



**GDSC SKHU**

Backend 2nd Session : More about JAVA

# Contents

## Part 1.

1. More about JAVA
2. EOF
3. 예외 처리
4. 스레드
5. 어노테이션

## Part 2.

1. 제네릭
2. 컬렉션
3. 람다
4. 옵셔널
5. 스트림

# Part 1.

# 1. More about JAVA

저번 시간에는 **Java**의 기초 내용을 객체지향 관점에서 알아봤습니다.

이번 시간에는 **Spring Framework**를 통해 개발할 때 필요한  
**JAVA**의 심화 개념들을 공부해보도록 하겠습니다.

**EOF**는 “**End of file**”의 약자입니다.

이름에서도 알 수 있다시피 끝을 나타내는 개념으로,  
데이터 소스로부터 더 이상 읽어들일 수 있는 데이터가 없음을  
나타냅니다.

## 2. EOF

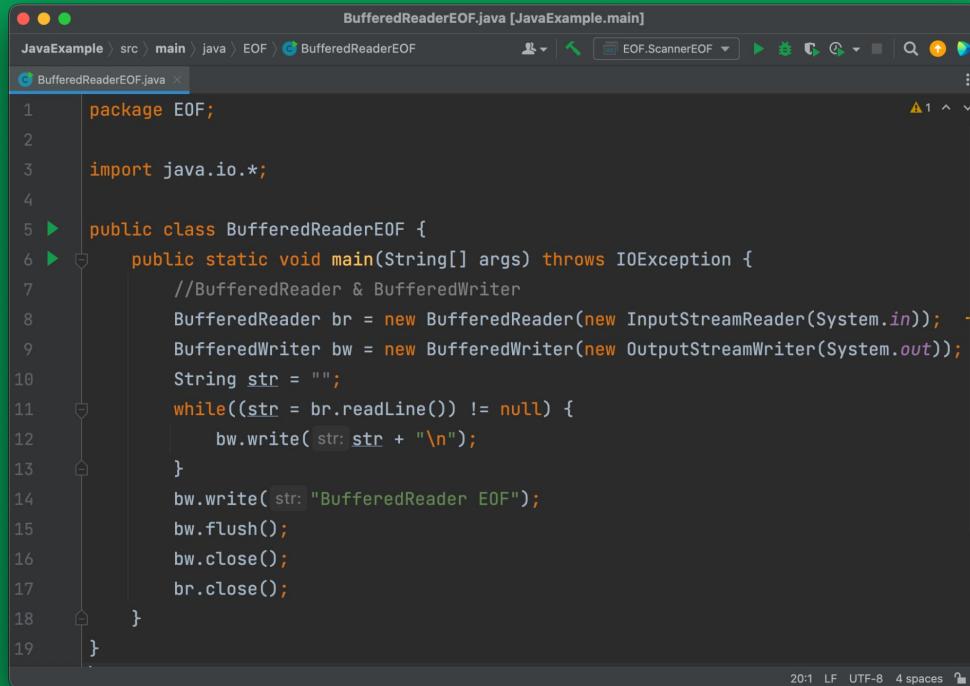
```
package EOF;

import java.util.Scanner;

public class ScannerEOF {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        while(scan.hasNext()) {
            System.out.println(scan.nextLine());
        }
        System.out.println("Scanner EOF");
    }
}
```

# Scanner 클래스를 사용할 때 EOF를 확인하는 방법

## 2. EOF



```
BufferedReaderEOF.java [JavaExample.main]
JavaExample > src > main > java > EOF > BufferedReaderEOF.java
BufferedReaderEOF.java

1 package EOF;
2
3 import java.io.*;
4
5 public class BufferedReaderEOF {
6     public static void main(String[] args) throws IOException {
7         //BufferedReader & BufferedWriter
8         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
9         BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));
10        String str = "";
11        while((str = br.readLine()) != null) {
12            bw.write(str + "\n");
13        }
14        bw.write("BufferedReader EOF");
15        bw.flush();
16        bw.close();
17        br.close();
18    }
19 }
```

**BufferedReader** 클래스를  
사용할 때 EOF를 확인하는 방법

**EOF**는 코딩테스트에서  
무한 루프 탈출 조건으로 사용하라고 나올 수도 있으니 알아두는게 좋습니다.

정말 쉬운 개념이지만 가끔 헷갈릴 수 있기 때문에 이 정도는 외워줍시다.  
물론 본래의 목적은 코테가 아닌 데이터의 끝을 확인하는데 있습니다.

예외(**Exception**)는 단순 오타로 발생하는 오류가 아닌  
프로그램 로직 실행 중이나 컴파일 중에 발생하는 오류들을  
말합니다.

JAVA 프로그래밍을 해보셨다면 자주 봤을 예외들이 있을 것입니다.

**NullPointerException...**

**ArrayIndexOutOfBoundsException...**

**ArithmetricException...**

**IOException... 등등...**

### 3. 예외 처리

보통 “예외 처리를 한다”함은 발생할 수 있는 예외를 예측하고 그에 대한 처리법을 미리 작성해두는 것을 의미합니다.

예외 처리에선 이 키워드들만 기억하면 됩니다.

**try, catch, finally, throw, throws**

### 3. 예외 처리

```
ExceptionExample.java [JavaExample.main]
ExceptionExample.java ×

1 package exception;
2
3 import java.util.Scanner;
4
5 public class ExceptionExample {
6     public static void main(String[] args) {
7         Scanner scan = new Scanner(System.in);
8         int A = scan.nextInt();
9         int B = scan.nextInt();
10        int result;
11
12        try {
13            result = A / B;
14            System.out.println(result);
15        } catch (Exception e) {
16            System.out.println("0으로는 나눌 수 없습니다.");
17        }
18    }
19}
```

The screenshot shows a Java code editor with the file "ExceptionExample.java" open. The code contains a main method that reads two integers from the user and attempts to divide them. If the divisor is zero, it catches the exception and prints an error message. The code editor interface includes tabs for "ExceptionExample.java" and "JavaExample.main", and various toolbars and status bars at the bottom.

**try catch**문  
예외 처리의 기초입니다.

**try**에 검사할 로직을,  
**catch**에 발생한 예외에 대한 처리를  
각각 작성합니다.

### 3. 예외 처리

```
ExceptionExample.java [JavaExample.main]
ExceptionExample.java

1 package exception;
2
3 import java.util.Scanner;
4
5 public class ExceptionExample {
6     public static void main(String[] args) {
7         Scanner scan = new Scanner(System.in);
8         int A = scan.nextInt();
9         int B = scan.nextInt();
10        int result;
11
12        try {
13            result = A / B;
14            System.out.println(result);
15        } catch (Exception e) {
16            System.out.println("0으로는 나눌 수 없습니다.");
17        } finally {
18            System.out.println("정말 재미있는 Java 스터디!");
19        }
20    }
21 }
```

**finally**문  
예외 처리와 상관 없이 항상  
실행합니다.

예외가 발생하더라도 실행되어야 하는  
코드를 작성할 때 사용합니다.

### 3. 예외 처리



The screenshot shows a Java IDE interface with the following details:

- Title Bar:** ThrowExample.java [JavaExample.main]
- Project Bar:** ava > exception > ThrowExample.ji
- Code Editor:** The file ThrowExample.java is open, displaying the following code:

```
package exception;

import java.util.Scanner;

public class ThrowExample {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("Java 스터디는 재미있습니까? (1 Yes / 2 No) : ");
        int choice = scan.nextInt();

        if(choice == 1) {
            throw new NotFunnyException();
        } else {
            System.out.println("지도 집에 가고 싶습니다.");
        }
    }

    class NotFunnyException extends RuntimeException {
        NotFunnyException() {
            super("공부는 항상 재미없습니다.");
        }
    }
}
```

The code includes several annotations and markers:

- Line 5: A green arrow pointing right indicates the start of the main method.
- Line 6: A green arrow pointing right indicates the start of the main method body.
- Line 11: A yellow box highlights the if-block where a new exception is thrown.
- Line 19: A yellow box highlights the class definition for NotFunnyException.
- Line 20: A yellow box highlights the constructor for NotFunnyException.
- Line 21: A yellow box highlights the call to super in the constructor.

**throw**문  
직접 예외를 만들고 발생시킬 수 있습니다.

내가 생각한 로직과 프로그램이  
다르게 실행되는 것을 방지할 수  
있습니다.

### 3. 예외 처리



# throws 문

로직 실행 중 발생한 예외의 처리를  
해당 로직 호출부에서 담당하도록  
합니다.

즉, 예외의 처리를 나중으로 미루는 것입니다.

스레드(**Thread**)는 하나의 프로세스 내에서  
여러 작업을 동시에 수행하는 실행 흐름의 최소 단위입니다.

즉, 한 번의 실행에서 여러 작업을 동시에 수행할 수 있게 해줍니다.  
기존의 우리가 작성한 코드는 한 번에 한가지 작업만 수행했습니다.

## 4. 스레드

The screenshot shows a Java code editor with the following code:

```
package thread;

public class ThreadExample extends Thread {
    private int order;

    public ThreadExample(int order) { this.order = order; }

    public void run() {
        System.out.println(this.order + "번째 스레드 시작합니다.");
        try {
            Thread.sleep( millis: 1000 );
        } catch (Exception e) {
        }
        System.out.println(this.order + "번째 스레드 종료합니다.");
    }

    public static void main(String[] args) {
        for(int i = 1; i <= 10; i++) {
            Thread thread = new ThreadExample(i);
            thread.start();
        }
        System.out.println("main 메소드 종료합니다.");
    }
}
```

The code includes several annotations and execution flow indicators:

- Annotations:
  - Line 1: `package` (green)
  - Line 3: `public class` (green)
  - Line 6: `public ThreadExample` (green)
  - Line 9: `int order` (yellow)
  - Line 10: `System.out.println` (green)
  - Line 12: `Thread.sleep` (green)
  - Line 15: `System.out.println` (green)
  - Line 18: `public static void main` (green)
  - Line 20: `for` loop (green)
  - Line 23: `System.out.println` (green)
- Execution Flow:
  - Line 3: A green triangle indicates the current method being executed.
  - Line 10: A blue circle with a red exclamation mark indicates a warning or error.
  - Line 18: A green triangle indicates the next method to be executed.
  - Line 23: A blue circle with a red exclamation mark indicates a warning or error.
- Other UI Elements:
  - Top bar: Shows the file name `ThreadExample.java`, the package `JavaExample.main`, and various tool icons.
  - Bottom bar: Shows the status `26:1 LF UTF-8 4 spaces`.

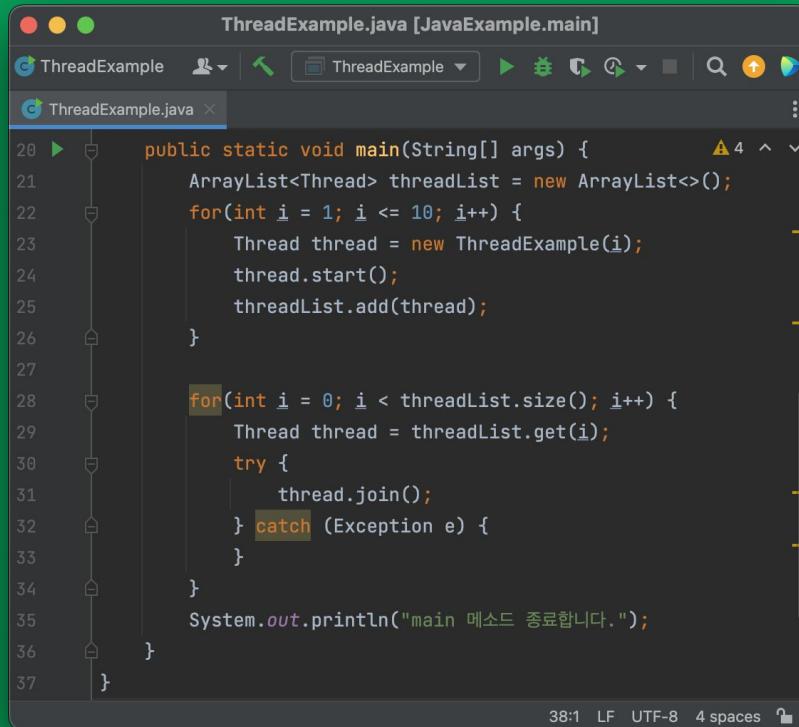
**thread**를 생성할 때 가장 간단한 방법은

**Thread** 클래스를 상속 받고,  
**run** 메소드를 오버라이딩한 후,  
스레드의 작업을 작성해주는 것입니다.

이 예제에서 중요한 점은 스레드는  
순서대로 실행되지 않으며, 종료 역시 순서대로 되지 않는다는  
것입니다.

심지어는 **main** 메소드마저 스레드보다 먼저 종료되어  
프로그램이 예상치 못한 결과를 낼을 수도 있습니다.

## 4. 스레드



The screenshot shows a Java IDE interface with the file `ThreadExample.java` open. The code implements a `main` method that creates a list of threads and starts them. It then iterates through the list, calling `join` on each thread to wait for them to finish before printing a message to the console.

```
public static void main(String[] args) {
    ArrayList<Thread> threadList = new ArrayList<>();
    for(int i = 1; i <= 10; i++) {
        Thread thread = new ThreadExample(i);
        thread.start();
        threadList.add(thread);
    }

    for(int i = 0; i < threadList.size(); i++) {
        Thread thread = threadList.get(i);
        try {
            thread.join();
        } catch (Exception e) {
        }
    }
    System.out.println("main 메소드 종료합니다.");
}
```

**main**이 먼저 종료되는 문제를  
해결하려면  
**join** 메소드를 사용하면 됩니다.

**join** 메소드는 자신을 호출한 스레드가  
종료될 때까지 대기하는 메소드입니다.

**Thread** 클래스를 통한 스레드 구현은  
치명적인 단점이 있습니다.

바로, **Thread** 클래스를 상속 받아야 하기 때문에  
다른 클래스를 상속 받을 수 없다는 점입니다.

**Thread** 클래스를 상속했을 때,  
역설적이게도 상속을 통한 여러 이점을 누릴 수 없게 됩니다.

이런 단점을 보완하기 위해 **Runnable** 인터페이스를  
사용해 스레드를 구현합니다.

## 4. 스레드

The screenshot shows the IntelliJ IDEA interface with the following details:

- Title Bar:** Shows the project name "JavaExample" and the current file "RunnableExample.java [JavaExample.main]".
- Status Bar:** Shows the build configuration as "RunnableExample" and the status "40:1 LF UTF-8 4 spaces".
- Code Editor:** Displays the Java code for "RunnableExample.java". The code uses multiple threads to print out messages. A yellow warning icon is visible in the top right corner of the editor area.

```
12     @Override
13     public void run() {
14         System.out.println(this.order + "번쨰 스레드 시작합니다.");
15         try {
16             Thread.sleep( millis: 1000 );
17         } catch (Exception e) {
18         }
19         System.out.println(this.order + "번쨰 스레드 종료합니다.");
20     }
21
22     public static void main(String[] args) {
23         ArrayList<Thread> threadList = new ArrayList<>();
24         for(int i = 1; i <= 10; i++) {
25             Thread thread = new Thread(new RunnableExample(i)); //바뀐 곳은 여기 뿐입니다.
26             thread.start();
27             threadList.add(thread);
28         }
29
30         for(int i = 0; i < threadList.size(); i++) {
31             Thread thread = threadList.get(i);
32             try {
33                 thread.join();
34             } catch (Exception e) {
35             }
36         }
37         System.out.println("main 메소드 종료합니다.");
38     }
39 }
```

**Runnable** 인터페이스를 상속  
받고  
**run** 메소드를 오버라이딩합니다.

사실 **Thread**로 구현하는 것과의  
차이점은 스레드 객체를 생성하는  
곳  
뿐입니다.

어노테이션(**Annotation**)은 주석이라는 뜻을 가진 단어입니다.  
어노테이션은 주석과 같이 데이터를 위한 데이터(메타 데이터)를  
알려줍니다.

즉, 어노테이션을 붙인 코드(데이터)에 대한 정보(데이터)를 줍니다.

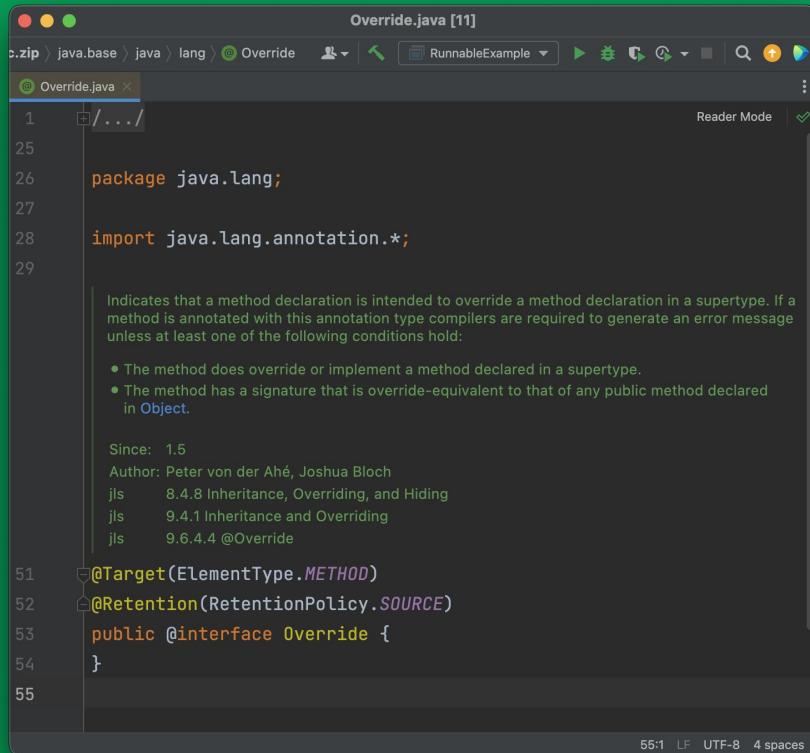
## 5. 어노테이션

어노테이션은 **@**를 통해 표시합니다.  
사실, 우리가 의도치 않게 자주 본 어노테이션이 하나 있습니다.

바로 **@Override** 어노테이션입니다.

**IDE**의 자동 완성 기능을 사용하다보면  
오버라이딩 한 메소드에 **@Override** 어노테이션이  
붙어서 작성되는 것을 확인할 수 있습니다.

# 5. 어노테이션



The screenshot shows a Java code editor with the file `Override.java` open. The code defines the `@Override` annotation. A tooltip for the `@Override` annotation is displayed, providing information about its purpose and usage conditions. The tooltip states: "Indicates that a method declaration is intended to override a method declaration in a supertype. If a method is annotated with this annotation type compilers are required to generate an error message unless at least one of the following conditions hold:" followed by two bullet points: "• The method does override or implement a method declared in a supertype." and "• The method has a signature that is override-equivalent to that of any public method declared in Object." Below the tooltip, the code shows the annotation definition with its target element and retention policy.

```
package java.lang;

import java.lang.annotation.*;

Indicates that a method declaration is intended to override a method declaration in a supertype. If a method is annotated with this annotation type compilers are required to generate an error message unless at least one of the following conditions hold:
• The method does override or implement a method declared in a supertype.
• The method has a signature that is override-equivalent to that of any public method declared in Object.

Since: 1.5
Author: Peter von der Ahé, Joshua Bloch
jls     8.4.8 Inheritance, Overriding, and Hiding
jls     9.4.1 Inheritance and Overriding
jls     9.6.4.4 @Override

@Override
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

**@Override** 내부를 살펴보면  
어노테이션 작성 방법을 알 수 있습니다.

다시 말해서 필요하다면  
우리가 직접 어노테이션을 만들 수  
있다는  
이야기입니다.

## 5. 어노테이션

```
CustomAnnotation.java [JavaExample.main]
@CustomAnnotation RunnableExample
CustomAnnotation.java

1 package annotation;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 @Target(ElementType.TYPE)
9 @Retention(RetentionPolicy.RUNTIME)
10 public @interface CustomAnnotation {
11     public 타입 메소드이름() default 값;
12 }
```

13:1 LF UTF-8 4 spaces

**@Target** 어노테이션과  
**@Retention** 어노테이션은  
어노테이션의 어노테이션,  
메타 어노테이션입니다.

각각 어노테이션의 적용 대상과  
어노테이션이 적용되는 범위를  
지정합니다.

### @Target 어노테이션의 파라미터 ElementType의 종류

- ElementType.PACKAGE : 패키지
- ElementType.TYPE : 타입
- ElementType.ANNOTATION\_TYPE : 어노테이션 타입
- ElementType.CONSTRUCTOR : 생성자
- ElementType.FIELD : 멤버 변수
- ElementType.LOCAL\_VARIABLE : 지역 변수
- ElementType.METHOD : 메소드
- ElementType.PARAMETER : 파라미터
- ElementType.TYPE\_PARAMETER : 파라미터 타입
- ElementType.TYPE\_USE : 타입

### @Retention 어노테이션의 파라미터 **RetentionPolicy**의 종류

- **RetentionPolicy.SOURCE** : 컴파일 전까지만 유효.
- **RetentionPolicy.CLASS** : 컴파일러가 클래스를 참조할 때까지 유효.
- **RetentionPolicy.RUNTIME** : 컴파일 이후에도 JVM에 의해 계속 참조가 가능.

**@Inherited, @Repeatable** 어노테이션은  
**@Target**과 **@Retention**과 마찬가지로 메타 어노테이션입니다.

각각 상속시킬 때, 반복해서 여러개 달 때 사용합니다.

또한, 어노테이션을 직접 작성해서 사용할 때는 지켜야 할 규칙이 있습니다.

- 요소의 타입은 기본형, **String**, **enum**, 어노테이션, **Class**만 허용된다.
- 괄호안에 매개변수를 선언할 수 없다.
- 예외를 선언할 수 없다.
- 요소의 타입을 매개변수로 정의할 수 없다.

# 5. 어노테이션

```
AnnotationExample.java [JavaExample.main]
AnnotationExample > src > main > java > annotation > AnnotationExample
```

```
AnnotationExample.java
```

```
1 package annotation;
2
3 import java.lang.reflect.Method;
4
5 public class AnnotationExample {
6     public static void main(String[] args) throws NoSuchMethodException {
7         Method m1 = AnnotationExample.class.getDeclaredMethod( name: "printAgeDefault");
8         CustomAnnotation customAnnotation1 = m1.getDeclaredAnnotation(CustomAnnotation.class);
9         System.out.println(customAnnotation1.myAge());
10
11         Method m2 = AnnotationExample.class.getDeclaredMethod( name: "printAgeCustom");
12         CustomAnnotation customAnnotation2 = m2.getDeclaredAnnotation(CustomAnnotation.class);
13         System.out.println(customAnnotation2.myAge());
14     }
15
16     @CustomAnnotation
17     public static void printAgeDefault() {
18
19     }
20
21     @CustomAnnotation(myAge = 28)
22     public static void printAgeCustom() {
23
24     }
25 }
```

26:1 LF UTF-8 4 spaces

직접 만든 어노테이션을  
메소드에 달아주고 값을 출력했습니다.

아무런 로직도 없는데  
서로 다른 값이 나오는 신기한 장면

## 5. 어노테이션

```
CustomAnnotation.java [JavaExample.main]
AnnotationExample
CustomAnnotation.java ×

1 package annotation;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 @Target(ElementType.METHOD)
9 @Retention(RetentionPolicy.RUNTIME)
10 public @interface CustomAnnotation {
11     int myAge() default 24;
12 }
13

13:1 LF UTF-8 4 spaces
```

직접 만든 어노테이션은 이런 형태를 띠고 있습니다.

어노테이션의 적용 대상은 메소드 적용 범위는 실행 시간 중입니다.

따라서 해당 어노테이션을 가지는 메소드는 값의 변경이 가능합니다.

직접 만든 어노테이션은 무분별하게 사용해서는 안됩니다.

내가 작성한 코드를 영원히 나만 보는게 아니라면  
다른 개발자가 내 커스텀 어노테이션이 무슨 역할을 하는지  
하나하나 확인해야하는 불상사가 일어날 수 있습니다.

# Part 2.

제네릭(**Generics**)은 클래스 내부에서 사용할 자료형을  
외부에서 사용자가 선언 시에 직접 지정할 수 있도록 한 개념입니다.

즉, 어떤 클래스가 자료형에 전혀 영향을 받지 않고  
모든 자료형에 같은 기능을 제공하도록 하는 일종의 다형성입니다.

우리는 **Java**를 사용하면서 이미 제네릭을 익숙하게 사용했습니다.

**Java** 내장 자료구조들을 사용할 때 <>안에 **Wrapper** 클래스를 통해 자료형을 지정하는 것을 분명 본 적 있을 것입니다.

**ex) Stack<Integer> myStack = new Stack<>();**

# 1. 제네릭

```
MyGenericClass.java [JavaExample.main]
main GenericExample
MyGenericClass.java

1 package generics;
2
3 public class MyGenericClass<T> {
4     private T value;
5     public T getValue() {
6         return value;
7     }
8     public void setValue(T value) {
9         this.value = value;
10    }
11 }
```

12:1 LF UTF-8 4 spaces

제네릭은 다음과 같이 사용할 수 있습니다.  
클래스나 인터페이스 이름 옆에 <>를 붙이고  
타입 파라미터를 지정해주면 됩니다.

클래스의 로직은 지정한 타입 파라미터를  
사용해 작성하면 됩니다.

이렇게 하면 **MyGenericClass**는 어떤 자료형이든 입력 받을  
수 있고  
불필요한 형변환이 일어나지 않게 됩니다.

또한, 자료형마다 여러 각각 클래스를 구현해주지 않아도 되어  
다형성의 이점을 온전히 가질 수 있게 됩니다.

# 1. 제네릭

```
GenericExample.java [JavaExample.main]
src > main > java > generics > GenericExample.java GenericExample
GenericExample.java

1 package generics;
2
3 public class GenericExample {
4     public static void main(String[] args) {
5         MyGenericClass<Integer> exampleInteger = new MyGenericClass<>();
6         MyGenericClass<String> exampleString = new MyGenericClass<>();
7
8         exampleInteger.setValue(1);
9         exampleString.setValue("Hello, generics!");
10        System.out.println(exampleInteger.getValue());
11        System.out.println(exampleString.getValue());
12    }
13 }
14

14:1 LF UTF-8 4 spaces
```

제네릭을 사용한 클래스를  
인스턴스화하려면 <>안에 **Wrapper** 클래스를  
적어주면 됩니다.

원래는 양쪽 <> 모두 **Wrapper** 클래스를  
작성해야했지만, **Java SE 7** 이후부터는  
“다이아몬드”라는 개념이 도입되어  
자료형을 명시하지 않아도 알아서 앞서 입력한  
자료형으로 설정됩니다.

제네릭에 사용할 수 있는 타입 파라미터의 종류는 다음과 같습니다.

- E : Element (Java 컬렉션 프레임워크에서 광범위하게 사용됨)
- K : Key
- N : Number
- T : Type
- V : Value
- ?: Wildcard (Unknown Type, 메소드와 변수에서만)

제네릭에는 타입 파라미터 관련해서 여러 활용법이 있습니다.

크게 **2**가지로 나눌 수 있는데, **Unbounded**와 **Bounded**로 나눌 수 있습니다.

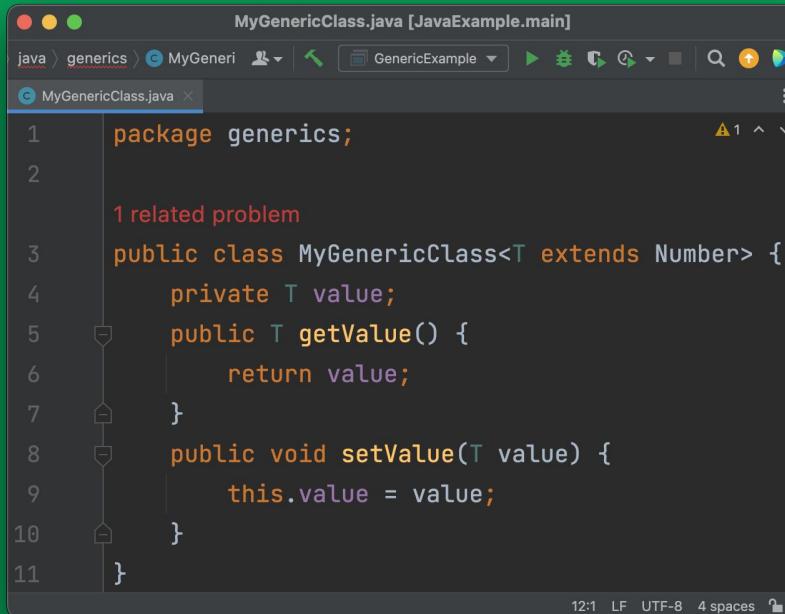
사실, 더 복잡하고 알아야 할게 많지만 당장에 제네릭을 사용하거나 제네릭을 사용한 코드를 이해하는데 문제 없는 정도로만 알아봅시다.

먼저 **Bounded** 방식의 활용법입니다.

지정한 클래스와 그 자식들만 가능하게 하는 **Upper Bound**와  
지정한 클래스와 그 부모들만 가능하게 하는 **Lower Bound**가 있습니다.

**Upper Bound**는 모든 파라미터 타입이 가능하지만  
**Lower Bound**는 **Wildcard**를 사용해서 구현할 수 있습니다.

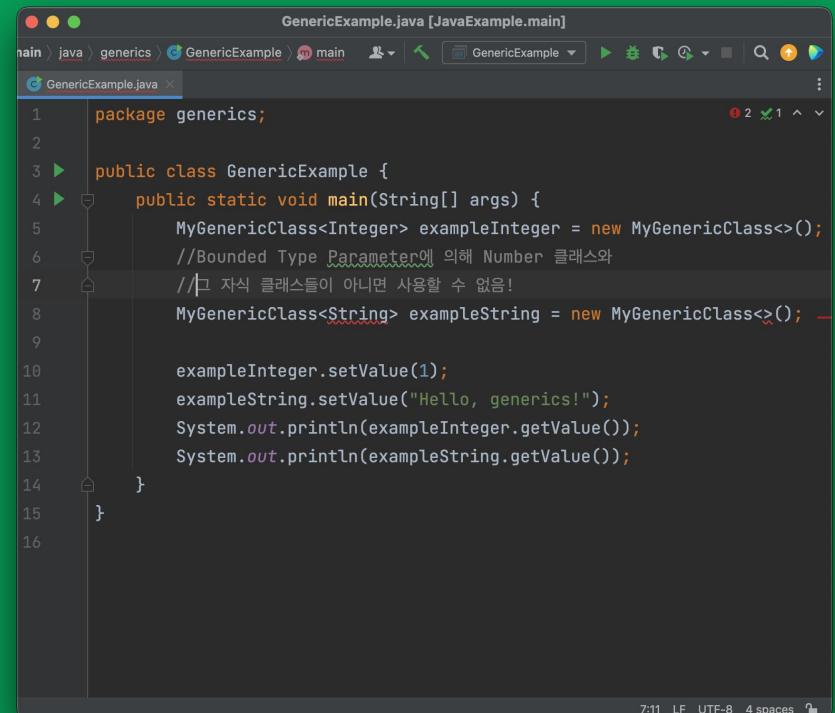
# 1. 제네릭



```
MyGenericClass.java [JavaExample.main]
java > generics > MyGeneri < GenericExample > MyGenericClass.java < GenericExample > MyGenericClass.java

1 package generics;
2
3 1 related problem
4
5 public class MyGenericClass<T extends Number> {
6     private T value;
7     public T getValue() {
8         return value;
9     }
10    public void setValue(T value) {
11        this.value = value;
12    }
13
14 }
```

12:1 LF UTF-8 4 spaces

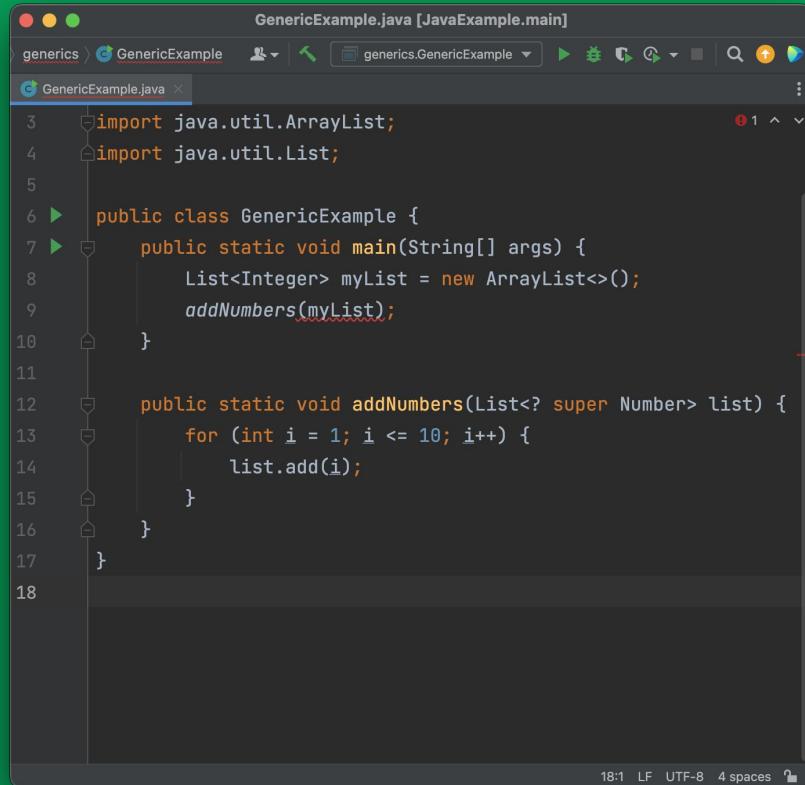


```
GenericExample.java [JavaExample.main]
main > java > generics > GenericExample < main < GenericExample > GenericExample.java < GenericExample > GenericExample.java

1 package generics;
2
3 public class GenericExample {
4     public static void main(String[] args) {
5         MyGenericClass<Integer> exampleInteger = new MyGenericClass<>();
6         //Bounded Type Parameter에 의해 Number 클래스와
7         //그 자식 클래스들이 아니면 사용할 수 없음!
8         MyGenericClass<String> exampleString = new MyGenericClass<>();
9
10        exampleInteger.setValue(1);
11        exampleString.setValue("Hello, generics!");
12        System.out.println(exampleInteger.getValue());
13        System.out.println(exampleString.getValue());
14    }
15
16 }
```

7:11 LF UTF-8 4 spaces

# 1. 제네릭



```
GenericExample.java [JavaExample.main]
generics > GenericExample generics.GenericExample
GenericExample.java x
import java.util.ArrayList;
import java.util.List;

public class GenericExample {
    public static void main(String[] args) {
        List<Integer> myList = new ArrayList<>();
        addNumbers(myList);
    }

    public static void addNumbers(List<? super Number> list) {
        for (int i = 1; i <= 10; i++) {
            list.add(i);
        }
    }
}
```

18:1 LF UTF-8 4 spaces

**extends** 키워드를 통해 **Bound**할 때는  
지정한 클래스의 부모 클래스로 객체를 생성한 후,  
사용하면 오류가 발생하고

반대로 **super** 키워드를 통해 **Bound**할 때는  
자식 클래스로 객체를 생성하고 사용할 때 오류가  
발생합니다.

이것이 **Upper Bound**와 **Lower Bound**입니다.

이렇게 **Bound** 개념이 필요한 이유는  
제네릭의 특성상 데이터 타입에 대한 불확실성이 있기 때문입니다.

데이터 타입의 불확실성은 다시 말해서 의도와는 전혀 다른 자료형을  
가진

객체가 만들어질 수도 있고, 그에 따른 잡하지 않는 버그들이  
쏟아져 나올 수도 있기 때문입니다.

컬렉션은 객체의 그룹을 표현하는 객체입니다.  
객체들을 모아두는 객체 자료형이라고 생각하면 쉽습니다.

Java는 JCF(**Java Collections Framework**)라고 불리는  
**Collection**들을 모아둔 프레임워크를 제공합니다.

컬렉션은 **java.util.Collection** 인터페이스를 상속 받는  
여러 인터페이스들이 있습니다.

대표적으로 **List**, **Set**, **Queue**가 있습니다.  
**Map**도 **Collection**으로 분류되지만 엄밀히 따지면  
**Collection** 인터페이스를 상속 받지 않아 조금 다릅니다.

컬렉션에는 정말 다양한 인터페이스들이 있고  
이를 구현하는 구현 클래스들도 정말 많습니다.

이번 시간에 전부 알아보는 것은 무리가 있고  
자주 사용하는 것들 위주로 알아보겠습니다.

## 2. 컬렉션

```
CollectionsExample.java [JavaExample.main]
CollectionsE generics.GenericExample ▶
CollectionsExample.java ×
import java.util.*;
public class CollectionsExample {
    public static void main(String[] args) {
        //List
        List<Integer> myArrayList = new ArrayList<>();
        List<Integer> myLinkedList = new LinkedList<>();

        //Set
        Set<Integer> myHashSet = new HashSet<>();
        Set<Integer> myTreeSet = new TreeSet<>();

        //Queue
        //Queue를 상속받은 Deque 인터페이스는 정말 많이 사용합니다.
        Queue<Integer> myPriorityQueue = new PriorityQueue<>();
        Queue<Integer> myArrayDeque = new ArrayDeque<>();

        //Map
        Map<String, Integer> myHashMap = new HashMap<>();
    }
}
```

사실 컬렉션을 사용하는 가장 큰 이유는 이미 구현되어 있는 자료구조를 사용하기 위함입니다.

실무에서 정말 특수한 경우가 아닌 이상 자료구조를 하나하나 직접 구현하는 경우는 없습니다.

**Hashing** 기법, **Tree**, 우선순위 큐 같은 세세한 자료구조와 알고리즘은 스스로 공부합시다!

람다, 정확히는 람다식(**Lambda Expression**)은  
함수를 하나의 식(**Expression**)으로 나타낸 것입니다.

식으로 나타내는 것이기 때문에 따로 이름이 필요 없는  
익명 함수(**Anonymous Function**)이며, 변수처럼 사용도 가능합니다.  
변수처럼 사용 가능하니 당연히 매개 변수로 전달도 가능합니다.

람다식의 기본 형태는 다음과 같습니다.  
( 매개 변수 ) -> { 함수 식; }

ex) (int a) -> { System.out.println(a); }

람다식은 극한의 효율을 추구하기 때문에 생략할 수 있는 표현이 많습니다.

매개변수 자료형이 코드 내에서 유추 가능하고 함수 식도 하나면 자료형, **return** 키워드를 생략할 수 있습니다.

심지어는 각각 하나씩이라고 보장되면 괄호들도 제거할 수 있습니다.

(int a) -> { System.out.println(a); }

==

a -> System.out.println(a);

람다를 사용하려면 함수형 인터페이스를 사용해야합니다.  
함수형 인터페이스는 멤버 메소드를 단 하나만 가지고 있어  
함수처럼 사용할 수 있는 인터페이스를 말합니다.

**@FunctionalInterface** 어노테이션을 인터페이스 위에 적어주면  
해당 인터페이스는 멤버로 함수 하나만을 가질 수 있게 강제됩니다.

### 3. 람다

```
LambdaExample.java [JavaExample.main]
LambdaExample.java

1 package lambda;
2
3 import java.util.Scanner;
4
5 public class LambdaExample {
6     public static void main(String[] args) {
7         Scanner scan = new Scanner(System.in);
8         int num1 = scan.nextInt();
9         int num2 = scan.nextInt();
10
11         //메소드 오버라이딩을 간편하게 해줍니다.
12         PrintResult pr = (a, b) -> a > b ? a : b;
13         System.out.println(pr.printResult(num1, num2));
14     }
15
16     //함수형 인터페이스
17     @FunctionalInterface
18     interface PrintResult {
19         int printResult(int a, int b);
20     }
}

22:1 LF UTF-8 4 spaces
```

코드를 살펴보면 함수형 인터페이스에 있는 멤버 메소드를 람다식으로 아주 간단하게 오버라이딩하고 원하는 결과를 출력하는 함수로 사용했습니다.

람다식의 가장 큰 장점이 바로 오버라이딩을 아주 쉽게 할 수 있다는 점입니다.

### 3. 람다

The screenshot shows an IDE window with the title "LambdaWithCollections.java [JavaExample.main]". The code is as follows:

```
1 package Lambda;
2
3 import java.util.*;
4
5 public class LambdaWithCollections {
6     public static void main(String[] args) {
7         List<String> myList = new ArrayList<>();
8         myList.add("Java");
9         myList.add("Python");
10        myList.add("C");
11
12        //원래라면 이렇게 해야하는데...
13        Collections.sort(myList, new Comparator<String>() {
14            @Override
15            public int compare(String o1, String o2) {
16                return o2.compareTo(o1);
17            }
18        });
19        for(String s : myList) {
20            System.out.println(s);
21        }
22        System.out.println();
23    }
24}
```

The code defines a class `LambdaWithCollections` with a `main` method. It creates a `List` of strings and adds "Java", "Python", and "C". Instead of using the standard `Collections.sort` method, it uses a lambda expression to create a `Comparator` that sorts in descending order (using `o2.compareTo(o1)`). Finally, it prints each string in the list.

이 프로그램은 컬렉션인 **List**에 담긴 문자열들을 사전순의 역순으로 정렬하는 프로그램입니다.

**sort** 메소드를 통해 정렬해야하는데,  
**Comparator** 클래스의 **compare** 메소드를  
오버라이딩해서 원하는 정렬법으로 정렬할 수 있습니다.

그런데 코드가 너무 복잡해보입니다...

### 3. 람다

```
LambdaWithCollections.java [JavaExample.main]
LambdaWi LambdaWithCollections.java

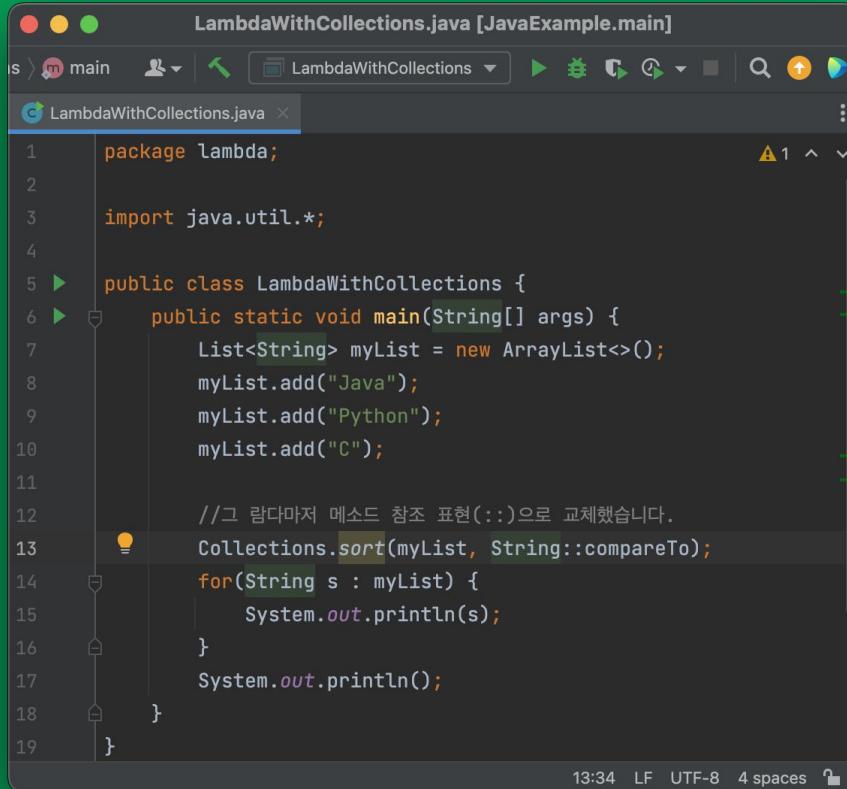
1 package lambda;
2
3 import java.util.*;
4
5 public class LambdaWithCollections {
6     public static void main(String[] args) {
7         List<String> myList = new ArrayList<>();
8         myList.add("Java");
9         myList.add("Python");
10        myList.add("C");
11
12        //그 람다마저 메소드 참조 표현(::)으로 교체했습니다.
13        Collections.sort(myList, (s1, s2) -> s1.compareTo(s2));
14        for(String s : myList) {
15            System.out.println(s);
16        }
17        System.out.println();
18    }
19 }
```

20:1 LF UTF-8 4 spaces

거추장스러웠던 코드들이 사라지고  
아주 간단하게 두 요소를 비교하는 식만  
남았습니다.

이렇게 람다를 사용하면  
메소드 오버라이딩이 간편해지고  
다형성의 효율을 극대화할 수 있습니다.

### 3. 람다



The screenshot shows an IDE window with the file `LambdaWithCollections.java` open. The code demonstrates the use of lambda expressions in Java:

```
1 package lambda;
2
3 import java.util.*;
4
5 public class LambdaWithCollections {
6     public static void main(String[] args) {
7         List<String> myList = new ArrayList<>();
8         myList.add("Java");
9         myList.add("Python");
10        myList.add("C");
11
12        //그 람다마저 메소드 참조 표현(():)으로 교체했습니다.
13        Collections.sort(myList, String::compareTo);
14        for(String s : myList) {
15            System.out.println(s);
16        }
17        System.out.println();
18    }
19 }
```

The code defines a class `LambdaWithCollections` with a `main` method. It creates a `List` of strings containing "Java", "Python", and "C". It then sorts the list using the `String::compareTo` lambda expression. Finally, it prints each string in the list followed by a new line.

이마저도 더 짧게 만들어버릴 수도 있습니다.  
메소드 참조 표현식(::)을 사용하면 코드는  
줄이고  
람다와 완전히 같은 동작을 합니다.

주의할 점은 (인스턴스의 자료형)::(메소드)  
형식으로  
작성해야한다는 점입니다.

옵셔널(Optional)은 Java 8 이후로 등장한  
**NullPointerException**을 더 효과적으로 처리할 수 있는 클래스입니다.

예외 처리나 조건문으로도 처리할 수 있지만  
코드가 너무 길어지고 재사용은 거의 불가능할 정도로  
**NPE**가 발생할 상황은 다양합니다.

옵셔널 객체는 기본적으로 **Wrapper** 클래스입니다.  
**Integer**나 **Double**처럼 내부의 값을 감싸 클래스의 내부 메소드를  
사용할 수 있도록 해줍니다.

즉, 옵셔널은 프로그래머가 사용하는 인스턴스를  
**NPE**로부터 보호하고 **null** 관련 다양한 메소드를 사용할 수 있도록  
도와주는 클래스입니다.

## 4. 옵션

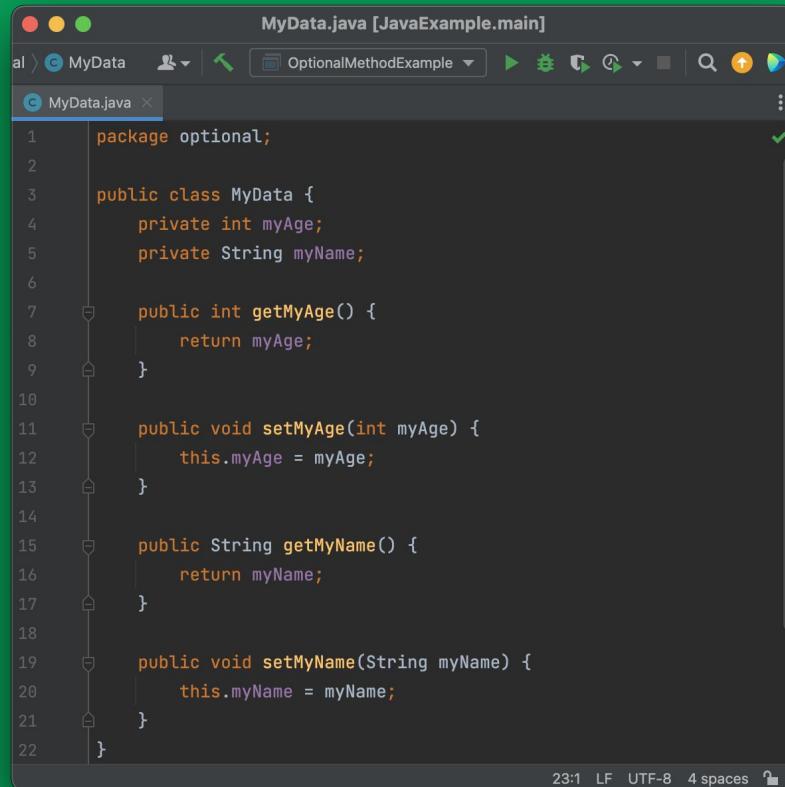
The screenshot shows an IDE interface with the following details:

- Title Bar:** OptionalExample.java [JavaExample.main]
- Toolbar:** Includes standard icons for file operations, search, and navigation.
- Project Explorer:** Shows JavaExample > src > m.
- Code Editor:** Displays the content of `OptionalExample.java`. The code demonstrates the use of the `Optional` class from `java.util`. It includes examples of creating `Optional` instances for non-null and nullable strings, and printing them to `System.out`.
- Status Bar:** Shows the current time as 20:1, line count as LF, encoding as UTF-8, and code style as 4 spaces.

```
1 package optional;
2
3 import java.util.Optional;
4
5 public class OptionalExample {
6     public static void main(String[] args) {
7         String nullableStr = null;
8         String neverNullStr = "nullable이 안됩니다!";
9
10        //nullable이 아닌 값이 넘어오면 해당 값을 가진 Optional 인스턴스 반환
11        Optional<String> optNotNull = Optional.of(neverNullStr);
12
13        //nullable이 넘어오면 빈 Optional 반환 (nullable이 아닌 값이 넘어오면 of와 같은 동작)
14        Optional<String> optNullable = Optional.ofNullable(nullableStr);
15
16        System.out.println(optNotNull);
17        System.out.println(optNullable);
18    }
19}
```

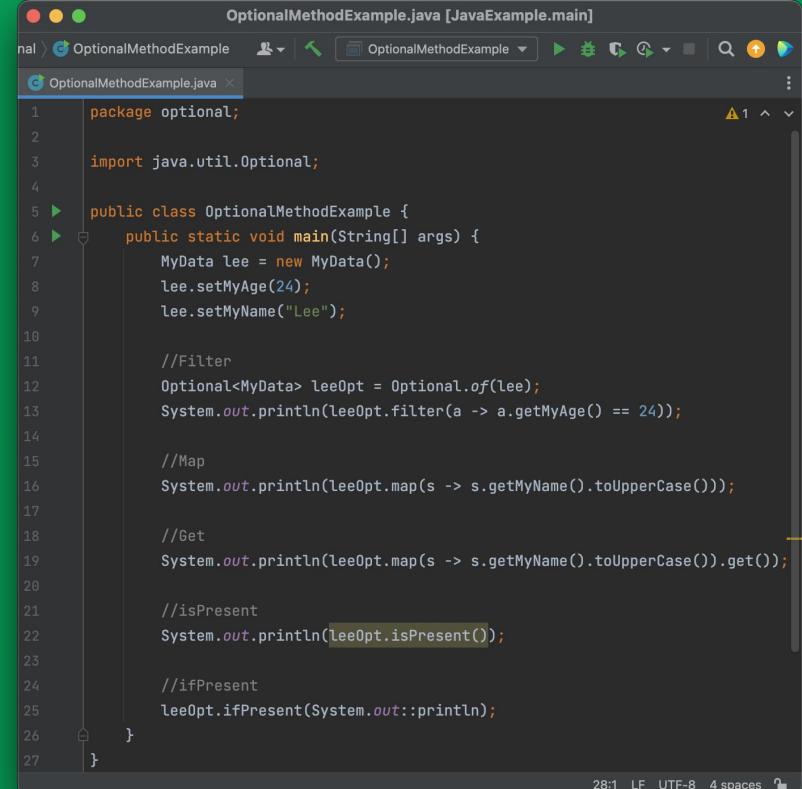
옵셔널은 메소드 체이닝을 활용합니다.  
먼저 **of**와 **ofNullable** 메소드를 통해  
인스턴스의 **null** 가능 여부를  
결정합니다.

# 4. 옵셔널



```
MyData.java [JavaExample.main]
MyData.java

1 package optional;
2
3 public class MyData {
4     private int myAge;
5     private String myName;
6
7     public int getMyAge() {
8         return myAge;
9     }
10
11    public void setMyAge(int myAge) {
12        this.myAge = myAge;
13    }
14
15    public String getMyName() {
16        return myName;
17    }
18
19    public void setMyName(String myName) {
20        this.myName = myName;
21    }
22 }
```



```
OptionalMethodExample.java [JavaExample.main]
OptionalMethodExample.java

1 package optional;
2
3 import java.util.Optional;
4
5 public class OptionalMethodExample {
6     public static void main(String[] args) {
7         MyData lee = new MyData();
8         lee.setMyAge(24);
9         lee.setMyName("Lee");
10
11         //Filter
12         Optional<MyData> leeOpt = Optional.of(lee);
13         System.out.println(leeOpt.filter(a -> a.getMyAge() == 24));
14
15         //Map
16         System.out.println(leeOpt.map(s -> s.getMyName().toUpperCase()));
17
18         //Get
19         System.out.println(leeOpt.map(s -> s.getMyName().toUpperCase()).get());
20
21         //isPresent
22         System.out.println(leeOpt.isPresent());
23
24         //ifPresent
25         leeOpt.ifPresent(System.out::println);
26     }
27 }
```

스트림(**Stream**)은 옵셔널과 마찬가지로 **Java 8**에서 추가되었습니다.

스트림 등장 이전까지는 배열이나 컬렉션에 저장된 데이터를 처리할 때 **for**문이나 **forEach**문 같은 반복문을 사용해야했습니다.

그러나 이는 불필요하게 코드가 중복되고 가독성이 떨어졌습니다.

그래서 등장한 것이 스트림입니다.

즉, 스트림은 연속적인 데이터에 대한 처리를 편리하게 하는 기술입니다.

스트림은 옵셔널과 마찬가지로 메소드 체이닝을 통해  
배열이나 컬렉션의 데이터를 스트림으로 만들고(생성하기)  
스트림으로 들어온 데이터들을 가공해서(가공하기)  
원하는 형태로 결과물을 만듭니다.(결과 만들기)

이 메소드 체이닝 과정에서 사용되는 메소드는 다양합니다.

모든 메소드를 이번 시간에 배우는 것은 무리일 것 같습니다.  
자주 사용하는 메소드 위주로 예제를 작성해보겠습니다.

더 많은 메소드들은 우리의 친구 구글에게 물어봅시다.

## 5. 스트림

The screenshot shows a Java IDE interface with the following details:

- Title Bar:** StreamExample.java [JavaExample.main]
- File Path:** JavaExample > src > main > java > stream
- Code Editor:** StreamExample.java
- Code Content:** StreamExample.java code demonstrating Java Streams.
- Toolbars and Status Bar:** Includes icons for file operations, search, and navigation, along with status indicators for line count (561), character count (LF, UTF-8), and code style (4 spaces).

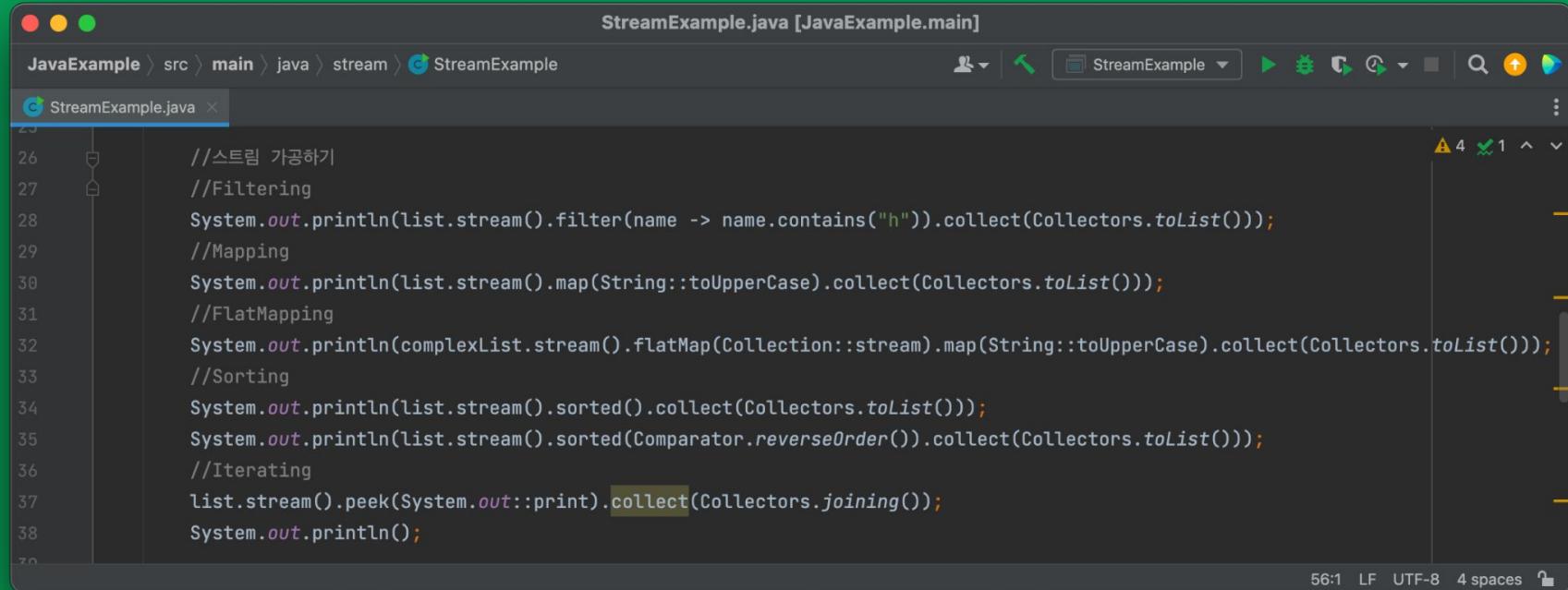
```
1 package stream;
2
3 import ...
4
5 public class StreamExample {
6     public static void main(String[] args) {
7         List<String> list = Arrays.asList("lee", "han", "cho");
8         List<String> concatTarget = Arrays.asList("GDSC CORE");
9         List<List<String>> complexList = Arrays.asList(
10             Arrays.asList("core", "lee"),
11             Arrays.asList("core", "han"),
12             Arrays.asList("core", "cho")
13         );
14     }
15
16     List<Integer> numList = Arrays.asList(1, 2, 3, 4, 5);
17
18
19     //스트림 생성하기
20     Stream<String> listStream = list.stream();
21     IntStream intStream = IntStream.range(1, 5); // [1, 2, 3, 4]
22     LongStream longStream = LongStream.rangeClosed(1, 5); // [1, 2, 3, 4, 5]
23     Stream<String> unitedStream = Stream.concat(concatTarget.stream(), listStream);
```

스트림은 여러 생성 방법이 있습니다.

컬렉션이나 배열을 **.stream** 메소드로  
스트림으로 만드는 방법,  
기본형 스트림을 직접 생성하는 방법,  
두 스트림을 합쳐 새 스트림을 만드는  
방법

이외에도 다양한 생성법이 있습니다.

# 5. 스트림



```
StreamExample.java [JavaExample.main]
JavaExample > src > main > java > stream > StreamExample
StreamExample.java ×

26 //스트림 가공하기
27 //Filtering
28 System.out.println(list.stream().filter(name -> name.contains("h")).collect(Collectors.toList()));
29 //Mapping
30 System.out.println(list.stream().map(String::toUpperCase).collect(Collectors.toList()));
31 //FlatMapping
32 System.out.println(complexList.stream().flatMap(Collection::stream).map(String::toUpperCase).collect(Collectors.toList()));
33 //Sorting
34 System.out.println(list.stream().sorted().collect(Collectors.toList()));
35 System.out.println(list.stream().sorted(Comparator.reverseOrder()).collect(Collectors.toList()));
36 //Iterating
37 list.stream().peek(System.out::print).collect(Collectors.joining());
38 System.out.println();

56:1 LF UTF-8 4 spaces
```

스트림 가공 방법에도 여러 가지가 있습니다.  
가장 많이 사용하는 것은 필터링과 매팅입니다.

## 5. 스트림

```
StreamExample.java [JavaExample.main]
java > stream > StreamExample StreamExample.java StreamExample.java ...
StreamExample.java ×
40 //스트림 결과 만들기
41 //Calculating
42 long count = intStream.count();
43 System.out.println(count);
44 long sum = longStream.sum();
45 System.out.println(sum);
46 //Reduction
47 System.out.println(numList.stream().reduce( identity: 0, Integer::sum));
48 //Collecting
49 System.out.println(list.stream()
50     .sorted(Comparator.reverseOrder())
51     .collect(Collectors.toList()))
52 );
53 //Matching
54 System.out.println(list.stream().allMatch(s -> s.length() == 3));
55 //Iterating
56 list.stream().forEach(System.out::println);
```

만들고 가공한 스트림의 결과를  
도출하는  
방법에도 여러가지가 있습니다.

# Q & A

## 과제

**PPT**의 예제 코드들을 따라쳐보고  
주석으로 해당 코드에 대해 설명하기

복불하지 말아주세요...

수고하셨습니다.