade == 2) {
            coupon = 2000;
        } else if (grade == 3) {
            coupon = 3000;
        } else {
            coupon = 500;
        }
        System.out.println("발급받은 쿠폰 " + coupon);
    }
}
```

실행 결과

발급받은 쿠폰 2000

switch 문

`switch` 문은 앞서 배운 `if` 문을 조금 더 편리하게 사용할 수 있는 기능이다.

참고로 `if` 문은 비교 연산자를 사용할 수 있지만, `switch` 문은 단순히 값이 같은지만 비교할 수 있다.

`switch` 문은 조건식에 해당하는 특정 값으로 실행할 코드를 선택한다.

```
switch (조건식) {  
    case value1:  
        // 조건식의 결과 값이 value1일 때 실행되는 코드  
        break;  
    case value2:  
        // 조건식의 결과 값이 value2일 때 실행되는 코드  
        break;  
    default:  
        // 조건식의 결과 값이 위의 어떤 값에도 해당하지 않을 때 실행되는 코드  
}
```

- 조건식의 결과 값이 어떤 `case`의 값과 일치하면 해당 `case`의 코드를 실행한다.
- `break` 문은 현재 실행 중인 코드를 끝내고 `switch` 문을 빠져나가게 하는 역할을 한다.
- 만약 `break` 문이 없으면, 일치하는 `case` 이후의 모든 `case` 코드들이 순서대로 실행된다.
- `default`는 조건식의 결과값이 모든 `case`의 값과 일치하지 않을 때 실행된다. `if` 문의 `else`와 같다.
`default` 구문은 선택이다.
- `if`, `else-if`, `else` 구조와 동일하다.

앞서 작성한 코드를 `switch` 문으로 변경해보자.

Switch2

```
package cond;  
  
public class Switch2 {  
  
    public static void main(String[] args) {  
        //grade 1:1000, 2:2000, 3:3000, 나머지: 500  
        int grade = 2;  
  
        int coupon;  
        switch (grade) {  
            case 1:  
                coupon = 1000;  
                break;  
            case 2:  
                coupon = 2000;  
                break;  
            case 3:  
                coupon = 3000;  
                break;  
        }  
    }  
}
```

```

        default:
            coupon = 500;
        }
        System.out.println("발급받은 쿠폰 " + coupon);
    }
}

```

실행 결과

발급받은 쿠폰 2000

break 문이 없으면?

만약 `break` 문이 없으면 어떻게 되는지 확인하기 위해 조건을 변경해보자.

비즈니스 요구사항이 변경되었다. **2등급도 3등급과 같이 3000원 쿠폰을 준다고** 해보자.

Switch3

```

package cond;

public class Switch3 {

    public static void main(String[] args) {
        //grade 1:1000, 2:3000(변경), 3:3000, 나머지: 500
        int grade = 2;

        int coupon;
        switch (grade) {
            case 1:
                coupon = 1000;
                break;
            case 2:
            case 3:
                coupon = 3000;
                break;
            default:
                coupon = 500;
                break;
        }
        System.out.println("발급받은 쿠폰 " + coupon);
    }
}

```

- 예를 들어서 `grade`가 2등급이면 먼저 `case 2`가 실행된다.

- 그런데 case 2에는 break 문이 없다. 그러면 중단하지 않고 바로 다음에 있는 case 3의 코드를 실행한다. 여기서 coupon = 3000; 을 수행하고 break 문을 만나서 switch 문 밖으로 빠져나간다.
- "발급받은 쿠폰 3000이 출력된다."

if문 vs switch문

switch 문의 조건식을 넣는 부분을 잘 보면 x > 10과 같은 참 거짓의 결과가 나오는 조건이 아니라, 단순히 값만 넣을 수 있다.

switch 문은 조건식이 특정 case와 같은지만 체크할 수 있다. 쉽게 이야기해서 값이 같은지 확인하는 연산만 가능하다. (문자도 가능)

반면에 if 문은 참 거짓의 결과가 나오는 조건식을 자유롭게 적을 수 있다. 예) x > 10, x == 10

정리하자면 switch 문 없이 if 문만 사용해도 된다. 하지만 특정 값에 따라 코드를 실행할 때는 switch 문을 사용하면 if 문 보다 간결한 코드를 작성할 수 있다.

자바 14 새로운 switch문

switch 문은 if 문 보다 조금 덜 복잡한 것 같지만, 그래도 코드가 기대보다 깔끔하게 나오지는 않는다.

이런 문제를 해결하고자 자바14부터는 새로운 switch 문이 정식 도입되었다.

기존 코드를 새로운 switch 문으로 개발하면 다음과 같다.

```
package cond;

public class Switch3 {

    public static void main(String[] args) {
        //grade 1:1000, 2:2000, 3:3000, 나머지: 500
        int grade = 2;

        int coupon = switch (grade) {
            case 1 -> 1000;
            case 2 -> 2000;
            case 3 -> 3000;
            default -> 500;
        };
        System.out.println("발급받은 쿠폰 " + coupon);
    }
}
```

기존 switch 문과 차이는 다음과 같다.

- -> 를 사용한다.
- 선택된 데이터를 반환할 수 있다.

새로운 switch문은 더 많은 내용을 담고 있다. 지금 이해하기에 어려운 내용들이 있으므로, 자세한 내용은 별도로 다룬다.

삼항 연산자

if 문을 사용할 때 다음과 같이 단순히 참과 거짓에 따라 특정 값을 구하는 경우가 있다.

CondOp1

```
package cond;

public class CondOp1 {

    public static void main(String[] args) {
        int age = 18;
        String status;
        if (age >= 18) {
            status = "성인";
        } else {
            status = "미성년자";
        }
        System.out.println("age = " + age + " status = " + status);
    }
}
```

실행 결과, age = 18

```
age = 18 status = 성인
```

실행 결과, age = 17

```
age = 17 status = 미성년자
```

이 예제는 참과 거짓에 따라 status 변수의 값이 달라진다.

이렇게 단순히 참과 거짓에 따라서 특정 값을 구하는 경우 **삼항 연산자** 또는 **조건 연산자**라고 불리는 `? :` 연산자를 사용할 수 있다.

이 연산자를 사용하면 `if` 문과 비교해서 코드를 단순화 할 수 있다.

우선 코드부터 보자. 삼항 연산자를 사용하면 다음과 같이 코드를 간결하게 만들 수 있다.

CondOp2

```
package cond;

public class CondOp2 {

    public static void main(String[] args) {
        int age = 18;
        String status = (age >= 18) ? "성인" : "미성년자";
        System.out.println("age = " + age + " status = " + status);
    }
}
```

실행 결과 분석

```
String status = (age >= 18) ? "성인" : "미성년자"; //age=18
String status = (true) ? "성인" : "미성년자"; //조건이 참이므로 참 표현식 부분이 선택된다.
String status = "성인"; //결과
```

삼항 연산자

(조건) `? 참_표현식 : 거짓_표현식`

- 삼항 연산자는 항이 3개라는 뜻이다. 조건, 참_표현식, 거짓_표현식 이렇게 항이 3개이다. 자바에서 유일하게 항이 3개인 연산자여서 삼항 연산자라 한다. 또는 특정 조건에 따라 결과가 나오기 때문에 조건 연산자라고도 한다.
- 조건에 만족하면 참_표현식 이 실행되고, 조건에 만족하지 않으면 거짓_표현식 이 실행된다. 앞의 `if`, `else` 문과 유사하다.
- `if` 문처럼 코드 블럭을 넣을 수 있는 것이 아니라 단순한 표현식만 넣을 수 있다.

삼항 연산자 없이 `if` 문만 사용해도 된다. 하지만 단순히 참과 거짓에 따라서 특정 값을 구하는 삼항 연산자를 사용하면 `if` 문 보다 간결한 코드를 작성할 수 있다.

문제와 풀이 1

코딩이 처음이라면 필독!

프로그래밍이 처음이라면 아직 코딩 자체가 익숙하지 않기 때문에 문제와 풀이에 상당히 많은 시간을 쓰게 될 수 있다. 강의를 들을 때는 다 이해가 되는 것 같았는데, 막상 혼자 생각해서 코딩을 하려니 잘 안되는 것이다. 이것은 아직 코딩이 익숙하지 않기 때문인데, 처음 코딩을 하는 사람이라면 누구나 겪는 자연스러운 현상이다.

문제를 스스로 풀기 어려운 경우, 너무 고민하기보다는 먼저 **강의 영상의 문제 풀이 과정을 코드로 따라하면서 이해하자. 반드시 코드로 따라해야 한다.** 그래야 코딩하는 것에 조금씩 익숙해질 수 있다. 그런 다음에 정답을 지우고 스스로 문제를 풀어보면 된다. 참고로 강의를 듣는 시간만큼 문제와 풀이에도 많은 시간을 들어야 제대로 성장할 수 있다!

문제: "학점 계산하기"

학생의 점수를 기반으로 학점을 출력하는 자바 프로그램을 작성하자. 다음과 같은 기준을 따른다.

- 90점 이상: "A"
- 80점 이상 90점 미만: "B"
- 70점 이상 80점 미만: "C"
- 60점 이상 70점 미만: "D"
- 60점 미만: "F"

점수는 변수(`int score`)로 지정하고, 해당 변수를 기반으로 학점을 출력하자.

출력 예시

`score: 95`

출력: 학점은 A입니다.

`score: 85`

출력: 학점은 B입니다.

`score: 75`

출력: 학점은 C입니다.

`score: 65`

출력: 학점은 D입니다.

`score: 55`

출력: 학점은 F입니다.

정답: "학점 계산하기"

```
package cond.ex;

public class ScoreEx {
    public static void main(String[] args) {
        int score = 85;

        if (score >= 90) {
            System.out.println("학점은 A입니다.");
        } else if (score >= 80) {
            System.out.println("학점은 B입니다.");
        } else if (score >= 70) {
            System.out.println("학점은 C입니다.");
        } else if (score >= 60) {
            System.out.println("학점은 D입니다.");
        } else {
            System.out.println("학점은 F입니다.");
        }
    }
}
```

문제: "거리에 따른 운송 수단 선택하기"

주어진 거리에 따라 가장 적합한 운송 수단을 선택하는 프로그램을 작성하자. 다음과 같은 기준을 따른다.

- 거리가 1km 이하이면: "도보"
- 거리가 10km 이하이면: "자전거"
- 거리가 100km 이하이면: "자동차"
- 거리가 100km 초과이면: "비행기"

거리는 변수(`int distance`)로 지정하고, 해당 변수를 기반으로 운송 수단을 출력하자.

출력 예시

```
distance: 1
출력: 도보를 이용하세요.
```

```
distance: 5
출력: 자전거를 이용하세요.
```

```
distance: 25
```

출력: 자동차를 이용하세요.

distance: 150

출력: 비행기를 이용하세요.

정답: "거리에 따른 운송 수단 선택하기"

```
package cond.ex;

public class DistanceEx {
    public static void main(String[] args) {
        int distance = 25;

        if (distance <= 1) {
            System.out.println("도보를 이용하세요.");
        } else if (distance <= 10) {
            System.out.println("자전거를 이용하세요.");
        } else if (distance <= 100) {
            System.out.println("자동차를 이용하세요.");
        } else {
            System.out.println("비행기를 이용하세요.");
        }
    }
}
```

문제: "환율 계산하기"

특정 금액을 미국 달러에서 한국 원으로 변환하는 프로그램을 작성하자. 환율은 1달러당 1300원이라고 가정하자. 다음과 같은 기준을 따른다.

- 달러가 0미만이면: "잘못된 금액입니다."
- 달러가 0일 때: "환전할 금액이 없습니다."
- 달러가 0 초과일 때: "환전 금액은 (계산된 원화 금액)원입니다."

금액은 변수(`int dollar`)로 지정하고, 해당 변수를 기반으로 한국 원으로의 환전 금액을 출력하자.

출력 예시

dollar: -5

출력: 잘못된 금액입니다.

dollar: 0

출력: 환전할 금액이 없습니다.

dollar: 10

출력: 환전 금액은 13000원입니다.

정답: "환율 계산하기"

```
package cond.ex;

public class ExchangeRateEx {
    public static void main(String[] args) {
        int dollar = 10;

        if (dollar < 0) {
            System.out.println("잘못된 금액입니다.");
        } else if (dollar == 0) {
            System.out.println("환전할 금액이 없습니다.");
        } else {
            int won = dollar * 1300;
            System.out.println("환전 금액은 " + won + "원입니다.");
        }
    }
}
```

문제와 풀이2

문제: "평점에 따른 영화 추천하기"

요청한 평점 이상의 영화를 찾아서 추천하는 프로그램을 작성하자.

- 어바웃타임 - 평점9
- 토이 스토리 - 평점8
- 고질라 - 평점7

평점 변수는 `double rating`을 사용하세요. `if` 문을 활용해서 문제를 풀자.

출력 예시

- rating: 9
- 출력:
• '어바웃타임'을 추천합니다.
- rating: 8
- 출력:
• '어바웃타임'을 추천합니다.
• '토이 스토리'를 추천합니다.
- rating: 7.1
- 출력:
• '어바웃타임'을 추천합니다.
• '토이 스토리'를 추천합니다.
- rating: 7
- 출력:
• '어바웃타임'을 추천합니다.
• '토이 스토리'를 추천합니다.
• '고질라'를 추천합니다.

정답: "평점에 따른 영화 추천하기"

```
package cond.ex;

public class MoveRateEx {
    public static void main(String[] args) {
        double rating = 7.1;

        if (rating <= 9) {
            System.out.println("'어바웃타임'을 추천합니다.");
        }

        if (rating <= 8) {
            System.out.println("'토이 스토리'를 추천합니다.");
        }

        if (rating <= 7) {
            System.out.println("'고질라'를 추천합니다.");
        }
    }
}
```

```
}
```

문제: "학점에 따른 성취도 출력하기"

String grade라는 문자열을 만들고, 학점에 따라 성취도를 출력하는 프로그램을 작성하자. 각 학점은 다음과 같은 성취도를 나타낸다.

- "A": "탁월한 성과입니다!"
- "B": "좋은 성과입니다!"
- "C": "준수한 성과입니다!"
- "D": "향상이 필요합니다."
- "F": "불합격입니다."
- 나머지: "잘못된 학점입니다."

switch 문을 사용해서 문제를 해결하자.

출력 예시

```
grade: "B"  
출력: "좋은 성과입니다!"
```

```
grade: "A"  
출력: "탁월한 성과입니다!"
```

```
grade: "F"  
출력: "불합격입니다."
```

정답: "학점에 따른 성취도 출력하기"

```
package cond.ex;  
  
public class GradeSwitchEx {  
    public static void main(String[] args) {  
        String grade = "B";  
  
        switch(grade) {  
            case "A":  
                System.out.println("탁월한 성과입니다!");  
                break;  
            case "B":  
                System.out.println("좋은 성과입니다!");  
        }  
    }  
}
```

```

        break;
    case "C":
        System.out.println("준수한 성과입니다!");
        break;
    case "D":
        System.out.println("향상이 필요합니다.");
        break;
    case "F":
        System.out.println("불합격입니다.");
        break;
    default:
        System.out.println("잘못된 학점입니다.");
    }
}
}

```

문제: 더 큰 숫자 찾기

여러분은 두 개의 정수 변수 `a` 와 `b` 를 가지고 있다. `a` 의 값은 10이고, `b` 의 값은 20이다. 삼항 연산자를 사용하여 두 숫자 중 더 큰 숫자를 출력하는 코드를 작성하자.

출력 예시

더 큰 숫자는 20입니다.

정답: 더 큰 숫자 찾기

```

package cond.ex;

public class CondOpEx {
    public static void main(String[] args) {
        int a = 10;
        int b = 20;

        int max = (a > b) ? a : b;

        System.out.println("더 큰 숫자는 " + max + "입니다.");
    }
}

```

문제: 홀수 짝수 찾기

정수 x 가 주어지면 x 가 짝수이면 "짝수"를, x 가 홀수이면 "홀수"를 출력하는 프로그램을 작성하자
삼항 연산자를 사용해야 한다.

참고로 $x \% 2$ 를 사용하면 홀수, 짝수를 쉽게 계산할 수 있다.

출력 예시

```
x: 2  
출력: x = 2, 짝수
```

```
x: 3  
출력: x = 3, 홀수
```

정답 홀수 짝수 찾기

```
package cond.ex;  
  
public class EvenOddEx {  
    public static void main(String[] args) {  
        int x = 2;  
        String result = (x % 2 == 0) ? "짝수" : "홀수";  
        System.out.println("x = " + x + ", " + result);  
    }  
}
```

정리

5. 반복문

#1.인강/0.자바/1.자바-입문

- /반복문 시작
- /while문1
- /while문2
- /do-while문
- /break, continue
- /for문1
- /for문2
- /중첩 반복문
- /문제와 풀이1
- /문제와 풀이2
- /정리

반복문 시작

반복문은 이름 그대로 특정 코드를 반복해서 실행할 때 사용한다.

자바는 다음 3가지 종류의 반복문을 제공한다.

while, do-while, for

먼저 간단한 예제를 통해 반복문이 왜 필요한지 이유를 알아보자.

1을 한 번씩 더해서 총 3번 더하는 간단한 코드를 만들어보자.

While1_1

```
package loop;

public class While1_1 {

    public static void main(String[] args) {
        int count = 0;

        count = count + 1;
        System.out.println("현재 숫자는:" + count);
        count = count + 1;
    }
}
```

```
        System.out.println("현재 숫자는 :" + count);
        count = count + 1;
        System.out.println("현재 숫자는 :" + count);
    }
}
```

출력

```
현재 숫자는 :1
현재 숫자는 :2
현재 숫자는 :3
```

단순히 `count`에 값을 1씩 3번 더하는 단순한 예제이다. 최종 결과는 3이다.

`count = count + 1`은 증감 연산자(`++`)를 사용해서 다음과 같이 개선할 수 있다.

```
//개선
count++;
System.out.println("현재 숫자는 :" + count);
count++;
System.out.println("현재 숫자는 :" + count);
count++;
System.out.println("현재 숫자는 :" + count);
```

하지만 같은 코드가 3번 반복되고 있다. 이번에는 1을 한 번씩 더해서 총 100번 더하는 코드를 만들어보자
아마도 직접 작성한다면 같은 코드가 100번 반복될 것이다.

이렇게 특정 코드를 반복해서 실행할 때 사용하는 것이 바로 반복문이다.

반복문에는 `while`, `for` 문이 있다. 먼저 `while` 문부터 알아보자.

while문1

while문은 조건에 따라 코드를 반복해서 실행할 때 사용한다.

```
while (조건식) {
    // 코드
}
```

- 조건식을 확인한다. 참이면 코드 블럭을 실행하고, 거짓이면 while문을 벗어난다.
- 조건식이 참이면 코드 블럭을 실행한다. 이후에 코드 블럭이 끝나면 다시 조건식 검사로 돌아가서 조건식을 검사한다.(무한 반복)

while문을 사용해서 1을 한 번씩 더해서 총 3번 더하는 코드를 만들어보자

While1_2

```
package loop;

public class While1_2 {

    public static void main(String[] args) {
        int count = 0;

        while (count < 3) {
            count++;
            System.out.println("현재 숫자는:" + count);
        }
    }
}
```

출력 결과

```
현재 숫자는:1
현재 숫자는:2
현재 숫자는:3
```

1. while(count(0) < 3)

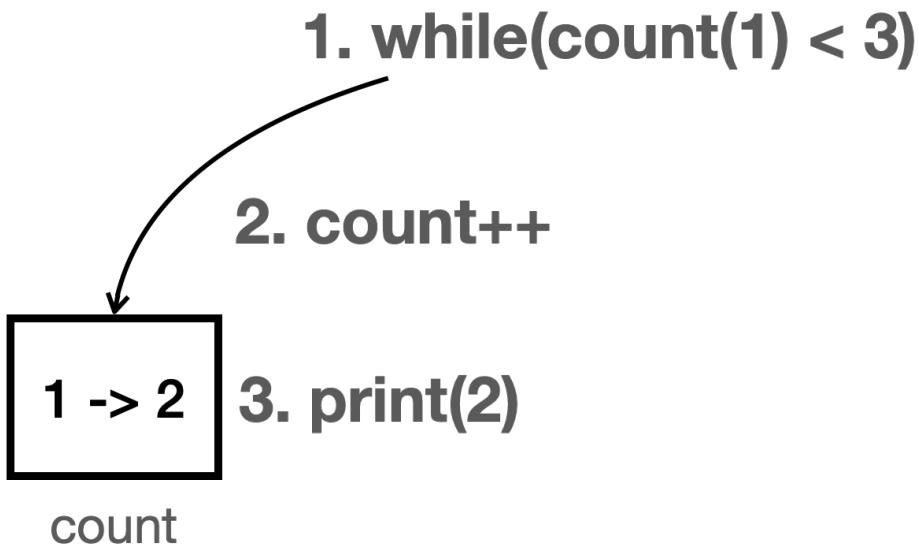
2. count++

0 -> 1

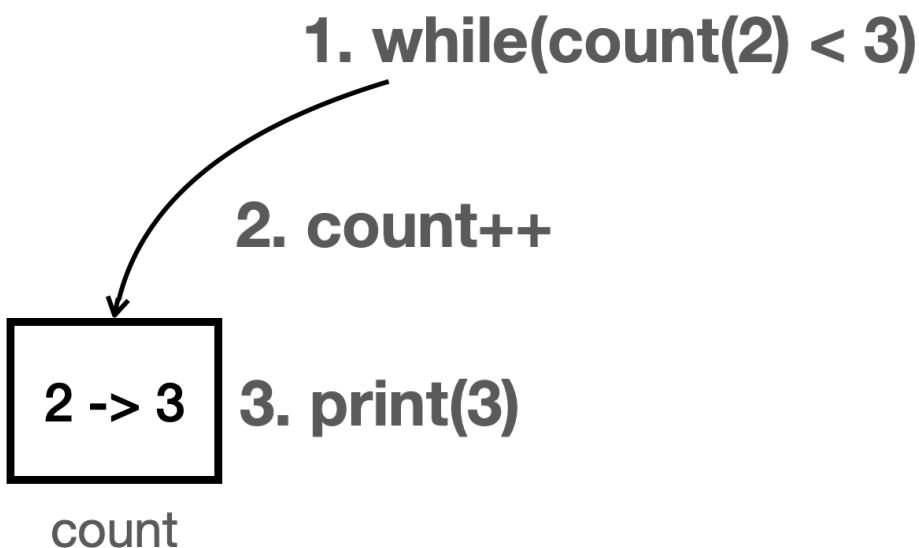
3. print(1)

count

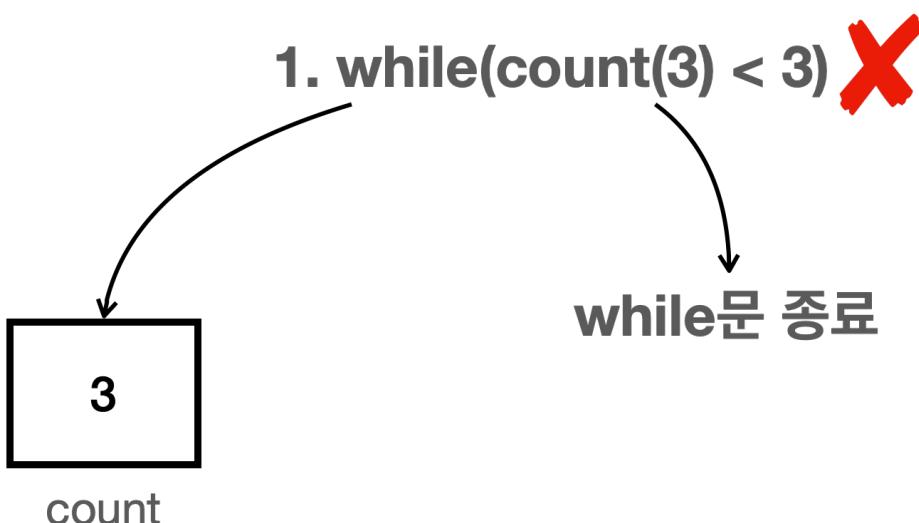
while문 실행1, count=0



while문 실행2, count=1



while문 실행3, count=2



while문 실행4, count=3

`while (count < 3)`에서 코드 블럭을 반복 실행한다. 여기서 `count`의 값이 `1, 2, 3`으로 점점 커지다가 결국 `count < 3`이 거짓이 되면서 `while` 문을 빠져나간다.

`while(count < 3)`에 있는 숫자를 `while(count < 100)`으로 변경하면 `while` 문의 코드 블럭을 100번 반복한다.

while문2

이번에는 난이도를 조금 높여보자. 다음 문제를 같이 풀어보자.

문제: 1부터 하나씩 증가하는 수를 3번 더해라 (1 ~ 3 더하기)

이 문제는 1부터 하나씩 증가하는 수이기 때문에 $1 + 2 + 3$ 을 더해야 한다.

우선 `while` 문을 사용하지 않고 단순 무식하게 풀어보자.

While2_1

```
package loop;

public class While2_1 {
    public static void main(String[] args) {
        int sum = 0;

        sum = sum + 1; //sum(0) + 1 -> sum(1)
        System.out.println("i=" + 1 + " sum=" + sum);

        sum = sum + 2; //sum(1) + 2 -> sum(3)
        System.out.println("i=" + 2 + " sum=" + sum);

        sum = sum + 3; //sum(3) + 3 -> sum(6)
        System.out.println("i=" + 3 + " sum=" + sum);
    }
}
```

출력 결과

```
i=1 sum=1
i=2 sum=3
```

```
i=3 sum=6
```

이 코드의 정답은 맞다. 하지만 개선할 점이 많이 있는데, 무엇보다 변경에 유연하지 않다.

다음과 같이 요구사항이 변경되었다.

문제: 10부터 하나씩 증가하는 수를 3번 더해라 (10 ~ 12더하기)

이렇게 되면 10 + 11 + 12를 계산 해야한다. 문제는 코드를 너무 많이 변경해야 한다는 점이다.

변수를 사용해서 더 변경하기 쉬운 코드로 만들어보자. 변경되는 부분을 변수 `i`로 바꾸어보자.

문제: i부터 하나씩 증가하는 수를 3번 더해라 (i ~ i+2더하기)

While2_2

```
package loop;

public class While2_2 {
    public static void main(String[] args) {
        int sum = 0;
        int i = 1;

        sum = sum + i; //sum(0) + i(1) -> sum(1)
        System.out.println("i=" + i + " sum=" + sum);
        i++; //i=2

        sum = sum + i; //sum(1) + i(2) -> sum(3)
        System.out.println("i=" + i + " sum=" + sum);
        i++; //i=3

        sum = sum + i; //sum(3) + i(3) -> sum(6)
        System.out.println("i=" + i + " sum=" + sum);
    }
}
```

출력 결과

```
//i=1
i=1 sum=1
i=2 sum=3
i=3 sum=6
```

변수 `i`를 사용한 덕분에 `i`의 값만 변경하면 나머지 코드를 전혀 변경하지 않아도 된다.

`i=10`으로 변경하면 다른 코드의 변경 없이 앞서 이야기한 10 + 11 + 12의 문제도 바로 풀 수 있다.

출력 결과

```
//i=10  
i=10 sum=10  
i=11 sum=21  
i=12 sum=33
```

좋은 코드인지 아닌지는 변경 사항이 발생했을 때 알 수 있다. 변경 사항이 발생했을 때 변경해야 하는 부분이 적을수록 좋은 코드이다.

지금까지 변수를 사용해서 하나의 문제를 잘 해결했다. 이번에는 새로운 변경사항이 등장했다.

기존 문제: i부터 하나씩 증가하는 수를 3번까지 더해라 ($i \sim i+2$ 더하기)

새로운 문제: i부터 하나씩 증가하는 수를 endNum(마지막 수)까지 더해라 ($i \sim endNum$ 더하기)

예)

- $i=1$, $endNum=3$ 이라고 하면 $1 \sim 3$ 까지 총 3번 더해야한다.
- $i=1$, $endNum=10$ 이라고 하면 $1 \sim 10$ 까지 총 10번 더해야한다.
- $i=10$, $endNum=12$ 이라고 하면 $10 \sim 12$ 까지 총 3번 더해야한다.

먼저 $i=1$, $endNum=3$ 이라고 생각하고 단순하게 문제를 풀어보자.

While2_3

```
package loop;  
  
public class While2_3 {  
    public static void main(String[] args) {  
        int sum = 0;  
        int i = 1;  
        int endNum = 3;  
  
        sum = sum + i;  
        System.out.println("i=" + i + " sum=" + sum);  
        i++;  
  
        sum = sum + i;  
        System.out.println("i=" + i + " sum=" + sum);  
        i++;  
  
        sum = sum + i;  
        System.out.println("i=" + i + " sum=" + sum);  
        i++;  
    }  
}
```

```
}
```

실행 결과

```
i=1 sum=1  
i=2 sum=3  
i=3 sum=6
```

i=1, endNum=3 이므로 다음 코드를 총 3번 반복해야 한다.

```
sum = sum + i;  
System.out.println("i=" + i + " sum=" + sum);  
i++;
```

그런데 i=1, endNum=10 와 같이 변경하면 이 코드를 총 10번 반복해야 한다. 따라서 같은 코드를 더 많이 추가해야 한다.

이 문제를 제대로 풀기 위해서는 코드가 실행되는 횟수를 유연하게 변경할 수 있어야 한다. 한마디로 같은 코드를 반복 실행할 수 있어야 한다.

while 문을 사용하면 원하는 횟수 만큼 같은 코드를 반복 실행할 수 있다.

While2_3 - 코드 변경

```
package loop;  
  
public class While2_3 {  
    public static void main(String[] args) {  
        int sum = 0;  
        int i = 1;  
        int endNum = 3;  
  
        while (i <= endNum) {  
            sum = sum + i;  
            System.out.println("i=" + i + " sum=" + sum);  
            i++;  
        }  
    }  
}
```

반복 횟수 정하기

i 가 endNum 이 될때 까지 반복해서 코드를 실행하면 된다.

- i=1, endNum=3 이라면 3번 반복하면 된다. i=1 -> 2 -> 3

- `i=3, endNum=4` 라면 2번 반복하면 된다. `i=3 -> 4`

while문 작성하기

- `while` 문에서 `i <= endNum` 조건을 통해 `i` 가 `endNum` 이 될 때 까지 코드 블럭을 실행한다.
- `i` 가 `endNum` 보다 크면 `while` 문을 종료한다.

실행 결과

```
//i=1, endNum=3
```

```
i=1 sum=1
```

```
i=2 sum=3
```

```
i=3 sum=6
```

```
//i=1, endNum=10
```

```
i=1 sum=1
```

```
i=2 sum=3
```

```
i=3 sum=6
```

```
i=4 sum=10
```

```
i=5 sum=15
```

```
i=6 sum=21
```

```
i=7 sum=28
```

```
i=8 sum=36
```

```
i=9 sum=45
```

```
i=10 sum=55
```

```
//i=10, endNum=12
```

```
i=10 sum=10
```

```
i=11 sum=21
```

```
i=12 sum=33
```

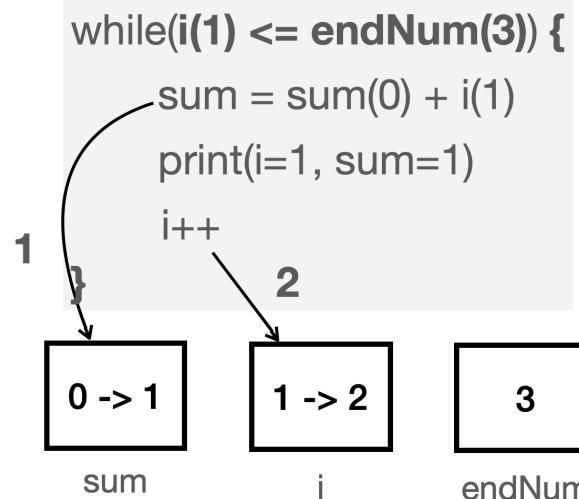
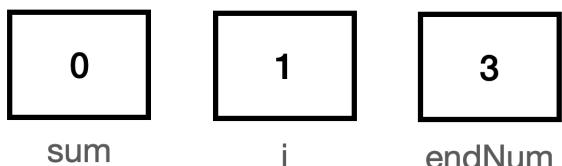
그림을 통해 코드를 분석해보자.

왼쪽은 `while` 의 조건식을 체크하는 단계이고, 오른쪽은 조건식을 통과하고 나서 `while` 문의 코드 블럭을 실행하는 부분이다.

```

while(i(1) <= endNum(3)) {
    sum = sum + i
    print(i=?, sum=?)
    i++
}

```



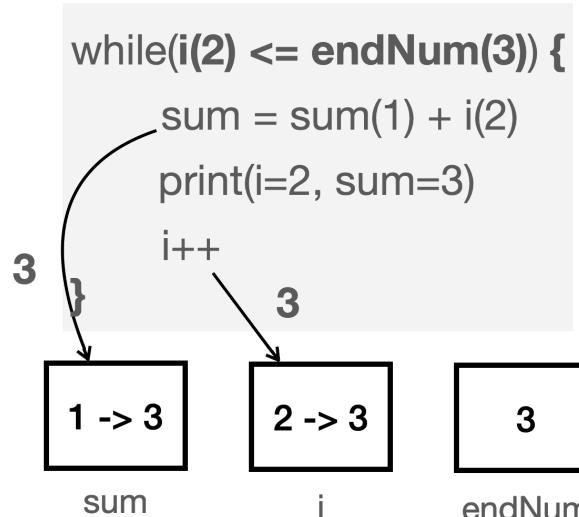
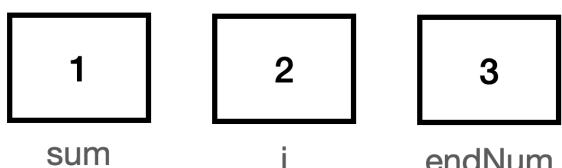
i=1, endNum=3

- 조건식을 만족한다.
- $i=1, sum=1$ 을 출력한다.

```

while(i(2) <= endNum(3)) {
    sum = sum + i
    print(i=?, sum=?)
    i++
}

```



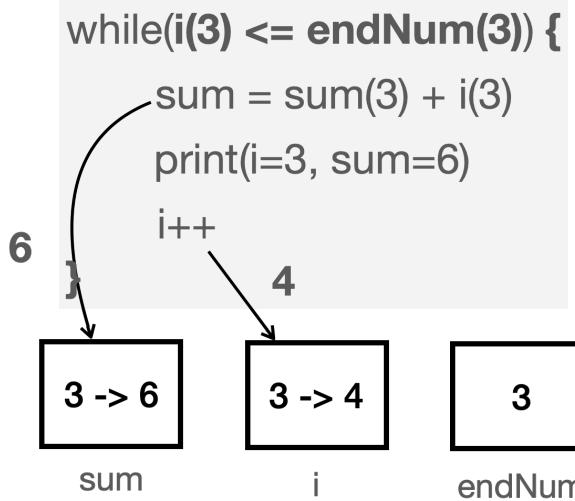
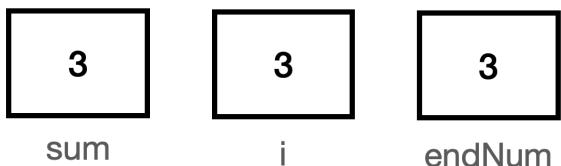
i=2, endNum=3

- 조건식을 만족한다.
- $i=2, sum=3$ 을 출력한다.

```

while(i(3) <= endNum(3)) {
    sum = sum + i
    print(i=?, sum=?)
    i++
}

```



i=3, endNum=3

- 조건식을 만족한다.
- i=3, sum=6 을 출력한다.

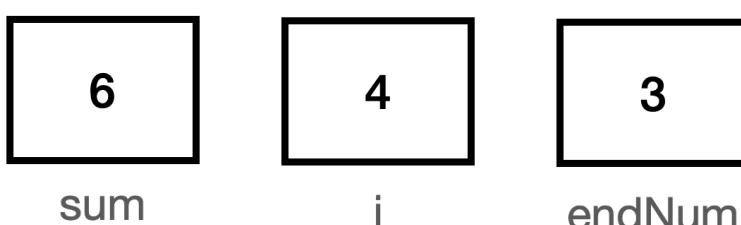
~~while(i(4) <= endNum(3)) {~~

```

sum = sum + i
print(i=?, sum=?)
i++
}

```

while문 종료



i=4, endNum=3

- 조건식을 만족하지 않는다.
- while 문을 종료한다.

실행 코드 분석

```

sum(0), i(1), endNum(3)
//루프 1
while (i(1) <= endNum(3)) -> true
sum(0)+i(1) -> sum(1)

```

```

i(1)++ -> i(2)

//루프 2
while (i(2) <= endNum(3)) -> true
sum(1)+i(2) -> sum(3)
i(2)++ -> i(3)

//루프 3
while (i(3) <= endNum(3)) -> true
sum(3)+i(3) -> sum(6)
i(3)++ -> i(4)

//루프 4
while (i(4) <= endNum(3)) -> false

```

do-while문

do-while 문은 while 문과 비슷하지만, 조건에 상관없이 무조건 한 번은 코드를 실행한다.

do-while문 구조

```

do {
    // 코드
} while (조건식);

```

예를 들어서 조건에 만족하지 않아도 한 번은 현재 값을 출력하고 싶다고 하자.

먼저 while 문을 사용한 예제를 보자

DoWhile1

```

package loop;

public class DoWhile1 {

    public static void main(String[] args) {
        int i = 10;
        while (i < 3) {
            System.out.println("현재 숫자는:" + i);
            i++;
    }
}

```

```
    }
}
}

i=10이기 때문에 while (i < 3) 조건식은 거짓이 된다. 따라서 아무것도 출력되지 않는다.
```

출력 결과

```
//없음
```

이번에는 do-while 문을 사용해보자.

DoWhile2

```
package loop;

public class DoWhile2 {

    public static void main(String[] args) {
        int i = 10;
        do {
            System.out.println("현재 숫자는:" + i);
            i++;
        } while (i < 3);

    }
}
```

do-while 문은 최초 한번은 항상 실행된다. 따라서 먼저 현재 숫자는:10이 출력된다.

코드 블럭을 실행 후에 조건식을 검증하는데, i=10이기 때문에 while (i < 3) 조건식은 거짓이 된다. 따라서 do-while 문을 빠져나온다.

출력 결과

```
현재 숫자는:10
```

do-while 문은 최초 한번은 코드 블럭을 꼭 실행해야 하는 경우에 사용하면 된다.

break, continue

break와 continue는 반복문에서 사용할 수 있는 키워드다.

`break`는 반복문을 즉시 종료하고 나간다. `continue`는 반복문의 나머지 부분을 건너뛰고 다음 반복으로 진행하는데 사용된다.

참고로 `while`, `do-while`, `for`와 같은 모든 반복문에서 사용할 수 있다.

break

```
while(조건식) {  
    코드1;  
    break; //즉시 while문 종료로 이동한다.  
    코드2;  
}  
//while문 종료
```

`break`를 만나면 코드2가 실행되지 않고 while문이 종료된다.

continue

```
while(조건식) {  
    코드1;  
    continue; //즉시 조건식으로 이동한다.  
    코드2;  
}
```

`continue`를 만나면 코드2가 실행되지 않고 다시 조건식으로 이동한다. 조건식이 참이면 while문을 실행한다.

예제를 통해서 알아보자.

문제: 1부터 시작해서 숫자를 계속 누적해서 더하다가 합계가 10보다 처음으로 큰 값은 얼마인가?

$1 + 2 + 3 \dots$ 계속 더하다가 처음으로 합이 10보다 큰 경우를 찾으면 된다.

Break1

```
package loop;  
  
public class Break1 {  
    public static void main(String[] args) {  
        int sum = 0;  
        int i = 1;  
  
        while (true) {  
            sum += i;  
            if (sum > 10) {  
                System.out.println("합이 10보다 크면 종료: i=" + i + " sum=" + sum);  
                break;  
            }  
        }  
    }  
}
```

```
i++;  
}  
}  
}
```

- 조건식을 잘 보면 `true`라고 되어있다. 조건이 항상 참이기 때문에 이렇게 두면 `while` 문은 무한 반복된다. 물론 `break` 문이 있기 때문에 중간에 빠져나올 수 있다.
- 만약 `sum > 10` 조건을 만족하면 결과를 출력하고, `break`를 사용해서 `while` 문을 빠져나간다.

실행 결과

```
합이 10보다 크면 종료: i=5 sum=15
```

문제: 1부터 5까지 숫자를 출력하는데, 숫자가 3일 때는 출력을 건너뛰어야 한다.

Continue1

```
package loop;  
  
public class Continue1 {  
    public static void main(String[] args) {  
        int i = 1;  
  
        while (i <= 5) {  
            if (i == 3) {  
                i++;  
                continue;  
            }  
            System.out.println(i);  
            i++;  
        }  
    }  
}
```

`i==3`인 경우 `i`를 하나 증가하고 `continue`를 실행한다. 따라서 이 경우에는 `i`를 출력하지 않고 바로 `while (i <= 5)` 조건식으로 이동한다.

실행 결과

```
1  
2  
4  
5
```

실행 결과를 보면 3일 때는 출력하지 않은 것을 확인할 수 있다.

for문1

for문도 while문과 같은 반복문이고, 코드를 반복 실행하는 역할을 한다. for문은 주로 반복 횟수가 정해져 있을 때 사용한다.

for문 구조

```
for (1. 초기식; 2. 조건식; 4. 증감식) {  
    // 3. 코드  
}
```

for문은 다음 순서대로 실행된다.

- 1. 초기식이 실행된다. 주로 반복 횟수와 관련된 변수를 선언하고 초기화 할 때 사용한다. 초기식은 딱 1번 사용된다.
- 2. 조건식을 검증한다. 참이면 코드를 실행하고, 거짓이면 for문을 빠져나간다.
- 3. 코드를 실행한다.
- 4. 코드가 종료되면 증감식을 실행한다. 주로 초기식에 넣은 반복 횟수와 관련된 변수의 값을 증가할 때 사용한다.
- 5. 다시 2. 조건식 부터 시작한다. (무한 반복)

for문은 복잡해 보이지만 while문을 조금 더 편하게 다룰 수 있도록 구조화 한 것 뿐이다.

예를 들어 1부터 10까지 출력하는 for문은 다음과 같다.

```
for (int i = 1; i <= 10; i++) {  
    System.out.println(i);  
}
```

- 1. 초기식이 실행된다. `int i = 1`
- 2. 조건식을 검증한다. `i <= 10`
- 3. 조건식이 참이면 코드를 실행한다. `System.out.println(i);`
- 4. 코드가 종료되면 증감식을 실행한다. `i++`
- 5. 다시 2. 조건식을 검증한다. (무한 반복) 이후 `i <= 10` 조건이 거짓이 되면 for문을 빠져나간다.

For1

```
package loop;
```

```

public class For1 {

    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i);
        }
    }
}

```

문제: i부터 하나씩 증가하는 수를 endNum(마지막 수)까지 더해라 (i ~ endNum 더하기)
for문을 사용해서 풀어보자

For2

```

package loop;

public class For2 {
    public static void main(String[] args) {
        int sum = 0;
        int endNum = 3;

        for (int i = 1; i <= endNum; i++) {
            sum = sum + i;
            System.out.println("i=" + i + " sum=" + sum);
        }
    }
}

```

출력 결과

```

i=1 sum=1
i=2 sum=3
i=3 sum=6

```

for vs while

앞서 같은 문제를 풀었던 while문과 for문을 서로 비교해보자.

While2_3

```
package loop;
```

```

public class While2_3 {
    public static void main(String[] args) {
        int sum = 0;
        int i = 1;
        int endNum = 3;

        while (i <= endNum) {
            sum = sum + i;
            System.out.println("i=" + i + " sum=" + sum);
            i++;
        }
    }
}

```

들을 비교했을 때 for문이 더 깔끔하다는 느낌을 받을 것이다. for문은 초기화, 조건 검사, 반복 후 작업 등이 규칙적으로 한 줄에 모두 들어 있어 코드를 이해하기 더 쉽다. 특히 반복을 위해 값이 증가하는 카운터 변수를 다른 부분과 명확하게 구분할 수 있다.

```
for (int i = 1; i <= endNum; i++)
```

여기서는 바로 변수 `i` 가 카운터 변수이다. 증가하면서 반복 횟수가 올라가고, 또 변수 `i` 를 사용해서 계속 반복할지 아니면 빠져나갈지 판단할 수 있다.

이렇게 반복 횟수에 직접적인 영향을 주는 변수를 선언부터, 값 증가, 또 조건식에 활용까지 `for (초기식; 조건식; 증감식)` 구조를 활용해서 처리하는 것이다.

덕분에 개발자는 루프 횟수와 관련된 코드와 나머지 코드를 명확하게 구분할 수 있다.

반면에 while을 보면 변수 `i` 를 선언하는 부분 그리고 `i++` 로 증가하는 부분이 기존 코드에 분산되어 있다.

for문2

for문 구조

```
for (초기식; 조건식; 증감식) {
    // 코드
}
```

for문에서 초기식, 조건식, 증감식은 선택이다. 다음과 같이 모두 생략해도 된다. 단 생략해도 각 영역을 구분하는 세미콜론(;)은 유지해야 한다.

```
for (;;) {  
    // 코드  
}
```

이렇게 하면 조건이 없기 때문에 무한 반복하는 코드가 된다. 따라서 다음과 같은 코드가 된다.

```
while (true) {  
    // 코드  
}
```

for문을 사용해서 다음 문제를 풀어보자.

문제: 1부터 시작하여 숫자를 계속 누적해서 더하다가 합계가 10보다 큰 처음 값은 얼마인가?

1 + 2 + 3 ... 계속 더하다가 처음으로 합이 10보다 큰 경우를 찾으면 된다.

Break2

```
package loop;  
  
public class Break2 {  
    public static void main(String[] args) {  
        int sum = 0;  
        int i = 1;  
  
        for (; ; ) {  
            sum += i;  
            if (sum > 10) {  
                System.out.println("합이 10보다 크면 종료: i=" + i + " sum=" + sum);  
                break;  
            }  
            i++;  
        }  
    }  
}
```

- `for (; ;)`를 보면 조건식이 없다. for문은 조건이 없으면 무한 반복한다.
- `sum > 10` 조건을 만족하면 `break`를 사용해서 while문을 빠져나간다.

실행 결과

```
합이 10보다 크면 종료: i=5 sum=15
```

for문은 증가하는 값이 무엇인지 초기식과 증감식을 통해서 쉽게 확인할 수 있다. 이 코드나 while문을 보면 어떤 값이 반복에 사용되는 증가 값인지 즉시 확인하기는 어렵다.

여기서는 `i` 가 증가하는 값이다. 따라서 다음과 같이 `i` 를 `for` 문에 넣어서 관리하도록 변경하면 더 깔끔한 코드가 된다.

Break3

```
package loop;

public class Break3 {
    public static void main(String[] args) {
        int sum = 0;

        for (int i = 1; ; i++) {
            sum += i;
            if (sum > 10) {
                System.out.println("합이 10보다 크면 종료: i=" + i + " sum=" + sum);
                break;
            }
        }
    }
}
```

정리하면 `for`문이 없이 `while`문으로 모든 반복을 다를 수 있다. 하지만 카운터 변수가 명확하거나, 반복 횟수가 정해진 경우에는 `for`문을 사용하는 것이 구조적으로 더 깔끔하고, 유지보수 하기 좋다.

참고

`for`문을 좀 더 편리하게 사용하도록 도와주는 향상된 `for`문 또는 `for-each`문으로 불리는 반복문도 있다. 이 부분은 뒤에서 설명한다.

중첩 반복문

반복문은 내부에 또 반복문을 만들 수 있다. `for`, `while` 모두 가능하다.

다음 코드를 작성하고 실행해보자.

Nested1

```
package loop;

public class Nested1 {
```

```

public static void main(String[] args) {
    for (int i = 0; i < 2; i++) {
        System.out.println("외부 for 시작 i:" + i);
        for (int j = 0; j < 3; j++) {
            System.out.println("-> 내부 for " + i + " - " + j);
        }
        System.out.println("외부 for 종료 i:" + i);
        System.out.println(); //라인 구분을 위해 실행
    }
}

```

실행 결과

```

외부 for 시작 i:0
-> 내부 for 0-0
-> 내부 for 0-1
-> 내부 for 0-2
외부 for 종료 i:0

```

```

외부 for 시작 i:1
-> 내부 for 1-0
-> 내부 for 1-1
-> 내부 for 1-2
외부 for 종료 i:1

```

외부 for는 2번, 내부 for는 3번 실행된다. 그런데 외부 for 1번당 내부 for가 3번 실행되기 때문에 외부(2) * 내부(3) 해서 총 6번의 내부 for 코드가 수행된다.

문제와 풀이 1

코딩이 처음이라면 필독!

프로그래밍이 처음이라면 아직 코딩 자체가 익숙하지 않기 때문에 문제와 풀이에 상당히 많은 시간을 쓰게 될 수 있다. 강의를 들을 때는 다 이해가 되는 것 같았는데, 막상 혼자 생각해서 코딩을 하려니 잘 안되는 것이다. 이것은 아직 코딩이 익숙하지 않기 때문인데, 처음 코딩을 하는 사람이라면 누구나 겪는 자연스러운 현상이다.

문제를 스스로 풀기 어려운 경우, 너무 고민하기보다는 먼저 **강의 영상의 문제 풀이 과정을 코드로 따라하면서 이해하자. 반드시 코드로 따라해야 한다.** 그래야 코딩하는 것에 조금씩 익숙해질 수 있다. 그런 다음에 정답을 지우

고 스스로 문제를 풀어보면 된다. 참고로 강의를 듣는 시간만큼 문제와 풀이에도 많은 시간을 들어야 제대로 성장할 수 있다!

문제: 자연수 출력

처음 10개의 자연수를 출력하는 프로그램을 작성해 보세요. 이때, `count`라는 변수를 사용해야 합니다.
`while`문, `for`문 2가지 버전의 정답을 만들어야 합니다.

출력 예시:

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

해답: 자연수 출력 - while

```
package loop.ex;  
  
public class WhileEx1 {  
  
    public static void main(String[] args) {  
        int count = 1;  
        while (count <= 10) {  
            System.out.println(count);  
            count++;  
        }  
    }  
}
```

해답: 자연수 출력 - for

```
package loop.ex;  
  
public class ForEx1 {  
  
    public static void main(String[] args) {  
        for (int count = 1; count <= 10; count++) {  
    }
```

```
        System.out.println(count);
    }
}
}
```

문제: 짝수 출력

반복문을 사용하여 처음 10개의 짝수를 출력하는 프로그램을 작성해 보세요. 이때, num이라는 변수를 사용하여 수를 표현해야 합니다.

while문, for문 2가지 버전의 정답을 만들어야 합니다.

출력 예시:

```
2  
4  
6  
8  
10  
12  
14  
16  
18  
20
```

해답: 짝수 출력 - while

```
package loop.ex;

public class WhileEx2 {

    public static void main(String[] args) {
        int num = 2;
        int count = 1;
        while (count <= 10) {
            System.out.println(num);
            num += 2;
            count++;
        }
    }
}
```

해답: 짝수 출력 - for

```

package loop.ex;

public class ForEx2 {

    public static void main(String[] args) {
        for (int num = 2, count = 1; count <= 10; num += 2, count++) {
            System.out.println(num);
        }
    }
}

```

문제: 누적 합 계산

반복문을 사용하여 1부터 max 까지의 합을 계산하고 출력하는 프로그램을 작성해 보세요. 이때, sum이라는 변수를 사용하여 누적 합을 표현하고, i라는 변수를 사용하여 카운트(1부터 max까지 증가하는 변수)를 수행해야 합니다.
while문, for문 2가지 버전의 정답을 만들어야 합니다.

출력 예시:

```

//max=1
1

//max=2
3

//max=3
6

//max=100
5050

```

정답: 누적 합 계산 - while

```

package loop.ex;

public class WhileEx3 {

    public static void main(String[] args) {
        int max = 100;

        int sum = 0;
        int i = 1;
    }
}

```

```
    while (i <= max) {
        sum += i;
        i++;
    }
    System.out.println(sum);
}
}
```

정답: 누적 합 계산 - for

```
package loop.ex;

public class ForEx3 {

    public static void main(String[] args) {
        int max = 100;

        int sum = 0;
        for (int i = 1; i <= max; i++) {
            sum += i;
        }
        System.out.println(sum);
    }
}
```

문제와 풀이2

문제: 구구단 출력

중첩 for문을 사용해서 구구단을 완성해라.

출력 형태

```
1 * 1 = 1
1 * 2 = 2
...
9 * 9 = 81
```

정답: 구구단 출력

```
package loop.ex;

public class NestedEx1 {
    public static void main(String[] args) {
        for(int i = 1; i <= 9; i++) {
            for(int j = 1; j <= 9; j++) {
                System.out.println(i + " * " + j + " = " + i * j);
            }
        }
    }
}
```

문제: 피라미드 출력

int rows를 선언해라.

이 수만큼 다음과 같은 피라미드를 출력하면 된다.

참고: `println()` 은 출력후 다음 라인으로 넘어간다. 라인을 넘기지 않고 출력하려면 `print()` 를 사용하면 된다.

예) `System.out.print("*")`

출력 형태

```
//rows = 2
*
**

//rows = 5
*
**
***
****
*****
```

정답: 피라미드 출력

```
package loop.ex;

public class NestedEx2 {
```

```

public static void main(String[] args) {
    int rows = 5;

    for(int i = 1; i <= rows; i++) {
        for(int j = 1; j <= i; j++) {
            System.out.print("*");
        }
        System.out.println();
    }
}

```

정리

while vs for

for문

장점:

1. 초기화, 조건 체크, 반복 후의 작업을 한 줄에서 처리할 수 있어 편리하다.
2. 정해진 횟수만큼의 반복을 수행하는 경우에 사용하기 적합하다.
3. 루프 변수의 범위가 for 루프 블록에 제한되므로, 다른 곳에서 이 변수를 실수로 변경할 가능성이 적다.

단점:

1. 루프의 조건이 루프 내부에서 변경되는 경우, for 루프는 관리하기 어렵다.
2. 복잡한 조건을 가진 반복문을 작성하기에는 while문이 더 적합할 수 있다.

while문

장점:

1. 루프의 조건이 루프 내부에서 변경되는 경우, while 루프는 이를 관리하기 쉽다.
2. for 루프보다 더 복잡한 조건과 시나리오에 적합하다.
3. 조건이 충족되는 동안 계속해서 루프를 실행하며, 종료 시점을 명확하게 알 수 없는 경우에 유용하다.

단점:

1. 초기화, 조건 체크, 반복 후의 작업이 분산되어 있어 코드를 이해하거나 작성하기 어려울 수 있다.
2. 루프 변수가 while 블록 바깥에서도 접근 가능하므로, 이 변수를 실수로 변경하는 상황이 발생할 수 있다.

한줄로 정리하자면 정해진 횟수만큼 반복을 수행해야 하면 for문을 사용하고 그렇지 않으면 while문을 사용하면 된다.
물론 이것이 항상 정답은 아니니 기준으로 삼는 정도로 이해하자

6. 스코프, 형변환

#1.인강/0.자바/1.자바-입문

- /스코프1 - 지역 변수와 스코프
- /스코프2 - 스코프 존재 이유
- /형변환1 - 자동 형변환
- /형변환2 - 명시적 형변환
- /계산과 형변환
- /정리

스코프1 - 지역 변수와 스코프

변수는 선언한 위치에 따라 지역 변수, 멤버 변수(클래스 변수, 인스턴스 변수)와 같이 분류된다.

우리가 지금까지 학습한 변수들은 모두 영어로 로컬 변수(Local Variable) 한글로 지역 변수라 한다. 나머지 변수들은 뒤에서 학습한다 지금은 지역 변수만 기억하자.

지역 변수는 이름 그대로 특정 지역에서만 사용할 수 있는 변수라는 뜻이다. 그 특정 지역을 벗어나면 사용할 수 없다.

여기서 말하는 지역이 바로 변수가 선언된 코드 블록({})이다. 지역 변수는 자신이 선언된 코드 블록({}) 안에서만 생존하고, 자신이 선언된 코드 블록을 벗어나면 제거된다. 따라서 이후에는 접근할 수 없다.

예제 코드로 확인해보자

Scope1

```
package scope;

public class Scope1 {
    public static void main(String[] args) {
        int m = 10; //m 생존 시작
        if (true) {
            int x = 20; //x 생존 시작
            System.out.println("if m = " + m); //블록 내부에서 블록 외부는 접근 가능
            System.out.println("if x = " + x);
        } //x 생존 종료

        //System.out.println("main x = " + x); //오류, 변수 x에 접근 불가
        System.out.println("main m = " + m);
    } //m 생존 종료
```

- ```
}
```
- int m
    - int m은 main{}의 코드 블록안에서 선언되었다. 따라서 변수를 선언한 시점부터 main{}의 코드 블록이 종료될 때 까지 생존한다.
    - if{} 블록 내부에서도 외부 블록에서 선언된 m에 접근할 수 있다. 쉽게 이야기해서 생존 범위만 맞으면 다 접근할 수 있다.
  - int x
    - int x는 if{} 블록안에서 선언되었다. 따라서 변수를 선언한 시점부터 if{}의 코드 블록이 종료될 때 까지 생존한다.
    - if{} 내부에서는 자신의 범위에서 선언한 x에 당연히 접근할 수 있다.
    - if{} 코드 블록이 끝나버리면 x는 제거된다. 따라서 더는 x에 접근할 수 없다. 따라서 이후에 접근하면 cannot find symbol이라는 변수 이름을 찾을 수 없다는 컴파일 오류가 발생한다.

정리하면 지역 변수는 본인의 코드 블록 안에서만 생존한다. 그리고 자신의 코드 블록 안에서는 얼마든지 접근할 수 있다. 하지만 자신의 코드 블록을 벗어나면 제거되기 때문에 접근할 수 없다.

이렇게 변수의 접근 가능한 범위를 스코프(Scope)라 한다. 참고로 Scope를 번역하면 범위라는 뜻이다.

int m은 main{} 전체에서 접근할 수 있기 때문에 스코프가 넓고, int x는 if{} 코드 블록 안에서만 접근할 수 있기 때문에 스코프가 좁다.

이번에는 if{} 대신에 for{}를 사용하는 예제를 보자

## Scope2

```
package scope;

public class Scope2 {
 public static void main(String[] args) {
 int m = 10;
 for (int i = 0; i < 2; i++) { //블록 내부, for문 내
 System.out.println("for m = " + m); //블록 내부에서 외부는 접근 가능
 System.out.println("for i = " + i);
 } //i 생존 종료

 //System.out.println("main i = " + i); //오류, i에 접근 불가
 System.out.println("main m = " + m);
 }
}
```

for 문으로 바뀐 것을 제외하면 앞의 예제와 비슷한 예제이다.

for 문의 경우 for(int i=0;...) 과 같이 for 문 안에서 초기식에 직접 변수를 선언할 수 있다. 그리고 이렇게 선

언한 변수는 `for` 문 코드 블록 안에서만 사용할 수 있다.

## 스코프2 - 스코프 존재 이유

변수를 선언한 시점부터 변수를 계속 사용할 수 있게 해도 되지 않을까? 왜 복잡하게 접근 범위(스코프)라는 개념을 만들었을까?

이해를 위해 다음 코드를 보자

### Scope3\_1

```
package scope;

public class Scope3_1 {
 public static void main(String[] args) {
 int m = 10;
 int temp = 0;
 if (m > 0) {
 temp = m * 2;
 System.out.println("temp = " + temp);
 }
 System.out.println("m = " + m);
 }
}
```

조건이 맞으면 변수 `m`의 값을 2배 증가해서 출력하는 코드이다. 여기서 2배 증가한 값을 저장해두기 위해 임시 변수 `temp`를 사용했다. 그런데 이 코드는 좋은 코드라고 보기는 어렵다. 왜냐하면 임시 변수 `temp`는 `if` 조건이 만족할 때 임시로 잠깐 사용하는 변수이다. 그런데 임시 변수 `temp` `main()` 코드 블록에 선언되어 있다. 이렇게 되면 다음과 같은 문제가 발생한다.

- **비효율적인 메모리 사용:** `temp`은 `if` 코드 블록에서만 필요하지만, `main()` 코드 블록이 종료될 때 까지 메모리에 유지된다. 따라서 불필요한 메모리가 낭비된다. 만약 `if` 코드 블록 안에 `temp`를 선언했다면 자바를 구현하는 곳에서 `if` 코드 블록의 종료 시점에 이 변수를 메모리에서 제거해서 더 효율적으로 메모리를 사용할 수 있다.
- **코드 복잡성 증가:** 좋은 코드는 군더더기 없는 단순한 코드이다. `temp`은 `if` 코드 블록에서만 필요하고, 여기서만 사용하면 된다. 만약 `if` 코드 블록 안에 `temp`를 선언했다면 `if`가 끝나고 나면 `temp`를 전혀 생각하지 않아도 된다. 머리속에서 생각할 변수를 하나 줄일 수 있다. 그런데 지금 작성한 코드는 `if` 코드 블록이 끝나도 `main()` 어디서나 `temp`를 여전히 접근할 수 있다. 누군가 이 코드를 유지보수 할 때 `m`은 물론이고 `temp`까지 계속 신경써야 한다. 스코프가 불필요하게 넓은 것이다. 지금은 코드가 매우 단순해서 이해하는데 어려움이 없겠지만 실무에서는 코드가 매우 복잡한 경우가 많다.

`temp`의 스코프를 꼭 필요한 곳으로 한정해보자

### Scope3\_2

```
package scope;

public class Scope3_2 {
 public static void main(String[] args) {
 int m = 10;
 if (m > 0) {
 int temp = m * 2;
 System.out.println("temp = " + temp);
 }
 System.out.println("m = " + m);
 }
}
```

`temp`를 `if` 코드 블록 안에서 선언했다. 이제 `temp`는 `if` 코드 블록 안으로 스코프가 줄어든다. 덕분에 `temp` 메모리를 빨리 제거해서 메모리를 효율적으로 사용하고, `temp` 변수를 생각해야 하는 범위를 줄여서 더 유지보수하기 좋은 코드를 만들었다.

## while문 vs for문 - 스코프 관점

이제 스코프 관점에서 `while` 문과 `for` 문을 비교해보자

다음 코드들은 기존에 반복문에서 학습했던 코드이다.

### While

```
package loop;

public class While2_3 {
 public static void main(String[] args) {
 int sum = 0;
 int i = 1;
 int endNum = 3;

 while (i <= endNum) {
 sum = sum + i;
 System.out.println("i=" + i + " sum=" + sum);
 }
 }
}
```

```

 i++;
 }
 //... 아래에 더 많은 코드들이 있다고 가정
}
}

```

## For

```

package loop;

public class For2 {
 public static void main(String[] args) {
 int sum = 0;
 int endNum = 3;

 for (int i = 1; i <= endNum; i++) {
 sum = sum + i;
 System.out.println("i=" + i + " sum=" + sum);
 }
 //... 아래에 더 많은 코드들이 있다고 가정
 }
}

```

변수의 스코프 관점에서 카운터 변수 `i`를 비교해보자.

- `while` 문의 경우 변수 `i`의 스코프가 `main()` 메서드 전체가 된다. 반면에 `for` 문의 경우 변수 `i`의 스코프가 `for`문 안으로 한정된다.
- 따라서 변수 `i`와 같이 `for` 문 안에서만 사용되는 카운터 변수가 있다면 `while` 문 보다는 `for` 문을 사용해서 스코프의 범위를 제한하는 것이 메모리 사용과 유지보수 관점에서 더 좋다.

## 정리

- 변수는 꼭 필요한 범위로 한정해서 사용하는 것이 좋다. 변수의 스코프는 꼭 필요한 곳으로 한정해서 사용하자. 메모리를 효율적으로 사용하고 더 유지보수하기 좋은 코드를 만들 수 있다.
- 좋은 프로그램은 무한한 자유가 있는 프로그램이 아니라 적절한 제약이 있는 프로그램이다.

## 형변환1 - 자동 형변환

## 형변환

- 작은 범위에서 큰 범위로는 당연히 값을 넣을 수 있다.
  - 예) int → long → double
- 큰 범위에서 작은 범위는 다음과 같은 문제가 발생할 수 있다.
  - 소수점 버림
  - 오버플로우

### 작은 범위에서 큰 범위로 대입은 허용한다

자바에서 숫자를 표현할 수 있는 범위는 다음과 같다.

```
int < long < double
```

int 보다는 long이, long 보다는 double이 더 큰 범위를 표현할 수 있다.

작은 범위에서 큰 범위에 값을 대입하는 다음 코드를 실행하면 특별한 문제없이 잘 수행된다.

## Casting1

```
package casting;

public class Casting1 {

 public static void main(String[] args) {
 int intValue = 10;
 long longValue;
 double doubleValue;

 longValue = intValue; // int -> long
 System.out.println("longValue = " + longValue); //longValue = 10

 doubleValue = intValue; // int -> double
 System.out.println("doubleValue1 = " + doubleValue); //doubleValue1 =
10.0

 doubleValue = 20L; // long -> double
 System.out.println("doubleValue2 = " + doubleValue); //doubleValue2 =
20.0
 }
}
```

## 실행 결과

```
longValue = 10
doubleValue1 = 10.0
doubleValue2 = 20.0
```

- 자바는 기본적으로 같은 타입에 값을 대입할 수 있다. 그런데 다른 타입에 값을 대입하면 어떻게 될까?
- `int` → `long`을 비교해보면 `long`이 `int` 보다 더 큰 숫자 범위를 표현한다. 작은 범위 숫자 타입에 대입을 하면 문제가 되지 않는다. 만약 이런 경우까지 오류가 발생한다면 개발이 너무 불편할 것이다.
- `long` → `double`의 경우에도 `double`은 부동 소수점을 사용하기 때문에 더 큰 숫자 범위를 표현한다. 따라서 대입할 수 있다.
- 정리하면 작은 범위에서 큰 범위로의 대입은 자바 언어에서 허용한다. 쉽게 이야기하면 큰 그릇은 작은 그릇에 담 긴 내용물을 담을 수 있다.

## 자동 형변환

하지만 결국 대입하는 형(타입)을 맞추어야 하기 때문에 개념적으로는 다음과 같이 동작한다.

```
//intValue = 10
doubleValue = intValue
doubleValue = (double) intValue //형 맞추기
doubleValue = (double) 10 //변수 값 읽기
doubleValue = 10.0 //형변환
```

이렇게 앞에 `(double)`과 같이 적어주면 `int` 형이 `double` 형으로 형이 변한다. 이렇게 형이 변경되는 것을 형변환 이라 한다.

작은 범위 숫자 타입에서 큰 범위 숫자 타입으로의 대입은 개발자가 이렇게 직접 형변환을 하지 않아도 된다. 이런 과정이 자동으로 일어나기 때문에 **자동 형변환**, 또는 **묵시적 형변환**이라 한다.

## 형변환2 - 명시적 형변환

이번에는 반대로 큰 범위에서 작은 범위로 대입해보자.

**큰 범위에서 작은 범위 대입은 명시적 형변환이 필요하다**

`double`은 실수를 표현할 수 있다. 따라서 `1.5`가 가능하다. 그런데 `int`는 실수를 표현할 수 없다. 이 경우 `double` → `int`로 대입하면 어떻게 될까?

## Casting2

```
package casting;

public class Casting2 {

 public static void main(String[] args) {
```

```
 double doubleValue = 1.5;
 int intValue = 0;

 //intValue = doubleValue; //컴파일 오류 발생
 intValue = (int) doubleValue; //형변환
 System.out.println(intValue); //출력:1
}
}
```

다음 코드의 앞부분에 있는 주석을 풀면(주석을 제거하면) 컴파일 오류가 발생한다.

```
intValue = doubleValue //컴파일 오류 발생
```

```
java: incompatible types: possible lossy conversion from double to int
//java: 호환되지 않는 유형: double에서 int로의 가능한 손실 변환
```

int 형은 double 형보다 숫자의 표현 범위가 작다. 그리고 실수를 표현할 수도 없다. 따라서 이 경우 숫자가 손실되는 문제가 발생할 수 있다. 쉽게 이야기해서 큰 컵에 담긴 물을 작은 컵에 옮겨 담으려고 하니, 손실이 발생할 수 있다는 것이다.

이런 문제는 매우 큰 버그를 유발할 수 있다. 예를 들어서 은행 프로그램이 고객에게 은행 이자를 계산해서 입금해야 하는데 만약 이런 코드가 아무런 오류 없이 수행된다면 끔찍한 문제를 만들 수 있다. 그래서 자바는 이런 경우 컴파일 오류를 발생시킨다. 항상 강조하지만 컴파일 오류는 문제를 가장 빨리 발견할 수 있는 좋은 오류이다.

## 형변환

하지만 만약 이런 위험을 개발자가 직접 감수하고도 값을 대입하고 싶다면 데이터 타입을 강제로 변경할 수 있다.

예를 들어서 대략적인 결과를 보고 싶은데, 이때 소수점을 버리고 정수로만 보고 싶을 수 있다.

형변환은 다음과 같이 변경하고 싶은 데이터 타입을 (int) 와 같이 괄호를 사용해서 명시적으로 입력하면 된다.

```
intValue = (int) doubleValue; //형변환
```

이것을 형(타입)을 바꾼다고 해서 형변환이라 한다. 영어로는 캐스팅이라 한다. 그리고 개발자가 직접 형변환 코드를 입력한다고 해서 명시적 형변환이라 한다.

## 캐스팅 용어

"캐스팅"은 영어 단어 "cast"에서 유래되었다. "cast"는 금속이나 다른 물질을 녹여서 특정한 형태나 모양으로 만드는 과정을 의미한다.

## 명시적 형변환 과정

```
//doubleValue = 1.5
```

```
intValue = (int) doubleValue;
intValue = (int) 1.5; //doubleValue에 있는 값을 읽는다.
intValue = 1; //(int)로 형변환 한다. intValue에 int형인 숫자 1을 대입한다.
```

형변환 후 출력해보면 숫자 1이 출력되는 것을 확인할 수 있다.

참고로 형변환을 한다고 해서 `doubleValue` 자체의 타입이 변경되거나 그 안에 있는 값이 변경되는 것은 아니다. `doubleValue`에서 읽은 값을 형변환 하는 것이다. `doubleValue` 안에 들어있는 값은 1.5로 그대로 유지된다. 참고로 변수의 값은 대입연산자(=)를 사용해서 직접 대입할 때만 변경된다.

## 형변환과 오버플로우

형변환을 할 때 만약 작은 숫자가 표현할 수 있는 범위를 넘어서면 어떻게 될까?

## Casting3

```
package casting;

public class Casting3 {

 public static void main(String[] args) {
 long maxValue = 2147483647; //int 최고값
 long maxIntOver = 2147483648L; //int 최고값 + 1(초과)
 int intValue = 0;

 intValue = (int) maxValue; //형변환
 System.out.println("maxValue casting=" + intValue); //출력:2147483647

 intValue = (int) maxIntOver; //형변환
 System.out.println("maxIntOver casting=" + intValue); //출력:-2147483648
 }
}
```

## 출력 결과

```
maxValue casting=2147483647
maxIntOver casting=-2147483648
```

## 정상 범위

`long maxValue = 2147483647`를 보면 `int`로 표현할 수 있는 가장 큰 숫자인 2147483647를 입력했다. 이 경우 `int`로 표현할 수 있는 범위에 포함되기 때문에 다음과 같이 `long → int`로 형변환을 해도 아무런 문제가 없다.

```
intValue = (int) maxIntValue; //형변환
```

어떻게 수행되는지 확인해보자.

```
maxIntValue = 2147483647; //int 최고값
intValue = (int) maxIntValue; //변수 값 읽기
intValue = (int) 2147483647L; //형변환
intValue = 2147483647;
```

## 초과 범위

다음으로 `long maxIntOver = 2147483648L`를 보면 `int`로 표현할 수 있는 가장 큰 숫자인 `2147483647`보다 1큰 숫자를 입력했다. 이 숫자는 리터럴은 `int` 범위를 넘어가기 때문에 마지막에 `L`을 붙여서 `long` 형을 사용해야 한다.

이 경우 `int`로 표현할 수 있는 범위를 넘기 때문에 다음과 같이 `long → int`로 형변환 하면 문제가 발생한다.

```
intValue = (int) maxIntOver; //형변환
```

어떻게 수행되는지 확인해보자.

```
maxIntOver = 2147483648L; //int 최고값 + 1
intValue = (int) maxIntOver; //변수 값 읽기
intValue = (int) 2147483648L; //형변환 시도
intValue = -2147483648;
```

- 결과를 보면 `-2147483648`이라는 전혀 다른 숫자가 보인다. `int` 형은 `2147483648L`를 표현할 수 있는 방법이 없다. 이렇게 기존 범위를 초과해서 표현하게 되면 전혀 다른 숫자가 표현되는데, 이런 현상을 오버플로우라 한다.
- 보통 오버플로우가 발생하면 마치 시계가 한바퀴 돈 것처럼 다시 처음부터 시작한다. 참고로 `-2147483648` 숫자는 `int`의 가장 작은 숫자이다.
- 중요한 것은 오버플로우가 발생하는 것 자체가 문제라는 점이다! 오버플로우가 발생했을 때 결과가 어떻게 되는지 계산하는데 시간을 낭비하면 안된다! 오버플로우 자체가 발생하지 않도록 막아야 한다. 이 경우 단순히 대입하는 변수(`intValue`)의 타입을 `int → long`으로 변경해서 사이즈를 늘리면 오버플로우 문제가 해결된다.

## 계산과 형변환

형변환은 대입 뿐만 아니라, 계산을 할 때도 발생한다.

## Casting4

```
package casting;

public class Casting4 {

 public static void main(String[] args) {
 int div1 = 3 / 2;
 System.out.println("div1 = " + div1); //1

 double div2 = 3 / 2;
 System.out.println("div2 = " + div2); //1.0

 double div3 = 3.0 / 2;
 System.out.println("div3 = " + div3); //1.5

 double div4 = (double) 3 / 2;
 System.out.println("div4 = " + div4); //1.5

 int a = 3;
 int b = 2;
 double result = (double) a / b;
 System.out.println("result = " + result); //1.5
 }
}
```

## 출력 결과

```
div1 = 1
div2 = 1.0
div3 = 1.5
div4 = 1.5
result = 1.5
```

자바에서 계산은 다음 2가지를 기억하자.

1. 같은 타입끼리의 계산은 같은 타입의 결과를 낸다.
  - `int + int`는 `int`를, `double + double`은 `double`의 결과가 나온다.
2. 서로 다른 타입의 계산은 큰 범위로 자동 형변환이 일어난다.
  - `int + long`은 `long + long`으로 자동 형변환이 일어난다.
  - `int + double`은 `double + double`로 자동 형변환이 일어난다.

다양한 타입별로 더 자세히 들어가면 약간 차이가 있지만 이 기준으로 이해하면 충분하다.

예시를 통해서 자세히 이해해보자.

```
int div1 = 3 / 2; //int / int
int div1 = 1; //int / int이므로 int타입으로 결과가 나온다.
```

```
double div2 = 3 / 2; //int / int
double div2 = 1; //int / int이므로 int타입으로 결과가 나온다.
double div2 = (double) 1; //int -> double에 대입해야 한다. 자동 형변환 발생
double div2 = 1.0; // 1(int) -> 1.0(double)로 형변환 되었다.
```

```
double div3 = 3.0 / 2; //double / int
double div3 = 3.0 / (double) 2; //double / int이므로, double / double로 형변환이 발생한다.
double div3 = 3.0 / 2.0; //double / double -> double이 된다.
double div3 = 1.5;
```

```
double div4 = (double) 3 / 2; //명시적 형변환을 사용했다. (double) int / int
double div4 = (double) 3 / (double) 2; //double / int이므로, double / double로 형변환이 발생한다.
double div4 = 3.0 / 2.0; //double / double -> double이 된다.
double div4 = 1.5;
```

3 / 2와 같이 int 형끼리 나눗셈을 해서 소수까지 구하고 싶다면 div4의 예제처럼 명시적 형변환을 사용하면 된다.

물론 변수를 사용하는 경우에도 다음과 같이 형변환을 할 수 있다.

```
int a = 3;
int b = 2;
double result = (double) a / b;
```

## 처리 과정

```
double result = (double) a / b; //(double) int / int
double result = (double) 3 / 2; //변수 값 읽기
double result = (double) 3 / (double) 2; //double + int 이므로 더 큰 범위로 형변환
double result = 3.0 / 2.0; //(double / double) -> double이 된다.
double result = 1.5;
```

# 정리

## 형변환

`int → long → double`

- 작은 범위에서 큰 범위로는 대입할 수 있다.
  - 이것을 묵시적 형변환 또는 자동 형변환이라 한다.
- 큰 범위에서 작은 범위의 대입은 다음과 같은 문제가 발생할 수 있다. 이때는 명시적 형변환을 사용해야 한다.
  - 소수점 버림
  - 오버플로우
- 연산과 형변환
  - 같은 타입은 같은 결과를 낸다.
  - 서로 다른 타입의 계산은 큰 범위로 자동 형변환이 일어난다.

# 7. 훈련

#1.인강/0.자바/1.자바-입문

- /Scanner 학습
- /Scanner - 기본 예제
- /Scanner - 반복 예제
- /문제와 풀이1
- /문제와 풀이2
- /문제와 풀이3
- /문제와 풀이4
- /정리

## Scanner 학습

### 훈련 시작

지금까지 학습한 변수, 연산자, 조건문, 반복문은 프로그래밍의 가장 기본이 되는 기능이다. 대부분의 프로그램 언어는 이 기능을 필수로 가진다. 그리고 프로그래머가 하는 일의 대부분은 지금까지 설명한 변수, 연산자, 조건문, 반복문을 다루는 일이다. 그래서 이 기능을 잘 다루는 것이 무엇보다 중요하다.

이번 시간에는 지금까지 배운 내용들을 훈련하는 시간이다. 여러분이 다음으로 나아가기 전에 최소한의 기본기를 훈련하는 시간으로 생각하자

지금까지 학습할 때 한가지 아쉬움이 있었는데, 바로 사용자의 입력이 없었다는 점이다.

이번 시간에는 사용자의 입력을 받는 방법을 배워서, 좀 더 그럴듯한 프로그램을 만들어보자.

#### 백문이 불여일타!

변수, 연산자, 조건문, 반복문을 머리로 이해하는 것은 전혀 어렵지 않다. 하지만 머리로 생각만 하는 것은 수영을 이렇게 해야하겠지? 라고 머리로 생각하는 것과 같다. 중요한 것은 코딩을 몸이 익히는 것이다. 그러기 위해서는 직접 코딩하는 것이 무엇보다 중요하다! 학생때처럼 단순히 외우는 방식으로는 좋은 프로그래머가 될 수 없다. 예제 코드는 모두 따라해보고, 문제도 직접 다 풀어보자, 문제가 안풀리면 답을 보고 코드를 따라친 다음에 기존 코드를 모두 지우고 처음부터 본인이 스스로 다시 풀어보아도 좋다. 백문이 불여일타!

## Scanner

System.out을 통해서 출력을 했듯이, System.in을 통해서 사용자의 입력을 받을 수 있다. 그런데 자바가 제공하

는 `System.in`을 통해서 사용자 입력을 받으려면 여러 과정을 거쳐야해서 복잡하고 어렵다.  
자바는 이런 문제를 해결하기 위해 `Scanner`라는 클래스를 제공한다. 이 클래스를 사용하면 사용자 입력을 매우 편리하게 받을 수 있다.

## Scanner 예제1

### Scanner1

```
package scanner;

import java.util.Scanner;

public class Scanner1 {

 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 System.out.print("문자열을 입력하세요:");
 String str = scanner.nextLine(); // 입력을 String으로 가져옵니다.
 System.out.println("입력한 문자열: " + str);

 System.out.print("정수를 입력하세요:");
 int intValue = scanner.nextInt(); // 입력을 int로 가져옵니다.
 System.out.println("입력한 정수: " + intValue);

 System.out.print("실수를 입력하세요:");
 double doubleValue = scanner.nextDouble(); // 입력을 double로 가져옵니다.
 System.out.println("입력한 실수: " + doubleValue);
 }
}
```

- `Scanner scanner = new Scanner(System.in);`
  - 이 코드는 객체와 클래스를 배워야 정확히 이해할 수 있다. 지금은 `Scanner`의 기능을 사용하기 위해 `new`를 사용해서 `Scanner`를 만든다 정도로 이해하면 된다. `Scanner`는 `System.in`을 사용해서 사용자의 입력을 편리하게 받도록 도와준다.
  - `Scanner scanner` 코드는 `scanner` 변수를 선언하는 것이다. 이제부터 `scanner` 변수를 통해서 `scanner`를 사용할 수 있다.
- `scanner.nextLine()`
  - 엔터(`\n`)을 입력할 때 까지 문자를 가져온다.
- `scanner.nextInt()`
  - 입력을 `int`형으로 가져온다. 정수 입력에 사용한다.

- `scanner.nextDouble()`
  - 입력을 `double` 형으로 가져온다. 실수 입력에 사용한다.

## 출력 예시

```
문자열을 입력하세요:hello
입력한 문자열: hello
정수를 입력하세요:10
입력한 정수: 10
실수를 입력하세요:1.5
입력한 실수: 1.5
```

## 주의! - 다른 타입 입력시 오류

타입이 다르면 오류가 발생한다. 예제와 같이 숫자에 문자를 입력하면 오류가 발생한다.

```
문자열을 입력하세요:hello
입력한 문자열: hello
정수를 입력하세요:백만원
Exception in thread "main" java.util.InputMismatchException
 at java.base/java.util.Scanner.throwFor(Scanner.java:939)
 at java.base/java.util.Scanner.next(Scanner.java:1594)
 at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
 at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
 at scanner.Scanner1.main(Scanner1.java:15)
```

## `print()` vs `println()`

다음 코드를 보면 `println()` 이 아니라 `print()` 를 사용한다.

```
System.out.print("문자열을 입력하세요:")
```

그 이유는 다음과 같다.

`print()` 출력하고 다음 라인으로 넘기지 않는다.

```
System.out.print("hello");
System.out.print("world");
//결과: helloworld
```

`println()` 출력하고 다음 라인으로 넘긴다.

```
System.out.println("hello");
System.out.println("world");
//결과:
hello
world
```

우리가 엔터 키를 치면 (\n)이라는 문자를 남기는 것이다.

이 문자는 영어로 new line character, 한글로 줄바꿈 문자 또는 개행 문자라고 하는데, 이를 그대로 새로운 라인으로 넘기라는 뜻이다. 콘솔에서는 이 문자를 보고 다음 라인으로 넘긴다.

`println()`은 `print()`의 마지막에 \n을 추가한다. 따라서 다음 코드는 `println()`과 같다.

```
System.out.print("hello\n");
System.out.print("world\n");
//결과:
hello
world
```

## Scanner - 기본 예제

### Scanner 예제2

이번에는 `Scanner`를 활용하는 간단한 예제를 만들어보자.

두 수를 입력받고 그 합을 출력하는 예제이다.

#### Scanner2

```
package scanner;

import java.util.Scanner;

public class Scanner2 {

 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 System.out.print("첫 번째 숫자를 입력하세요:");
 int num1 = scanner.nextInt();

 System.out.print("두 번째 숫자를 입력하세요:");
 int num2 = scanner.nextInt();
 }
}
```

```

 int sum = num1 + num2;
 System.out.println("두 숫자의 합: " + sum);
 }
}

```

## 실행 결과

```

첫 번째 숫자를 입력하세요:10
두 번째 숫자를 입력하세요:20
두 숫자의 합: 30

```

이해하는데 어려움은 없을 것이다.

## Scanner 예제3

사용자로부터 두 개의 정수를 입력 받고, 더 큰 수를 출력하는 프로그램을 작성해보자. 두 숫자가 같은 경우 두 숫자는 같다고 출력하면 된다.  
조건문을 사용해서 처리할 수 있다.

### Scanner3

```

package scanner;

import java.util.Scanner;

public class Scanner3 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 System.out.print("첫 번째 숫자를 입력하세요:");
 int num1 = scanner.nextInt();

 System.out.print("두 번째 숫자를 입력하세요:");
 int num2 = scanner.nextInt();

 if (num1 > num2) {
 System.out.println("더 큰 숫자: " + num1);
 } else if (num2 > num1) {
 System.out.println("더 큰 숫자: " + num2);
 } else {
 System.out.println("두 숫자는 같습니다.");
 }
 }
}

```

```
 }
}
```

## 실행 결과

```
첫 번째 숫자를 입력하세요:10
두 번째 숫자를 입력하세요:20
더 큰 숫자: 20
```

```
첫 번째 숫자를 입력하세요:5
두 번째 숫자를 입력하세요:5
두 숫자는 같습니다.
```

## Scanner - 반복 예제

우리가 지금까지 개발한 프로그램들은 단 한 번의 결과 출력 후 종료되는 일회성 프로그램이었다. 실제 프로그램들은 이렇게 한 번의 결과만 출력하고 종료되지 않는다. 한 번 실행하면 사용자가 프로그램을 종료할 때 까지 반복해서 실행된다. 이제부터는 사용자의 입력을 지속해서 받아들이고, 반복해서 동작하는 프로그램을 개발해보자.

Scanner 와 반복문을 함께 사용하면 된다.

### Scanner 반복 예제1

- 사용자가 입력한 문자열을 그대로 출력하는 예제를 만들어보자.
- `exit`라는 문자가 입력되면 프로그램을 종료한다.
- 프로그램은 반복해서 실행된다.

#### ScannerWhile1

```
package scanner;

import java.util.Scanner;

public class ScannerWhile1 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
```

```

 while (true) {
 System.out.print("문자열을 입력하세요(exit: 종료):");
 String str = scanner.nextLine();
 if (str.equals("exit")) {
 System.out.println("프로그램을 종료합니다.");
 break;
 }

 System.out.println("입력한 문자열: " + str);
 }
 }
}

```

## 실행 결과

```

문자열을 입력하세요(exit: 종료):hello
입력한 문자열: hello
문자열을 입력하세요(exit: 종료):안녕?
입력한 문자열: 안녕?
문자열을 입력하세요(exit: 종료):exit
프로그램을 종료합니다.

```

while (true) : 중간에 break 문을 만나기 전까지 무한 반복한다.

```

if (str.equals("exit")) {
 System.out.println("프로그램을 종료합니다.");
 break;
}

```

입력 받은 문자가 "exit" 이면 "프로그램을 종료합니다."를 출력하고 break 문을 통해서 while문을 빠져나간다.

## Scanner 반복 예제2

- 첫 번째 숫자와 두 번째 숫자를 더해서 출력하는 프로그램을 개발하자.
- 첫 번째 숫자와 두 번째 숫자 모두 0을 입력하면 프로그램을 종료한다.
- 프로그램은 반복해서 실행된다.

### ScannerWhile2

```

package scanner;

import java.util.Scanner;

```

```

public class ScannerWhile2 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 System.out.println("첫 번째 숫자와 두 번째 숫자 모두 0을 입력하면 프로그램을 종료합니다.");
 while (true) {
 System.out.print("첫 번째 숫자를 입력하세요:");
 int num1 = scanner.nextInt();

 System.out.print("두 번째 숫자를 입력하세요:");
 int num2 = scanner.nextInt();

 if (num1 == 0 && num2 == 0) {
 System.out.println("프로그램 종료");
 break;
 }

 int sum = num1 + num2;
 System.out.println("두 숫자의 합: " + sum);
 }
 }
}

```

## 실행 결과

```

첫 번째 숫자와 두 번째 숫자 모두 0을 입력하면 프로그램을 종료합니다.
첫 번째 숫자를 입력하세요:10
두 번째 숫자를 입력하세요:20
두 숫자의 합: 30
첫 번째 숫자를 입력하세요:0
두 번째 숫자를 입력하세요:0
프로그램 종료

```

## 종료 부분

```

if (num1 == 0 && num2 == 0) {
 System.out.println("프로그램 종료");
 break;
}

```

```
num1 == 0 && num2 == 0
```

- 입력 받은 num1, num2 둘다 함께 0일 때 "프로그램 종료" 를 출력하고 break를 통해 while문을 빠져나간다.
- 비교 연산자를 사용했다. true && true → true 이다. 따라서 두 조건이 모두 참이어야 if문의 코드 블럭이 실행된다.

## Scanner 반복 예제3

사용자 입력을 받아 그 합계를 계산하는 프로그램을 작성해야 한다.

사용자는 한 번에 한 개의 정수를 입력할 수 있으며, 사용자가 0을 입력하면 프로그램은 종료된다. 이 때, 프로그램이 종료될 때까지 사용자가 입력한 모든 정수의 합을 출력해야 한다.

### ScannerWhile3

```
package scanner;

import java.util.Scanner;

public class ScannerWhile3 {
 public static void main(String[] args) {
 Scanner input = new Scanner(System.in);

 int sum = 0;

 while (true) {
 System.out.print("정수를 입력하세요 (0을 입력하면 종료) : ");
 int number = input.nextInt();

 if (number == 0) {
 break;
 }

 sum += number;
 }

 System.out.println("입력한 모든 정수의 합 : " + sum);
 }
}
```

### 실행 결과

```
정수를 입력하세요 (0을 입력하면 종료) : 1
정수를 입력하세요 (0을 입력하면 종료) : 2
```

```
정수를 입력하세요 (0을 입력하면 종료) : 3
정수를 입력하세요 (0을 입력하면 종료) : 0
입력한 모든 정수의 합 : 6
```

- `int sum`: 이곳에 사용자가 입력한 값을 누적한다.
- `sum += number`: 사용자가 입력한 `number` 값을 `sum`에 누적해서 더한다.
  - `+=` 을 사용했으므로 다음 코드와 같다: `sum = sum + number`

## 문제와 풀이 1

### 코딩이 처음이라면 필독!

프로그래밍이 처음이라면 아직 코딩 자체가 익숙하지 않기 때문에 문제와 풀이에 상당히 많은 시간을 쓰게 될 수 있다. 강의를 들을 때는 다 이해가 되는 것 같았는데, 막상 혼자 생각해서 코딩을 하려니 잘 안되는 것이다. 이것은 아직 코딩이 익숙하지 않기 때문인데, 처음 코딩을 하는 사람이라면 누구나 겪는 자연스러운 현상이다.

문제를 스스로 풀기 어려운 경우, 너무 고민하기보다는 먼저 **강의 영상의 문제 풀이 과정을 코드로 따라하면서 이해하자. 반드시 코드로 따라해야 한다.** 그래야 코딩하는 것에 조금씩 익숙해질 수 있다. 그런 다음에 정답을 지우고 스스로 문제를 풀어보면 된다. 참고로 강의를 듣는 시간만큼 문제와 풀이에도 많은 시간을 들어야 제대로 성장 할 수 있다!

### 문제 - 이름 나이 입력받고 출력하기

사용자로부터 입력받은 이름과 나이를 출력하세요. 출력 형태는 "당신의 이름은 [이름]이고, 나이는 [나이]살입니다." 이어야 합니다.

### 실행 결과 예시

```
당신의 이름을 입력하세요 :자바
당신의 나이를 입력하세요 :30
당신의 이름은 자바이고, 나이는 30살입니다.
```

### 정답

```
package scanner.ex;

import java.util.Scanner;

public class ScannerEx1 {
```

```

public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 System.out.print("당신의 이름을 입력하세요:");
 String name = scanner.nextLine(); // 이름을 입력받는다.

 System.out.print("당신의 나이를 입력하세요:");
 int age = scanner.nextInt(); // 나이를 입력받는다. 나이는 정수이므로 nextInt()를
사용한다.

 System.out.println("당신의 이름은 " + name + "이고, 나이는 " + age + "살입니다.");
}
}

```

## 문제 - 홀수 짝수

사용자로부터 하나의 정수를 입력받고, 이 정수가 홀수인지 짝수인지 판별하는 프로그램을 작성하세요.

### 실행 결과 예시

```

하나의 정수를 입력하세요:1
입력한 숫자 1는 홀수입니다.

```

### 실행 결과 예시

```

하나의 정수를 입력하세요:4
입력한 숫자 4는 짝수입니다.

```

### 정답

```

package scanner.ex;

import java.util.Scanner;

public class ScannerEx2 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 System.out.print("하나의 정수를 입력하세요:");
 int number = scanner.nextInt();

 if (number % 2 == 0) {

```

```

 System.out.println("입력한 숫자 " + number + "는 짹수입니다.");
 } else {
 System.out.println("입력한 숫자 " + number + "는 홀수입니다.");
 }
}
}

```

## 문제 - 음식점 주문

- 사용자로부터 음식의 이름( `foodName` ), 가격( `foodPrice` ), 그리고 수량( `foodQuantity` )을 입력받아, 주문한 음식의 총 가격을 계산하고 출력하는 프로그램을 작성하세요.
- 음식의 총 가격을 계산하세요. 총 가격은 각 음식의 가격( `foodPrice` )과 수량( `foodQuantity` )을 곱한 값이며, 이를 `totalPrice` 라는 이름의 변수에 저장하세요.
- 주문 정보와 총 가격을 출력하세요. 출력 형태는 "[음식 이름] [수량]개를 주문하셨습니다. 총 가격은 [총 가격]원입니다." 이어야 합니다.

### 실행 결과 예시

```

음식 이름을 입력해주세요: 피자
음식의 가격을 입력해주세요: 20000
음식의 수량을 입력해주세요: 2
피자 2개를 주문하셨습니다. 총 가격은 40000원입니다.

```

### 정답

```

package scanner.ex;

import java.util.Scanner;

public class ScannerEx3 {
 public static void main(String[] args) {
 Scanner input = new Scanner(System.in);

 System.out.print("음식 이름을 입력해주세요: ");
 String foodName = input.nextLine();

 System.out.print("음식의 가격을 입력해주세요: ");
 int foodPrice = input.nextInt();

 System.out.print("음식의 수량을 입력해주세요: ");
 int foodQuantity = input.nextInt();
 }
}

```

```

 int totalPrice = foodPrice * foodQuantity;

 System.out.println(foodName + " " + foodQuantity + "개를 주문하셨습니다. 총 가
격은 " + totalPrice + "원입니다.");
 }
}

```

## 문제 - 구구단 출력

사용자로부터 하나의 정수  $n$ 을 입력받고, 입력받은 정수  $n$ 의 구구단을 출력하는 프로그램을 작성하세요.

### 실행 결과 예시

구구단의 단 수를 입력해주세요: 8

8단의 구구단:

```

8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72

```

### 정답

```

package scanner.ex;

import java.util.Scanner;

public class ScannerEx4 {
 public static void main(String[] args) {
 Scanner input = new Scanner(System.in);

 System.out.print("구구단의 단 수를 입력해주세요: ");
 int n = input.nextInt();

 System.out.println(n + "단의 구구단: ");
 for (int i = 1; i <= 9; i++) {
 System.out.println(n + " x " + i + " = " + n * i);
 }
 }
}

```

```
}
```

## 문제와 풀이2

### 문제 - 변수 값 교환

변수 `a=10` 이 들어있고, `b=20` 이 들어있다.

변수 `a`의 값과 변수 `b`의 값을 서로 바꾸어라

다음 코드에서 시작과 종료 부분 사이에 변수의 값을 교환하는 코드를 작성하면 된다.

힌트: `temp` 변수를 활용해야 한다.

### 출력 결과

```
a = 20
b = 10
```

### 문제 예시

```
package scanner.ex;

public class ChangeVarEx {
 public static void main(String[] args) {
 int a = 10;
 int b = 20;
 int temp;

 //시작: 코드를 작성하세요

 //종료: 코드를 작성하세요

 System.out.println("a = " + a);
 System.out.println("b = " + b);
 }
}
```

### 정답

```
package scanner.ex;

public class ChangeVarEx {
```

```

public static void main(String[] args) {
 int a = 10;
 int b = 20;
 int temp;

 // 시작: 코드를 작성하세요
 temp = a;
 a = b;
 b = temp;
 // 종료: 코드를 작성하세요

 System.out.println("a = " + a);
 System.out.println("b = " + b);
}
}

```

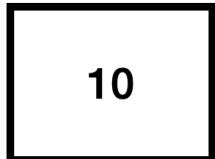
## 풀이

- a 와 b 를 한번에 서로 교환할 수는 없다.
- a = b 라고 하면 a 의 값인 10은 사라져 버린다. 따라서 a, b 둘다 20이 되어버린다.
- a = b 라고 하기 전에 a 의 값을 어딘가에 보관해두어야 한다. 여기서는 임시로 사용할 변수인 temp 에 보관해 두었다.

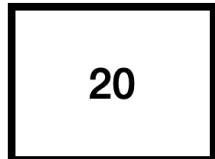
## 진행 과정 - 초기 상태



temp

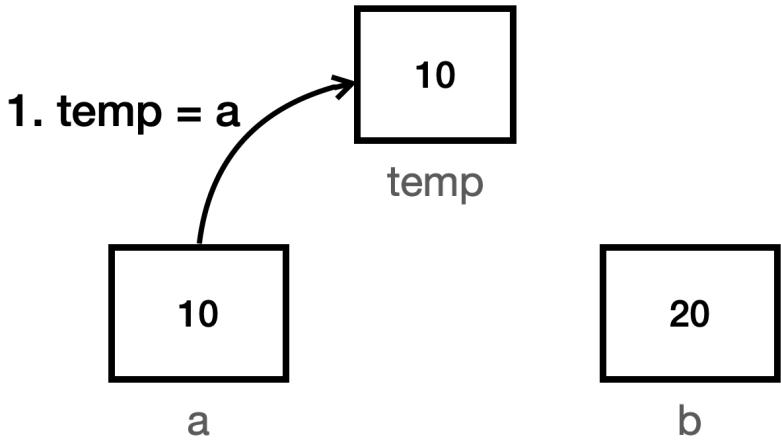


a

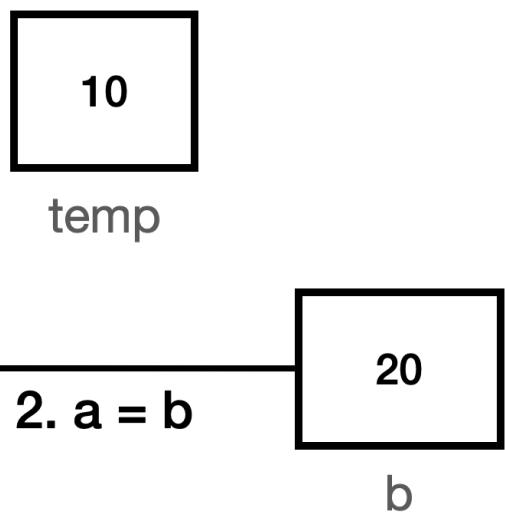


b

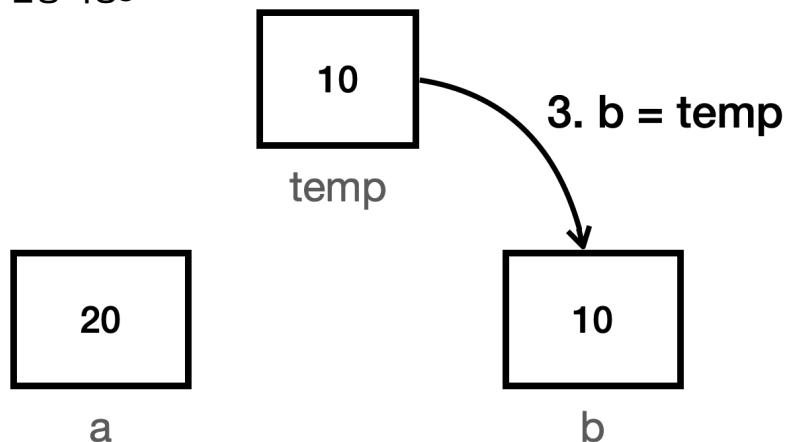
## 진행 과정1



진행 과정2



진행 과정3



### 문제 - 사이 숫자

사용자로부터 두 개의 정수를 입력받고, 이 두 정수 사이의 모든 정수를 출력하는 프로그램을 작성하세요.

- 사용자에게 첫 번째 숫자를 입력받으세요. 변수의 이름은 num1 이어야 합니다.

- 사용자에게 두 번째 숫자를 입력받으세요. 변수의 이름은 num2 이어야 합니다.
- 만약 첫 번째 숫자 num1 이 두 번째 숫자 num2 보다 크다면, 두 숫자를 교환하세요.
  - 참고: 교환을 위해 임시 변수 사용을 고려하세요.
- num1 부터 num2 까지의 각 숫자를 출력하세요.
- 출력 결과에 유의하세요. 다음 예시와 같이 2, 3, 4, 5처럼 , 로 구분해서 출력해야 합니다.

### 실행 결과 예시

```
첫 번째 숫자를 입력하세요:2
두 번째 숫자를 입력하세요:5
두 숫자 사이의 모든 정수:2,3,4,5
```

### 실행 결과 예시

```
첫 번째 숫자를 입력하세요:7
두 번째 숫자를 입력하세요:5
두 숫자 사이의 모든 정수:5,6,7
```

### 정답

```
package scanner.ex;

import java.util.Scanner;

public class ScannerEx5 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 System.out.print("첫 번째 숫자를 입력하세요:");
 int num1 = scanner.nextInt();

 System.out.print("두 번째 숫자를 입력하세요:");
 int num2 = scanner.nextInt();

 // num1이 num2보다 큰 경우, 두 숫자를 교환합니다.
 if (num1 > num2) {
 int temp = num1;
 num1 = num2;
 num2 = temp;
 }

 System.out.print("두 숫자 사이의 모든 정수:");
 for (int i = num1; i <= num2; i++) {
```

```

 System.out.print(i);
 if (i != num2) {
 System.out.print(", ");
 }
 }
}

```

## 문제와 풀이3

### 문제 - 이름과 나이 반복

- 사용자로부터 이름과 나이를 반복해서 입력받고, 입력받은 이름과 나이를 출력하는 프로그램을 작성하세요. 사용자가 "종료"를 입력하면 프로그램이 종료되어야 합니다.
- 다음 실행 결과 예시를 참고해주세요.

### 실행 결과 예시

```

이름을 입력하세요 (종료를 입력하면 종료) : 자바
나이를 입력하세요 : 30
입력한 이름: 자바, 나이: 30
이름을 입력하세요 (종료를 입력하면 종료) : 하니
나이를 입력하세요 : 20
입력한 이름: 하니, 나이: 20
이름을 입력하세요 (종료를 입력하면 종료) : 종료
프로그램을 종료합니다.

```

### 정답

```

package scanner.ex;

import java.util.Scanner;

public class ScannerWhileEx1 {
 public static void main(String[] args) {
 Scanner input = new Scanner(System.in);

 while (true) {
 System.out.print("이름을 입력하세요 (종료를 입력하면 종료) : ");
 String name = input.nextLine();

```

```

 if (name.equals("종료")) {
 System.out.println("프로그램을 종료합니다. ");
 break;
 }

 System.out.print("나이를 입력하세요: ");
 int age = input.nextInt();
 input.nextLine(); // 숫자 입력 후의 줄바꿈 처리

 System.out.println("입력한 이름: " + name + ", 나이: " + age);
 }
}
}

```

## 문제 - 상품 가격 계산

- 사용자로부터 상품의 가격(price)과 수량(quantity)을 입력받고, 총 비용을 출력하는 프로그램을 작성하세요.
- 가격과 수량을 입력받은 후에는 이들의 곱을 출력하세요. 출력 형태는 "총 비용: [곱한 결과]"이어야 합니다.
- 1을 입력하여 가격 입력을 종료합니다.

### 실행 결과 예시

```

상품의 가격을 입력하세요 (-1을 입력하면 종료): 1000
구매하려는 수량을 입력하세요: 3
총 비용: 3000

상품의 가격을 입력하세요 (-1을 입력하면 종료): 2000
구매하려는 수량을 입력하세요: 4
총 비용: 8000

상품의 가격을 입력하세요 (-1을 입력하면 종료): -1
프로그램을 종료합니다.

```

### 정답

```

package scanner.ex;

import java.util.Scanner;

public class ScannerWhileEx2 {
 public static void main(String[] args) {
 Scanner input = new Scanner(System.in);

```

```

while (true) {
 System.out.print("상품의 가격을 입력하세요 (-1을 입력하면 종료) : ");
 int price = input.nextInt();

 if (price == -1) {
 System.out.println("프로그램을 종료합니다.");
 break;
 }

 System.out.print("구매하려는 수량을 입력하세요 : ");
 int quantity = input.nextInt();

 int totalCost = price * quantity;
 System.out.println("총 비용: " + totalCost);
}
}
}

```

## 문제와 풀이4

### 문제 - 입력한 숫자의 합계와 평균

- 사용자로부터 여러 개의 숫자를 입력 받고, 그 숫자들의 합계와 평균을 계산하는 프로그램을 작성하세요. 사용자는 숫자를 입력하고, 마지막에는 -1을 입력하여 숫자 입력을 종료한다고 가정합니다.
- 모든 숫자의 입력이 끝난 후에는, 숫자들의 합계 sum과 평균 average를 출력하세요. 평균은 소수점 아래까지 계산해야 합니다.
- 다음 실행 결과 예시를 참고해주세요.

### 실행 결과 예시

숫자를 입력하세요. 입력을 중단하려면 -1을 입력하세요 :

1  
2  
3  
4  
-1

입력한 숫자들의 합계: 10

입력한 숫자들의 평균: 2.5

## 정답

```
package scanner.ex;

import java.util.Scanner;

public class ScannerWhileEx3 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 int sum = 0;
 int count = 0;
 int input = 0;

 System.out.println("숫자를 입력하세요. 입력을 중단하려면 -1을 입력하세요 : ");
 //반복문 예제1
 while (true) {
 input = scanner.nextInt();
 if (input == -1) {
 break;
 }
 sum += input;
 count++;
 }

 //반복문 예제2
 /*
 while ((input = scanner.nextInt()) != -1) {
 sum += input;
 count++;
 }
 */

 double average = (double) sum / count;
 System.out.println("입력한 숫자들의 합계: " + sum);
 System.out.println("입력한 숫자들의 평균: " + average);
 }
}
```

이 문제에서 while문은 주석으로 처리한 것처럼 축약할 수 있다.

## 반복문 실행

```
while ((input = scanner.nextInt()) != -1) //사용자 3입력
```

```
while ((input = 3) != -1) //input에 3 대입
while ((input(3)) != -1) //input의 값 읽기
while ((3) != -1) //() 제거
while (3 != -1) // 부등식 연산
while (true) //while문 실행
```

## 반복문 종료

```
while ((input = scanner.nextInt()) != -1) //사용자 -1입력
while ((input = -1) != -1) //input에 -1 대입
while ((input(-1)) != -1) //input의 값 읽기
while ((-1) != -1) //() 제거
while (-1 != -1) // 부등식 연산
while (false) //while문 종료
```

## 문제 - 상품 구매

- 사용자로부터 상품 정보(상품명, 가격, 수량)를 입력받고, 이들의 총 가격을 출력하는 프로그램을 작성하세요. 사용자는 여러 상품을 추가하고 결제할 수 있으며, 프로그램을 언제든지 종료할 수 있습니다.
- 사용자에게 다음 세 가지 옵션을 제공해야 합니다: 1: 상품 입력, 2: 결제, 3: 프로그램 종료. 옵션은 정수로 입력받으며, 옵션을 저장하는 변수의 이름은 `option`이어야 합니다.
- 상품 입력 옵션을 선택하면, 사용자에게 상품명과 가격, 수량을 입력받으세요.
- 결제 옵션을 선택하면, 총 비용을 출력하고 총 비용을 0으로 초기화하세요. (사용자가 총 비용을 확인하고, 결제를 완료했다고 가정합니다. 따라서 다음 사용자를 위해 총 비용을 다시 0으로 초기화 해야합니다.)
- 프로그램 종료 옵션을 선택하면, "프로그램을 종료합니다."라는 메시지를 출력하고 프로그램을 종료하세요.
- 위의 세 가지 옵션 외에 다른 값을 입력하면, "올바른 옵션을 선택해주세요."라는 메시지를 출력하세요.

## 실행 결과 예시

```
1: 상품 입력, 2: 결제, 3: 프로그램 종료
1
상품명을 입력하세요: 스프링
상품의 가격을 입력하세요: 30000
구매 수량을 입력하세요: 1
상품명:스프링 가격:30000 수량:1 합계:30000
1: 상품 입력, 2: 결제, 3: 프로그램 종료
1
상품명을 입력하세요: JPA
상품의 가격을 입력하세요: 40000
구매 수량을 입력하세요: 2
상품명:JPA 가격:40000 수량:2 합계:80000
```

```
1: 상품 입력, 2: 결제, 3: 프로그램 종료
```

```
2
```

```
총 비용: 110000
```

```
1: 상품 입력, 2: 결제, 3: 프로그램 종료
```

```
3
```

```
프로그램을 종료합니다.
```

## 정답

```
package scanner.ex;

import java.util.Scanner;

public class ScannerWhileEx4 {
 public static void main(String[] args) {
 Scanner input = new Scanner(System.in);
 int totalCost = 0;

 while (true) {
 System.out.println("1: 상품 입력, 2: 결제, 3: 프로그램 종료");
 int option = input.nextInt();

 if (option == 1) {
 input.nextLine(); // 이전에 입력된 개행문자 제거

 System.out.print("상품명을 입력하세요: ");
 String product = input.nextLine();

 System.out.print("상품의 가격을 입력하세요: ");
 int price = input.nextInt();

 System.out.print("구매 수량을 입력하세요: ");
 int quantity = input.nextInt();

 totalCost += price * quantity;
 System.out.println("상품명:" + product + " 가격:" + price + " 수량:"
+ quantity + " 합계:" + price * quantity);
 } else if (option == 2) {
 System.out.println("총 비용: " + totalCost);
 totalCost = 0; // 결제 후 총 비용 초기화
 } else if (option == 3) {
 System.out.println("프로그램을 종료합니다.");
 break;
 } else {
```

```
 System.out.println("올바른 옵션을 선택해주세요.");
 }
}
}
```

## 정리

# 8. 배열

#1.인강/0.자바/1.자바-입문

- /배열 시작
- /배열의 선언과 생성
- /배열 사용
- /배열 리펙토링
- /2차원 배열 - 시작
- /2차원 배열 - 리펙토링1
- /2차원 배열 - 리펙토링2
- /향상된 for문
- /문제와 풀이1
- /문제와 풀이2
- /문제와 풀이3
- /정리

## 배열 시작

### 배열이 필요한 이유

학생의 점수를 출력하는 간단한 프로그램을 작성해보자.

#### Array1

```
package array;

public class Array1 {
 public static void main(String[] args) {
 int student1 = 90;
 int student2 = 80;
 int student3 = 70;
 int student4 = 60;
 int student5 = 50;

 System.out.println("학생1 점수: " + student1);
 System.out.println("학생2 점수: " + student2);
 System.out.println("학생3 점수: " + student3);
 System.out.println("학생4 점수: " + student4);
 System.out.println("학생5 점수: " + student5);
```

```
 }
}
```

## 실행 결과

```
학생1 점수: 90
학생2 점수: 80
학생3 점수: 70
학생4 점수: 50
학생5 점수: 90
```

- 학생을 몇 명 더 추가해야 한다면 변수를 선언하는 부분과 점수를 출력하는 부분의 코드도 추가해야한다. 학생을 몇 명 더 추가하는 것은 개발자가 코딩으로 해결할 수 있겠지만, 학생을 수백 명 이상 추가해야 한다면 코드가 상당히 많이 늘어날 것이다. 결국 학생 수가 증가함에 따라 코딩 양이 비례해서 증가하는 문제가 발생한다.
- 변수를 선언하는 부분을 보면 학생 수가 증가함에 따라 int 형 변수를 계속해서 추가해야 한다. 학생 수가 5명이면 int 형 변수를 5개 선언해야 하고, 학생 수가 100명이라면 int 형 변수를 100개 선언해야 한다. 결국 비슷한 변수를 반복해서 선언하는 문제가 발생한다.
- 반복문으로 해결할 수 있을 것 같지만, 점수를 출력하는 부분을 보면 변수의 이름이 다르기 때문에 반복문도 적용할 수 없다.

이렇게 같은 타입의 변수를 반복해서 선언하고 반복해서 사용하는 문제를 해결하는 것이 바로 배열이다.

## 배열의 선언과 생성

배열은 같은 타입의 변수를 사용하기 편하게 하나로 묶어둔 것이다. 이전 예제를 배열을 사용하도록 변경해보자. 참고로 단계적으로 구조를 변경해 나갈 것이다.

### Array1Ref1

```
package array;

public class Array1Ref1 {
 public static void main(String[] args) {
 int[] students; //배열 변수 선언
 students = new int[5]; //배열 생성

 //변수 값 대입
 students[0] = 90;
```

```

 students[1] = 80;
 students[2] = 70;
 students[3] = 60;
 students[4] = 50;

 //변수 값 사용
 System.out.println("학생1 점수: " + students[0]);
 System.out.println("학생2 점수: " + students[1]);
 System.out.println("학생3 점수: " + students[2]);
 System.out.println("학생4 점수: " + students[3]);
 System.out.println("학생5 점수: " + students[4]);
}
}

```

지금부터 아주 간단해보이는 다음 두 줄을 아주 자세히 설명하겠다. 집중해서 따라오자.

```

int[] students; //1. 배열 변수 선언
students = new int[5]; //2. 배열 생성

```

## 1. 배열 변수 선언

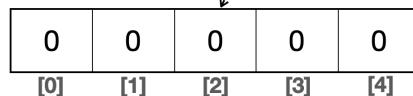
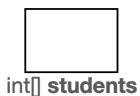
### 1. int[] students; //배열 변수 선언



- 배열을 사용하려면 `int[] students;` 와 같이 배열 변수를 선언해야 한다.
- 일반적인 변수와 차이점은 `int[]`처럼 타입 다음에 대괄호(`[]`)가 들어간다는 점이다.
- 배열 변수를 선언한다고해서 아직 사용할 수 있는 배열이 만들어진 것은 아니다!
  - `int a`에는 정수를, `double b`에는 실수를 담을 수 있다.
  - `int[] students`와 같은 배열 변수에는 배열을 담을 수 있다. (배열 변수에는 10, 20 같은 값이 아니라 배열이라는 것을 담을 수 있다)

## 2. 배열 생성

### 2. students = new int[5]; //배열 생성



5개의 int 공간이 생성, 0으로 자동 초기화

- 배열을 사용하려면 배열을 생성해야 한다.
- `new int[5]`라고 입력하면 오른쪽 그림과 같이 총 5개의 `int` 형 변수가 만들어진다.
- `new`는 새로 생성한다는 뜻이고, `int[5]`는 `int` 형 변수 5개라는 뜻이다. 따라서 `int` 형 변수 5개를 다룰 수 있는 배열을 새로 만든다는 뜻이다.

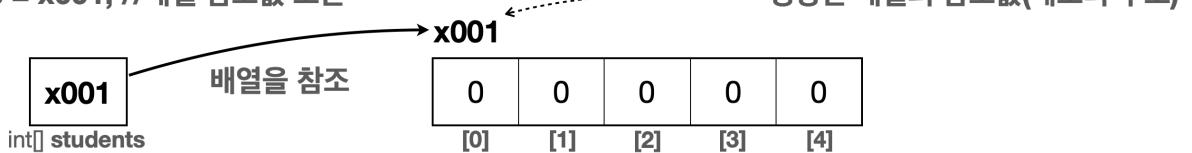
- 앞서 `int student1, int student2 ... int student5` 까지 총 5개의 변수를 직접 선언했다. 배열을 사용하면 이런 과정을 한번에 깔끔하게 처리할 수 있다.

## 배열과 초기화

- `new int[5]` 라고 하면 총 5개의 `int` 형 변수가 만들어진다. 자바는 배열을 생성할 때 그 내부값을 자동으로 초기화한다.
- 숫자는 `0`, `boolean`은 `false`, `String`은 `null`(없다는 뜻이다.)로 초기화 된다.

## 3. 배열 참조값 보관

### 3. `students = x001; //배열 참조값 보관`



- `new int[5]`로 배열을 생성하면 배열의 크기만큼 메모리를 확보한다.
  - `int` 형을 5개 사용하면 `4byte * 5 → 20byte`를 확보한다.
- 배열을 생성하고 나면 자바는 메모리 어딘가에 있는 이 배열에 접근할 수 있는 참조값(주소)(`x001`)을 반환한다.
  - 여기서 `x001`이라고 표현한 것이 참조값이다. (실제로 `x001`처럼 표현되는 것은 아니고 이해를 돋기 위한 예시이다.)
- 앞서 선언한 배열 변수인 `int[] students`에 생성된 배열의 참조값(`x001`)을 보관한다.
- `int[] students` 변수는 `new int[5]`로 생성한 배열의 참조값을 가지고 있다.
  - 이 변수는 참조값을 가지고 있다. 이 참조값을 통해 배열을 참조할 수 있다. 쉽게 이야기해서 참조값을 통해 메모리에 있는 실제 배열에 접근하고 사용할 수 있다.
  - 참고로 배열을 생성하는 `new int[5]` 자체에는 아무런 이름이 없다! 그냥 `int` 형 변수를 5개 연속으로 만드는 것이다. 따라서 생성한 배열에 접근하는 방법이 필요하다. 따라서 배열을 생성할 때 반환되는 참조값을 어딘가에 보관해두어야 한다. 앞서 `int[] students` 변수에 참조값(`x001`)을 보관해두었다. 이 변수를 통해서 이 배열에 접근할 수 있다.

이 부분을 풀어서 설명하면 다음과 같다.

```
int[] students = new int[5]; //1. 배열 생성
int[] students = x001; //2. new int[5]의 결과로 x001 참조값 반환
students = x001 //3. 최종 결과
```

참조값을 확인하고 싶다면 다음과 같이 배열의 변수를 출력해보면 된다.

```
System.out.println(students); // [I@4617c264 @앞의 [I는 int형 배열을 뜻한다. @뒤에 16진수는 참조값을 뜻한다.
```

- 참조값에 대한 더 자세한 내용은 뒤에서 다룬다. 지금은 생성한 배열을 참조할 수 있는, 메모리의 주소를 나타내는

특별한 값이 있다는 정도만 이해하면 충분하다.

## 배열 사용

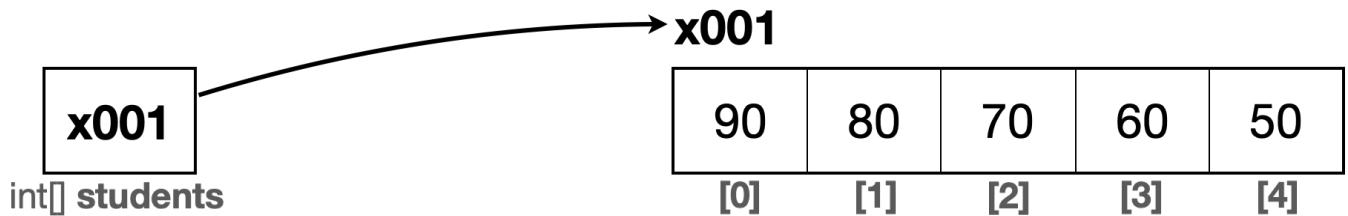
### 인덱스

배열은 변수와 사용법이 비슷한데, 차이점이 있다면 다음과 같이 [ ] 사이에 숫자 번호를 넣어주면 된다. 배열의 위치를 나타내는 숫자를 인덱스(index)라 한다.

```
//변수 값 대입
students[0] = 90;
students[1] = 80;

//변수 값 사용
System.out.println("학생1 점수: " + students[0]);
System.out.println("학생2 점수: " + students[1]);
```

### 배열 참조 그림



### 배열은 0부터 시작한다

`new int[5]` 와 같이 5개의 요소를 가지는 `int` 형 배열을 만들었다면 인덱스는 `0, 1, 2, 3, 4` 가 존재한다.

여기서 주의해야 할 점이 있는데 인덱스는 0부터 시작한다는 것이다. 배열의 요소를 5개로 생성했지만, 인덱스는 0부터 시작한다. 따라서 사용 가능한 인덱스의 범위는 `0 ~ (n-1)` 이 된다. 그래서 `students[4]` 가 배열의 마지막 요소이다.

만약 `students[5]` 와 같이 접근 가능한 배열의 인덱스 범위를 넘어가면 다음과 같은 오류가 발생한다.

### 인덱스 허용 범위를 넘어설 때 발생하는 오류

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out
of bounds for length 5 at array.Array1Ref1.main(Array1Ref1.java:14)
```

## 배열에 값 대입

배열에 값을 대입하든 배열의 값을 사용하든 간에 일반적인 변수와 사용법은 같다. 추가로 `[]`를 통해 인덱스만 넣어주면 된다.

```
students[0] = 90; //1. 배열에 값을 대입
x001[0] = 90; //2. 변수에 있는 참조값을 통해 실제 배열에 접근. 인덱스를 사용해서 해당 위치의 요소에 접근, 값 대입
```

```
students[1] = 80; //1. 배열에 값을 대입
x001[1] = 80; //2. 변수에 있는 참조값을 통해 실제 배열에 접근. 인덱스를 사용해서 해당 위치의 요소에 접근, 값 대입
```

## 배열 값 읽기

```
//1. 변수 값 읽기
System.out.println("학생1 점수: " + students[0]);
//2. 변수에 있는 참조값을 통해 실제 배열에 접근. 인덱스를 사용해서 해당 위치의 요소에 접근
System.out.println("학생1 점수: " + x001[0]);
//3. 배열의 값을 읽어옴
System.out.println("학생1 점수: " + 90);
```

배열을 사용하면 이렇게 참조값을 통해서 실제 배열에 접근하고 인덱스를 통해서 원하는 요소를 찾는다.

## 기본형 vs 참조형

자바의 변수 데이터 타입을 가장 크게 보면 기본형과 참조형으로 분류할 수 있다. 사용하는 값을 직접 넣을 수 있는 기본형, 그리고 방금 본 배열 변수와 같이 메모리의 참조값을 넣을 수 있는 참조형으로 분류할 수 있다.

- **기본형(Primitive Type):** 우리가 지금까지 봤던 `int`, `long`, `double`, `boolean`처럼 변수에 사용할 값을 직접 넣을 수 있는 데이터 타입을 기본형(Primitive Type)이라 한다.
- **참조형(Reference Type):** `int[] students`와 같이 데이터에 접근하기 위한 참조(주소)를 저장하는 데이터 타입을 참조형(Reference Type)이라 한다. 뒤에서 학습하는 객체나 클래스를 담을 수 있는 변수들도 모두 참조형이다.

## 참고

배열은 왜 이렇게 복잡하게 참조형을 사용할까? 지금까지 배운 변수처럼 단순히 그 안에 값을 넣고 사용하면 되는 것 아닐까?

기본형은 모두 사이즈가 명확하게 정해져있다.

```
int i; //4byte
long l; //8byte
double d; //8byte
```

그런데 배열은 다음과 같이 동적으로 사이즈를 변경할 수 있다.

```
int size=10000; //사용자가 입력한 값을 넣었다고 가정해보자.
new int[size]; //이 코드가 실행되는 시점에 배열의 크기가 정해진다.
```

- 기본형은 선언과 동시에 크기가 정해진다. 따라서 크기를 동적으로 바꾸거나 할 수는 없다. 반면에 앞서본 배열과 같은 참조형은 크기를 동적으로 할당할 수 있다. 예를 들어서 Scanner를 사용해서 사용자 입력에 따라 size 변수의 값이 변하고, 생성되는 배열의 크기도 달라질 수 있다. 이런 것을 동적 메모리 할당이라 한다. 기본형은 선언과 동시에 사이즈가 정적으로 정해지지만, 참조형을 사용하면 이처럼 동적으로 크기가 변해서 유연성을 제공할 수 있다.
- 기본형은 사용할 값을 직접 저장한다. 반면에 참조형은 메모리에 저장된 배열이나 객체의 참조를 저장한다. 이로 인해 참조형은 더 복잡한 데이터 구조를 만들고 관리할 수 있다. 반면 기본형은 더 빠르고 메모리를 효율적으로 처리한다.

기본형과 참조형에 대한 내용은 객체를 설명할 때 더 자세히 다룬다. 지금은 기본형과 참조형이라는 2가지 구분이 있다는 점만 이해하면 충분하다.

## 배열 리펙토링

### 배열 리펙토링 - 변수 값 사용

이제 배열을 사용해서 코드를 단계별로 리펙토링 해보자.

먼저 변수 값을 사용한 부분을 변경해보자.

```
//변수 값 사용
System.out.println("학생1 점수: " + students[0]);
System.out.println("학생2 점수: " + students[1]);
System.out.println("학생3 점수: " + students[2]);
System.out.println("학생4 점수: " + students[3]);
System.out.println("학생5 점수: " + students[4]);
```

변수명이 students로 같기 때문에 숫자가 반복되는 부분만 해결하면 반복문을 도입할 수 있을 것 같다. for 문을 사용해서 문제를 해결해보자.

#### 참고: 리펙토링

리펙토링(Refactoring)은 기존의 코드 기능은 유지하면서 내부 구조를 개선하여 가독성을 높이고, 유지보수를 용이하게 하는 과정을 뜻한다. 이는 중복을 제거하고, 복잡성을 줄이며, 이해하기 쉬운 코드로 만들기 위해 수행된다.

다. 리펙토링은 버그를 줄이고, 프로그램의 성능을 향상시킬 수도 있으며, 코드의 설계를 개선하는 데에도 도움이 된다.

쉽게 이야기해서 작동하는 기능은 똑같은데, 코드를 개선하는 것을 리펙토링이라 한다.

## Array1Ref2

```
package array;

public class Array1Ref2 {
 public static void main(String[] args) {
 int[] students; //배열 변수 선언
 students = new int[5]; //배열 생성

 //변수 값 대입
 students[0] = 90;
 students[1] = 80;
 students[2] = 70;
 students[3] = 60;
 students[4] = 50;

 //변수 값 사용
 for (int i = 0; i < students.length; i++) {
 System.out.println("학생" + (i + 1) + " 점수: " + students[i]);
 }
 }
}
```

- 반복문을 사용해서 배열을 통해 값을 사용하는 부분을 효과적으로 변경했다.
- 배열의 인덱스는 0부터 시작하기 때문에 반복문에서 `i = 0`을 초기값으로 사용했다.
- `students.length`
  - 배열의 길이를 제공하는 특별한 기능이다.
  - 참고로 이 값은 조회만 할 수 있다. 대입은 할 수는 없다.
  - 현재 배열의 크기가 5이기 때문에 여기서는 5가 출력된다.
- for문의 조건이 `i < students.length`이기 때문에 `i`는 `0, 1, 2, 3, 4`까지만 반복한다.
  - `i`가 5가 되면 `5 < 5`가 되면서 조건이 거짓이 되고, 반복을 종료한다.

## 배열 리펙토링 - 초기화

배열은 `{}`를 사용해서 생성과 동시에 편리하게 초기화 하는 기능을 제공한다.

```
int[] students;
```

```
students = new int[]{90, 80, 70, 60, 50}; //배열 생성과 초기화
```

### Array1Ref3

```
package array;

public class Array1Ref3 {
 public static void main(String[] args) {
 int[] students;
 students = new int[]{90, 80, 70, 60, 50}; //배열 생성과 초기화

 for (int i = 0; i < students.length; i++) {
 System.out.println("학생" + (i + 1) + " 점수: " + students[i]);
 }
 }
}
```

이해를 돋기 위해 배열 변수의 선언과 배열의 생성 및 초기화를 두 줄로 나누었지만 다음과 같이 한줄도 가능하다.

```
int[] students = new int[]{90, 80, 70, 60, 50}; //배열 변수 선언, 배열 생성과 초기화
```

### 배열 리펙토링 - 간단한 배열 생성

배열은 {} 만 사용해서 생성과 동시에 편리하게 초기화 하는 기능을 제공한다.

#### 배열의 편리한 초기화

```
int[] students = {90, 80, 70, 60, 50};
```

단 이때는 예제와 같이 배열 변수의 선언을 한 줄에 함께 사용할 때만 가능하다.

물론 이렇게 하더라도 자바가 내부에서 배열 요소의 크기를 보고 new int[5] 을 사용해서 배열을 생성한다. 따라서 기존 코드를 조금 더 편리하게 사용할 수 있는 편의 기능이라 생각하면 된다.

### 오류

```
int[] students;
students = {90, 80, 70, 60, 50};
```

### Array1Ref4

```
package array;

public class Array1Ref4 {
 public static void main(String[] args) {
```

```

//배열 생성 간략 버전, 배열 선언과 함께 사용시 new int[] 생략 가능
int[] students = {90, 80, 70, 60, 50};

for (int i = 0; i < students.length; i++) {
 System.out.println("학생" + (i + 1) + " 점수: " + students[i]);
}
}

```

이제 학생의 점수를 추가해도 `{90, 80, 70, 60, 50}`의 내용만 변경하면 된다. 그러면 나머지 코드는 변경하지 않아도 된다.

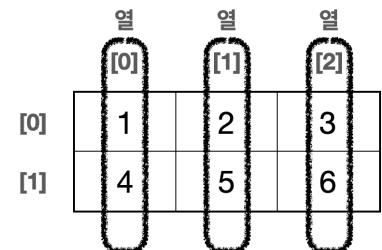
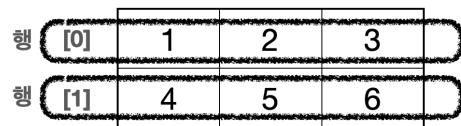
배열을 사용한 덕분에 프로그램을 전체적으로 잘 구조화 할 수 있었다.

## 2차원 배열 - 시작

지금까지 학습한 배열은 단순히 순서대로 나열되어 있었다. 이것을 1차원 배열이라 한다.

이번에 학습할 2차원 배열은 이를 그대로 하나의 차원이 추가된다. 2차원 배열은 행과 열로 구성된다.

|     | [0] | [1] | [2] |
|-----|-----|-----|-----|
| [0] | 1   | 2   | 3   |
| [1] | 4   | 5   | 6   |



2차원 배열은 `int[][] arr = new int[2][3]` 와 같이 선언하고 생성한다. 그리고 `arr[1][2]` 와 같이 사용하는데, 먼저 행 번호를 찾고, 그 다음에 열 번호를 찾으면 된다.

행은 영어로 row(로우), 열은 영어로 column(컬럼)이라 한다. 자주 사용하는 단어이니 알아두자.

2차원 배열의 사용법은 `[ ]` 가 하나 추가되는 것을 제외하고는 앞서본 1차원 배열과 같다.

그림의 배열에 들어있는 데이터는 다음과 같다.

`arr[행][열], arr[row][column]`

### 그림의 2차원 배열 데이터

- `arr[0][0]: 1`
- `arr[0][1]: 2`
- `arr[0][2]: 3`

- arr[1][0]: 4
- arr[1][1]: 5
- arr[1][2]: 6

코드를 통해서 2차원 배열의 사용법을 알아보자.

## ArrayDi0

```
package array;

public class ArrayDi0 {
 public static void main(String[] args) {
 // 2x3 2차원 배열을 만든다.
 int[][] arr = new int[2][3]; // 행(row), 열(column)

 arr[0][0] = 1; // 0행, 0열
 arr[0][1] = 2; // 0행, 1열
 arr[0][2] = 3; // 0행, 2열
 arr[1][0] = 4; // 1행, 0열
 arr[1][1] = 5; // 1행, 1열
 arr[1][2] = 6; // 1행, 2열

 // 0행 출력
 System.out.print(arr[0][0] + " ");
 System.out.print(arr[0][1] + " ");
 System.out.print(arr[0][2] + " ");
 System.out.println(); // 한 행이 끝나면 라인을 변경한다.

 // 1행 출력
 System.out.print(arr[1][0] + " ");
 System.out.print(arr[1][1] + " ");
 System.out.print(arr[1][2] + " ");
 System.out.println(); // 한 행이 끝나면 라인을 변경한다.
 }
}
```

- 이 코드는 2차원 배열을 만들고, 배열에 값을 1부터 6까지 순서대로 직접 입력한다.
- 다음과 같은 결과를 만들기 위해 0행에 있는 0,1,2열을 출력한다. 그리고 다음으로 1행에 있는 0,1,2열을 출력한다.

## 실행 결과

```
1 2 3 // [0][0], [0][1], [0][2]
4 5 6 // [1][0], [1][1], [1][2]
```

## 2차원 배열 - 리펙토링1

### 구조 개선 - 행 출력 반복

#### 구조 변경

코드 구조를 보면 비슷한 부분이 반복된다.

```
//0행 출력
System.out.print(arr[0][0] + " ");
System.out.print(arr[0][1] + " ");
System.out.print(arr[0][2] + " ");
System.out.println(); //한 행이 끝나면 라인을 변경한다.

//1행 출력
System.out.print(arr[1][0] + " ");
System.out.print(arr[1][1] + " ");
System.out.print(arr[1][2] + " ");
System.out.println(); //한 행이 끝나면 라인을 변경한다.
```

코드를 보면 행을 출력하는 부분이 거의 같다. 차이가 있는 부분은 행에서 arr[0]으로 시작할지 arr[1]로 시작할지의 차이다.

다음과 같이 행(row)에 들어가는 숫자만 하나씩 증가하면서 반복하면 될 것 같다.

```
//row를 0, 1로 변경하면서 다음 코드를 반복
System.out.print(arr[row][0] + " ");
System.out.print(arr[row][1] + " ");
System.out.print(arr[row][2] + " ");
System.out.println(); //한 행이 끝나면 라인을 변경한다.
```

반복문을 사용하도록 코드를 변경해보자.

### ArrayDi1

```
package array;

public class ArrayDi1 {
 public static void main(String[] args) {
 // 2x3 2차원 배열을 만듭니다.
```

```

int[][] arr = new int[2][3]; //행(row), 열(column)

arr[0][0] = 1; //0행, 0열
arr[0][1] = 2; //0행, 1열
arr[0][2] = 3; //0행, 2열
arr[1][0] = 4; //1행, 0열
arr[1][1] = 5; //1행, 1열
arr[1][2] = 6; //1행, 2열

for (int row = 0; row < 2; row++) {
 System.out.print(arr[row][0] + " "); //0열 출력
 System.out.print(arr[row][1] + " "); //1열 출력
 System.out.print(arr[row][2] + " "); //2열 출력
 System.out.println(); //한 행이 끝나면 라인을 변경한다.
}
}
}

```

- for문을 통해서 행(row)을 반복해서 접근한다. 각 행 안에서 열(column)이 3개이므로 arr[row][0], arr[row][1], arr[row][2] 3번 출력한다. 이렇게하면 for문을 한번 도는 동안 3개의 열을 출력할 수 있다.
  - row=0 의 for문이 실행되면 arr[0][0], arr[0][1], arr[0][2]로 1 2 3 이 출력된다.
  - row=1 의 for문이 실행되면 arr[1][0], arr[1][1], arr[1][2]로 4 5 6 이 출력된다.

## 실행 결과

```

1 2 3
4 5 6

```

## 구조 개선 - 열 출력 반복

다음 부분을 보면 같은 코드가 반복된다.

```

System.out.print(arr[row][0] + " "); //0열 출력
System.out.print(arr[row][1] + " "); //1열 출력
System.out.print(arr[row][2] + " "); //2열 출력

```

다음과 같이 열(column)에 들어가는 숫자만 하나씩 증가하면서 반복하면 될 것 같다.

```

//column을 0, 1, 2로 변경하면서 다음 코드를 반복
System.out.print(arr[row][column] + " "); //column열 출력

```

코드를 수정해보자.

## ArrayDi2

```
package array;

public class ArrayDi2 {
 public static void main(String[] args) {
 // 2x3 2차원 배열을 만듭니다.
 int[][] arr = new int[2][3]; // 행(row), 열(column)

 arr[0][0] = 1; // 0행, 0열
 arr[0][1] = 2; // 0행, 1열
 arr[0][2] = 3; // 0행, 2열
 arr[1][0] = 4; // 1행, 0열
 arr[1][1] = 5; // 1행, 1열
 arr[1][2] = 6; // 1행, 2열

 for (int row = 0; row < 2; row++) {
 for (int column = 0; column < 3; column++) {
 System.out.print(arr[row][column] + " ");
 }
 System.out.println(); // 한 행이 끝나면 라인을 변경한다.
 }
 }
}
```

- for문을 2번 중첩해서 사용하는데, 첫번째 for문은 행을 탐색하고, 내부에 있는 두번째 for문은 열을 탐색한다.
- 내부에 있는 for문은 앞서 작성한 다음 코드와 같다. for문을 사용해서 열을 효과적으로 출력했다.

```
System.out.print(arr[row][0] + " "); // 0열 출력
System.out.print(arr[row][1] + " "); // 1열 출력
System.out.print(arr[row][2] + " "); // 2열 출력
```

## 2차원 배열 - 리펙토링2

### 구조 개선 - 초기화, 배열의 길이

이 코드를 보니 2가지 개선할 부분이 있다.

1. 초기화: 기존 배열처럼 2차원 배열도 편리하게 초기화 할 수 있다.

2. `for` 문에서 배열의 길이 활용: 배열의 길이가 달라지면 `for` 문에서 `row < 2, column < 3` 같은 부분을 같이 변경해야 한다. 이 부분을 배열의 길이를 사용하도록 변경해보자. 배열이 커지거나 줄어들어도 `for`문의 코드를 변경하지 않고 그대로 유지할 수 있다.

코드를 개선해보자.

### ArrayDi3

```
package array;

public class ArrayDi3 {
 public static void main(String[] args) {
 // 2x3 2차원 배열, 초기화
 int[][] arr = {
 {1, 2, 3},
 {4, 5, 6}
 };

 // 2차원 배열의 길이를 활용
 for (int row = 0; row < arr.length; row++) {
 for (int column = 0; column < arr[row].length; column++) {
 System.out.print(arr[row][column] + " ");
 }
 System.out.println();
 }
 }
}
```

### 초기화

1차원 배열은 다음과 같이 초기화 한다.

```
{1, 2, 3}
```

2차원 배열은 다음과 같이 초기화 한다. 밖이 행이 되고, 안이 열이 된다. 그런데 이렇게 하면 행열이 잘 안느껴진다.

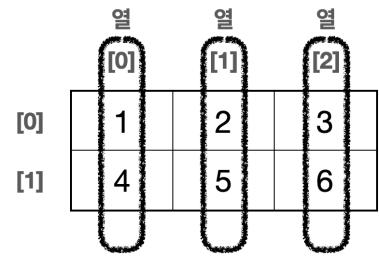
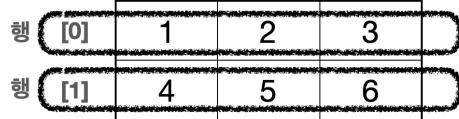
```
{ {1, 2, 3}, {4, 5, 6} }
```

이렇게 라인을 적절하게 넘겨주면 행과 열이 명확해진다. 따라서 코드를 더 쉽게 이해할 수 있다.

```
{
 {1, 2, 3},
 {4, 5, 6}
}
```

## 배열의 길이

|     | [0] | [1] | [2] |
|-----|-----|-----|-----|
| [0] | 1   | 2   | 3   |
| [1] | 4   | 5   | 6   |



for 문에서 2차원 배열의 길이를 활용했다.

- arr.length는 행의 길이를 뜻한다. 여기서는 2가 출력된다.
  - {} , {} 를 생각해보면 arr 배열은 {}, {} 2개의 배열 요소를 가진다.
- arr[row].length는 열의 길이를 뜻한다. 여기서는 3이 출력된다.
  - arr[0] 은 {1, 2, 3} 배열을 뜻한다. 이 배열에는 3개의 요소가 있다.
  - arr[1] 은 {4, 5, 6} 배열을 뜻한다. 이 배열에는 3개의 요소가 있다.

잘 동작하는지 확인해보자.

이제 배열의 초기화 부분만 다음과 같이 변경하면 바로 적용된 결과를 확인할 수 있다. 나머지 코드는 변경하지 않아도 된다.

```
{1, 2, 3},
{4, 5, 6},
{7, 8, 9}
```

## 구조 개선 - 값 입력

이번에는 배열에 직접 1, 2, 3 숫자를 적어서 값을 할당하는 것이 아니라, 배열의 크기와 상관없이 배열에 순서대로 1씩 증가하는 값을 입력하도록 변경해보자.

```
package array;

public class ArrayDi4 {
 public static void main(String[] args) {
 // 2x3 2차원 배열, 초기화
 int[][] arr = new int[2][3];

 int i = 1;
 // 순서대로 1씩 증가하는 값을 입력한다.
 for (int row = 0; row < arr.length; row++) {
 for (int column = 0; column < arr[row].length; column++) {
```

```

 arr[row][column] = i++;
 }

}

// 2차원 배열의 길이를 활용
for (int row = 0; row < arr.length; row++) {
 for (int column = 0; column < arr[row].length; column++) {
 System.out.print(arr[row][column] + " ");
 }
 System.out.println();
}
}
}

```

- 중첩된 `for` 문을 사용해서 값을 순서대로 입력한다.
- `arr[row][column] = i++` 후의 증감 연산자(`++`)를 사용해서 값을 먼저 대입한 다음에 증가한다.
  - 코드에서 `int i = 1`으로 `i`가 1부터 시작한다.

2차원 배열 선언 부분인 `new int[2][3]`을 `new int[4][5]`처럼 다른 숫자로 변경해도 잘 동작하는 것을 확인할 수 있다.

### new int[4][5]로 변경시 출력

```

1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20

```

## 향상된 for문

앞서 반복문에서 설명하지 않은 내용이 하나 있는데, 바로 향상된 `for`문(Enhanced For Loop)이다. 향상된 `for`문을 이해하려면 배열을 먼저 알아야 한다. 각각의 요소를 탐색한다는 의미로 `for-each`문이라고도 많이 부른다. 향상된 `for` 문은 배열을 사용할 때 기존 `for` 문 보다 더 편리하게 사용할 수 있다.

### 향상된 `for`문 정의

```

for (변수 : 배열 또는 컬렉션) {
 // 배열 또는 컬렉션의 요소를 순회하면서 수행할 작업
}

```

코드로 확인해보자.

### EnhancedFor1

```
package array;

public class EnhancedFor1 {
 public static void main(String[] args) {
 int[] numbers = {1, 2, 3, 4, 5};

 //일반 for문
 for(int i = 0; i < numbers.length; ++i) {
 int number = numbers[i];
 System.out.println(number);
 }

 //향상된 for문, for-each문
 for (int number : numbers) {
 System.out.println(number);
 }

 //for-each문을 사용할 수 없는 경우, 증가하는 index 값 필요
 for(int i = 0; i < numbers.length; ++i) {
 System.out.println("number" + i + "번의 결과는: " + numbers[i]);
 }
 }
}
```

### 일반 for문

```
//일반 for문
for(int i = 0; i < numbers.length; ++i) {
 int number = numbers[i];
 System.out.println(number);
}
```

먼저 일반 for문을 살펴보자. 배열에 있는 값을 순서대로 읽어서 number 변수에 넣고, 출력한다.

배열은 처음부터 끝까지 순서대로 읽어서 사용하는 경우가 많다. 그런데 배열의 값을 읽으려면 int i와 같은 인덱스를 탐색할 수 있는 변수를 선언해야 한다. 그리고 i < numbers.length와 같이 배열의 끝 조건을 지정해야주어야 한다. 마지막으로 배열의 값을 하나 읽을 때마다 인덱스를 하나씩 증가해야 한다.

개발자 입장에서는 그냥 배열을 순서대로 처음부터 끝까지 탐색하고 싶은데, 너무 번잡한 일을 해주어야 한다. 그래서 향상된 for문이 등장했다.

## 향상된 for문

```
//향상된 for문, for-each문
for (int number : numbers) {
 System.out.println(number);
}
```

- 앞서 일반 for문과 동일하게 작동한다.
- 향상된 for문은 배열의 인덱스를 사용하지 않고, 종료 조건을 주지 않아도 된다. 단순히 해당 배열을 처음부터 끝까지 탐색한다.
- :의 오른쪽에 numbers와 같이 탐색할 배열을 선택하고, :의 왼쪽에 int number와 같이 반복할 때마다 찾은 값을 저장할 변수를 선언한다. 그러면 배열의 값을 하나씩 꺼내서 왼쪽에 있는 number에 담고 for문을 수행한다. for문의 끝에 가면 다음 값을 꺼내서 number에 담고 for문을 반복 수행한다. numbers 배열의 끝에 도달해서 더 값이 없으면 for문이 완전히 종료된다.
- 향상된 for문은 배열의 인덱스를 사용하지 않고도 배열의 요소를 순회할 수 있기 때문에 코드가 간결하고 가독성이 좋다.

## 향상된 for문을 사용하지 못하는 경우

그런데 향상된 for문을 사용하지 못하는 경우가 있다.

향상된 for문에는 증가하는 인덱스 값이 감추어져 있다. 따라서 int i와 같은 증가하는 인덱스 값을 직접 사용해야 하는 경우에는 향상된 for문을 사용할 수 없다.

```
//for-each문을 사용할 수 없는 경우, 증가하는 index 값 필요
for(int i = 0; i < numbers.length; ++i) {
 System.out.println("number" + i + "번의 결과는: " + numbers[i]);
}
```

이 예제에서는 증가하는 i 값을 출력해야 하므로 향상된 for문 대신에 일반 for문을 사용해야 한다.

물론 다음과 같이 억지스럽게 향상된 for문을 사용하는 것이 가능하지만, 이런 경우 일반 for문을 사용하는 것이 더 좋다.

```
int i = 0;
for (int number : numbers) {
 System.out.println("number" + i + "번의 결과는: " + number);
 i++;
}
```

# 문제와 풀이 1

## 코딩이 처음이라면 필독!

프로그래밍이 처음이라면 아직 코딩 자체가 익숙하지 않기 때문에 문제와 풀이에 상당히 많은 시간을 쓰게 될 수 있다. 강의를 들을 때는 다 이해가 되는 것 같았는데, 막상 혼자 생각해서 코딩을 하려니 잘 안되는 것이다. 이것은 아직 코딩이 익숙하지 않기 때문인데, 처음 코딩을 하는 사람이라면 누구나 겪는 자연스러운 현상이다.

문제를 스스로 풀기 어려운 경우, 너무 고민하기보다는 먼저 **강의 영상의 문제 풀이 과정을 코드로 따라하면서 이해하자. 반드시 코드로 따라해야 한다.** 그래야 코딩하는 것에 조금씩 익숙해질 수 있다. 그런 다음에 정답을 지우고 스스로 문제를 풀어보면 된다. 참고로 강의를 듣는 시간만큼 문제와 풀이에도 많은 시간을 들어야 제대로 성장할 수 있다!

## 문제 - 배열을 사용하도록 변경

다음 문제를 배열을 사용해서 개선하자.

```
package array.ex;

public class ArrayEx1 {
 public static void main(String[] args) {
 int student1 = 90;
 int student2 = 80;
 int student3 = 70;
 int student4 = 60;
 int student5 = 50;

 int total = student1 + student2 + student3 + student4 + student5;
 double average = (double) total / 5;

 System.out.println("점수 총합: " + total);
 System.out.println("점수 평균: " + average);
 }
}
```

## 실행 결과 예시

```
점수 총합: 350
점수 평균: 70.0
```

## 정답

```

package array.ex;

public class ArrayEx1Ref {
 public static void main(String[] args) {
 int[] students = {90, 80, 70, 60, 50};

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

 double average = (double) total / students.length;
 System.out.println("점수 총합: " + total);
 System.out.println("점수 평균: " + average);
 }
}

```

## 문제 - 배열의 입력과 출력

사용자에게 5개의 정수를 입력받아서 배열에 저장하고, 입력 순서대로 출력하자.

출력시 출력 포맷은 1, 2, 3, 4, 5와 같이 , 쉼표를 사용해서 구분하고, 마지막에는 쉼표를 넣지 않아야 한다.

실행 결과 예시를 참고하자.

### 실행 결과 예시

5개의 정수를 입력하세요 :

1  
2  
3  
4  
5

출력

1, 2, 3, 4, 5

### 정답

```

package array.ex;

import java.util.Scanner;

public class ArrayEx2 {

```

```

public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
 int[] numbers = new int[5];

 System.out.println("5개의 정수를 입력하세요:");
 for (int i = 0; i < numbers.length; i++) {
 numbers[i] = scanner.nextInt();
 }

 System.out.println("출력");
 for (int i = 0; i < numbers.length; i++) {
 System.out.print(numbers[i]);
 if (i < numbers.length - 1) {
 System.out.print(", ");
 }
 }
}
}

```

## 문제 - 배열과 역순 출력

사용자에게 5개의 정수를 입력받아서 배열에 저장하고, 입력받은 순서의 반대인 역순으로 출력하자.

출력시 출력 포맷은 5, 4, 3, 2, 1과 같이 , 쉼표를 사용해서 구분하고, 마지막에는 쉼표를 넣지 않아야 한다.

실행 결과 예시를 참고하자.

### 실행 결과 예시

5개의 정수를 입력하세요 :

1  
2  
3  
4  
5

입력한 정수를 역순으로 출력 :

5, 4, 3, 2, 1

### 정답

```

package array.ex;

import java.util.Scanner;

```

```

public class ArrayEx3 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
 int[] numbers = new int[5];

 System.out.println("5개의 정수를 입력하세요:");
 for (int i = 0; i < 5; i++) {
 numbers[i] = scanner.nextInt();
 }

 System.out.println("입력한 정수를 역순으로 출력:");
 for (int i = 4; i >= 0; i--) {
 System.out.print(numbers[i]);
 if (i > 0) {
 System.out.print(", ");
 }
 }
 }
}

```

## 문제 - 합계와 평균

사용자에게 5개의 정수를 입력받아서 이들 정수의 합계와 평균을 계산하는 프로그램을 작성하자.

### 실행 결과 예시

5개의 정수를 입력하세요:

1  
2  
3  
4  
5

입력한 정수의 합계: 15

입력한 정수의 평균: 3.0

### 정답

```

package array.ex;

import java.util.Scanner;

```

```

public class ArrayEx4 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
 int[] numbers = new int[5];
 int sum = 0;
 double average;

 System.out.println("5개의 정수를 입력하세요:");
 for (int i = 0; i < 5; i++) {
 numbers[i] = scanner.nextInt();
 sum += numbers[i];
 }

 average = (double) sum / 5;

 System.out.println("입력한 정수의 합계: " + sum);
 System.out.println("입력한 정수의 평균: " + average);
 }
}

```

## 문제 - 합계와 평균2

이전 문제에서 입력받을 숫자의 개수를 입력받도록 개선하자.

실행 결과 예시를 참고하자

### 실행 결과 예시1

입력받을 숫자의 개수를 입력하세요:3

3개의 정수를 입력하세요:

1

2

3

입력한 정수의 합계: 6

입력한 정수의 평균: 2.0

### 실행 결과 예시2

입력받을 숫자의 개수를 입력하세요:5

5개의 정수를 입력하세요:

1

2

```
3
4
5
입력한 정수의 합계: 15
입력한 정수의 평균: 3.0
```

## 정답

```
package array.ex;

import java.util.Scanner;

public class ArrayEx5 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
 System.out.print("입력 받을 숫자의 개수를 입력하세요:");
 int count = scanner.nextInt();

 int[] numbers = new int[count];
 int sum = 0;
 double average;

 System.out.println(count + "개의 정수를 입력하세요:");
 for (int i = 0; i < count; i++) {
 numbers[i] = scanner.nextInt();
 sum += numbers[i];
 }

 average = (double) sum / count;

 System.out.println("입력한 정수의 합계: " + sum);
 System.out.println("입력한 정수의 평균: " + average);
 }
}
```

## 문제와 풀이2

## 문제 - 가장 작은 수, 큰 수 찾기

사용자로부터 n개의 정수를 입력받아 배열에 저장한 후, 배열 내에서 가장 작은 수와 가장 큰 수를 찾아 출력하는 프로그램을 작성하자. 실행 결과 예시를 참고하자.

### 실행 결과 예시

```
입력받을 숫자의 개수를 입력하세요 : 3
```

```
3개의 정수를 입력하세요 :
```

```
1
```

```
2
```

```
5
```

```
가장 작은 정수 : 1
```

```
가장 큰 정수 : 5
```

### 정답

```
package array.ex;

import java.util.Scanner;

public class ArrayEx6 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);

 System.out.print("입력받을 숫자의 개수를 입력하세요 : ");
 int n = scanner.nextInt();

 int[] numbers = new int[n];
 int minNumber, maxNumber;

 System.out.println(n + "개의 정수를 입력하세요 : ");
 for (int i = 0; i < n; i++) {
 numbers[i] = scanner.nextInt();
 }

 minNumber = maxNumber = numbers[0];
 for (int i = 1; i < n; i++) {
 if (numbers[i] < minNumber) {
 minNumber = numbers[i];
 }
 if (numbers[i] > maxNumber) {
 maxNumber = numbers[i];
 }
 }
 }
}
```

```
 }

 }

System.out.println("가장 작은 정수: " + minNumber);
System.out.println("가장 큰 정수: " + maxNumber);
}

}
```

## 문제 - 2차원 배열1

사용자로부터 4명 학생의 국어, 수학, 영어 점수를 입력받아 각 학생의 총점과 평균을 계산하는 프로그램을 작성하자.  
2차원 배열을 사용하고, 실행 결과 예시를 참고하자.

### 실행 결과 예시

```
1번 학생의 성적을 입력하세요:
국어 점수:100
영어 점수:80
수학 점수:70
2번 학생의 성적을 입력하세요:
국어 점수:30
영어 점수:40
수학 점수:50
3번 학생의 성적을 입력하세요:
국어 점수:60
영어 점수:70
수학 점수:50
4번 학생의 성적을 입력하세요:
국어 점수:90
영어 점수:100
수학 점수:80
1번 학생의 총점: 250, 평균: 83.33333333333333
2번 학생의 총점: 120, 평균: 40.0
3번 학생의 총점: 180, 평균: 60.0
4번 학생의 총점: 270, 평균: 90.0
```

### 정답

```
package array.ex;

import java.util.Scanner;
```

```

public class ArrayEx7 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
 int[][] scores = new int[4][3];
 String[] subjects = {"국어", "영어", "수학"};

 for(int i=0; i<4; i++){
 System.out.println((i+1) + "번 학생의 성적을 입력하세요:");
 for(int j=0; j<3; j++){
 System.out.print(subjects[j] + " 점수:");
 scores[i][j] = scanner.nextInt();
 }
 }

 for(int i=0; i<4; i++){
 int total = 0;
 for(int j=0; j<3; j++){
 total += scores[i][j];
 }
 double average = total / 3.0;
 System.out.println((i+1) + "번 학생의 총점: " + total + ", 평균: " +
average);
 }
 }
}

```

## 문제 - 2차원 배열2

이전 문제에서 학생수를 입력받도록 개선하자.

실행 결과 예시를 참고하자.

### 실행 결과 예시

```

학생수를 입력하세요:3
1번 학생의 성적을 입력하세요:
국어 점수:10
영어 점수:20
수학 점수:30
2번 학생의 성적을 입력하세요:
국어 점수:10
영어 점수:10

```

수학 점수:10

3번 학생의 성적을 입력하세요:

국어 점수:20

영어 점수:20

수학 점수:20

1번 학생의 총점: 60, 평균: 20.0

2번 학생의 총점: 30, 평균: 10.0

3번 학생의 총점: 60, 평균: 20.0

## 정답

```
package array.ex;

import java.util.Scanner;

public class ArrayEx8 {
 public static void main(String[] args) {
 Scanner scanner = new Scanner(System.in);
 System.out.print("학생수를 입력하세요:");
 int studentCount = scanner.nextInt();

 int[][] scores = new int[studentCount][3];
 String[] subjects = {"국어", "영어", "수학"};

 for (int i = 0; i < studentCount; i++) {
 System.out.println((i + 1) + "번 학생의 성적을 입력하세요:");
 for (int j = 0; j < 3; j++) {
 System.out.print(subjects[j] + " 점수:");
 scores[i][j] = scanner.nextInt();
 }
 }

 for (int i = 0; i < studentCount; i++) {
 int total = 0;
 for (int j = 0; j < 3; j++) {
 total += scores[i][j];
 }
 double average = total / 3.0;
 System.out.println((i + 1) + "번 학생의 총점: " + total + ", 평균: " +
average);
 }
 }
}
```

}

## 문제와 풀이3

### 상품 관리 프로그램 만들기

자바를 이용하여 상품 관리 프로그램을 만들어 보자. 이 프로그램은 다음의 기능이 필요하다:

- 상품 등록: 상품 이름과 가격을 입력받아 저장한다.
- 상품 목록: 지금까지 등록한 모든 상품의 목록을 출력한다.

다음과 같이 동작해야 한다:

- 첫 화면에서 사용자에게 세 가지 선택을 제시한다: "1. 상품 등록", "2. 상품 목록", "3. 종료"
- "1. 상품 등록"을 선택하면, 사용자로부터 상품 이름과 가격을 입력받아 배열에 저장한다.
- "2. 상품 목록"을 선택하면, 배열에 저장된 모든 상품을 출력한다.
- "3. 종료"를 선택하면 프로그램을 종료한다.

### 제약 조건

상품은 최대 10개까지 등록할 수 있다.

다음은 사용해야 하는 변수 및 구조이다:

- Scanner scanner : 사용자 입력을 받기 위한 Scanner 객체
- String[] productNames : 상품 이름을 저장할 String 배열
- int[] productPrices : 상품 가격을 저장할 int 배열
- int productCount : 현재 등록된 상품의 개수를 저장할 int 변수

실행 결과 예시를 참고하자.

### 실행 결과 예시

1. 상품 등록 | 2. 상품 목록 | 3. 종료

메뉴를 선택하세요 : 1

상품 이름을 입력하세요 : JAVA

상품 가격을 입력하세요 : 10000

1. 상품 등록 | 2. 상품 목록 | 3. 종료

메뉴를 선택하세요 : 1

상품 이름을 입력하세요 : SPRING

상품 가격을 입력하세요:20000  
1. 상품 등록 | 2. 상품 목록 | 3. 종료  
메뉴를 선택하세요:2  
JAVA: 10000원  
SPRING: 20000원  
1. 상품 등록 | 2. 상품 목록 | 3. 종료  
메뉴를 선택하세요:3  
프로그램을 종료합니다.

### 상품을 더 등록할 수 없는 경우

1. 상품 등록 | 2. 상품 목록 | 3. 종료  
메뉴를 선택하세요:1  
더 이상 상품을 등록할 수 없습니다.

### 등록된 상품이 없는 경우

1. 상품 등록 | 2. 상품 목록 | 3. 종료  
메뉴를 선택하세요:2  
등록된 상품이 없습니다.

### 정답

```
package array.ex;

import java.util.Scanner;

public class ProductAdminEx {
 public static void main(String[] args) {
 int maxProducts = 10;
 String[] productNames = new String[maxProducts];
 int[] productPrices = new int[maxProducts];
 int productCount = 0;

 Scanner scanner = new Scanner(System.in);
 while (true) {
 System.out.print("1. 상품 등록 | 2. 상품 목록 | 3. 종료\n메뉴를 선택하세요:");
 int menu = scanner.nextInt();
 scanner.nextLine();

 if (menu == 1) {
 if (productCount >= maxProducts) {
 System.out.println("더 이상 상품을 등록할 수 없습니다.");
 } else {
 System.out.print("상품 이름: ");
 String name = scanner.nextLine();
 System.out.print("상품 가격: ");
 int price = scanner.nextInt();
 productNames[productCount] = name;
 productPrices[productCount] = price;
 productCount++;
 }
 } else if (menu == 2) {
 for (int i = 0; i < productCount; i++) {
 System.out.println(productNames[i] + " (" + productPrices[i] + ")");
 }
 } else if (menu == 3) {
 System.out.println("프로그램을 종료합니다.");
 break;
 } else {
 System.out.println("잘못된 메뉴 번호입니다.");
 }
 }
 }
}
```

```

 continue;
 }

System.out.print("상품 이름을 입력하세요:");
productNames[productCount] = scanner.nextLine();

System.out.print("상품 가격을 입력하세요:");
productPrices[productCount] = scanner.nextInt();

productCount++;
} else if (menu == 2) {
 if (productCount == 0) {
 System.out.println("등록된 상품이 없습니다.");
 continue;
 }
 for (int i = 0; i < productCount; i++) {
 System.out.println(productNames[i] + ":" + productPrices[i]
+ "원");
 }
} else if (menu == 3) {
 System.out.println("프로그램을 종료합니다.");
 break;
} else {
 System.out.println("잘못된 메뉴를 선택하셨습니다.");
}
}
}
}

```

## 정리

# 9. 메서드

#1.인강/0.자바/1.자바-입문

- /메서드 시작
- /메서드 사용
- /메서드 정의
- /반환 타입
- /메서드 호출과 값 전달1
- /메서드 호출과 값 전달2
- /메서드와 형변환
- /메서드 오버로딩
- /문제와 풀이1
- /문제와 풀이2
- /정리

## 메서드 시작

두 숫자를 입력 받아서 더하고 출력하는 단순한 기능을 개발해보자.

먼저 1 + 2를 수행하고, 그 다음으로 10 + 20을 수행할 것이다.

### Method1

```
package method;

public class Method1 {

 public static void main(String[] args) {
 //계산1
 int a = 1;
 int b = 2;
 System.out.println(a + "+" + b + " 연산 수행");
 int sum1 = a + b;
 System.out.println("결과1 출력:" + sum1);

 //계산2
 int x = 10;
 int y = 20;
 System.out.println(x + "+" + y + " 연산 수행");
 }
}
```

```

 int sum2 = x + y;
 System.out.println("결과2 출력:" + sum2);
}

}

```

- 같은 연산을 두 번 수행한다.
- 코드를 잘 보면 계산1 부분과, 계산2 부분이 거의 같다.

## 계산1

```

int a = 1;
int b = 2;
System.out.println(a + "+" + b + " 연산 수행");
int sum1 = a + b;

```

## 계산2

```

int x = 10;
int y = 20;
System.out.println(x + "+" + y + " 연산 수행");
int sum2 = x + y;

```

계산1, 계산2 둘 다 변수를 두 개 선언하고, 어떤 연산을 수행하는지 출력하고, 두 변수를 더해서 결과를 구한다.

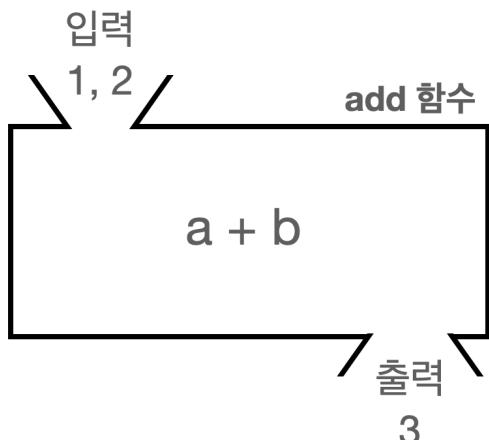
만약 프로그램의 여러 곳에서 이와 같은 계산을 반복해야 한다면? 같은 코드를 여러번 반복해서 작성해야 할 것이다.

더 나아가서 어떤 연산을 수행하는지 출력하는 부분을 변경하거나 또는 제거하고 싶다면 해당 코드를 다 찾아다니면서 모두 수정해야 할 것이다.

이런 문제를 어떻게 깔끔하게 해결할 수 있을까?

잠깐 아주 간단하게 수학의 함수를 알아보자.

## 함수(function)



수학 용어가 나왔다고 전혀 어렵게 생각할 것이 없다! 숫자를 2개 입력하면 해당 숫자를 더한 다음에 그 결과를 출력하는 아주 단순한 함수이다. 이 함수의 이름은 `add`이다.

### 함수 정의

```
add(a, b) = a + b
```

- 이름이 `add`이고 `a`, `b`라는 두 값을 받는 함수이다. 그리고 이 함수는 `a + b` 연산을 수행한다.

### 함수 사용

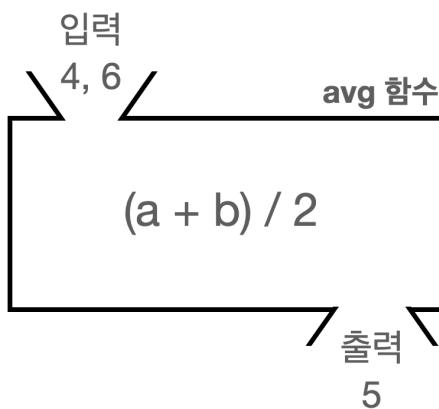
```
add(1, 2) -> 결과:3
```

```
add(5, 6) -> 결과:11
```

```
add(3, 5) -> 결과:8
```

- 함수에 값을 입력하면, 함수가 가진 연산을 처리한 다음 결과를 출력한다. 여기서는 단순히 `a+b`라는 연산을 수행한다.
- 여러번 같은 계산을 해야 한다면 지금처럼 함수를 만들어두고(정의), 필요한 입력 값을 넣어서 해당 함수를 호출하면 된다. 그러면 계산된 결과가 나온다.
- 함수는 마치 블랙박스와 같다. 함수를 호출할 때는 외부에서는 필요한 값만 입력하면 된다. 그러면 계산된 결과가 출력된다.
- 같은 함수를 다른 입력 값으로 여러번 호출할 수 있다.
- 여기서 핵심은 함수를 한번 정의해두면 계속해서 재사용할 수 있다는 점이다!

### 평균 함수



만약 두 수의 평균을 구해야 한다면 매번  $(a + b) / 2$ 라는 공식을 사용해야 할 것이다.

이것을 함수로 만들어두면 다음과 같이 사용하면 된다.

### 함수 정의

```
avg(a, b) = (a + b) / 2
```

### 함수 사용

```
avg(4, 6) -> 결과:5
avg(10, 20) -> 결과:15
avg(100, 200) -> 결과:150
```

수학의 함수의 개념을 프로그래밍에 가지고 온다면 어떨까? 필요한 기능을 미리 정의해두고 필요할 때마다 호출해서 사용할 수 있기 때문에 앞서 고민한 문제들을 해결할 수 있을 것 같다.

프로그램 언어들은 오래 전부터 이런 문제를 해결하기 위해 수학의 함수라는 개념을 차용해서 사용한다.

## 메서드 사용

자바에서는 함수를 메서드(Method)라 한다.

메서드도 함수의 한 종류라고 생각하면 된다. 지금은 둘을 구분하지 않고, 이정도만 알아두자.

메서드를 사용하면 앞서 고민한 문제를 한번에 해결할 수 있다.

메서드에 대한 자세한 설명보다는 우선 메서드를 사용해서 코드를 작성해보자. 참고로 앞에서 작성한 코드와 완전히 동일하게 작동하는 코드이다.

### Method1Ref

```
package method;

public class Method1Ref {

 public static void main(String[] args) {
 int sum1 = add(5, 10);
 System.out.println("결과1 출력:" + sum1);

 int sum2 = add(15, 20);
 System.out.println("결과2 출력:" + sum2);
 }

 //add 메서드
 public static int add(int a, int b) {
 System.out.println(a + "+" + b + " 연산 수행");
 int sum = a + b;
 return sum;
 }
}
```

```
}
```

## 실행 결과

```
5+10 연산 수행
```

```
결과1 출력:15
```

```
15+20 연산 수행
```

```
결과2 출력:35
```

중복이 제거되고, 코드가 상당히 깔끔해진 것을 느낄 수 있을 것이다.

## 메서드 정의

```
public static int add(int a, int b) {
 System.out.println(a + "+" + b + " 연산 수행");
 int sum = a + b;
 return sum;
}
```

이 부분이 바로 메서드이다. 이것을 함수를 정의하는 것과 같이, 메서드를 정의한다고 표현한다.

메서드는 수학의 함수와 유사하게 생겼다. 함수에 값을 입력하면, 어떤 연산을 처리한 다음에 결과를 반환한다.

(수학에 너무 집중하지는 말자, 단순히 무언가 정의해두고 필요할 때 불러서 사용한다는 개념으로 이해하면 충분하다)

메서드는 크게 **메서드 선언**과 **메서드 본문**으로 나눌 수 있다.

## 메서드 선언(Method Declaration)

```
public static int add(int a, int b)
```

메서드의 선언 부분으로, 메서드 이름, 반환 타입, 매개변수(파라미터) 목록을 포함한다.

이름 그대로 이런 메서드가 있다고 선언하는 것이다. 메서드 선언 정보를 통해 다른 곳에서 해당 메서드를 호출할 수 있다.

- `public static`
  - `public`: 다른 클래스에서 호출할 수 있는 메서드라는 뜻이다. 접근 제어에서 학습한다.
  - `static`: 객체를 생성하지 않고 호출할 수 있는 정적 메서드라는 뜻이다. 자세한 내용은 뒤에서 다룬다.
  - 두 키워드의 자세한 내용은 뒤에서 다룬다. 지금은 단순하게 메서드를 만들 때 둘을 사용해야 한다고 생각하자.
- `int add(int a, int b)`
  - `int`: 반환 타입을 정의한다. 메서드의 실행 결과를 반환할 때 사용할 반환 타입을 지정한다.
  - `add`: 메서드에 이름을 부여한다. 이 이름으로 메서드를 호출할 수 있다.
  - `(int a, int b)`: 메서드를 호출할 때 전달하는 입력 값을 정의한다. 이 변수들은 해당 메서드 안에서만

사용된다. 이렇게 메서드 선언에 사용되는 변수를 영어로 파라미터(parameter), 한글로 매개변수라 한다.

## 메서드 본문(Method Body)

```
{
 System.out.println(a + "+" + b + " 연산 수행");
 int sum = a + b;
 return sum;
}
```

- 메서드가 수행해야 하는 코드 블록이다.
- 메서드를 호출하면 메서드 본문이 순서대로 실행된다.
- 메서드 본문은 블랙박스이다. 메서드를 호출하는 곳에서는 메서드 선언은 알지만 메서드 본문은 모른다.
- 메서드의 실행 결과를 반환하려면 `return` 문을 사용해야 한다. `return` 문 다음에 반환할 결과를 적어주면 된다.
  - `return sum;`: `sum` 변수에 들어있는 값을 반환한다.

## 메서드 호출

앞서 정의한 메서드를 호출해서 실행하려면 메서드 이름에 입력 값을 전달하면 된다. 보통 메서드를 호출한다고 표현한다.

```
int sum1 = add(5, 10);
int sum2 = add(15, 20);
```

## 메서드를 호출하면 어떻게 실행되는지 순서대로 확인해보자

```
int sum1 = add(5, 10); //add라는 메서드를 숫자 5,10을 전달하면서 호출한다.
int sum1 = 15; //add(5, 10)이 실행된다. 실행 결과는 반환 값은 15이다.
//sum1에 15 값이 저장된다.
```

메서드를 호출하면 메서드는 계산을 끝내고 결과를 반환한다. 쉽게 이야기하자면, 메서드 호출이 끝나면 해당 메서드가 반환한 결과 값으로 치환된다.

조금 더 자세히 알아보자. 메서드의 코드는 일부 축약했다.

```
//1: 메서드 호출
int sum1 = add(5, 10);

//2: 파라미터 변수 a=5, b=10이 전달되면서 메서드가 수행된다.
public static int add(int a=5, int b=10) {
 int sum = a + b;
 return sum;
}

//3: 메서드가 수행된다.
```

```

public static int add(int a=5, int b=10) {
 int sum = a(5) + b(10);
 return sum;
}

//4: return을 사용해서 메서드 실행의 결과인 sum을 반환한다. sum에는 값 15가 들어있으므로 값 15가 반환된다.
public static int add(int a=5, int b=10) {
 int sum = 15;
 return sum(15);
}

//5: 메서드 호출 결과로 메서드에서 반환한 값 15가 나온다. 이 값을 sum1에 대입했다.
int sum1 = 15;

```

메서드 호출이 끝나면 더 이상 해당 메서드가 사용한 메모리를 낭비할 이유가 없다. 메서드 호출이 끝나면 메서드 정의에 사용한 파라미터 변수인 `int a`, `int b`는 물론이고, 그 안에서 정의한 `int sum`도 모두 제거된다.

## 메서드 호출과 용어정리

메서드를 호출할 때는 다음과 같이 메서드에 넘기는 값과 매개변수(파라미터)의 타입이 맞아야 한다. 물론 넘기는 값과 매개변수(파라미터)의 순서와 갯수도 맞아야 한다.

```

호출: call("hello", 20)
메서드 정의: int call(String str, int age)

```

### 인수(Argument)

여기서 "hello", 20처럼 넘기는 값을 영어로 Argument(아규먼트), 한글로 인수 또는 인자라 한다.  
실무에서는 아규먼트, 인수, 인자라는 용어를 모두 사용한다.

### 매개변수(Parameter)

메서드를 정의할 때 선언한 변수인 `String str`, `int age`를 매개변수, 파라미터라 한다.

메서드를 호출할 때 인수를 넘기면, 그 인수가 매개변수에 대입된다.

실무에서는 매개변수, 파라미터 용어를 모두 사용한다.

### 용어정리

- **인수**라는 용어는 '인'과 '수'의 합성어로, '들어가는 수'라는 의미를 가진다. 즉, 메서드 내부로 들어가는 값을 의미 한다. 인자도 같은 의미이다.
- **매개변수, parameter**는 '매개'와 '변수'의 합성어로, '중간에서 전달하는 변수'라는 의미를 가진다. 즉, 메서드 호

출부와 메서드 내부 사이에서 값을 전달하는 역할을 하는 변수라는 뜻이다.

## 메서드 정의

메서드는 다음과 같이 정의한다.

```
public static int add(int a, int b) {
 //메서드 본문, 실행 코드
}
```

```
제어자 반환타입 메서드이름(매개변수 목록) {
 메서드 본문
}
```

- **제어자(Modifier)**: `public`, `static`과 같은 부분이다. 제어자는 뒤에서 설명한다. 지금은 항상 `public static` 키워드를 입력하자.
- **반환 타입(Return Type)**: 메서드가 실행 된 후 반환하는 데이터의 타입을 지정한다. 메서드가 값을 반환하지 않는 경우, 없다는 뜻의 `void`를 사용해야 한다. 예) `void print(String str)`
- **메서드 이름(Method Name)**: 메서드의 이름이다. 이 이름은 메서드를 호출하는 데 사용된다.
- **매개변수(Parameter)**: 입력 값으로, 메서드 내부에서 사용할 수 있는 변수이다. 매개변수는 옵션이다. 입력값이 필요 없는 메서드는 매개변수를 지정하지 않아도 된다. 예) `add()`
- **메서드 본문(Method Body)**: 실제 메서드의 코드가 위치한다. 중괄호 `{}` 사이에 코드를 작성한다.

## 매개변수가 없거나 반환 타입이 없는 경우

매개변수가 없고, 반환 타입도 없는 메서드를 확인해보자.

```
package method;

public class Method2 {

 public static void main(String[] args) {
 printHeader();
 System.out.println("프로그램이 동작합니다.");
 printFooter();
 }

 public static void printHeader() {
```

```

 System.out.println("= 프로그램을 시작합니다 =");
 return; //void의 경우 생략 가능
 }

 public static void printFooter() {
 System.out.println("= 프로그램을 종료합니다 =");
 }

}

```

## 실행 결과

```

= 프로그램을 시작합니다 =
프로그램이 동작합니다.
= 프로그램을 종료합니다 =

```

`printHeader()`, `printFooter()` 메서드는 매개변수가 없고, 반환 타입도 없다.

- 매개변수가 없는 경우
  - 선언: `public static void printHeader()` 와 같이 매개변수를 비워두고 정의하면 된다.
  - 호출: `printHeader();` 와 같이 인수를 비워두고 호출하면 된다.
- 반환 타입이 없는 경우
  - 선언: `public static void printHeader()` 와 같이 반환 타입을 `void`로 정의하면 된다.
  - 호출: `printHeader();` 와 같이 반환 타입이 없으므로 메서드만 호출하고 반환 값을 받지 않으면 된다.
    - ◆ `String str = printHeader();` 반환 타입이 `void`이기 때문에 이렇게 반환 값을 받으면 컴파일 오류가 발생한다.

## void와 return 생략

모든 메서드는 항상 `return`을 호출해야 한다. 그런데 반환 타입 `void`의 경우에는 예외로 `printFooter()` 와 같이 생략해도 된다. 자바가 반환 타입이 없는 경우에는 `return`을 마지막줄에 넣어준다. 참고로 `return`을 만나면 해당 메서드는 종료된다.

## 반환 타입

반환 타입이 있으면 반드시 값을 반환해야 한다.

반환 타입이 있는 메서드는 반드시 `return`을 사용해서 값을 반환해야 한다.

이 부분은 특히 조건문과 함께 사용할 때 주의해야 한다.

### MethodReturn1

```
package method;

public class MethodReturn1 {

 public static void main(String[] args) {
 boolean result = odd(2);
 System.out.println(result);
 }

 public static boolean odd(int i) {
 if (i % 2 == 1) {
 return true;
 }
 }
}
```

이 코드에서 `if` 조건이 만족할 때는 `true` 가 반환되지만, 조건을 만족하지 않으면 어떻게 될까? 조건을 만족하지 않은 경우에는 `return` 문이 실행되지 않는다. 따라서 이 코드를 실행하면 `return` 문을 누락했다는 다음과 같은 컴파일 오류가 발생한다.

### 컴파일 오류

```
java: missing return statement
```

### MethodReturn1 - 수정 코드

```
package method;

public class MethodReturn1 {

 public static void main(String[] args) {
 boolean result = odd(2);
 System.out.println(result);
 }

 public static boolean odd(int i) {
 if (i % 2 == 1) {
 return true;
 } else {
 return false;
 }
 }
}
```

```
 }
}
```

이렇게 수정하면 `if` 조건을 만족하지 않아도 `else`를 통해 `return`문이 실행된다.

**return** 문을 만나면 그 즉시 메서드를 빠져나간다.

`return` 문을 만나면 그 즉시 해당 메서드를 빠져나간다.

다음 로직을 수행하는 메서드를 만들어보자.

- 18살 미만의 경우: 미성년자는 출입이 불가합니다.
- 18살 이상의 경우: 입장하세요.

## MethodReturn2

```
package method;

public class MethodReturn2 {

 public static void main(String[] args) {
 checkAge(10);
 checkAge(20);
 }

 public static void checkAge(int age) {
 if (age < 18) {
 System.out.println(age + "살, 미성년자는 출입이 불가능합니다.");
 return;
 }

 System.out.println(age + "살, 입장하세요.");
 }
}
```

- 18세 미만의 경우 "미성년자는 출입이 불가능합니다"를 출력하고 바로 `return`문이 수행된다. 따라서 다음 로직을 수행하지 않고, 해당 메서드를 빠져나온다.
- 18세 이상의 경우 "입장하세요"를 출력하고, 메서드가 종료된다. 참고로 반환 타입이 없는 `void` 형이기 때문에 마지막 줄의 `return`은 생략할 수 있다.

## 반환 값 무시

반환 타입이 있는 메서드를 호출했는데 만약 반환 값이 필요없다면 사용하지 않아도 된다.

예시1: int sum = add(1,2) //반환된 값을 받아서 sum에 저장했다.

예시2: add(1,2) //반환된 값을 사용하지 않고 버린다. 여기서는 예시1과 같이 호출 결과를 변수에 담지 않았다. 단순히 메서드만 호출했다.

## 메서드 호출과 값 전달1

지금부터 자바에서 아주 중요한 대원칙 하나를 이야기하겠다. 밑줄 100번 긋자!

**자바는 항상 변수의 값을 복사해서 대입한다.**

이 대원칙은 반드시 이해해야 한다. 그러면 아무리 복잡한 상황에도 코드를 단순하게 이해할 수 있다.

### 변수와 값 복사

다음 코드를 보고 어떤 결과가 나올지 먼저 생각해보자.

#### MethodValue0

```
package method;

public class MethodValue0 {

 public static void main(String[] args) {
 int num1 = 5;
 int num2 = num1;
 num2 = 10;
 System.out.println("num1=" + num1);
 System.out.println("num2=" + num2);
 }
}
```

#### 실행 결과

```
num1=5
num2=10
```

#### 실행 과정

```
int num2 = num1; //num1의 값은 5이다. num1(5)
int num2 = 5; //num2 변수에 대입하기 전에 num1의 값 5를 읽는다. 결과: num1(5), num2(5)
num2 = 10; // num2에 10을 대입한다. 결과: num1(5), num2(10)
```

여기서 값을 복사해서 대입한다는 부분이 바로 이 부분이다.

```
int num2 = num1;
```

- 이 부분은 생각해보면 num1에 있는 값 5를 복사해서 num2에 넣는 것이다.
  - 복사한다고 표현한 이유는 num1의 값을 읽어도 num1에 있는 기존 값이 유지되고, 새로운 값이 num2에 들어가기 때문이다. 마치 num1의 값이 num2에 복사가 된 것 같다.
  - num1이라는 변수 자체가 num2에 들어가는 것이 아니다. num1에 들어있는 값을 읽고 복사해서 num2에 넣는 것이다.
  - 간단하게 num1에 있는 값을 num2에 대입한다고 표현한다. 하지만 실제로는 그 값을 복사해서 대입하는 것이다.

너무 당연한 이야기를 왜 이렇게 장황하게 풀어서 하지? 라고 생각한다면 이제 진짜 문제를 만나보자.

## 메서드 호출과 값 복사

다음은 숫자를 2배 곱하는 메서드이다. 다음 코드를 보고 어떤 결과가 나올지 먼저 생각해보자.

### MethodValue1

```
package method;

public class MethodValue1 {

 public static void main(String[] args) {
 int num1 = 5;
 System.out.println("1. changeNumber 호출 전, num1: " + num1);
 changeNumber(num1);
 System.out.println("4. changeNumber 호출 후, num1: " + num1);
 }

 public static void changeNumber(int num2) {
 System.out.println("2. changeNumber 변경 전, num2: " + num2);
 num2 = num2 * 2;
 System.out.println("3. changeNumber 변경 후, num2: " + num2);
 }
}
```

실행 결과를 먼저 예측해보고 그 다음에 실행 결과를 확인해보자.

혹시라도 실행 결과의 마지막이 10이라고 생각했다면 대원칙을 떠올려보자

## 실행 결과

1. changeNumber 호출 전, num1: 5
2. changeNumber 변경 전, num2: 5
3. changeNumber 변경 후, num2: 10
4. changeNumber 호출 후, num1: 5

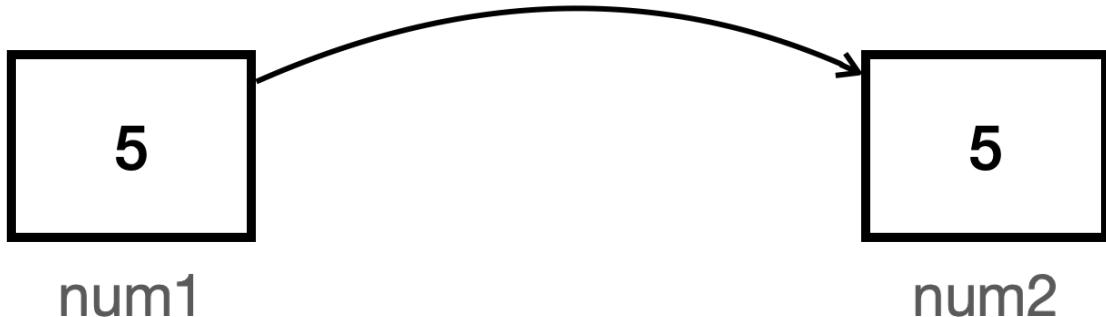
다음 대원칙을 따라간다면 문제를 정확하게 풀 수 있다.

자바는 항상 변수의 값을 복사해서 대입한다.

## 실행 과정 그림

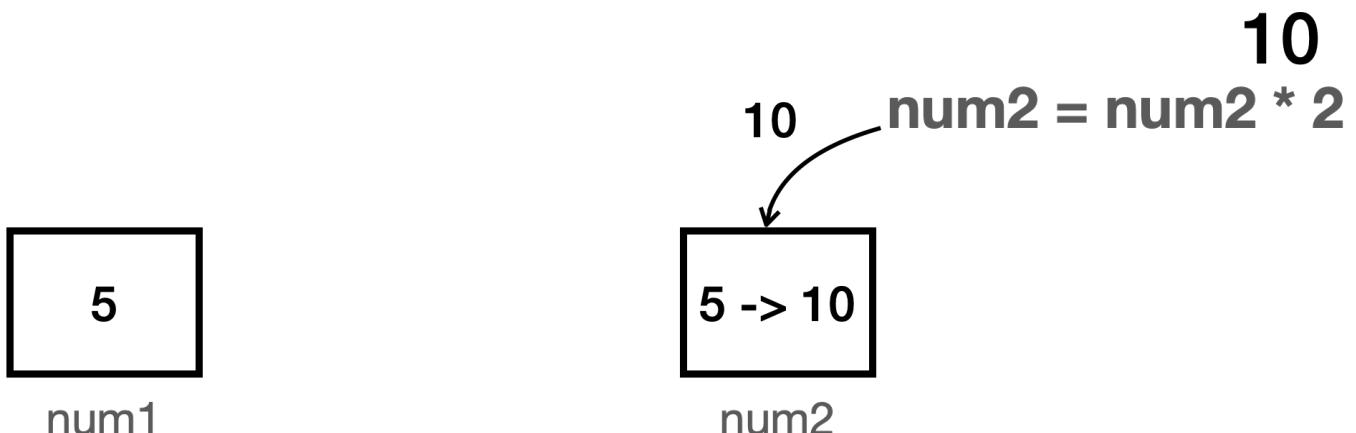
### 변수의 값을 복사해서 전달

#### changeNumber(num1)



#### changeNumber(num1) 호출 시점

- `num1`의 값 5를 읽고 복사해서 `num2`에 전달 → 이 부분이 핵심



#### changeNumber메서드 실행 중

- `num2`의 변경은 `num1`에 영향을 주지 않는다. 왜냐하면 앞서 값을 복사해서 전달했기 때문이다.

5

num1

10

num2

최종 결과

실행 과정 코드

```
changeNumber(num1); //changeNumber를 호출한다. num1(5)
changeNumber(5); //num1의 값을 읽는다.

void changeNumber(int num2=5) //num1의 값 5가 num2에 복사된다. 결과: num1(5), num2(5)
num2 = num2 * 2; //num2에 2를 곱한다. 결과: num1(5), num2(5)
num2 = 5 * 2; //num2의 값을 읽어서 2를 곱한다. 결과: num1(5), num2(5)
num2 = 10; //num2에 계산 결과인 값 10을 대입한다. 결과: num1(5), num2(10)
```

num2를 출력한다: num2의 값인 10이 출력된다.

num1을 출력한다: num1의 값인 5가 출력된다.

결과적으로 매개변수 num2의 값만 10으로 변경되고 num1의 값은 변경되지 않고 기존 값인 5로 유지된다. 자바는 항상 값을 복사해서 전달하기 때문에 num2의 값을 바꾸더라도 num1에는 영향을 주지 않는다.

## 메서드 호출과 값 전달2

### 메서드 호출과 이름이 같은 변수

같은 문제를 호출자의 변수 이름과 매개변수의 이름을 같게 해서 한번 더 풀어보자

#### MethodValue2

```
package method;

public class MethodValue2 {

 public static void main(String[] args) {
 int number = 5;
 System.out.println("1. changeNumber 호출 전, number: " + number); // 출력:
```

```

5
 changeNumber(number);
 System.out.println("4. changeNumber 호출 후, number: " + number); // 출력:
5
}

public static void changeNumber(int number) {
 System.out.println("2. changeNumber 변경 전, number: " + number); // 출력:
5
 number = number * 2;
 System.out.println("3. changeNumber 변경 후, number: " + number); // 출력:
10
}
}

```

이번에는 `main()`에 정의한 변수와 메서드의 매개변수(파라미터)의 이름이 둘다 `number`로 같다.

## 실행 결과

```

1. changeNumber 호출 전, number: 5
2. changeNumber 변경 전, number: 5
3. changeNumber 변경 후, number: 10
4. changeNumber 호출 후, number: 5

```

`main()`도 사실은 메서드이다. 각각의 메서드 안에서 사용하는 변수는 서로 완전히 분리된 다른 변수이다. 물론 이름이 같아도 완전히 다른 변수다. 따라서 `main()`의 `number`와 `changeNumber()`의 `number`는 서로 다른 변수이다.

## 실행 과정

```

changeNumber(number); //changeNumber를 호출한다. main의 number(5)
changeNumber(5); //number의 값을 읽는다.

//main의 number값 5가 changeNumber의 number에 복사된다.
//결과: main의 number(5), changeNumber의 number(5)
void changeNumber(int number=5)

//changeNumber의 number에 값 10을 대입한다.
//결과: main의 number(5), changeNumber의 number(10)
number = number * 2;
main의 number를 출력한다: main의 number의 값인 5가 출력된다.

```

## 메서드 호출과 값 반환받기

그렇다면 메서드를 사용해서 값을 변경하려면 어떻게 해야할까?

메서드의 호출 결과를 반환 받아서 사용하면 된다.

### MethodValue3

```
package method;

public class MethodValue3 {

 public static void main(String[] args) {
 int num1 = 5;
 System.out.println("changeNumber 호출 전, num1: " + num1); // 출력: 5
 num1 = changeNumber(num1);
 System.out.println("changeNumber 호출 후, num1: " + num1); // 출력: 10
 }

 public static int changeNumber(int num2) {
 num2 = num2 * 2;
 return num2;
 }
}
```

### 실행 결과

```
changeNumber 호출 전, num1: 5
changeNumber 호출 후, num1: 10
```

### 실행 과정

```
num1 = changeNumber(num1); //num1(5)
num1 = changeNumber(5);

//호출 시작:changeNumber()
//num1의 값 5가 num2에 대입된다. num1의 값을 num2에 복사한다. num1(5), num2(5)
int changeNumber(int num2=5)
num2 = num2 * 2; //계산 결과: num1(5), num2(10)
return num2; // num2의 값은 10이다.
return 10;
//호출 끝: changeNumber()

num1 = changeNumber(5); //반환 결과가 10이다.
num1 = 10; //결과: num1(10)
```

## 정리

꼭 기억하자! 자바는 항상 변수의 값을 복사해서 대입한다.

(참고로 뒤에서 참조형이라는 것을 학습하는데, 이때도 똑같다. 결국 참조형 변수에 있는 값인 참조값을 복사하는 것이다! 이것은 나중에 알아본다.)

## 메서드와 형변환

메서드를 호출할 때도 형변환이 적용된다. 메서드 호출과 명시적 형변환, 자동 형변환에 대해 알아보자.

### 명시적 형변환

메서드를 호출하는데 인자와 매개변수의 타입이 맞지 않다면 어떻게 해야 할까?

다음 예제 코드를 보자

#### MethodCasting1

```
package method;

public class MethodCasting1 {

 public static void main(String[] args) {
 double number = 1.5;
 //printNumber(number); // double을 int형에 대입하므로 컴파일 오류
 printNumber((int) number); // 명시적 형변환을 사용해 double을 int로 변환
 }

 public static void printNumber(int n) {
 System.out.println("숫자: " + n);
 }
}
```

먼저 주석으로 처리해둔 부분의 주석을 풀고 실행해보자.

```
printNumber(number) // double을 int형에 대입하므로 컴파일 오류
```

### 실행 결과 - 컴파일 오류

```
java: incompatible types: possible lossy conversion from double to int
```

다음과 같은 이유로 컴파일 오류가 발생한다.

```
printNumber(number) //number는 1.5 실수
printNumber(1.5) //메서드를 호출하기 전에 number 변수의 값을 읽음
void printNumber(int n=1.5) //int형 매개변수 n에 double형 실수인 1.5를 대입 시도, 컴파일 오류
```

이 경우 메서드 호출이 꼭 필요하다면 다음과 같이 명시적 형변환을 사용해야 한다.

```
printNumber((int) number); // 명시적 형변환을 사용해 double을 int로 변환
printNumber(1); // (double) 1.5 -> (int) 1로 변환
void printNumber(int n=1) //int형 파라미터 변수 n에 int형 1을 대입
```

## 실행 결과

```
숫자: 1
```

## 자동 형변환

```
int < long < double
```

메서드를 호출할 때 매개변수에 값을 전달하는 것도 결국 변수에 값을 대입하는 것이다. 따라서 앞서 배운 자동 형변환이 그대로 적용된다.

```
package method;

public class MethodCasting2 {

 public static void main(String[] args) {
 int number = 100;
 printNumber(number); // int에서 double로 자동 형변환
 }

 public static void printNumber(double n) {
 System.out.println("숫자: " + n);
 }
}
```

- double형 매개변수(파라미터)에 int형 인수를 전달하는데 문제없이 잘 동작한다.

## 실행 결과

숫자: 100.0

다음과 같이 자동 형변환이 동작한다.

```
printNumber(number); // number는 int형 100
printNumber(100); //메서드를 호출하기 전에 number 변수의 값을 읽음

void printNumber(double n=100) //double형 파라미터 변수 n에 int형 값 100을 대입
void printNumber(double n=(double) 100) //double이 더 큰 숫자 범위이므로 자동 형변환 적용
void printNumber(double n=100.0) //자동 형변환 완료
```

## 정리

메서드를 호출할 때는 전달하는 인수의 타입과 매개변수의 타입이 맞아야 한다. 단 타입이 달라도 자동 형변환이 가능한 경우에는 호출할 수 있다.

## 메서드 오버로딩

다음과 같은 메서드를 만들고 싶다.

- 두 수를 더하는 메서드
- 세 수를 더하는 메서드

이 경우 둘다 더하는 메서드이기 때문에 가급적 같은 이름인 `add`를 사용하고 싶다.

자바는 메서드의 이름 뿐만 아니라 매개변수 정보를 함께 사용해서 메서드를 구분한다.

따라서 다음과 같이 이름이 같고, 매개변수가 다른 메서드를 정의할 수 있다.

### 오버로딩 성공

```
add(int a, int b)
add(int a, int b, int c)
add(double a, double b)
```

이렇게 이름이 같고 매개변수가 다른 메서드를 여러개 정의하는 것을 메서드 오버로딩(Overloading)이라 한다.

오버로딩은 번역하면 과적인데, 과하게 물건을 담았다는 뜻이다. 따라서 같은 이름의 메서드를 여러개 정의했다고 이해하면 된다.

### 오버로딩 규칙

메서드의 이름이 같아도 매개변수의 타입 및 순서가 다르면 오버로딩을 할 수 있다. 참고로 반환 타입은 인정하지 않는다.

다음 케이스는 메서드 이름과 매개변수의 타입이 같으므로 컴파일 오류가 발생한다. 반환 타입은 인정하지 않는다.

## 오버로딩 실패

```
int add(int a, int b)
double add(int a, int b)
```

### 용어: 메서드 시그니처(method signature)

메서드 시그니처 = 메서드 이름 + 매개변수 타입(순서)

메서드 시그니처는 자바에서 메서드를 구분할 수 있는 고유한 식별자나 서명을 뜻한다. 메서드 시그니처는 메서드의 이름과 매개변수 타입(순서 포함)으로 구성되어 있다. 쉽게 이야기해서 메서드를 구분할 수 있는 기준이다. 자바 입장에서는 각각의 메서드를 고유하게 구분할 수 있어야한다. 그래야 어떤 메서드를 호출 할지 결정할 수 있다.

따라서 메서드 오버로딩에서 설명한 것 처럼 메서드 이름이 같아도 메서드 시그니처가 다르면 다른 메서드로 간주한다. 반환 타입은 시그니처에 포함되지 않는다. 방금 오버로딩이 실패한 두 메서드를 보자. 두 메서드는 `add(int a, int b)`로 메서드 시그니처가 같다. 따라서 메서드의 구분이 불가능하므로 컴파일 오류가 발생한다.

다양한 예제를 통해서 메서드 오버로딩을 알아보자.

먼저 매개변수의 갯수가 다른 오버로딩 예제를 보자

### Overloading1

```
package overloading;

public class Overloading1 {

 public static void main(String[] args) {
 System.out.println("1: " + add(1, 2));
 System.out.println("2: " + add(1, 2, 3));
 }

 // 첫 번째 add 메서드: 두 정수를 받아서 합을 반환한다.
 public static int add(int a, int b) {
 System.out.println("1번 호출");
 return a + b;
 }

 // 두 번째 add 메서드: 세 정수를 받아서 합을 반환한다.
 // 첫 번째 메서드와 이름은 같지만, 매개변수 목록이 다르다.
 public static int add(int a, int b, int c) {
 System.out.println("2번 호출");
 return a + b + c;
 }
}
```

```
}
```

1: 정수 1,2를 호출했으므로 `add(int a, int b)` 가 호출된다.

2: 정수 1,2,3을 호출했으므로 `add(int a, int b, int c)` 가 호출된다.

## 실행 결과

```
1번 호출
```

```
1: 3
```

```
2번 호출
```

```
2: 6
```

이번에는 매개변수의 타입이 다른 오버로딩 예제를 보자

## Overloading2

```
package overloading;

public class Overloading2 {

 public static void main(String[] args) {
 myMethod(1, 1.2);
 myMethod(1.2, 2);
 }

 public static void myMethod(int a, double b) {
 System.out.println("int a, double b");
 }

 public static void myMethod(double a, int b) {
 System.out.println("double a, int b");
 }
}
```

1: 정수1, 실수 1.2를 호출했으므로 `myMethod(int a, double b)` 가 호출된다.

2: 실수 1.2, 정수 2를 호출했으므로 `myMethod(double a, int b)` 가 호출된다.

## 실행 결과

```
int a, double b
double a, int b
```

마지막으로 매개변수의 타입이 다른 경우를 추가로 확인해보자.

### Overloading3

```
package overloading;

public class Overloading3 {

 public static void main(String[] args) {
 System.out.println("1: " + add(1, 2));
 System.out.println("2: " + add(1.2, 1.5));
 }

 // 첫 번째 add 메서드: 두 정수를 받아서 합을 반환한다.
 public static int add(int a, int b) {
 System.out.println("1번 호출");
 return a + b;
 }

 // 두 번째 add 메서드: 두 실수를 받아서 합을 반환한다.
 // 첫 번째 메서드와 이름은 같지만, 매개변수의 유형이 다르다.
 public static double add(double a, double b) {
 System.out.println("2번 호출");
 return a + b;
 }

}
```

1: 정수1, 정수 2를 호출했으므로 `add(int a, int b)` 가 호출된다.

2: 실수 1.2, 실수 1.5를 호출했으므로 `add(double a, double b)` 가 호출된다.

### 실행 결과

```
1번 호출
1: 3
2번 호출
2: 2.7
```

여기서 만약 다음 첫 번째 메서드를 삭제하면 어떻게 될까?

```
public static int add(int a, int b) {
 System.out.println("1번 호출");
 return a + b;
}
```

1: int 형 정수 1, int 형 정수 2를 호출했으므로 자동 형변환이 발생해서 add(double a, double b) 가 호출된다.

2: 실수 1.2, 실수 1.5를 호출했으므로 add(double a, double b) 가 호출된다.

## 실행 결과

2번 호출

1: 3.0

2번 호출

2: 2.7

정리하면 먼저 본인의 타입에 최대한 맞는 메서드를 찾아서 실행하고, 그래도 없으면 형 변환 가능한 타입의 메서드를 찾아서 실행한다.

# 문제와 풀이 1

## 코딩이 처음이라면 필독!

프로그래밍이 처음이라면 아직 코딩 자체가 익숙하지 않기 때문에 문제와 풀이에 상당히 많은 시간을 쓰게 될 수 있다. 강의를 들을 때는 다 이해가 되는 것 같았는데, 막상 혼자 생각해서 코딩을 하려니 잘 안되는 것이다. 이것은 아직 코딩이 익숙하지 않기 때문인데, 처음 코딩을 하는 사람이라면 누구나 겪는 자연스러운 현상이다.

문제를 스스로 풀기 어려운 경우, 너무 고민하기보다는 먼저 **강의 영상의 문제 풀이 과정을 코드로 따라하면서 이해하자. 반드시 코드로 따라해야 한다.** 그래야 코딩하는 것에 조금씩 익숙해질 수 있다. 그런 다음에 정답을 지우고 스스로 문제를 풀어보면 된다. 참고로 강의를 듣는 시간만큼 문제와 풀이에도 많은 시간을 들어야 제대로 성장할 수 있다!

## 문제 - 평균값 리펙토링

메서드를 잘 이해하고 있는지 확인하기 위해 다음 코드를 메서드를 사용하도록 리펙토링해보자.

### MethodEx1

```
package method.ex;

public class MethodEx1 {
 public static void main(String[] args) {
 int a = 1;
```

```

int b = 2;
int c = 3;

int sum = a + b + c;
double average = sum / 3.0;
System.out.println("평균값: " + average);

int x = 15;
int y = 25;
int z = 35;

sum = x + y + z;
average = sum / 3.0;
System.out.println("평균값: " + average);
}

}

```

## 실행 결과

```

평균값: 2.0
평균값: 25.0

```

## 정답

```

package method.ex;

public class MethodEx1Ref {

 public static void main(String[] args) {
 System.out.println("평균값: " + average(1, 2, 3));
 System.out.println("평균값: " + average(15, 25, 35));
 }

 public static double average(int a, int b, int c) {
 int sum = a + b + c;
 return sum / 3.0;
 }
}

```

## 문제 - 반복 출력 리펙토링

다음은 특정 숫자만큼 같은 메시지를 반복 출력하는 기능이다.

메서드를 사용해서 리팩토링해보자.

## MethodEx2

```
package method.ex;

public class MethodEx2 {
 public static void main(String[] args) {
 String message = "Hello, world!";

 for (int i = 0; i < 3; i++) {
 System.out.println(message);
 }

 for (int i = 0; i < 5; i++) {
 System.out.println(message);
 }

 for (int i = 0; i < 7; i++) {
 System.out.println(message);
 }
 }
}
```

## 실행 결과

```
Hello, world!
Hello, world!
... //여러번 반복
```

## 정답

```
package method.ex;

public class MethodEx2Ref {
 public static void main(String[] args) {
 printMessage("Hello, world!", 3);
 printMessage("Hello, world!", 5);
 printMessage("Hello, world!", 7);
 }

 public static void printMessage(String message, int times) {
 for (int i = 0; i < times; i++) {
 System.out.println(message);
 }
 }
}
```

```
 }
}
```

## 문제 - 입출금 리펙토링

다음은 입금, 출금을 나타내는 코드이다.

입금(deposit)과, 출금(withdraw)을 메서드로 만들어서 리펙토링 해보자.

```
package method.ex;

public class MethodEx3 {
 public static void main(String[] args) {
 int balance = 1000;

 // 입금 1000
 int depositAmount = 1000;
 balance += depositAmount;
 System.out.println(depositAmount + "원을 입금하였습니다. 현재 잔액: " + balance
+ "원");

 // 출금 2000
 int withdrawAmount = 2000;
 if (balance >= withdrawAmount) {
 balance -= withdrawAmount;
 System.out.println(withdrawAmount + "원을 출금하였습니다. 현재 잔액: " +
balance + "원");
 } else {
 System.out.println(withdrawAmount + "원을 출금하려 했으나 잔액이 부족합니다.");
 }

 System.out.println("최종 잔액: " + balance + "원");
 }
}
```

## 실행 결과

```
1000원을 입금하였습니다. 현재 잔액: 11000원
```

```
2000원을 출금하였습니다. 현재 잔액: 9000원
```

```
최종 잔액: 9000원
```

## 정답

```

package method.ex;

public class MethodEx3Ref {
 public static void main(String[] args) {
 int balance = 10000;

 balance = deposit(balance, 1000);
 balance = withdraw(balance, 2000);

 System.out.println("최종 잔액: " + balance + "원");
 }

 public static int deposit(int balance, int amount) {
 balance += amount;
 System.out.println(amount + "원을 입금하였습니다. 현재 잔액: " + balance +
"원");
 return balance;
 }

 public static int withdraw(int balance, int amount) {
 if (balance >= amount) {
 balance -= amount;
 System.out.println(amount + "원을 출금하였습니다. 현재 잔액: " + balance +
"원");
 } else {
 System.out.println(amount + "원을 출금하려 했으나 잔액이 부족합니다.");
 }
 return balance;
 }
}

```

리펙토링 결과를 보면 `main()`은 세세한 코드가 아니라 전체 구조를 한눈에 볼 수 있게 되었다. 쉽게 이야기해서 책의 목차를 보는 것 같다. 더 자세히 알고 싶으면 해당 메서드를 찾아서 들어가면 된다. 그리고 입금과 출금 부분이 메서드로 명확하게 분리되었기 때문에 이후에 변경 사항이 발생하면 관련된 메서드만 수정하면 된다. 특정 메서드로 수정 범위가 한정되기 때문에 더 유지보수 하기 좋다.

이런 리펙토링을 메서드 추출(Extract Method)이라 한다. 메서드를 재사용하는 목적이 아니어도 괜찮다. 메서드를 적절하게 사용해서 분류하면 구조적으로 읽기 쉽고 유지보수 하기 좋은 코드를 만들 수 있다.

그리고 메서드의 이름 덕분에 프로그램을 더 읽기 좋게 만들 수 있다.

## 문제와 풀이2

### 문제 - 은행 계좌 입출금

- 다음 실행 예시를 참고해서, 사용자로부터 계속 입력을 받아 입금과 출금을 반복 수행하는 프로그램을 작성하자.  
또한 간단한 메뉴를 표시하여 어떤 동작을 수행해야 할지 선택할 수 있게 하자
- 출금시 잔액이 부족하다면 "x원을 출금하려 했으나 잔액이 부족합니다."라고 출력해야 한다.

### 실행 예시

-----  
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료  
-----

선택: 1

입금액을 입력하세요: 10000

10000원을 입금하였습니다. 현재 잔액: 10000원

-----  
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료  
-----

선택: 2

출금액을 입력하세요: 8000

8000원을 출금하였습니다. 현재 잔액: 2000원

-----  
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료  
-----

선택: 2

출금액을 입력하세요: 3000

3000원을 출금하려 했으나 잔액이 부족합니다.

-----  
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료  
-----

선택: 3

현재 잔액: 2000원

-----  
1.입금 | 2.출금 | 3.잔액 확인 | 4.종료  
-----

선택: 4

시스템을 종료합니다.

### 정답

```
package method.ex;
```

```
import java.util.Scanner;

public class MethodEx4 {
 public static void main(String[] args) {
 int balance = 0;
 Scanner scanner = new Scanner(System.in);

 while (true) {
 System.out.println("-----");
 System.out.println("1.입금 | 2.출금 | 3.잔액 확인 | 4.종료");
 System.out.println("-----");
 System.out.print("선택: ");

 int choice = scanner.nextInt();
 int amount;

 switch (choice) {
 case 1:
 System.out.print("입금액을 입력하세요: ");
 amount = scanner.nextInt();
 balance = deposit(balance, amount);
 break;

 case 2:
 System.out.print("출금액을 입력하세요: ");
 amount = scanner.nextInt();
 balance = withdraw(balance, amount);
 break;

 case 3:
 System.out.println("현재 잔액: " + balance + "원");
 break;

 case 4:
 System.out.println("시스템을 종료합니다.");
 return;

 default:
 System.out.println("올바른 선택이 아닙니다. 다시 선택해주세요.");
 }
 }
 }
}
```

```

public static int deposit(int balance, int amount) {
 balance += amount;
 System.out.println(amount + "원을 입금하였습니다. 현재 잔액: " + balance +
"원");
 return balance;
}

public static int withdraw(int balance, int amount) {
 if (balance >= amount) {
 balance -= amount;
 System.out.println(amount + "원을 출금하였습니다. 현재 잔액: " + balance +
"원");
 } else {
 System.out.println(amount + "원을 출금하려 했으나 잔액이 부족합니다.");
 }
 return balance;
}
}

```

## 정리

### 변수명 vs 메서드명

변수 이름은 일반적으로 명사를 사용한다. 한편 메서드는 무언가 동작하는데 사용하기 때문에 일반적으로 동사로 시작한다.

이런 차이점 외에는 변수 이름과 메서드 이름에 대한 규칙은 둘다 같다.

- **변수명 예):** `customerName`, `totalSum`, `employeeCount`, `isAvailable`
- **메서드명 예):** `printReport()`, `calculateSum()`, `addCustomer()`, `getEmployeeCount()`, `setEmployeeName()`

### 메서드 사용의 장점

- **코드 재사용:** 메서드는 특정 기능을 캡슐화하므로, 필요할 때마다 그 기능을 다시 작성할 필요 없이 해당 메서드를 호출함으로써 코드를 재사용할 수 있다.
- **코드의 가독성:** 이름이 부여된 메서드는 코드가 수행하는 작업을 명확하게 나타내므로, 코드를 읽는 사람에게 주관적인 문맥을 제공한다.

- **모듈성**: 큰 프로그램을 작은, 관리 가능한 부분으로 나눌 수 있다. 이는 코드의 가독성을 향상시키고 디버깅을 쉽게 만든다.
- **코드 유지 관리**: 메서드를 사용하면, 코드의 특정 부분에서 문제가 발생하거나 업데이트가 필요한 경우 해당 메서드만 수정하면 된다. 이렇게 하면 전체 코드 베이스에 영향을 주지 않고 변경 사항을 적용할 수 있다.
- **재사용성과 확장성**: 잘 설계된 메서드는 다른 프로그램이나 프로젝트에서도 재사용할 수 있으며, 새로운 기능을 추가하거나 기존 기능을 확장하는 데 유용하다.
- **추상화**: 메서드를 사용하는 곳에서는 메서드의 구현을 몰라도 된다. 프로그램의 다른 부분에서는 복잡한 내부 작업에 대해 알 필요 없이 메서드를 사용할 수 있다.
- **테스트와 디버깅 용이성**: 개별 메서드는 독립적으로 테스트하고 디버그할 수 있다. 이는 코드의 문제를 신속하게 찾고 수정하는 데 도움이 된다.

따라서, 메서드는 효율적이고 유지 보수가 가능한 코드를 작성하는 데 매우 중요한 도구이다.

# 10. 다음으로

#1.인강/0.자바/1.자바-입문

## 학습 내용 정리

### 전체 목차

## 1. Hello World

- /개발 환경 설정
- /다운로드 소스 코드 실행 방법
- /자바 프로그램 실행
- /주석(comment)
- /자바란?

## 2. 변수

- /변수 시작
- /변수 값 변경
- /변수 선언과 초기화
- /변수 타입1
- /변수 타입2
- /변수 명명 규칙
- /문제와 풀이
- /정리

## 3. 연산자

- /산술 연산자
- /문자열 더하기
- /연산자 우선순위
- /증감 연산자
- /비교 연산자
- /논리 연산자

- /대입 연산자
- /문제와 풀이
- /정리

## 4. 조건문

- /if문1 - if, else
- /if문2 - else if
- /if문3 - if문과 else if문
- /switch문
- /삼항 연산자
- /문제와 풀이1
- /문제와 풀이2
- /정리

## 5. 반복문

- /반복문 시작
- /while문1
- /while문2
- /do-while문
- /break, continue
- /for문1
- /for문2
- /중첩 반복문
- /문제와 풀이1
- /문제와 풀이2
- /정리

## 6. 스코프, 형변환

- /스코프1 - 지역 변수와 스코프
- /스코프2 - 스코프 존재 이유
- /형변환1 - 자동 형변환
- /형변환2 - 명시적 형변환

- /계산과 형변환
- /정리

## 7. 훈련

- /Scanner 학습
- /Scanner - 기본 예제
- /Scanner - 반복 예제
- /문제와 풀이1
- /문제와 풀이2
- /문제와 풀이3
- /문제와 풀이4
- /정리

## 8. 배열

- /배열 시작
- /배열의 선언과 생성
- /배열 사용
- /배열 리펙토링
- /2차원 배열 - 시작
- /2차원 배열 - 리펙토링1
- /2차원 배열 - 리펙토링2
- /향상된 for문
- /문제와 풀이1
- /문제와 풀이2
- /문제와 풀이3
- /정리

## 9. 메서드

- /메서드 시작
- /메서드 사용
- /메서드 정의
- /반환 타입

- /메서드 호출과 값 전달1
- /메서드 호출과 값 전달2
- /메서드와 형변환
- /메서드 오버로딩
- /문제와 풀이1
- /문제와 풀이2
- /정리

## 로드맵 소개

### 실전 자바 로드맵

- 김영한의 자바 입문 - 코드로 시작하는 자바 첫걸음 (오픈)
- 김영한의 실전 자바 - 기본편 (오픈)
- 김영한의 실전 자바 - 중급편 (2024년 초 예정)
- 김영한의 실전 자바 - 고급편 (2024년 초 예정)

### 실전 데이터베이스 로드맵(2024년 중순 예정)

### 백엔드 개발자 로드맵 소개



#### 김영한 백엔드 개발자 자바 스프링 JPA 실무 로드맵

백엔드 개발자 로드맵 소개 영상 링크: <https://youtu.be/ZgtvcyH58ys>

#### 스프링 완전 정복 로드맵

- 스프링을 완전히 마스터 할 수 있는 로드맵

- URL: <https://www.inflearn.com/roadmaps/373>

### 스프링 부트와 JPA 실무 완전 정복 로드맵

- 최신 실무 기술로 웹 애플리케이션을 만들어보면서 학습
- URL: <https://www.inflearn.com/roadmaps/149>

## 하고 싶은 이야기

### 개발자?

세상 어떤 직업보다 개발만큼 학벌보다 실력과 노력으로 잘 될 수 있는 곳은 드물다 생각한다.

적성에 맞는다면 내가 하고 싶은 개발을 하면서 돈도 벌 수 있다.

### 적성?

특정 분야에서 몰입할 수 있는 능력

- 그 분야의 일을 할 때 크게 피곤하지 않음
- 그 분야의 일을 할 때 자리에 오래 앉아 있음

### 개발자에게 가장 중요한 것은 꾸준함

- 머리는 타고 나는 것
- 개발자는 꾸준한 노력으로 극복할 수 있는 분야
  - 프로그래머는 프로 선수처럼 천재 + 노력의 소수의 사람만 직업을 구하는 분야가 아니다.
  - 취업, 아직 시장에 프로그래머는 항상 부족한 상황 (물론 어느정도 기준은 넘어야 한다.)
- 적성에 맞아야 꾸준한 노력이 가능
  - 새로운 기술이 계속 나옴, 좋은 개발자가 되려면 실무에서도 꾸준하게 공부해야 함

### 내가 개발자가 적성이 맞을까?

코딩의 80%는 변수, 연산자, 조건문, 반복문, 배열(자료구조), 메서드를 사용하는 것

- 입문편 강의가 잘 이해가 되고 재미있다. → 개발자로 성공 가능성 높음
- 입문편 강의가 어렵지만 복습하니 이해된다. → 아직 컴퓨터가 익숙하지 않은 것, 개발자로 성공 가능성 있음
- 복습해도 모르겠다. → 한번 정도 처음부터 복습, 익숙하지 않아서 그럴 수 있음
  - 복습해보고 그래도 이해가 안되면 다른 분야로 가자! 포기하는 것도 용기다!
  - 내 적성이 다른 곳에 있을 수 있다.

### 어떤 분야의 개발자를 하는게 좋을까?

- 앱, 프론트엔드, 백엔드, 기타(인프라, 데이터 과학자 등등)

- 미래는 아무도 모른다.
- 결국 본인이 가장 즐거운 분야를 선택하는 것이 가장 길게 갈 수 있다.

## 하고 싶은 이야기 정리 링크

### 개발 인생 전반의 이야기

#### EO 인터뷰 영상

- 한국 개발자 최고 1타강사 김영한의 인생 [1부]: [https://youtu.be/\\_HTj5b59Em0](https://youtu.be/_HTj5b59Em0)
- 한국 개발자 최고 1타강사 김영한의 인생 [2부]: <https://youtu.be/MNyNRraMU8Y>

#### 개발바닥 - 시골 청년 개발왕 되다

- 1편: <https://youtu.be/Pb69UQ6f8n0>
- 2편: <https://youtu.be/b4QP5RsuJts>
- 3편: <https://youtu.be/l0h1pQ96u2g>

### 취업과 이직에 대한 고민

#### 인프콘 - 어느 날 고민 많은 주니어 개발자가 찾아왔다, 성장과 취업, 이직 이야기

- <https://youtu.be/QHlyr8soUDM>

### 인프런 최초 20만 명 달성 기념 QA

- <https://youtu.be/psXdWq008DA>