

Day 1

1장 Hello, 자바!

안녕하세요! <코딩 자율학습 자바 입문>을 시작하는 여러분 반갑습니다. 이 책을 공부하는데 조금 더 도움을 드리고자 학습 가이드를 준비했습니다. 대부분은 책을 보시면 되고 추가로 볼 만한 내용을 곁들였으니 가볍게 읽고 참고해 주세요. 그밖에 궁금한 점은 [코딩 자율학습단 카페](https://cafe.naver.com/gilbutitbook)(https://cafe.naver.com/gilbutitbook)의 질문 게시판을 이용해 주세요.

첫날에는 자바 개발 환경을 설정해 보고 자바 프로젝트를 생성해 보겠습니다! 오늘 공부할 내용은 다음과 같습니다.



공부할 내용(19~38p)

- 자바 개발 환경 설정
- 첫 번째 자바 프로젝트

1. 자바 개발 환경 설정

책에서는 윈도우 환경에서 자바 개발 환경 설정합니다. 그래서 맥 사용자를 위해 macOS 기반으로 자바 개발 환경을 설정하는 방법을 소개하겠습니다.

JDK 설치하기

macOS에서는 Homebrew로 설치하는 방법이 가장 간단하고 가장 권장됩니다. 컴퓨터에 Homebrew가 설치되어 있지 않다면 Homebrew를 먼저 설치해 주세요. 이미 설치되어 있다면 3번 과정으로 넘어갑니다.

1. 맥의 기본 터미널을 열고 다음 명령어를 입력합니다.

```
Unset
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2. 설치가 끝나면 다음 명령어를 입력합니다. **USER_ID**는 각자 **PC**에 맞게 입력해 주세요.

```
Unset
echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >>
/Users/<USER_ID>/.zprofile
eval "$(/opt/homebrew/bin/brew shellenv)"
```

3. **homebrew**가 설치가 끝나면 **JDK**를 설치합니다.

```
Unset
brew install openjdk@21
```

JDK는 **LTS**와 **non-LTS** 버전이 있습니다. **LTS**는 **Long Term Support**로 오랫동안 지원되는 버전을 말합니다. **8, 11, 17, 21**버전이 **LTS** 버전이고 가장 최신 버전 **LTS**는 **21**입니다. 이 책에서는 **21** 버전을 사용합니다. 만약 다른 버전의 **JDK**가 필요하다면 **brew search openjdk** 명령어로 설치 가능한 버전 목록을 확인한 후 **brew install openjdk@<버전>** 명령어로 설치할 수 있습니다.

4. 설치된 **JDK** 버전을 확인하려면 다음과 같이 입력하세요:

```
Unset
java -version
```

멘토 Tip

Homebrew는 **macOS**와 **Linux** 시스템에서 사용되는 패키지 관리자입니다. **Apple**이나 기본 **Linux** 시스템에서 제공하지 않는 다양한 소프트웨어 패키지를 손쉽게 설치, 관리, 업데이트할 수 있도록 도와줍니다.

Homebrew는 패키지를 전용 디렉토리에 설치하고, **/opt/homebrew** 위치로 심볼릭 링크(파일 시스템에서 특정 파일이나 디렉토리의 경로에 대한 참조를 저장하는 파일)를 연결해 시스템의 다른 부분과 통합합니다. 심볼릭 링크는 파일 시스템에서 특정 파일이나 디렉토리의 경로에 대한 참조를 저장하는 파일입니다. 심볼릭 링크를 생성하면 원본 파일이 있는 위치와 관계없이 해당 링크를 통해 원하는 파일에 접근할 수 있습니다.

또한, **Homebrew**는 사용자가 직접 패키지를 만들거나 수정할 수 있는 유연성을 제공합니다. 이를 통해 사용자는 필요에 따라 자신만의 패키지를 생성하거나 기존 패키지를 커스터마이징할 수 있습니다.

Homebrew는 **macOS**의 개발 환경을 보완해주며, **gem** 명령으로 **RubyGems**를, **brew** 명령으로 **RubyGems**의 의존성 모듈을 설치할 수 있습니다. 또한, **Homebrew Cask** 명령을 통해 **macOS** 앱, 폰트, 플러그인, 오픈소스가 아닌 소프트웨어를 설치할 수 있습니다.



인텔리제이 설치하기

인텔리제이도 **homebrew**로 간편하게 설치할 수 있습니다.

```
Unset
# IntelliJ IDEA Community Edition(무료 버전)
brew install --cask intellij-idea-ce

# IntelliJ IDEA Ultimate Edition (유료 버전)
brew install --cask intellij-idea
```

설치가 완료되면 애플리케이션 폴더에 **IntelliJ IDEA** 아이콘이 추가됩니다. 이 아이콘을 클릭하면 여 인텔리제이를 실행할 수 있습니다.

2. 첫 번째 자바 프로젝트

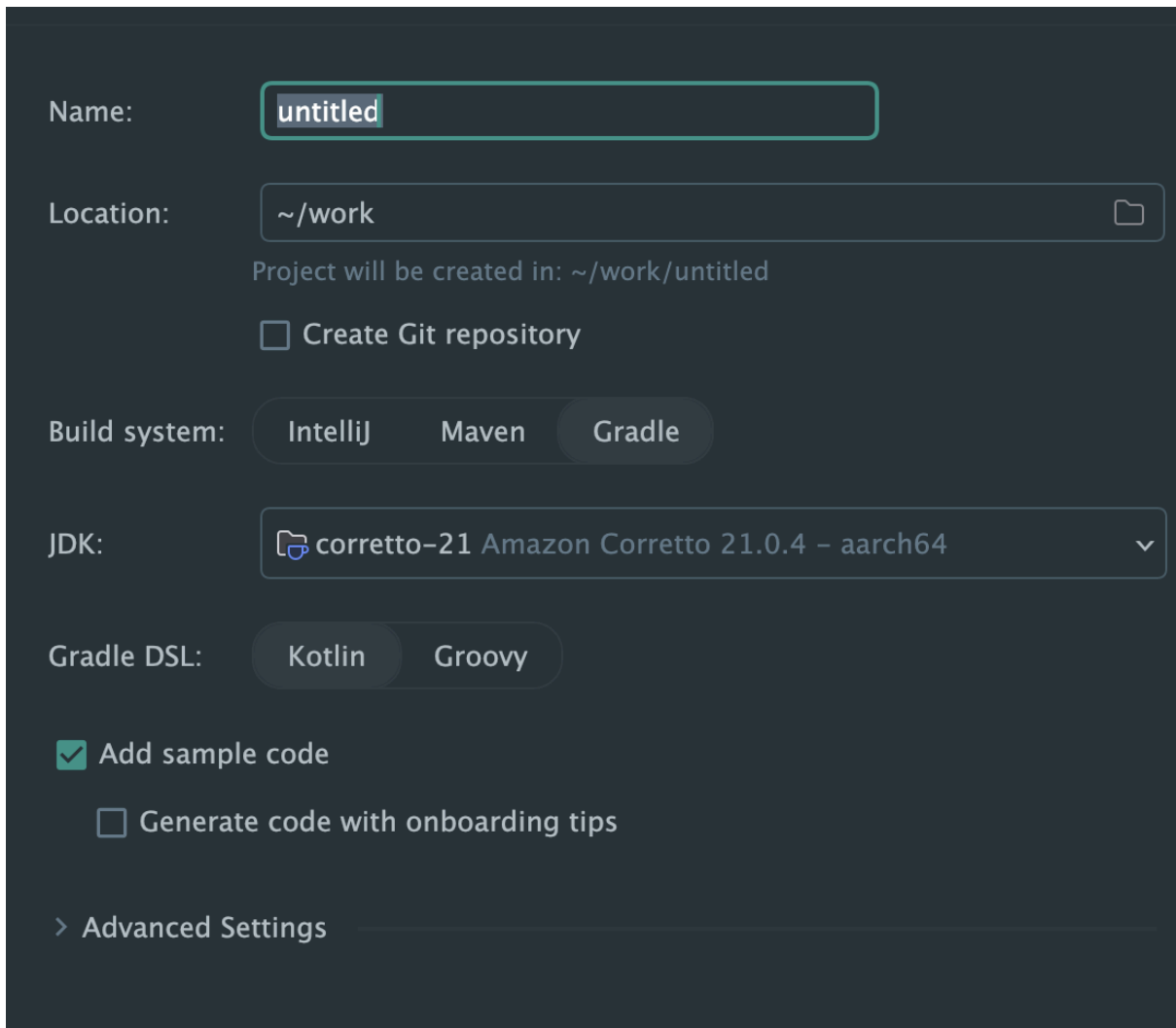
1. 인텔리제이 실행하고 시작 화면에서 **New Project** 버튼을 클릭합니다. **IntelliJ IDEA**가 이미 열려 있다면, 메뉴에서 **File > New > Project...**를 선택합니다.

2. 왼쪽 패널에서 **Java**를 선택합니다.

3. 프로젝트를 설정할 수 있는 화면이 나옵니다.


- **Name** 필드에 프로젝트 이름을 입력합니다.
- **Location** 필드에 프로젝트를 저장할 경로를 지정합니다. 기본값을 그대로 사용해도 됩니다.
- **Build System** 필드는 책처럼 **IntelliJ**로 선택합니다.

- JDK 필드는 앞에서 설치한 21 버전을 선택합니다. JDK가 설정되어 있지 않다면, "Add JDK..."를 클릭하여 설치한 JDK 경로(/Library/Java/JavaVirtualMachines/)를 추가합니다.





The image shows the 'New Project' dialog in IntelliJ IDEA. The 'Name' field is 'untitled'. The 'Location' is '~/work', with a note 'Project will be created in: ~/work/untitled'. The 'Build system' is set to 'Gradle'. The 'JDK' is set to 'corretto-21 Amazon Corretto 21.0.4 - aarch64'. The 'Gradle DSL' is set to 'Kotlin'. The 'Add sample code' checkbox is checked, and 'Generate code with onboarding tips' is unchecked. There is an 'Advanced Settings' link at the bottom.

Name:

Location: 
Project will be created in: ~/work/untitled

☐ Create Git repository

Build system: IntelliJ Maven Gradle

JDK:  corretto-21 Amazon Corretto 21.0.4 - aarch64 

Gradle DSL: Kotlin Groovy

☒ Add sample code

☐ Generate code with onboarding tips

[> Advanced Settings](#)

3. 프로젝트를 생성하면 다음과 같은 코드가 작성되어 있습니다.



The image shows the 'Main.java' file in the IntelliJ IDEA editor. The code is as follows:

```
1 public class Main {  
2     public static void main(String[] args) { System.out.println("Hello world!"); }  
5 }
```

```
Main.java x
1  > public class Main {
2  >     public static void main(String[] args) {
3      System.out.println("Hello world!");
4  }
5  }
```

자바는 모든 코드가 클래스 안에 작성됩니다.

- 클래스는 자바에서 프로그램을 구성하는 기본 단위이자 인스턴스를 생성하는 '틀'과 같습니다.
- 메서드는 클래스에 종속된 함수입니다. `main()`은 JVM에서 가장 먼저 실행하는 메서드입니다.
- 자바는 명령어 끝에 항상 세미콜론을 넣습니다.

멘토 Tip

맥에서 해당 함수가 어떤 역할을 하는지 보려면 단축키 **command + (click)** 또는 **command + b**를 누릅니다. 해당 함수의 구현부를 확인할 수 있습니다.



Day 2

2장 자료형과 변수

안녕하세요! 오늘은 2장 <입출력 다루기>를 살펴봅니다.

공부할 내용(39~58p)

- 출력하기
- 입력받기

1. 출력하기

자바에서 출력은 대표적으로 다음 3개 메서드를 사용합니다.

- `System.out.printf()`
- `System.out.println()`
- `System.out.print()`

다른 언어를 공부해봤다면 출력 명령어가 친숙할 것 같습니다.

- `printf()`는 `print + format`의 의미로, 출력할 문자열의 서식을 지정할 수 있습니다.
- `println()`은 `print + line`의 의미로, 출력한 후 한 줄을 띄웁니다.
- `print()`는 출력만을 담당합니다.

멘토 Tip

`System`에서도 얻어갈 것이 있습니다. 우선 객체를 생성하지 않았는데 메서드 호출이 된다는 점입니다. `System` 클래스는 `static` 클래스로, 객체 생성 없이 사용이 가능합니다. 그 중 `out`이라는 `public static` 변수에 접근해서, 해당 변수의 `println()`, `printf()` 등의 메서드를 호출하는 것입니다.

IntelliJ에는 자동완성 기능이 있습니다. 그래서 `sout` 을 입력하면 자동으로 `System.out.println()` 작성해줍니다. 다만, 자바가 처음이라면 `System.out`을 직접 입력하며 연습해 보기 바랍니다.



2. 입력받기

자바에서 입력을 받는 방법은 몇 가지가 있습니다. 우선 책에서 다루는 두 가지를 먼저 살펴봅시다.

System.in.read()

`read()` 는 사용자가 입력한 값을 읽어오는데, 이를 **ASCII 코드**(ASCII가 무엇인지 모른다면 아래 링크를 참고해 주세요.)로 읽어옵니다. 그래서 `1`을 입력하더라도 `49`가 출력되는 것이죠. 또한 하나만 입력받을 수 있는 것도 문제입니다.

📖 함께 보면 좋은 자료

- <https://ko.wikipedia.org/wiki/ASCII>

그래서 이에 대안으로 **Scanner**가 존재합니다.

🥕 멘토 Tip

mac에서는 import하는 단축키는 `option + enter` 입니다.



Scanner

Scanner를 설명하기 이전에 **Stream**에 대해 알아야 합니다. **Stream**은 단어 자체로만 보면 물줄기 인데요, 단방향으로 흐른다는 것이 특징입니다. 그래서 입/출력 역시 단 방향으로 흐르는 특성을 가지고 있습니다. 또한 **Stream**은 한번 읽으면 다시 읽지 못하니 주의해서 사용해야 합니다.

Java

```
Scanner sc = new Scanner(System.in);
```

Scanner가 지원하는 메서드는 많지만 여기서는 `nextLine()`을 배워봅시다.

Java

```
Scanner sc = new Scanner(System.in);  
String line = sc.nextLine();  
System.out.println(line);
```

위 코드를 보면 `nextLine()`의 반환 타입은 `String`인 것을 알 수 있습니다. `nextLine()`으로 읽은 그 값은 문자열로 취급이 된다는 것이죠.

그리고 `Scanner`를 사용했을 때는 반드시 닫아줘야 합니다.

Java

```
Scanner sc = new Scanner(System.in);  
String line = sc.nextLine();  
System.out.println(line);  
sc.close();
```

이는 자바의 권장사항으로 자원을 사용했을 경우 반드시 닫아줘야 합니다. 자세한 내용은 다음 참고해 주세요.

 함께 보면 좋은 자료

java resource 해제

- <https://www.oracle.com/technical-resources/articles/java/trywithresources.html>

1분 퀴즈풀이

46p 1번

- `System.out.print("Hello")`는 줄을 바꾸지 않습니다. 줄을 바꾸는 메서드는 `println()`입니다. -> X
- `println`을 사용했기 때문에 "Java"를 출력한 후 줄을 바꿉니다. -> O
- `\n`은 줄 바꿈을 의미하는 특수 문자가 맞지만, `println()` 자체가 이미 줄바꿈을 포함하고 있습니다. -> X

- 42p System 클래스에 대한 설명입니다. -> O

55p. 2번

Java

```
public class Main {  
    public static void main(String[] args) throws IOException{  
        System.out.print((char)System.in.read()); // 1이  
        들어갑니다.  
        System.out.print((char)System.in.read()); // 2가  
        들어갑니다.  
        System.out.print((char)System.in.read()); // 3이  
        들어갑니다.  
        // print() 메서드이기 때문에 줄 바꿈 없이 123이 출력됩니다.  
    }  
}
```

58p. 셀프체크 풀이

Java

```
import java.util.Scanner;  
  
public class Main {  
    public static void main(String[] args) {  
        // scanner로 입력을 받습니다.  
        Scanner scanner = new Scanner(System.in);  
        // nextLine()으로 문자열을 입력받습니다.  
        System.out.println(scanner.nextLine());  
        System.out.println(scanner.nextLine());  
        System.out.println(scanner.nextLine());  
    }  
}
```

Day 3

3장 기초 문법 배우기

안녕하세요! 오늘은 본격적으로 자바 문법을 배워봅니다.

공부할 내용(59~76p)

- 변수
- 자료형
- 상수와 리터럴

1. 변수

모든 언어에서 변수 선언 방식은 매우 중요합니다. 변수 선언은 메모리 공간에 데이터를 저장하는 것입니다. 그리고 자바는 다음 형태로 변수를 선언합니다.

```
Java  
자료형 변수명;
```

멘토 Tip

Java 10버전 이후부터는 **var**가 추가되어 다음과 같은 형태로 변수 선언도 가능해졌습니다.

```
Java  
var a = 10;
```

그러나 이 방법은 반드시 값을 할당해야 합니다. 그리고 타입 추론의 한계도 있어서 우선 자료형을 명시하는 형태로 익히기 바랍니다.



변수를 초기화하려면 값을 할당하면 됩니다.

Java

```
int a = 10;
```

멘토 Tip

자바에서는 변수명을 **CamelCase**로, 클래스명을 **PascalCase**로 작성합니다.



2. 자료형

자료형은 변수가 어떤 자료를 저장하는 변수인지를 나타냅니다. 자료형에는 기본 자료형과 참조 자료형 이 있습니다. 기본 자료형에는 다음과 같은 것들이 있습니다.

- 정수형
 - byte
 - short
 - int
 - long
- 실수형
 - float
 - double
- 문자형
 - char
- 논리형
 - boolean

그리고 참조 자료형은 배열, 열거형, **class** 들이 있습니다. 그래서 자바의 **String**은 기본 자료형이 아니라 참조 자료형입니다.

3. 상수와 리터럴

많은 언어에서 상수의 경우 **const** 라는 예약어를 통해서 상수를 명시하고 있지만 자바는 **const**가 없습니다. 하지만 상수를 구현할 수 있는데요, 상수는 저장한 값이 바뀌지 않는 변수를 말하고, 이는 자바에서 **final**로 구현할 수 있습니다.

Java

```
final String HELLO = "hello" // 변경 불가.
```

이렇게 **final**이 붙으면 값을 변경할 수 없습니다. 상수에 대한 설명은 구글 스타일 가이드 문서를 첨부합니다.

함께 보면 좋은 자료

- google style guide constant
- <https://google.github.io/styleguide/javaguide.html#s5.2.4-constant-names>

멘토 Tip

자바에는 컴파일 타임 상수라는 개념이 존재합니다.다음은 컴파일 타임 상수로 간주되기 위한 조건입니다.

- **final** 로 선언되어야 합니다.
- 기본 자료형 또는 **String**이어야 합니다.
- 초기화할 때 상수 표현식을 사용해야 합니다.
- 상수 표현식은 반드시 컴파일 시점에 평가될 수 있는 값이어야 하며, 다른 상수 값이나 리터럴 값으로 초기화해야 합니다.

자바 공식 문서를 첨부합니다.

함께 보면 좋은 자료

- java 상수 공식문서
- <https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.28>



리터럴은 코드에서 직접 표현된 고정된 값을 의미합니다. 상수와 리터럴의 관계는 상수가 되려면 반드시 리터럴 혹은 리터럴의 조합이어야 하지만 리터럴은 반드시 상수일 필요는 없습니다. 또한 리터럴은 반드시 변수에 할당될 필요는 없습니다.

1분 퀴즈풀이

66p. 1번

- 자바에서 변수명은 숫자로 시작할 수 없습니다. -> X
- 변수를 선언하면 즉시 메모리 공간이 할당 되지 않고, 초기화가 이루어져야합니다. -> X (62p. 참고)
- 변수에 값을 처음 저장하는 것을 초기화라고 합니다. -> O
- 초기화되지 않은 변수를 사용하려고 하면 오류가 발생할 수 있습니다. -> O
Nullpointer Exception과 같은 에러가 발생할 수 있습니다.
- 자바에서는 여러 변수를 한 번에 선언하고 초기화 할 수 있습니다. -> X (64p.)

75p. 2번

printf를 사용할 때 정수는 %d를 사용하고, 실수는 %f를 사용합니다. 이때 소수점 4째자리까지 출력하고 싶기 때문에 %.4f를 사용하면 됩니다.

Day 4

3장 기초 문법 배우기

안녕하세요! 오늘은 기초 문법 나머지 부분을 배워 봅니다.

공부할 내용(77~100p)

- 연산자
- 형변환
- 주석
- 코드 작성 규칙

1. 연산자

연산자에는 일반적으로 쓰이는 사칙연산자가 있고 추가적으로 모듈러(%) 연산자와 대입 연산자가 있습니다. 각 연산자를 어떻게 사용하고 결과는 어떻게 되는지 보여드리겠습니다.

Java

```
public class Main {  
    public static void main(String[] args) {  
        int a = 10; // 대입 연산자  
        int b = 5;  
        System.out.println(a + b); // 15  
        System.out.println(a - b); // 5  
        System.out.println(a * b); // 50  
        System.out.println(a / b); // 2  
        System.out.println(a % b); // 0  
    }  
}
```

+ 연산자로는 문자열을 연결할 수도 있습니다.

Java

```
String a = "hello"  
String b = "world"  
System.out.println(a + b) // helloworld
```

2. 형변환

형변환은 어떤 자료형의 값을 다른 자료형으로 바꾸는 것을 말합니다.

Java

```
int a = 100;  
float b = (float)a;  
System.out.println(b); //100.0
```

자바에는 명시적 형변환과 묵시적 형변환이 있습니다. 명시적 형변환은 명시적으로 A 타입인 변수를 B로 변환하는 코드를 입력해서 변환하는 것입니다.

Java

```
int num1 = 12;  
float num2 = 12.3F;  
int result = num1 + (int)num2;  
System.out.println(result);
```

묵시적 형변환은 자동으로 자료형이 바뀌는 것으로 자 형변환이라고도 합니다. 이때 묵시적 형변환은 범위가 작은 자료형에서 범위가 큰 자료형으로만 변환만 허용합니다.

Java

```
int a = 10;  
float b = 15.3F;  
System.out.println(a + b); // float으로 변환 됨
```

3. 주석

주석은 코드의 동작을 설명하거나, 추가 정보를 제공하고, 특정 부분에 대한 이해를 돕기 위해 작성된 메모입니다. 이 책에서는 다양한 주석을 소개합니다.

먼저 블록 주석입니다. 블록 주석은 `/* */` 기호를 사용해 범위에 안에 든 내용을 주석으로 처리합니다.

```
Java
/*
This is block comment
*/
```

멘토 Tip

자바에는 **JavaDoc**이라는 특별한 주석이 있습니다. **JavaDoc**은 자바에서 코드에 대한 문서를 생성하기 위해 사용되는 주석 스타일로, 주로 클래스, 메서드, 필드 등에 대한 설명을 작성하여 자동으로 **API** 문서를 생성할 수 있도록 해줍니다. **IntelliJ**에서는 **JavaDoc**을 사용하고 싶은 곳 위에 `/**` 을 입력하고 엔터키를 눌러주시면 **JavaDoc**이 생성됩니다.

```
Java
/**
 * * @param data
 * * @return
 */
public String solution(String data) {

}
```

이렇게 하면 **data**에 대한 추가적인 설명과 **return** 값에 대한 설명을 작성할 수 있습니다.



한 줄 주석은 블록주석과 마찬가지로 `/* */` 기호를 사용한 주석이고, 한 줄에만 작성합니다.

꼬리 주석은 코드 뒤에 아주 짧은 주석이 필요할 때 씁니다. 줄 끝 주석은 `//` 를 사용해 작성합니다. 한 줄 전체를 주석 처리하거나 일부분을 주석 처리해야 할 때 사용합니다.

4. 코드 작성 규칙

코드 작성 규칙은 혼자 코드를 작성할 때보다 여러명과 협업할 때 중요합니다. 각기 다른 코드 작성 규칙을 가지고 있다면 A에게는 10번째 라인이 B에게는 12번 라인일 수 있기 때문이죠.

멘토 Tip

대표적인 코드 작성 규칙으로 널리 사용되고 있는 구글 스타일 가이드를 소개하려고 합니다. 구글 스타일 가이드는 **import**하는 순서부터 변수명을 작성하는 규칙, 라인 수 등 다양한 코드 작성에 대한 컨벤션을 제시합니다. 시간을 내서서 읽어볼 가치가 충분한 문서입니다.

함께 보면 좋은 자료

- google style guide
- <https://google.github.io/styleguide/javaguide.html>



1분 퀴즈풀이

84p. 3번

"456" + 7을 하면 7을 문자열로 인식해 4567이 출력됩니다.

90p. 4번

정수 값과 실수 값을 덧셈할 때는 자료형을 명시해야하기 때문에 명시적 형변환해야 하고 자동으로 실수값으로 변환되는 것을 묵시적 형변환이라고 합니다.

90p. 5번

Java

```
public class Main {  
    public static void main(String[] args) {  
        int a = 1;  
        char b = '1'; '1'은 ASCII 코드값으로 49입니다.  
        int c = a + b; // 49 + 1 = 50  
    }  
}
```

```
        System.out.println(c); // 50 출력
    }
}
```

100p. 셀프체크 풀이

Java

```
import java.util.Scanner;
```

```
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("정수를 입력하세요. --> ");
        int a = scanner.nextInt();
        System.out.print("실수를 입력하세요. --> ");

        double b = scanner.nextDouble();
        int sum = a + (int) b;
        double diff = (double) a - b;
        double product = (double) a * b;
        int quotient = a / (int) b;

        System.out.println("덧셈 결과(정수): " + sum);
        System.out.println("뺄셈 결과(실수): " + diff);
        System.out.println("곱셈 결과(실수): " + product);
        System.out.println("나눗셈 결과(정수): " + quotient);
        scanner.close();
    }
}
```

Day 5

4장 조건에 따라 흐름 바꾸기: 조건문

안녕하세요! 오늘부터 이틀에 걸쳐 조건문에 대해 공부해 봅니다.

공부할 내용(101~112p)

- 조건이 하나일 때: **if-else**
- 조건이 여러 개일때: **else if**

1. 조건이 하나일 때

우선 조건문에 대해서 알아보시다. 조건문은 조건이 참일 때 실행하는 문입니다. 자바에서 조건에 들어갈 수 있는 건 **boolean** 타입으로 평가되는 값입니다. 다음 내용이 모두 가능합니다.

- **true, false**
- **==, !=, <, >, >=, <=**
- 논리 연산자
- 메서드 호출 결과(**boolean**을 반환하는 경우)

Java

```
public static void main(String[] args) {  
    int score = 90;  
    if (score >= 90) {  
        System.out.println("A 학점입니다.");  
    }  
}
```

위는 조건을 만족하는 경우고, 조건을 만족하지 않는 경우도 있을 것입니다. 조건을 만족하지 않는 경우 다른 문을 실행하고 싶다면 이때 **else**를 사용합니다.

Java

```
public static void main(String[] args) {  
    int score = 80;  
    if (score >= 90) {  
        System.out.println("A학점입니다.");  
    } else {  
        System.out.println("A학점이 아닙니다.");  
    }  
}
```

2. 조건이 여러 개일 때: else if

if-else 문은 조건이 하나고, 이를 만족하는 경우와 아닌 경우로 나뉩니다. 조건을 여러 개 설정하고 싶다면 **if-else**가 아닌 **if-else if-else**를 사용할 수 있습니다.

Java

```
public static void main(String[] args) {  
    int score = 80;  
    if (score >= 90) {  
        System.out.println("A학점입니다.");  
    } else if (score >= 80 && score < 90) {  
        System.out.println("B학점입니다.");  
    } else if (score >= 70 && score < 80) {  
        System.out.println("C학점입니다.");  
    } else {  
        System.out.println("D학점입니다.");  
    }  
}
```

멘토 Tip

앞의 코드는 완벽하게 최적화된 코드는 아닙니다. 점수가 75점이라고 생각해봅시다. 첫 번째 **if**문에서 90점 미만이기 때문에 다음 **else-if**문으로 내려갑니다. 여기서 **score**를 80점 이상이면서 90점 미만을 것을 확인하는데, 첫 번째 **if**문에서 이미 90점 미만인 것이 확인되었기 때문에 **score < 90**은 불필요한 코드입니다. 마찬가지로 **score < 80** 도 같은 이유로 필요 없습니다.

수정하면 다음과 같습니다.

Java

```
public static void main(String[] args) {
    int score = 80;
    if (score >= 90) {
        System.out.println("A학점입니다.");
    } else if (score >= 80) {
        System.out.println("B학점입니다.");
    } else if (score >= 70) {
        System.out.println("C학점입니다.");
    } else {
        System.out.println("D학점입니다.");
    }
}
```



3. 비교 연산자와 논리 연산자

비교 연산자는 두 값을 비교하는 연산자입니다. 수학에서는 $a < b < c$ 와 같은 연산이 가능하지만, 자바에서는 불가능합니다. 자바에서는 위와 같은 연산을 하려면 논리 연산자를 활용해야 합니다.

$a < b < c$ 는 b 가 a 보다 크면서 c 보다 작다는 것을 의미합니다. 이를 논리 연산자인 `&&`로 묶어서 표현해야 합니다.

Java

```
if (a < b && b < c) {

}
```

논리 연산자에 설명을 더하자면 `&&` 연산자는 모두 참이어야 하기에, `false` 인 것이 나오지 않으면 `true` 입니다. `||` 연산자는 하나만 참이면 되기 때문에, `true`가 나오기만 하면 `true` 입니다.

- &&: false가 나올 때까지 찾는다.
- ||: true가 나올 때까지 찾는다.

1분 퀴즈풀이

107p. 1번

60점을 초과하면 합격, 60점 이하면 불합격이므로 if-else를 사용하면 됩니다.

Java

```
public class Main {  
    public static void main(String[] args) {  
        int score = 60;  
        if (score > 60) {  
            System.out.println("합격입니다.");  
        } else {  
            System.out.println("불합격입니다.");  
        }  
    }  
}
```

107p. 2번

짝수와 홀수를 판별하는 문제입니다. 짝수의 경우 2로 나누어 떨어지면 짝수이고, 그렇지 않으면 홀수입니다. 풀이는 다양하지만, 해설과 다르게 풀이해 보겠습니다.

Java

```
public class Main {  
    public static void main(String[] args) {  
        Scanner scan = new Scanner(System.in);  
        int num = scan.nextInt();  
        if (num % 2 != 0) { // if (num % 2 == 1) 도 가능  
            System.out.println("홀수");  
        } else {  
            System.out.println("짝수");  
        }  
    }  
}
```

111p. 3번

멘토 팁에 나옴 방법입니다.

Java

```
public static void main(String[] args) {  
    int score = 80;  
    if (score >= 90) {  
        System.out.println("A학점입니다.");  
    } else if (score >= 80) {  
        System.out.println("B학점입니다.");  
    } else if (score >= 70) {  
        System.out.println("C학점입니다.");  
    } else {  
        System.out.println("D학점입니다.");  
    }  
}
```

Day 6

4장 조건에 따라 흐름 바꾸기: 조건문

안녕하세요! 오늘도 어제에 이어 4장 조건문을 공부해 봅시다.

공부할 내용(113~126p)

- 조건문 안에 조건문이 있을 때: 중첩 조건
- 조건과 일치하는 값 찾기: switch
- 삼항 연산자

1. 조건문 안에 조건문이 있을 때: 중첩 조건문

if-else문 안에 if-else문이 들어 있는 형태를 중첩 조건문이라고 합니다. 중첩 조건문은 앞에서 배운 조건문과 크게 다르지 않습니다.

이전 장에 나왔던 문제를 한 번 응용해보겠습니다. 점수가 90점이 넘는 학생에 대해서 95점이 넘으면 A+라고하고, 95점 미만인 경우 A, 85점을 넘으면 B+, 80점을 넘으면 B라고 하는 코드를 작성해보겠습니다.

Java

```
public static void main(String[] args) {  
    int score = 80;  
    if (score >= 90) {  
        if (score >= 95) {  
            System.out.println("A+학점입니다.");  
        } else {  
            System.out.println("A학점입니다.");  
        }  
    } else if (score >= 80) {  
        if (score >= 85) {  
            System.out.println("B+학점입니다.");  
        }  
    }  
}
```



```

        } else {
            System.out.println("B학점입니다.");
        }
    } else if (score >= 70) {
        if (score >= 75) {
            System.out.println("C+학점입니다.");
        } else {
            System.out.println("C학점입니다.");
        }
    } else {
        System.out.println("D학점입니다.");
    }
}

```

이렇게 중첩문 안에 중첩을 할 수 있습니다.

멘토 Tip

중첩 조건문이 하나 생길 때 마다 이를 "깊이가 깊어진다"라고 합니다. 중첩 조건문은 깊이가 깊어질 수록 가독성이 좋지 않은 문제가 있습니다. 그렇기 때문에 최대한 깊이를 깊게 하지 않는게 중요합니다. 최대한 같은 깊이에서 조건문을 처리할 수 있게 하는 것이 좋습니다.



2. 조건과 일치하는 값 찾기 **switch**

조건문에는 **if-else**문 말고도 **switch** 문이 있습니다. **switch**문 역시 조건에 따라 명령을 실행할 수 있습니다. **switch** 문을 사용할 때 주의해야할 점은 **break**문을 반드시 해야한다는 것입니다. 그렇지 않으면 다음 **case**도 실행하게 됩니다.

switch 문은 **if**문과 다르게 **switch** 명시한 자료형만 **case** 문에 넣을 수 있습니다. 그래서 다음과 같은 코드 작성이 불가능합니다.

Java

```
public static void main(String[] args) {
    System.out.println("양수를 입력하세요.");
    Scanner scanner = new Scanner(System.in);
    int num = scanner.nextInt();
    switch (num) {
        case (num > 90):
            System.out.println("A 학점 입니다.");
    }
}
```

멘토 Tip

Java 14 버전에는 정식으로 향상된 **switch** 문이 추가되었습니다.

- 람다 스타일 문법을 지원합니다.
- **switch** 를 표현식으로 사용할 수 있습니다.

Java

```
int score = 85;
String grade = switch (score) {
    case int s when s >= 90 -> "A";
    case int s when s >= 80 -> {
        System.out.println("Good job!");
        yield "B";
    }
    case int s when s >= 70 -> "C";
    case int s when s >= 60 -> "D";
    default -> "F";
};
System.out.println(grade); // "Good job!" "B"
```

- **yield** 키워드가 추가되었습니다. **yield**는 **return**과 비슷한 역할을 합니다.

Java

```
int score = 85;
String grade = switch (score) {
```

```

case int s when s >= 90 -> "A";
case int s when s >= 80 -> {
    System.out.println("Good job!");
    yield "B";
}
case int s when s >= 70 -> "C";
case int s when s >= 60 -> "D";
default -> "F";
};
System.out.println(grade); // "Good job!" "B"

```

- 여러 **case**에 대해 중복으로 처리할 수 있습니다.

```

Java
String type = "apple";
int calories = switch (type) {
    case "apple", "banana" -> 100;
    case "orange", "grape" -> 80;
    default -> 0;
};
System.out.println(calories); // 100

```



3. 삼항 연산자

삼항 연산자는 **Ternary operator**라고도 하며, 조건을 한 줄로 작성할 수 있게 해줍니다. 덕분에 다음과 같은 코드가 가능합니다. 하지만 책에도 나와있듯, 삼항 연산자를 무분별하게 사용하면 코드가 읽기 어려워집니다.

```

Java
public class Main {
    public static void main(String[] args) {
        int a = 1;
    }
}

```

```

        System.out.println(a != 1? a : 10);
    }
}

```

1분 퀴즈풀이

115p. 4번

양수라면 홀수 인지 짝수인지 확인하고, 양수가 아니면 양수를 입력하는 코드를 작성하려고 합니다.

```

Java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        System.out.println("양수를 입력하세요.");
        Scanner scanner = new Scanner(System.in);
        int num = scanner.nextInt();
        if (num > 0) {
            if (num % 2 != 0) { // 혹은 if (num % 2 == 1)
                System.out.println("홀수");
            } else {
                System.out.println("짝수");
            }
        } else {
            System.out.println("양수가 아닙니다. 양수를 입력하세요.");
        }
    }
}

```

119p. 5번

input이 1이기 때문에 case(1)에서 걸리게 되고 print() 메서드에 의해 1이 출력됩니다. 그러나 break가 없기 때문에 2와 3도 출력되기 때문에 결과적으로 "123"이 출력됩니다.

122p. 6번

짝수이면 "짝수", 홀수이면 "홀수"를 grade에 할당하는 삼항연산자 문제입니다.

Java

```
String grade = (num % 2 == 0) ? "짝수" : "홀수";
```

126p. 셀프체크 풀이

다음 코드는 향상된 Switch 문으로 작성했습니다.

Java

```
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        String a = scanner.next();
        switch (a) {
            case "a" -> System.out.println("A");
            case "b" -> System.out.println("B");
            case "c" -> System.out.println("C");
            default -> System.out.println("일치하는 알파벳이 없습니다.");
        }
        scanner.close();
    }
}
```

Day 7

5장 같은 작업 반복하기: 반복문

안녕하세요! 오늘은 5장에 나온 반복문을 공부해 보겠습니다.

공부할 내용(127~147p)

- 범위 안에서 반복할 때: **for** 문
- 조건이 참일 동안 반복할 때: **while** 문

1. 범위 안에서 반복할 때: **for** 문

for 문은 반복하는 작업을 줄일 수 있는 강력한 기능입니다. 기본 형식은 다음과 같습니다.

Java

```
public static void main(String[] args) {  
    for(초깃값; 조건식; 증감식) {  
    }  
}
```

초깃값에 변수는 주로 **i**를 많이 사용하곤 합니다.

Java

```
for(int i = 0; i < 10; i++) {  
    System.out.println(i);  
}
```

조건식은 **for**문이 종료되는 조건식을 의미하고, 증감식을 통해 반복 횟수를 조절합니다.

멘토 Tip

for문에서 초깃값, 조건식, 증감식을 모두 생략해도 됩니다.

Java

```
for(;;) {  
    System.out.println("1");  
}
```

이렇게 하면 멈추지 않고 1을 출력하게 됩니다. 이렇게 멈추지 않는 상황을 무한루프에 빠졌다고 합니다. 혹은 조건식을 생략하거나, 증감식을 생략해도 됩니다.

Java

```
for(int i = 0;;i += 1) {  
    System.out.println(i); //역시 무한루프 1, 2, 3, 4....  
}
```

Java

```
for(int i = 0; i < 10;) {  
    System.out.println(i); // 역시 무한루프. 0만 출력.  
}
```



증감 연산자는 간편한 연산을 제공합니다. 다만 ++i와 i++가 헷갈릴 수 있습니다.

Java

```
int a = 10;  
System.out.println(a++); // 10 출력후 +1  
System.out.println(++a); // a + 1한 후 12 출력
```

증감 연산자가 변수 앞에 오는 것을 전위 연산자라고 하고 뒤에 오는 것을 후위 연산자라고 합니다. 전위 연산자는 값을 사용한 후에 증감을 수행하고, 후위 연산자는 증감을 먼저 수행하고 값을 사용합니다.

중첩 반복문도 중첩 조건문과 마찬가지로 일반 for 문과 다르지 않습니다. 반복문 안에 반복문이 있는 것을 의미합니다. 중첩 조건문처럼 반복문을 중첩해서 사용하게 되면 가독성이 떨어질 수 있습니다.

또한 반복문을 중첩해서 사용하면 루프를 도는 횟수가 지수적으로 증가하게 됩니다.(같은 횟수를 도는 경우) 그렇기 때문에 프로그램 실행 시간이 길어지게 되고, 이는 성능 저하를 야기합니다.

2. While문

while문은 조건식이 참이면 실행되는 반복문입니다. while문은 for 문과 유사한 반복문이지만, for문과 다르게 증감식과 초기값 설정 문이 없습니다.

```
Java
while (조건식) {
    실행문
}
```

while 문은 for 문처럼 조건식을 항상 True로 설정할 수 있습니다. 이렇게 되면 무한 루프가 발생하게 되는데, 실행문에서 적절한 탈출식을 정의해야 합니다.

```
Java
while (true) {
    실행문
}
```

1분 퀴즈풀이

138p. 1번

sum을 for 문 밖에서 초기화하지 않고 매번 0으로 초기화하기 때문에 항상 sum은 i값이 됩니다. 그래서 1부터 100까지 출력하게 됩니다.

144p. 2번

첫 번째 for 문에서 0부터 6까지 총 7번을 실행합니다. 그리고 각각의 숫자마다, j를 0부터 4까지 5번 실행하기 때문에 35번 명령문이 실행되게 됩니다.

147p. 3번

for 문에서 조건식에 해당하는 부분을 보면 i가 9일때까지만 실행합니다. 10일때는 멈추면 되는 것이죠. j도 마찬가지 입니다.

Java

```
int i = 2;
while (i < 10) {
    System.out.println("--- " + i + "단 ---");
    int j = 1;
    while (j < 10) {
        System.out.println(i + " x " + j + " = " + (i * j));
    }
}
```

Day 8

5장 같은 작업 반복하기: 반복문

안녕하세요! 어제에 이어 반복문을 더 공부해 보겠습니다.

공부할 내용(148~163p)

- 무조건 한 번은 실행할 때: **do - while** 문
- 무한 반복문
- 프로그램 흐름 제어하기

1. do-while 문

do-while 문은 **while**문과 다르게 조건식을 나중에 확인합니다. 그렇기 때문에 최소 한 번의 실행을 보장합니다.

```
Java
do {
    실행문
} while (조건식);
```

do-while 문은 편의를 위해서 존재하기 때문에 **do-while**로 작성된 코드를 **while**로 작성하지 못하는 것은 아닙니다. p.149의 예제 역시 **while**을 **do-while**로 변경하는 예를 들었습니다.

2. 무한 반복문

반복문이 끝나지 않는 않고 무한히 반복되는 경우를 무한 반복문 또는 무한 루프라고 합니다. 7일차에서 설명한 것처럼 다음과 같은 코드에서 발생할 수 있습니다.

Java

```
for(;;) {  
    // infinite-loop  
}  
  
while(true) {  
    // 탈출문 없음  
}
```

이렇게 되면 **CPU**와 메모리를 계속 낭비하게 되어 다른 작업이 현저하게 느려질 수 있습니다. 그래서 반드시 종료 조건을 명시해야 합니다.

3. 프로그램 흐름 제어하기

while문과 **for**문에서 프로그램 혹은 함수를 종료하기 위해서는 어떻게 해야 할까요? 먼저 **break**문이 있습니다. 반복문에서 **break** 문을 만나면 조건과 상관없이 반복문을 탈출합니다.

151p.의 1분 퀴즈의 코드를 **break**문을 사용해 바꿔보면 다음과 같이 작성할 수 있습니다.

Java

```
Scanner scan = new Scanner(System.in);  
int num;  
while (true) {  
    num = scan.nextInt();  
    if (num == 0) {  
        break;  
    }  
}
```

continue는 반복 구간을 건너뛰는 문입니다. 원하지 않는 부분을 뛰어넘을 수 있습니다.

Java

```
for(int i = 1; i < 10; i++) {  
    if (i % 2 == 0) {  
        continue;  
    }  
    System.out.println(i) // 1, 3, 5, 7, 9  
}
```

1분 퀴즈풀이

151. 4번

0을 입력하면 종료되는 프로그램입니다. `scanner`에서 `num`을 입력받고, `num`이 0인지 확인하면 됩니다.

Java

```
num == 0
```

160p. 5번

반복문 안에서 조건을 확인하고, `break`로 반복문을 종료해버립니다. 이때 `i`가 0부터 시작하는데 `if(0%2 == 0)`는 `true`이므로, '짝수'가 한 번 출력되고 프로그램이 종료됩니다.

162p. 셀프체크 풀이

일반적인 자판기와 로직이 조금 다릅니다. 일반적인 자판기라면 금액을 먼저 넣고, 음료를 선택하거나 음료 선택 후 금액을 투입하게 되지만, 이 자판기는 그렇지 않습니다. 그 점에 유의하시고 코드를 작성해주세요!

Java

```
package org.example;
```

```
import java.util.Scanner;
```

```
public class Main {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        int balance = 0;  
        System.out.println("**** 자판기 프로그램을 시작합니다. ****");  
        while (true) {  
            System.out.println("\n현재 투입 금액: " + balance);  
            System.out.println("1. 콜라 (1,500원)");  
            System.out.println("2. 오렌지주스 (2,000원)");  
            System.out.println("3. 생수(1,000원)");  
            System.out.println("4. 종료");
```

```

System.out.print("음료를 고르세요. (번호 입력): ");
int choice = scanner.nextInt();

switch (choice) {
    case 1 -> { // 향상된 switch 문 사용
        if (balance >= 1500) {
            balance -= 1500;
            System.out.println("콜라를 선택했습니다. 남은
금액: " + balance + "원\n");
        } else {
            System.out.println("금액이 부족합니다. 돈을 더
투입하세요.\n");
        }
    }
    case 2 -> {
        if (balance >= 2000) {
            balance -= 2000;
            System.out.println("오렌지주스를 선택했습니다.
남은 금액: " + balance + "원\n");
        } else {
            System.out.println("금액이 부족합니다. 돈을 더
투입하세요.\n");
        }
    }
    case 3 -> {
        if (balance >= 1000) {
            balance -= 1000;
            System.out.println("생수를 선택했습니다. 남은
금액: " + balance + "원\n");
        } else {
            System.out.println("금액이 부족합니다. 돈을 더
투입하세요.\n");
        }
    }
    case 4 -> {
        System.out.println("프로그램을 종료합니다.");
        scanner.close();
        return;
    }
    default -> {
        System.out.println("프로그램을 종료합니다.");
    }
}

```

```
        System.out.println("돈을 더 투입하세요. (0을 입력하면 메뉴로  
돌아갑니다.): ");  
        int money = scanner.nextInt();  
        if (money > 0) {  
            balance += money;  
        } else {  
            System.out.println("메뉴로 돌아갑니다.\n");  
        }  
    }  
}  
}
```

Day 9

6장 여러 값 한 번에 저장하기: 배열

안녕하세요! 오늘은 6장 배열에 대해 배워봅니다.

공부할 내용(165~194p)

- 배열의 기본
- 이차원 배열
- 배열 심화

1. 배열의 기본

배열은 같은 자료형의 값을 여러 개 저장할 수 있는 자료구조입니다.

멘토 Tip

자료구조(데이터 구조)는 데이터를 효율적으로 저장하고 관리하기 위한 방법이나 형식을 의미합니다. 자료구조는 데이터의 조직, 관리, 저장 방식을 정의하며, 알고리즘과 함께 사용되어 데이터 처리의 효율성을 높이는 데 중요한 역할을 합니다.



배열은 우선 크기가 정해져 있고, 한 배열 안에서는 같은 자료형만 사용해야 합니다.

Java

```
int[] ages;  
String[] names;
```

선언만 해서는 사용할 수 없고, 반드시 초기화가 필요합니다.

Java

```
int[] ages = {10, 20, 30};  
String[] names = {"John", "Doe"};
```

배열은 초기화를 하고 값을 지정하지 않으면 자료형의 기본값으로 초기화됩니다.

Java

```
int[] scoreArray = new int[4];
```

배열에 접근할 때는 인덱스를 사용해서 접근합니다. 배열의 인덱스는 항상 0부터 시작합니다.

Java

```
int[] ages = {10, 20, 30};  
int age = ages[0]; // 10
```

배열 순회도 간단합니다.

Java

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

2. 2차원 배열

앞에서 배운 배열은 일차원 배열입니다. 2차원 배열은 배열 안에 배열이 존재하는 구조로 더 많은 데이터를 표현할 수 있습니다.

Java

```
int[][] arr2 = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}};
```

2차원 배열도 1차원 배열처럼 똑같이 인덱스로 접근합니다. 다만 2차원이기 때문에 조금 헷갈릴 수 있습니다. 이는 178p. 표를 보면 쉽게 이해할 수 있습니다.

2차원 배열 순회는 for문 두 개를 중첩해서 순회할 수 있습니다.

Java

```
for (int i = 0; i < arr2.length; i++) {  
    for(int j = 0; j < arr2[i].length; j++) {  
    }  
}
```

3. 배열 심화

이 단원에서는 복사에 대해서 알아봅니다. 자바에는 참조 자료형과 기본 자료형이 존재합니다. 참조 자료형은 힙메모리에 저장되고, 기본 자료형은 스택메모리에 저장됩니다.

배열은 참조 변수입니다. 그래서 힙 메모리를 할당해서 값을 저장합니다. 힙 메모리는 전역적인 주소 값을 가지게 되는데, 이 때문에 복사에 있어서 기본 자료형보다 까다롭게 됩니다.

Java

```
int a = 1;  
int b = a;  
  
b = 3;  
  
System.out.println(a); // 1  
System.out.println(b); // 3
```

이 코드는 b가 변경된다고 해서 a의 값이 변하지 않습니다. 하지만 다음 코드는 다릅니다.

Java

```
int[] a = {1, 2, 3};  
int[] b = a;  
  
System.out.println(a);  
System.out.println(b);
```

우선 이 코드를 출력하면 두 출력 모두 동일한 주소값을 가리키고 있습니다. 그렇기 때문에 **a**가 변경이 되면 **b**도 영향을 받게 되죠. 이런 코드를 얕은 복사라고 합니다. 얕은 복사시에 같은 주소를 바라보기 때문에 문제가 생깁니다. 그래서 깊은 복사가 필요합니다. 깊은 복사는 다른 주소 값을 가진 변수를 새로 생성하고 만 옮기는 것이죠. 190p.의 예가 깊은 복사에 해당합니다.

1분 퀴즈풀이

175. 1번

`value * (i + 1)`이 10, 20, 30, 40의 값을 가지게 됩니다. 그렇기 때문에 **numbers**를 순회하는 `for` 문에서 10, 20, 30, 40이 출력되게 됩니다.

186p. 2번

첫 번째 `for` 문에는 학생들을 순회하는 `for` 문이기 때문에 학생의 숫자만큼에 대한 식이 들어가야합니다. 그리고 두 번째로는 학생의 각 과목별 점수를 순회합니다.

Java

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

191p. 3번

이 문제는 깊은 복사에 대한 문제입니다. **copy** 배열의 값을 **original** 배열에서 깊은 복사했기 때문에 **copy** 배열을 바꾼다고 해도 **original** 배열에 영향을 주지 않습니다. 그렇기 때문에 **copy[3]**은 여전히 4입니다.

194p. 셀프체크 풀이

2차원 배열에 관한 문제로, 반복문과 배열을 활용하면 문제를 풀 수 있습니다.

Java

```
public class Main {  
    public static void main(String[] args) {  
        int[][] scores = new int[3][4];  
        scores[0] = new int[]{85, 70, 90, 95};  
    }  
}
```

```
scores[1] = new int[]{80, 95, 90, 75};
scores[2] = new int[]{75, 85, 90, 80};

System.out.println("학생들의 성적은 다음과 같습니다.");
for (int i = 0; i < scores.length; i++) {
    int sum = 0;
    System.out.print("학생 1: ");
    for (int j = 0; j < scores[i].length; j++) {
        sum += scores[i][j];
        System.out.print(scores[i][j] + " ");
    }
    System.out.println("| 평균: " + (double) sum / 4);
}
}
```

Day 10

7장 클래스

안녕하세요! 오늘부터 자바에서 핵심이라고 할 수 있는 클래스를 배웁니다. 오늘 공부할 내용은 다음과 같습니다.

공부할 내용(195~218p)

- 객체지향 프로그래밍
- 클래스와 객체

1. 객체지향 프로그래밍

프로그래밍에는 여러가지 패러다임이 있습니다.

- 절차지향 프로그래밍
- 함수형 프로그래밍
- 관점지향 프로그래밍

그 중 현대의 많은 언어가 채택하고 있는 것이 바로 객체지향 프로그래밍입니다. 객체지향 프로그래밍은 데이터를 객체 단위로 묶어 프로그래밍을 구성하는 방식입니다. 객체는 각각의 역할을 가지고 있고, 메시지를 통해 서로 상태의 변화시키거나 협력합니다.

이런 객체지향을 구성하는 4가지 요소가 있습니다.

1. 캡슐화: 캡슐화는 객체로 자원을 감추는 것을 말합니다. 캡슐화는 기본적으로 내부의 구현 세부 사항을 감추고, 외부에 필요한 인터페이스만을 노출하는 것을 말합니다.
2. 상속: 상속은 객체가 다른 객체의 기능을 이어 받는 것을 말합니다. 'B가 A를 상속했다' 라고 하면 B는 A의 기능을 물려받게 됩니다.
3. 다형성: 다형성은 같은 메서드에 대해 서로 다른 객체가 다른 방식으로 응답하는 것을 말합니다. 자바에서는 오버라이딩과 오버로딩으로 구현이 가능합니다.
4. 추상화: 추상화는 객체마다 구체적으로 동작을 정의하지 않고, 공통적인 데이터와 동작을 추출하여 일반화 하고, 중요한 속성과 동작을 정의하는 것을 말합니다.

멘토 Tip

객체지향에 대해서 간략하게 다루었지만 객체지향은 몇 줄로 이해하기 쉽지 않은 분야입니다. 먼저 앞에서 언급한 네 가지 객체지향의 특징을 이해하고 다양한 객체지향 책을 찾아보기를 권장합니다.



2. 클래스와 객체

클래스 == 객체라고 오해하기 쉽지만 클래스와 객체는 다릅니다. 간략하게 설명하면 클래스는 객체를 만드는 틀 혹은 설계도 정도로 설명할 수 있고, 객체는 그 설계도를 바탕으로 만들어진 산출물입니다. 그리고 우리는 그 산출물을 객체 혹은 인스턴스라고 부릅니다.

자바에서는 객체를 만들기 위해서는 클래스가 있어야합니다.

Java

```
public class Car {  
  
}
```

그리고 클래스는 필드, 메서드, 생성자로 구성될 수 있습니다. 필드는 멤버 변수(인스턴스 변수), 클래스 변수(정적 변수)를 말합니다. 인스턴스 변수는 객체마다 각각 다른 값을 가질 수 있고, 정적 변수는 정적 변수로, 같은 클래스의 인스턴스는 모두 같은 값을 가집니다.

메서드 역시 인스턴스 메서드가 있고, 정적 메서드가 있습니다. 인스턴스 메서드는 인스턴스에 종속된 메서드로 인스턴스 변수의 값을 활용하는 함수입니다. 정적 메서드는 특정 객체에 종속되지 않은 메서드로, 클래스 자체에 종속되어 정적 변수(클래스 변수)를 활용하는 함수입니다. 메서드는 매개변수를 가질 수 있고, 반환형이 존재합니다.

생성자는 인스턴스(객체)를 만들 때 사용하는 특별한 메서드입니다. 이름이 클래스와 같고 반환형을 명시하지 않습니다. 생성자에도 매개변수를 정의할 수 있는데, 매개변수를 통해 인스턴스를 생성할 때 인스턴스 변수를 초기화할 수 있습니다.

Java

```
public Person {
```

```

    int age;
    int height;

    public Person(int age, int height) {
        this.age = age;
        this.height = height;
    }

    public int getAge() {
        return this.age;
    }
}

```

위 코드에서 **this**는 객체 자기 자신을 말합니다. 해석하면 '나를 생성할 때, **age**는 매개변수로 넘어온 **age**로 설정하고, **height**은 매개변수로 넘어온 **height**로 설정해.' 입니다.

객체를 생성할 때는 배열에서 사용한 것처럼 **new** 키워드를 활용합니다.

Java

```

Person person = new Person(20, 180);

```

멘토 Tip

자바에서는 생성자를 작성하지 않으면 기본 생성자가 기본적으로 생성됩니다.

Java

```

class Person {
    public Person() {} // 기본 생성자
}

```

하지만 하나라도 생성자를 정의하면 기본 생성자를 사용하기 위해서는 명시적으로 생성해야 합니다.

Java

```

class Person {

```

```

    int age;
    public Person(int age) {
        this.age = age;
    }
}

class Main {
    public void main() {
        Person person = new Peron(); // 불가능
    }
}

```

1분 퀴즈풀이

218p. 1번

Java

```

public class Main {
    public static void main(String[] args) {
        Car otherCar = new Car("Genesis", 2025, "Red");
        otherCar.carInfo();
    }
}

```

Day 11

7장 클래스

안녕하세요! 오늘은 7장 클래스의 나머지 부분을 이어서 공부해 보겠습니다.

공부할 내용(219~235p)

- 클래스 심화

1. 클래스 심화

static

static은 자바의 예약어로 정적인 것을 나타낼 때 사용하는 예약어입니다. **static**은 클래스, 변수, 메서드에 모두 붙을 수 있습니다. **static**과 결합하게 되면 정적 클래스, 정적 변수, 정적 메서드가 됩니다.

static이 붙으면 **JVM**의 메서드 영역에 저장됩니다. 그리고 클래스가 처음 로드될 때(클래스가 처음으로 참조될 때) 정적 변수와 메서드가 메서드 영역에 올라갑니다. 그리고 이때 올라간 정적 변수와 메서드, 클래스는 **JVM**이 종료되기 전까지 메모리에서 유지됩니다.

멘토 Tip

static 키워드는 조심해서 사용해야 합니다. 인스턴스에 종속되지 않기 때문에 많은 것을 정적으로 선언할 경우 메모리에 부담을 줄 수 있습니다.

특히 정적 변수의 경우 여러 인스턴스에서 정적 변수를 수정하는 코드를 피해야 합니다. 다음과 같이 식으로 코드를 작성한다면 여러 스레드에서 접근했을 때 원하지 않는 결과를 얻게 될 수도 있습니다.

Java

```
class Car {  
    static int speed = 10;  
  
    // 해서는 안 됨.  
    public void speedUp() {  
        speed += 10;  
    }  
}
```



접근 제한자

자바에는 객체지향 프로그래밍 특징 중 하나인 캡슐화를 보조하는 수단으로 접근 제한자가 존재합니다. 접근 제한자를 통해 클래스 내부의 데이터와 메서드에 대한 외부 접근을 제한함으로써 객체 지향의 핵심 원칙 중 하나인 캡슐화를 구현하고 있습니다.

자바에는 4가지 접근 제한자가 있습니다.

1. **public** **public** 접근 제한자는 어디에서나 접근 가능한 접근 제한자입니다.
2. **protected** **protected**는 같은 패키지의 클래스나 다른 패키지의 상속 받은 클래스에서 접근할 수 있습니다. 클래스에는 사용할 수 없습니다.
3. **default** **default**는 **package-private**이라고도 부르며, 따로 명시하지 않으면 설정되는 접근 제한자입니다.
4. **private** **private**은 같은 클래스에서만 접근할 수 있습니다. 클래스, 필드, 메서드, 생성자 모두에 사용할 수 있지만, 클래스의 경우 중첩 클래스에서만 사용이 가능합니다.

getter와 setter

자바에서는 **private** 혹은 **protected**인 필드에 접근할 때, **getter**와 **setter**를 사용해서 우회해 접근합니다.

getter는 **public**으로 선언해서 다른 클래스에서 필드의 값을 읽어오는 데 사용합니다. 다음과 같이 작성하면 **name**는 **Car** 클래스 내부에서만 접근 가능하면서 값은 외부로 노출할 수 있게 됩니다.

Java

```
class Car {  
    private String name;  
  
    public String getName() {  
        return this.name;  
    }  
}
```

getter가 값을 읽었다면 **setter**는 값을 쓰는 용도로 사용됩니다.

Java

```
class Car {  
    private String name;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

1분 퀴즈풀이

224p. 2번

add 메서드를 호출 할 때마다 정적변수 **count**를 증가시킵니다. 정적 변수이기 때문에 모든 인스턴스에서 접근이 가능하고, 모든 인스턴스에서 동일한 값을 가집니다. 가장 먼저 **count**의 값인 2가 출력이 되고 15, 10이 연어 출력되면서, 마지막으로 **count** 2가 출력됩니다.

Unset

```
호출 횟수 : 2  
result1: 15  
result2: 10  
메서드가 2번 호출됐습니다.
```

231p. 3번

getter는 **get+Pascal** 케이스 필드명입니다.

- `public String getCarNumber()`
- `this.carNumber = carNumber`
- `oldCar.getCarNumber()`
- `oldCar.setCarNumber`
- `oldCar.getCarNumber()`

235p. 셀프체크 풀이

1. 3번째 라인(`Person person = new Person`)에서 생성자의 파라미터가 2개라는 사실을 알 수 있습니다.
2. 첫 번째 파라미터는 `String`, 두 번째는 `int` 임을 알 수 있습니다.
3. 4번째 라인에서 `displayInfo()`라는 메서드가 있다는 것을 알 수 있습니다.
4. 5,6번째 라인에서 `name`과 `age` 필드가 존재하는 것을 알 수 있습니다.
5. 7,8번째 라인에서 출력을 하기 때문에, `displayInfo()`메서드가 이름과 나이를 출력해야하는 메서드인 것을 알 수 있습니다.

Java

```
public class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
```

```
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void displayInfo() {
        System.out.println("이름: " + this.name + "\n");
        System.out.println("나이: " + this.age + "\n");
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Day 12

8장 상속과 다형성

안녕하세요! 오늘부터 3일간 객체지향 언어의 중요한 특징인 상속과 다형성에 대해 공부해 봅니다.

🧐 공부할 내용(237~259p)

- 상속
- 다형성

1. 상속

상속은 객체지향의 특징 중 하나입니다. 세상에도 부모와 자식 관계가 있든, 클래스에도 부모 클래스와 자식 클래스가 있습니다. 자식 클래스는 부모 클래스를 상속합니다. 사용 방법은 다음과 같습니다.

Java

```
class ChildClass extends ParentClass{  
  
}
```

언뜻 보기에는 어려워 보이지만 실습을 따라 하면 쉽게 이해할 수 있습니다. 실습에서 강아지와 고양이 클래스가 먼저 정의되어 있습니다. 이 두 클래스는 **name**, **age** 필드가 겹치고, **eat()**이라는 메서드도 겹칩니다. 이렇게 겹치는 부분을 분리할 수 있습니다.

Java

```
public class Animal {  
    String name;  
    int age;
```

```

    public void eat() {
        System.out.println(name + "가 밥을 먹습니다.");
    }
}

```

그리고 강아지와 고양이가 이를 상속합니다.

```

Java
public class Dog extends Animal {}

public class Cat extends Animal {}

```

이렇게 상속하면 **Dog** 클래스와 **Cat** 클래스는 모두 **eat()**에 접근이 가능함과 동시에 **name**과 **age**에도 접근이 가능합니다.

```

Java
Cat cat = new Cat();
cat.name = "나비";
cat.age = 5;

cat.eat();

```

2. 다형성

다형성은 객체지향 프로그래밍의 특징 중 하나로, 동일한 인터페이스나 부모 클래스를 공유하는 객체가 각기 다른 방식으로 동작할 수 있는 것을 말합니다.

자바에서는 다양한 방식으로 다형성을 구현하고 있습니다.

메서드 오버라이딩

메서드 오버라이딩은 자식 클래스가 부모 클래스에 정의된 메서드를 재정의하여 사용하는 것을 의미합니다. 이때, 메서드의 이름, 반환 타입, 매개변수(시그니처)가 부모 클래스와 완전히 동일해야 합니다.

@Override 애너테이션을 사용해 오버라이딩 여부를 명확히 나타낼 수 있습니다.

Java

```
class Animal {
    void sound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Woof woof");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("Meow");
    }
}
```

메서드 오버로딩

메서드 오버로딩은 같은 클래스 내에서 이름이 같은 메서드를 매개변수의 종류, 개수, 순서 등을 다르게 정의하여 사용하는 것을 의미합니다.

Java

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }

    double add(double a, double b) {
```

```

        return a + b;
    }

    double add(int a, double b) {
        return a + b;
    }
}

```

형변환

앞서 명시적 형변환과 묵시적 형변환을 배웠습니다. 상속관계에서는 2가지 형변환이 있습니다.

- 업캐스팅
- 다운캐스팅

업캐스팅은 자식 클래스의 객체를 부모 클래스형으로 변환하는 것을 말합니다.

```

Java
Animal dog = new Dog();
Animal cat = new Cat();

```

이는 컴파일러에 의해 자식 클래스의 객체가 부모 클래스로 자동 형변환됩니다. 이때, 자식 클래스에 선언한 필드와 메서드에는 접근할 수 없고, 오직 부모 메서드에 정의된 것에만 접근할 수 있습니다.

다운캐스팅은 부모 클래스형 객체를 자식 클래스형으로 형변환하는 것을 말합니다. 다운캐스팅하려면 명시적 형변환처럼 괄호 안에 자식 클래스 명을 지정해줘야 합니다.

```

Java
Animal animal = new Dog();
Dog dog = (Dog)animal;

```

다운 캐스팅에는 주의가 필요합니다. 앞의 코드에서 `Animal animal = new Cat()`이라면 `Dog`으로 형 변환이 이루어졌을 것이고, `dog.roll()`을 호출 할 수도 있을 것입니다. 하지만 `animal`은 `Cat`의 객체이기 때문에 `roll`이 정의되어 있지 않아서 런타임 에러가 발생합니다.

super

앞서 **this**를 배웠습니다. **this**는 인스턴스 자신을 의미합니다. **super**는 **this**의 부모 클래스를 참조하는데 사용되고, 부모 클래스를 말합니다.

- 부모 클래스의 메서드를 호출할 때 사용합니다. ex: `super().eat()`
- 자식 클래스의 생성자에서 부모 클래스의 생성자를 호출 할 때 사용합니다 기존 예시에서는 `super()`를 호출하는 코드를 보지 못하셨을 것입니다. 생성자에서 `super()`를 명시적으로 호출하지 않으면 부모 클래스의 기본 생성자를 호출하게 됩니다.
- `super` 키워드는 부모 클래스의 필드에 접근할 때 사용합니다. ex: `super.name`, `super.age`

멘토 Tip

만약 부모 클래스에 기본 생성자가 없다면 컴파일 에러를 발생시킵니다.



1분 퀴즈풀이

245p. 1번

1. 상속하는 클래스는 부모 클래스입니다. 상속 받는 클래스는 자식 클래스입니다.
2. `extends` 키워드를 사용해서 상속을 표현합니다.

253p. 2번

1. 첫 번째 출력에서 <나는 학생입니다.>를 출력했기 때문에 `person`입니다.
2. 두 번째 출력에서 <대학교 1학년입니다.>를 출력했기에 `student`입니다.

259p. 3번

1. 부모 클래스의 생성자를 호출해줘야하기 때문에 `super(name, age)`
2. `Student` 클래스의 `grade`를 초기화해야하므로 `this.grade = grade`

Day 13

8장 상속과 다형성

안녕하세요! 오늘은 어제에 이어 8장 <상속과 다형성>을 공부합니다. 그중에서 추상 클래스를 자세히 살펴보겠습니다.



공부할 내용(260~267p)

- 추상 클래스

1. 추상 클래스

이번 장에서는 추상화에 대해서 다룹니다. 추상화는 객체지향 프로그램에서 역시 중요한 개념입니다. 추상은 '여러 가지 사물이나 개념에서 공통되는 특성이나 속성을 추출해 파악하는 작용'입니다. 그리고 추상적이란 '어떤 사물이 일정한 형태와 성질을 갖추고 있지 않은 것'을 말합니다.

프로그래밍 세계에도 추상으로 표현할 수 있는 것이 있습니다. 바로 추상 클래스입니다. 추상 클래스는 일반 클래스의 공통 부분을 추출해서 만든 완전하지 않은 클래스입니다.

Java

```
public abstract class Person {  
    abstract getAge();  
}
```

추상 클래스는 다음과 같은 특징을 가지고 있습니다.

1. 추상 메서드를 하나 이상 가질 수 있습니다.(갖지 않아도 됩니다.)
2. 추상 클래스는 **new** 키워드를 사용해서 객체를 생성할 수 없습니다.
3. 일반 메서드를 가질 수 있습니다.

262p의 예를 살펴봅시다.

추상 클래스는 필드를 가질 수 있습니다. 또한 생성자를 가질 수 있습니다. 추상 클래스는 추상 메서드를 가질 수 있고, 일반 메서드도 가질 수 있습니다.

Java

```
public abstract class BankAccount {
    String accountNumber;
    double balance;

    public BankAccount(String accountNumber, double balance)
    {
        this.accountNumber = accountNumber;
        this.balance = balance;
    }

    public abstract void withdraw(double amount);

    public abstract void deposit(double amount);

    public void displayAccountInfo() {
        System.out.println("계좌 번호: " + accountNumber);
        System.out.println("잔액: " + balance + "원");
    }
}
```

이를 상속하면 **BankAccount**에 정의된 **withdraw**, **deposit** 추상 메서드는 반드시 구현해야 하기 때문에 상속받는 클래스인 **CheckingAccount** 클래스에서 이를 재정의하고 있습니다. 또한 **super**를 사용해서 **BankAccount**의 생성자를 호출하는 코드도 볼 수 있습니다.

Java

```
public class CheckingAccount extends BankAccount{
    public CheckingAccount(String accountNumber, double
balance) {
        super(accountNumber, balance);
    }

    @Override
    public void withdraw(double amount) {
```

```

        if(amount > 0 && balance >= amount) {
            balance -= amount;
        } else {
            System.out.println("잔액이 부족합니다.");
        }
    }

    @Override
    public void deposit(double amount) {
        if(amount > 0) {
            balance += amount;
        } else {
            System.out.println("입금액은 0보다 커야 합니다.");
        }
    }
}

```

이렇게 추상클래스는 여러 클래스에서 반복되는 코드를 정의해서 공통 기능을 한 번만 작성할 수 있게 해주면서 다형성을 제공합니다. 또한 이후 배울 인터페이스보다 멤버 변수와 생성자를 포함할 수 있어서 더 유연하게 설계가 가능합니다.

1분 퀴즈풀이

267p. 4번

Person은 추상 클래스이기 때문에 인스턴스를 생성할 수 없습니다. 그렇기 때문에 1, 3, 4 번 코드가 모두 에러가 발생합니다.

Day 14

8장 상속과 다형성

안녕하세요! 오늘은 8장 <상속과 다형성> 마지막 날로, 인터페이스에 대해 공부합니다.

 공부할 내용(268~281p)

- 인터페이스

1. 인터페이스

인터페이스 역시 추상에 속합니다. 인터페이스는 메서드의 시그니처만을 포함하고, 구현을 포함하지 않는 일종의 계약입니다. 그리고 상태를 가지고 있는 추상 클래스와 달리 동작의 규칙을 정의하는데 중점을 둡니다.

```
Java
interface Charger {
    void charge(String type);
}
```

이렇게 상태는 인터페이스의 관심사가 아닙니다. 오직 동작만이 인터페이스의 관심사입니다. 인터페이스는 몇 가지 특징을 가지고 있습니다.

1. 추상 메서드와 **default** 메서드로 이루어질 수 있습니다. **default** 메서드는 구현을 포함한 메서드입니다.
2. 여러 인터페이스를 상속하는 다중 상속을 지원합니다.
3. 정적 메서드를 가질 수 있습니다.
4. 인터페이스는 상수만을 필드로 포함할 수 있습니다.
5. 접근 제한자를 따로 설정하지 않으면 모두 **public** 입니다. **public**, **default**만 가능합니다.

6. 객체를 생성할 수 없습니다.

예제 코드를 통해서 인터페이스를 익혀봅시다.

Java

```
interface Payment {  
    void processPayment(double amount);  
}
```

Java

```
class CreditCardPayment implements Payment {  
    @Override  
    public void processPayment(double amount) {  
        System.out.println("Processing credit card payment of  
$" + amount);  
    }  
}  
  
class PayPalPayment implements Payment {  
    @Override  
    public void processPayment(double amount) {  
        System.out.println("Processing PayPal payment of $" +  
amount);  
    }  
}
```

Payment 인터페이스에서 **processPayment**라는 추상 메서드(동작)을 정의했습니다. 그리고 이를 **CreditCardPayment**, **PaypalPayment** 클래스에서 구현했습니다. 이렇게 하나의 동작을 정의하고, 여러 클래스에서 역할에 맞게 구현해서 사용할 수 있습니다.

멘토 Tip

자바 8 버전 이전까지는 인터페이스는 오직 추상 메서드만 가질 수 있었습니다. 자바 8 버전이 되면서 인터페이스가 **default** 메서드도 가질 수 있게 되었습니다. 그 외에도 자바 8버전에는 다양한 기능이 추가되었습니다.

- 람다 표현식 추가
- 함수형 인터페이스가 추가되었습니다.

- 스트림 API가 추가되었습니다.
- **Optional** 클래스가 추가되었습니다.

람다 표현식과 함수형 인터페이스는 뒤에서 배웁니다.



1분 퀴즈풀이

276p. 5번

1. **Person**이 인터페이스이기 때문에 **implements**가 들어갑니다.
2. **Override**할 수 있는 메서드는 **introduce** 밖에 없습니다.

281p. 셀프체크 풀이

SavingsAccount클래스가 **BankAccount**와 **InterestBearing** 인터페이스를 구현하기 때문에 **withdraw(double amount)**, **deposit(double amount)**, **displayAccountInfo()**와 **addInterest()** 메서드는 반드시 구현해야 합니다. 그리고 **addMonthlyDeposit()** 메서드를 추가해서 매월 **balance**에 **monthlyDeposit**을 추가하는 코드를 작성하면 됩니다.

Day 15

9장 예외 처리

안녕하세요! 오늘은 3주차 마지막 날로 9장 <예외 처리>에 대해 공부합니다. 오늘까지 공부하고 주말에 쉴 수 있으니 힘을 내봅시다.

공부할 내용(283~302p)

- 오류와 예외
- 예외 처리하기

1. 오류와 예외

오류

오류는 프로그램에서 발생할 수 있는 문제로, 제어할 수 없고, 정상적인 실행도 불가능한 상황을 의미합니다. 프로그램에서의 오류는 크게 3가지로 나눌 수 있습니다.

1. 컴파일 타임 오류

컴파일 타임 오류가 발생하면 애플리케이션이 실행조차 될 수 없습니다. 주로 문법 오류, 타입 오류, 참조 오류 등이 여기에 해당합니다. IDE를 사용했을 때 쉽게 확인할 수 있습니다.

2. 런타임 오류

런타임 오류는 애플리케이션 실행 도중에 발생하는 오류입니다. 런타임 오류는 대부분 예외 처리를 통해서 해결할 수 있습니다.

3. 논리 오류

논리 오류는 컴파일과 실행은 정상적으로 되지만 결과가 의도한 대로 나오지 않아서 발생하는 오류입니다. 코드 작성과정에서 생기는 문제로, 디버거를 활용해서 해결할 수 있습니다.

예외

예외는 예측할 수 있는 오류를 말합니다. 그리고 이 예외에 대해 미리 조치를 취하는 것을 예외 처리라고 합니다. 자바에서는 예외를 두 가지로 나눕니다.

1. 확인 예외(**checked exception**)

확인 예외는 컴파일러가 예외를 확인하고 예외처리를 하는지 검사합니다. 예외 처리를 하지 않으면 컴파일 오류가 발생하기 때문에 위험부담이 상대적으로 적습니다.

2. 미확인 예외(**unchecked exception**)

미확인 예외는 컴파일 과정에서 예외 처리를 확인하지 않으며, 명시적으로 예외 처리를 강제하지도 않습니다. 우리가 마주하는 대부분의 예러는 미확인 예외에 해당합니다.

가장 흔하게 볼 수 있는 미확인 예외는 다음과 같습니다.

- `NullPointerException`: null 값 참조로 필드나 메서드에 접근할 때 발생.
- `ArrayIndexOutOfBoundsException`: 잘못된 인덱스로 배열에 접근했을 때.

Java

```
public class Main {  
    public static void main(String[] args) {  
        String s = null;  
        boolean res = s.isEmpty(); // NullPointerException  
    }  
}
```

발생

2. 예외 처리하기

자바에서는 예외를 처리하는 몇 가지 방법이 있습니다.

try-catch-finally

try-catch-finally 문은 다음과 같이 사용할 수 있습니다.

```

Java
try {
    // 메서드 실행
} catch(e Exception) {
    // 예외 발생 시 처리 로직
} finally {
    // 예외와 상관없이 항상 수행할 동작
}

```

try 블록 안에는 예외가 발생할 수 있는 코드를 작성합니다. **try** 블록 안에서 예외가 발생하면 **catch** 블록으로 넘어갑니다. **catch** 블록은 여러 개의 예외를 처리할 수 있습니다. 마지막으로 **finally** 문에서 예외와 상관없이 항상 수행할 동작을 수행하게 됩니다. **finally**는 명시하지 않아도 됩니다.

```

Java
try {
    String s = null;
    s.isEmpty();
} catch (NullPointerException e) {
    System.out.println(e);
}

```

예외를 잡는 방법을 알아보았으니 예외를 던지는 방법을 알아보겠습니다.

```

Java
int a = 10;
if (a >= 10) {
    throw new IllegalArgumentException("a는 10보다 작아야합니다.");
}

```

이렇게 **throw**를 활용해서 예외를 던질 수 있습니다. 던져진 예외는 해당 메서드를 호출한 곳에서 **try-catch**를 이용해서 오류를 잡을 수도 있고, 그냥 예외를 사용자에게 리턴할 수도 있습니다.

멘토 Tip

어떤 예러가 날지 명확하게 알고 있을 때는 **catch** 절에 해당 예러를 명시해서 처리할 수 있지만, 그렇지 않은 경우라면 **Exception**을 **catch**절에 명시해서 모든 예외를 다 처리하게 할 수 있습니다.

```
Java
try {
    String s = null;
    s.isEmpty();
} catch (Exception e) {
    System.out.println(e);
}
```

이게 가능한 이유는 모든 예외는 **Exception**의 자식 클래스이고, **catch**절은 자식 클래스이기만 하면 모두 예외를 잡습니다.

마지막으로 **throws**는 예외 처리를 호출하는 곳에 넘길 때 사용합니다. 다음과 같이 메서드를 정의하면 메서드를 사용할 때는 반드시 해당 예러를 발생시킬 수 있다는 것을 개발자가 인지할 수 있기 때문에 **try-catch**로 해당 예러를 핸들링하게 됩니다.

```
Java
public void maybeThrowError() throws IllegalArgumentException
{
    //
}
```

1분 퀴즈풀이

289p. 1번 5.

실행도 불가능한 상황은 컴파일 타임 과정에서 발생하며 프로그램이 컴파일 되지 않게 만듭니다.

294p. 2번

throw new를 이용해 예러를 던질 수 있습니다. 그리고 **IllegalArgumentException**은 던졌기 때문에 **IllegalArgumentException**을 처리해줘야합니다.

298p. 3번

2. 정의된 예외와 일치하지 않으면 프로그램이 종료됩니다.

301p. 셀프체크 풀이

`withdraw`시에 출금 금액보다 잔고가 적으면 `LowBalanceException`을 던집니다. 또한 `throws`로 예외를 던지고 있기 때문에 호출하는 곳에서 예외를 처리해줘야 합니다.

Java

```
BankAccount account = new BankAccount(100000);
try {
    account.deposit(50000);
    account.withdraw(200000);
} catch (LowBalanceException e) {
    System.out.println("오류: " + e.getMessage());
} finally {
    System.out.println("현재 잔액: " + account.getBalance() +
        "원");
}
```

Day 16

10장 컬렉션 프레임워크

안녕하세요! 오늘은 10장 <컬렉션 프레임워크>에 대해 공부해 봅시다.

공부할 내용(305~314p)

- 컬렉션 프레임워크 개요

1. 컬렉션과 컬렉션 프레임워크 개요

6장에서 배운 배열은 자바의 기본 자료구조입니다. 자료구조는 어떤 데이터를 담는 여러 형태의 그릇을 의미합니다. 자바에서 제공하는 기본 자료구조로는 배열, 리스트, 스택, 큐 등이 있습니다.

이런 자료구조를 제공하는 패키지는 `java.util`에 담겨 있으며, 대부분 `Collection`을 상속합니다. 즉, 컬렉션(`Collection`)은 데이터를 그룹으로 묶어 하나의 객체로 관리할 수 있게 하는 자료구조인 동시에 자료구조들의 부모입니다.

멘토 Tip

`java.util`에 있는 모든 자료구조가 `Collection` 인터페이스를 상속하고 있는 것은 아닙니다.

Collection을 상속하는 자료구조

- List, Stack, HashSet, LinkedList 등

Collection을 상속하지 않는 자료구조

- HashMap, TreeMap 등

307p에 있는 그림 10-1이 상속 구조를 자세하게 보여줍니다.



List: 삽입한 순서대로 요소를 저장하는 컬렉션입니다.

Set: 집합으로 중복을 허용하지 않습니다.

Map: Key-Value로 이루어진 자료구조입니다. **key**는 중복을 허용하지 않습니다.

Queue: 선입선출 자료구조로, 중복을 허용합니다.

Deque: 덱은 양방향 큐입니다. 양쪽에서 데이터를 빼는 속도가 빠릅니다.

래퍼 클래스

래퍼 클래스(Wrapper Class)는 자바의 기본형을 참조형으로 변환한 클래스입니다. 참조형으로 변환해서 사용하는 이유는 여러 가지입니다. 그 중 대표적인 이유는 컬렉션이 제네릭 기반으로 동작하는데, 제네릭은 참조형으로만 선언이 가능하기 때문입니다.

기본형을 래퍼 클래스로 변환하는 것을 박싱이라고 하고 그 역을 언박싱이라고 합니다.

제네릭

제네릭을 도입하기 전에는 자료구조를 다음과 같이 사용했습니다.

```
Java
List list = new ArrayList<>();
list.add("Hello");
String value = (String)list.get(0);
```

이때 다음과 같은 문제가 발생할 수 있습니다.

1. 타입에 대한 정의가 없습니다. 그래서 어떤 타입의 값을 넣든 모두 동작합니다.
2. 타입 정의가 없어서 **get()**을 호출하는 쪽에서 어떤 타입인지 알 수 없습니다. 그래서 형변환이 실패하기 전까지 어떤 타입인지 알 수 없습니다.

코드를 보면 **list**에 저장된 타입이 어떤 타입인지 **list.get()**을 하는 상황에서는 알 수가 없습니다. 설사 안다고 해도 사용하려면 값을 사용하려는 타입으로 변환해줘야 합니다.

결국 앞의 코드는 타입에 대한 안정성을 제공하지 못하는 문제가 있습니다. 그래서 자바는 제네릭(generic)을 도입해 이 문제를 해결했습니다.

Java

```
List<String> list = new ArrayList<>();  
list.add("Hello");  
list.add(1); // 에러  
String value = list.get(0);
```

제네릭을 사용하면 앞의 문제들을 해결 할 수 있습니다.

제네릭은 데이터 타입을 일반화(**generalize**)해 다양한 타입에서 재사용할 수 있도록 설계하는 방법입니다. 타입 안정성을 보장하고 컴파일 시 타입 검사를 강화해 더 안전하고 유연한 코드를 작성할 수 있게 합니다..

1분 퀴즈풀이

314p. 1번

① 컬렉션은 기본형 데이터를 직접 저장할 수 없습니다. 기본형은 오토크싱을 통해 참조형으로 변환 후 저장 가능합니다.

Day 17

10장 컬렉션 프레임워크

안녕하세요. 오늘도 지난 시간에 이어 10장 <컬렉션 프레임워크>를 공부해 봅시다.

공부할 내용(315~334p)

- 컬렉션 프레임워크와 주요 인터페이스

1. 컬렉션 프레임워크와 주요 인터페이스

List 인터페이스

List 인터페이스는 기본 자료구조인 리스트를 정의한 인터페이스입니다. List는 다음과 같은 특징이 있습니다.

- 요소들이 삽입된 순서를 유지합니다.
- 중복을 허용합니다.
- 인덱스를 통해 접근할 수 있습니다.
- 크기가 변경될 수 있습니다.

List를 구현하는 대표적인 클래스로 ArrayList, LinkedList가 있습니다.

ArrayList는 배열을 리스트화 시킨 동적 배열입니다. '배열을 리스트화 시켰다'는 말은 정적이고 고정된 배열의 특성을 동적인 리스트 형태로 바꾸어 유연하게 사용할 수 있도록 만들었다는 뜻입니다. 배열과 ArrayList의 특징을 비교하면 다음과 같습니다.

1. 배열의 특징

- 배열은 고정된 크기를 가집니다. 배열을 생성할 때 크기를 지정하며, 생성한 후에는 크기를 변경할 수 없습니다.
- 배열은 특정 인덱스를 통해 요소에 접근할 수 있는 빠른 랜덤 액세스(Random Access)를 제공합니다.

- 요소를 추가하거나 제거하려면 새로운 배열을 생성하거나 복사해야 하므로 유연성이 떨어집니다.

2. ArrayList의 특징

- **ArrayList**는 내부적으로 배열을 사용하지만, 크기를 동적으로 조정할 수 있는 구조를 제공합니다.
- 요소를 추가하거나 제거할 때 필요한 경우 **ArrayList**는 내부적으로 새로운 배열을 생성하여 데이터를 복사하는 방식으로 크기를 늘립니다.
- 사용자는 크기를 신경 쓰지 않고 요소를 추가(**add**)하거나 제거(**remove**)할 수 있습니다.

Java

```
List<String> list = new ArrayList<>();
list.add("a");
list.add("b");
list.isEmpty();
```

멘토 Tip

LinkedList는 **Deque**의 인터페이스도 구현하고, **Queue** 인터페이스도 구현하므로 용도가 다양합니다.

Java

```
Queue<String> queue = new LinkedList<>();
queue.add("queue");
queue.add("LL");

String peeked = queue.peek(); // queue
```



Set 인터페이스

세트는 수학의 집합을 생각하면 됩니다. 중복을 허용하지 않고, 기본적으로 순서를 보장하지 않습니다. 대표적인 구현체로는 **HashSet**이 있습니다. **HashSet**은 **hash**를 기반으로 저장하는 자료구조입니다.

Java

```
Set<String> set = new HashSet<>();
set.add("1");
set.add("1");
set.add("2");

set.size(); // 2
```

멘토 Tip

Hash는 데이터를 고정된 크기의 값으로 변환하는 함수 또는 알고리즘을 의미합니다.

- 고정된 크기를 가집니다.
- 빠른 연산을 제공합니다.
- 입력에 민감해 입력 데이터가 조금이라도 바뀌면 완전히 다른 해시 값이 생성됩니다.
- 단방향 함수로 설계되어 원래 데이터를 복원할 수 없습니다.
- 같은 해시 값을 가지는 해시 충돌이 일어날 수 있습니다.



Map 인터페이스

Map 인터페이스는 **Key - Value** 형태의 데이터를 저장하는 자료구조입니다. **key**에 대한 중복을 허용하지 않고, 기본적으로 순서를 보장하지 않고, 같은 키에 값을 저장하면 값이 덮어써지는 특징이 있습니다. 대표적인 구현체로 **HashMap**이 있습니다.

Java

```
Map<String, Integer> map = new HashMap<>();
map.put("hello", 0);
map.put("hello", map.get("hello") + 1)

map.get("hello") // 2
```

1분 퀴즈풀이

323p. 2번

① numbers는 int가 들어가는 배열이므로 Integer

② 리스트의 크기를 저장하므로 size

③ 요소를 삭제하므로 remove

326p. 3번

① Set<Integer>

② 요소의 포함 여부를 확인하므로 contains

③ 요소를 반복하므로 iterator

330p. 4번

② HashMap은 키와 값의 순서를 유지하지 않습니다.

301p. 셀프체크 풀이

Java

```
List<String> students = new ArrayList<>();
students.add("홍길동");
students.add("김길벗");
students.add("이코천");
students.add("홍길동");

Set<String> studentSet = new HashSet<>();
for(String student : students) {
    studentSet.add(student);
}

HashMap<String, Integer> scores = new HashMap<>();
scores.put("홍길동", 85);
scores.put("김길벗", 92);
scores.put("이코천", 78);
scores.put("강남순", 90);

for (String name: scores.keySet()) {
    System.out.println(name + ", " + scores.get(name));
}

String name = "홍길동";
if (scores.containsKey(name)) {
    System.out.println(name + ", " + scores.get(name));
}
```


Day 18

11장 입출력 스트림

안녕하세요. 오늘은 11장 <입출력 스트림>을 공부해봅니다.

🧐 공부할 내용(335 ~ 350p)

- 입출력 스트림이란
- 스트림의 종류
- 파일 입출력

1. 입출력 스트림이란

입출력은 프로그램이 컴퓨터 내/외부 장치와 데이터를 주고받는 것을 뜻합니다. 자바에서는 컴퓨터 내/외부와 통신할 때 스트림을 사용합니다. 스트림은 입력 스트림과 출력 스트림이 따로 존재합니다. 입력 스트림은 프로그램으로 들어오는 스트림을 의미하고, 출력 스트림은 프로그램에서 외부로 데이터를 내보내는 스트림을 의미합니다.

스트림은 큐와 마찬가지로 **FIFO**(선입선출) 형태로 동작하며 내부적으로 데이터를 일시적으로 저장하는 버퍼를 가지고 있습니다.

2. 스트림의 종류

스트림은 크게 바이트 스트림과 문자 스트림으로 나뉩니다.

바이트 스트림

바이트 스트림은 1바이트 단위로 데이터를 처리합니다. 이미지, 비디오, 바이너리 파일 등 비 텍스트 데이터를 처리 할 때 사용합니다.

주요 클래스는 다음과 같습니다.

- 입력: `InputStream`, `FileInputStream`, `BufferedInputStream`
- 출력: `OutputStream`, `FileOutputStream`, `BufferedOutputStream`

문자 스트림

문자 스트림은 데이터를 문자 단위로 처리합니다. 텍스트 데이터 (`ASCII`, `UTF-8`)를 처리할 때 사용합니다.

주요 클래스는 다음과 같습니다.

- 입력: `Reader`, `FileReader`, `BufferedReader`
- 출력: `Writer`, `FileWriter`, `BufferedWriter`

3. 파일 입출력

파일 입출력은 프로그램이 외부에 저장된 데이터를 읽거나, 프로그램의 데이터를 파일로 저장하는 작업입니다. 기본적으로 자바의 파일 입출력은 스트림 기반으로 작동합니다.

파일 읽기

`FileInputStream`을 활용해서 파일을 읽을 수 있습니다. 바이트 단위로 파일을 읽고, 오디오, 비디오 등의 파일을 처리합니다.

```
Java
try (FileInputStream fis = new FileInputStream("input.txt")) {
    int data;
    while ((data = fis.read()) != -1) { // 한 바이트씩 읽기
        System.out.print((char) data);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

파일 쓰기

`FileOutputStream`으로 바이트 단위로 파일에 데이터를 쓸 수 있습니다.

Java

```
import java.io.*;

public class FileOutputExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new
FileOutputStream("output.txt")) {
            String content = "Hello, File Output!";
            // 문자열을 바이트 배열로 변환하여 쓰기
            fos.write(content.getBytes());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

멘토 Tip

Java 7 이상에서는 `java.nio.file` 패키지의 `Path`, `Paths`, `Files` 클래스를 사용하여 파일 작업을 더 간단하게 처리할 수 있습니다.

파일 읽기

Java

```
import java.nio.file.*;
import java.io.IOException;
import java.util.List;

public class NioReadExample {
    public static void main(String[] args) {
        try {
            Path path = Paths.get("input.txt");
            // 모든 줄 읽기
            List<String> lines = Files.readAllLines(path);
            for (String line : lines) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
}  
}
```

파일 쓰기

Java

```
import java.nio.file.*;  
import java.io.IOException;  
  
public class NioWriteExample {  
    public static void main(String[] args) {  
        try {  
            Path path = Paths.get("output.txt");  
            String content = "Hello, NIO.2!";  
            Files.write(path, content.getBytes()); // 파일 쓰기  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



1분 퀴즈풀이

341p. 1번

- ① 스트림은 데이터를 FIFO로 처리합니다.
- ② 스트림은 데이터를 단방향으로 주고 받습니다.
- ③ 바이트 스트림은 바이너리 데이터를 처리할 때 사용합니다. 텍스트 데이터는 문자 스트림이 처리합니다.
- ④ 스트림을 사용한 후 반드시 닫아야 메모리 누수가 발생하지 않습니다.
- ⑤ `InputStream`은 바이트 입력 스트림을 위한 추상클래스입니다.

350p. 2번

`try-with-resources` 문법은 `try` 블록에 선언한 리소스를 블록 종료 시 자동으로 해제해주는 것입니다. 코드에서 ①, ② 문장이 하나로 합쳐지고, ④ 문장은 더 이상 필요 없게 됩니다.

Day 19

11장 입출력 스트림

안녕하세요. 어제에 이어 11장 <입출력 스트림> 나머지 부분을 공부해 봅시다.

 공부할 내용(351~357p)

- 표준 입출력 스트림

1. 표준 입출력 스트림

자바의 표준 입출력 스트림은 콘솔을 통해 데이터를 입력받거나 출력하는 기능을 제공합니다. 기본적으로 다음의 세 가지 표준 스트림을 지원하며, 이를 통해 사용자는 콘솔에서 데이터를 입력하거나 출력할 수 있습니다.

표준 입력

- `System.in` 을 사용합니다.
- 데이터를 콘솔에서 읽는 데 사용됩니다.
- 기본적으로 바이트 기반 입력 스트림(`InputStream`)으로 제공됩니다.

표준 입력은 기본 입력(`System.in`)을 사용하고, 데이터를 효율적으로 처리하려면 보조 클래스를 사용해야 합니다. `Scanner` 혹은 `BufferedReader`와 같은 보조클래스를 활용할 수 있습니다.

표준 출력

- `System.out`을 사용합니다.
- 데이터를 콘솔에 출력하는 데 사용됩니다.
- 기본적으로 바이트 기반 출력 스트림(`PrintStream`)으로 제공됩니다.

표준 출력은 많이 사용하는 `System.out.println()`과 같은 메서드를 사용하여 데이터를 출력할 수 있습니다.

표준 오류

- `System.err`를 사용합니다.
- 에러 메시지를 콘솔에 출력하는 데 사용됩니다.
- 기본적으로 바이트 기반 출력 스트림(`PrintStream`)으로 제공됩니다.

표준 오류는 에러 메시지를 출력하는 데 사용되며, 기본적으로 콘솔에 출력됩니다. 일반적인 출력인 `System.out`과는 분리되어 있어 에러와 일반 메시지를 구분할 수 있습니다.

1분 퀴즈풀이

354p. 3번

③ 에러 스트림은 `ErrStream`이 아닌 `System.out`과 동일한 `PrintStream` 타입입니다.

357p. 셀프체크

여기서는 책의 풀이와 다르게 **Java 7**의 `nio` 패키지를 사용해 작성해 봤습니다.

```
Java
import java.nio.file.*;
import java.io.*;
import java.nio.channels.FileChannel;

public class FileCopyWithBuffer {
    public static void main(String[] args) {
        Path inputPath = Paths.get("inputTest.txt");
        Path outputPath = Paths.get("outputTest.txt");
        // 파일을 쓰기 모드로 열고 없으면 생성
        try (FileChannel inputChannel = FileChannel.open(inputPath,
            StandardOpenOption.READ);
            FileChannel outputChannel = FileChannel.open(outputPath,
            StandardOpenOption.WRITE, StandardOpenOption.CREATE)) {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            while (inputChannel.read(buffer) > 0) {
                buffer.flip(); // 버퍼를 읽기 모드로 전환합니다.
                outputChannel.write(buffer);
                // 버퍼를 쓰기 모드로 초기화하여
            }
        }
    }
}
```

```
        // 다음 데이터를 읽을 준비를 합니다.  
        buffer.clear();  
    }  
} catch (IOException e) {  
    e.printStackTrace();  
}  
}  
}
```

Day 20

12장 람다 표현식

오늘은 마지막으로 12장 <람다 표현식>을 공부해 봅니다.

 공부할 내용(359~377p)

- 람다 표현식 사용하기
- 함수형 인터페이스 사용하기
- 자바에서 제공하는 함수형 인터페이스

1. 람다 표현식

함수형 프로그래밍은 함수를 중심으로 하는 프로그래밍 패러다임으로, 수학의 함수처럼 데이터의 입력을 받아 출력을 반환하는 순수 함수들을 조합해 작성하는 것을 목표로 합니다.

함수형 프로그래밍은 코드를 다음과 같은 3가지 기준으로 분류합니다.

1. 액션: 액션이란 결과가 부르는 시점, 횟수에 의존하는 것을 말합니다. `getCurrentTime()`과 같은 함수를 예로 들 수 있습니다. 부수 효과를 동반할 수 있습니다.
2. 계산: 계산은 입력 값을 계산해 출력하는 것으로 언제, 어디서 계산해도 같은 인풋으로 같은 아웃풋이 나와야 합니다.
3. 데이터: 데이터는 기록된 사실을 말합니다.

함수형 프로그램이 지향하는 것은 코드를 3가지 기준에 따라 나누고, 액션을 최소화하고, 최대한 많은 로직을 계산으로 이동하는 것입니다.

람다식은 함수를 하나의 표현식(expression)으로 나타내는 방법으로, 익명함수로 작성됩니다.

Java

```
() -> System.out.println("a")  
(a,b) -> a - b
```

멘토 Tip

익명 함수란 이름이 없는 함수를 의미합니다. 프로그래밍에서 보통 함수는 이름으로 호출하거나 참조할 수 있습니다. 그러나 익명 함수는 이름 없이 정의되며, 특정 작업을 수행하기 위해 일시적으로 사용되는 경우가 많습니다.

특징

1. 일반 함수와 달리 함수의 이름을 정의하지 않고 사용할 수 있습니다.
2. 주로 함수가 단 한 번만 사용되거나 간단한 작업을 수행할 때 사용됩니다.
3. 다른 변수에 저장되거나 다른 함수의 인자로 전달될 수 있습니다.



함수를 화살표 함수로 변경하면 다음과 같습니다.

Java

```
int add(int a, int b){  
    return a + b;  
}
```

```
(a, b) -> a + b
```

이처럼 람다식으로 작성하면 절대적인 코드량이 줄어들어 코드 작성에 수월합니다.

2. 함수형 인터페이스 사용하기

람다식을 사용하려면 변수의 자료형이 필요합니다. 이때 자료형 역할을 함수형 인터페이스가 수행합니다. 함수형 인터페이스는 한 개의 추상 메서드만 가지는 인터페이스입니다.

Java

```
@FunctionalInterface
interface Fun {
    someAbstractMethod();
}
```

이를 람다식으로 구현하면 다음과 같습니다. 가장 먼저 함수형 인터페이스를 선언합니다. 그리고 이를 구현하는 코드를 따로 `isEven()`이라는 함수로 분리해서 작성하거나 람다식으로 작성하면 됩니다.

Java

```
@FunctionalInterface
public interface Func {

    boolean isEven(int a);
}

public class Main {
    public static void main(String[] args) {
        Func func = Main::isEven;

    }
    private static boolean isEven(int n) {
        return n % 2 == 0;
    }
}

public class Main {
    public static void main(String[] args) {
        Func func = n -> n % 2 == 0;
        var result = func.isEven(15); // false
    }
}
```

3. 자바에서 제공하는 함수형 인터페이스

자바에서는 직접 구현하는 불편한 과정을 덜어주기 위해서 기본으로 제공하는 함수형 인터페이스가 있습니다.

Function

Function은 순수 함수를 구현할 수 있는 가장 기본적인 인터페이스입니다.

```
Java
public interface Function<T, R> {
    R apply(T t);
}
```

T를 매개변수로 받아 R을 반환하는 순수함수입니다. Function의 내부를 열어보면 apply() 메서드 말고, 기본 메서드가 2개 더 있습니다.

- compose()
- applyThen()

compose() 메서드는 '선 처리' 메서드로, compose()의 인자를 먼저 처리한 후 본 함수를 처리합니다.

```
Java
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        Function<Integer, Integer> add = number -> number + 10;
        Function<Integer, Integer> multiply = number -> number * 2;
        System.out.println(add.compose(multiply).apply(5)); // 20
    }
}
```

applyThen()은 compose()와 반대로 후 처리 메서드입니다.

Java

```
import java.util.function.Function;

public class Main {
    public static void main(String[] args) {
        Function<Integer, Integer> add = number -> number + 10;
        Function<Integer, Integer> multiply = number -> number * 2;
        System.out.println(add.andThen(multiply).apply(5)); // 30
    }
}
```

Predicate

Predicate는 어떤 인자를 받아서 **boolean** 값을 리턴하는 함수형 인터페이스입니다.

Java

```
public interface Predicate<T> {
    boolean test(T t);
}
```

Java

```
Predicate<Integer> even = number -> number % 2 == 0;
Predicate<Integer> odd = number -> number % 2 == 1;

System.out.println(even.test(5));
System.out.println(odd.test(5));
```

Runnable

Runnable 인터페이스는 스레드를 실행하기 위해 사용되는 함수형 인터페이스입니다. 이 인터페이스는 **run** 메서드를 구현하여 스레드에서 실행할 작업을 정의하는 데 사용됩니다. 스레드 기반 프로그래밍의 핵심 요소로, 실행 가능한 작업(=Runnable)을 스레드에 전달하여 비동기 실행을 수행합니다.

Java

```
public class LambdaRunnableExample {
```

```

public static void main(String[] args) {
    Thread thread = new Thread(() -> {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread running: " + i);
        }
    });
    thread.start();
}
}

```

멘토 Tip

스레드는 프로세스의 실행 단위입니다. 프로세스는 실행 중인 프로그램을 말합니다. 하나의 프로세스에 여러 개의 스레드가 존재할 수 있으며, 여러개의 스레드가 동시에 프로세스를 처리하는 것을 멀티스레딩라고 합니다.

멀티 스레딩에서 **Thread** 클래스와 **Runnable** 인터페이스는 비동기 처리의 핵심으로 반드시 필요합니다.



1분 퀴즈풀이

364p. 1번

Java

```
(s) -> str.length() >= 5
```

367p. 2번

Java

```

CalculatorInterface add = ①(a, b) -> a + b;
int result = add.②operate(10, 20);

```

374p. 3번

① run() 메서드는 매개변수를 받을 수 없습니다.

② **Predicate** 인터페이스는 **boolean** 값을 반환합니다.

③ **Function**의 **apply**는 결과값을 반환합니다.

④ **Runnable**은 **java.lang**에 포함되어 있어 임포트가 필요 없습니다.

⑤ **Runnable** 인터페이스는 스레드에서 실행할 작업을 정의하기 위해 사용하는 인터페이스입니다. 단 하나의 추상 메서드 **run()**을 가지며, 이를 구현하여 실행할 작업을 정의합니다.

⑥ **Thread** 클래스를 사용해 스레드를 생성할 때 반드시 **Runnable** 인터페이스를 구현할 필요는 없습니다. **Thread** 클래스는 자체적으로 **Runnable** 인터페이스를 구현하고 있으며, 직접 상속하여 **run()** 메서드를 오버라이드하는 방식으로 스레드를 생성할 수 있습니다.

377p. 셀프체크

책의 해설과 조금 다르게 작성했습니다. 책에서는 **filterList**라는 함수를 사용해서 직접 **predicate.test**를 호출하지만, 다음 코드만으로도 간편하게 구현할 수 있습니다.

Java

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Predicate;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        for (int i = 1; i <= 6; i++) {
            numbers.add(i);
        }
        Predicate<Integer> even = (a) -> a % 2 == 0;
        List<Integer> evenNumbers = numbers.stream()
            .filter(even)
            .toList();
        evenNumbers.forEach(System.out::println);
    }
}
```

