

1. 람다가 필요한 이유

#1.인강/0.자바/7.자바-고급3편

- /프로젝트 환경 구성
- /람다가 필요한 이유1
- /람다가 필요한 이유2
- /람다가 필요한 이유3
- /함수 vs 메서드
- /람다 시작

프로젝트 환경 구성

자바 입문편에서 인텔리제이 설치, 선택 이유 설명

프로젝트 환경 구성에 대한 자세한 내용은 자바 입문편 참고

여기서는 입문편을 들었다는 가정하에 설정 진행

인텔리제이 실행하기

New Project



- New Project를 선택해서 새로운 프로젝트를 만들자

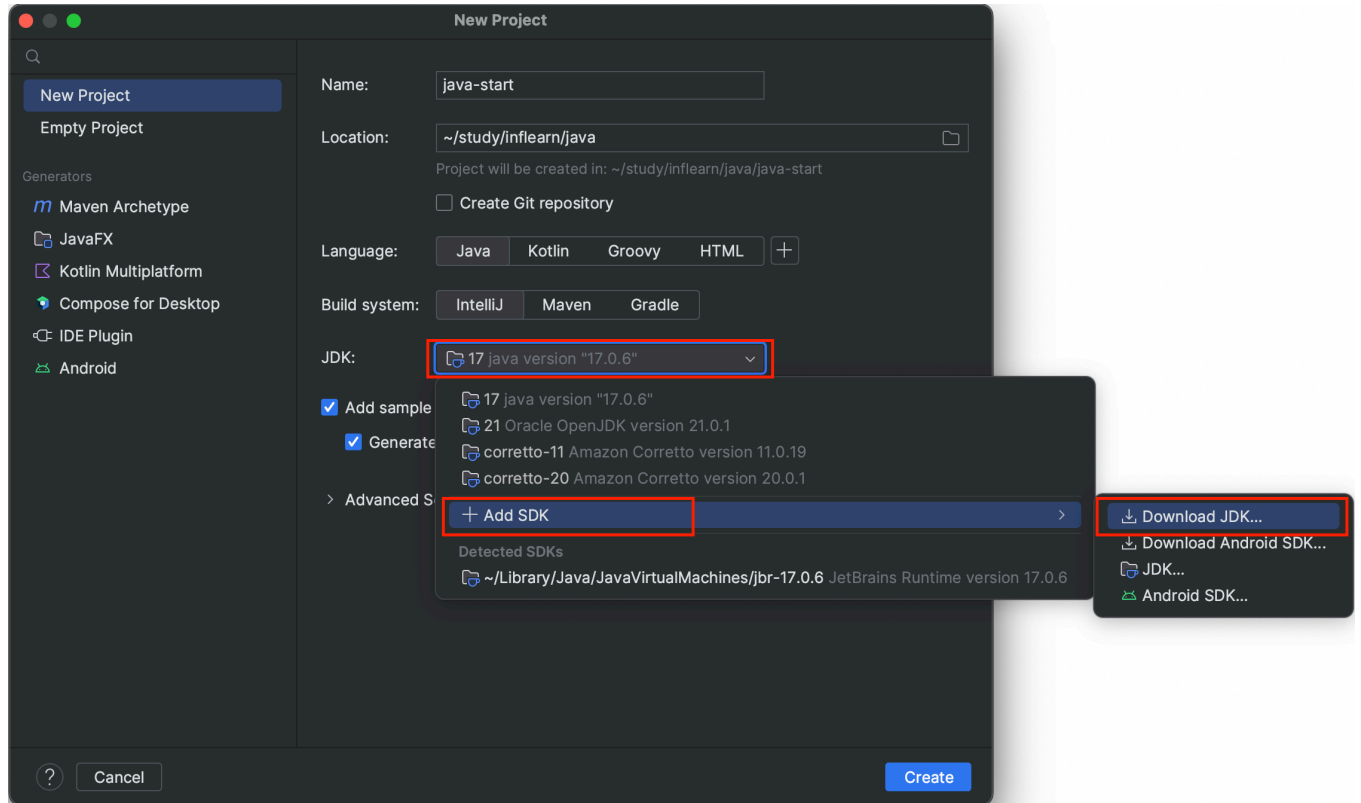
New Project 화면

- Name:
 - 자바 입문편 강의: java-start
 - 자바 기본편 강의: java-basic
 - 자바 중급1편 강의: java-mid1
 - 자바 중급2편 강의: java-mid2
 - 자바 고급1편 강의: java-adv1
 - 자바 고급2편 강의: java-adv2
 - 자바 고급3편 강의: **java-adv3**
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: **자바 버전 21 이상(주의!)**

- Add sample code 선택

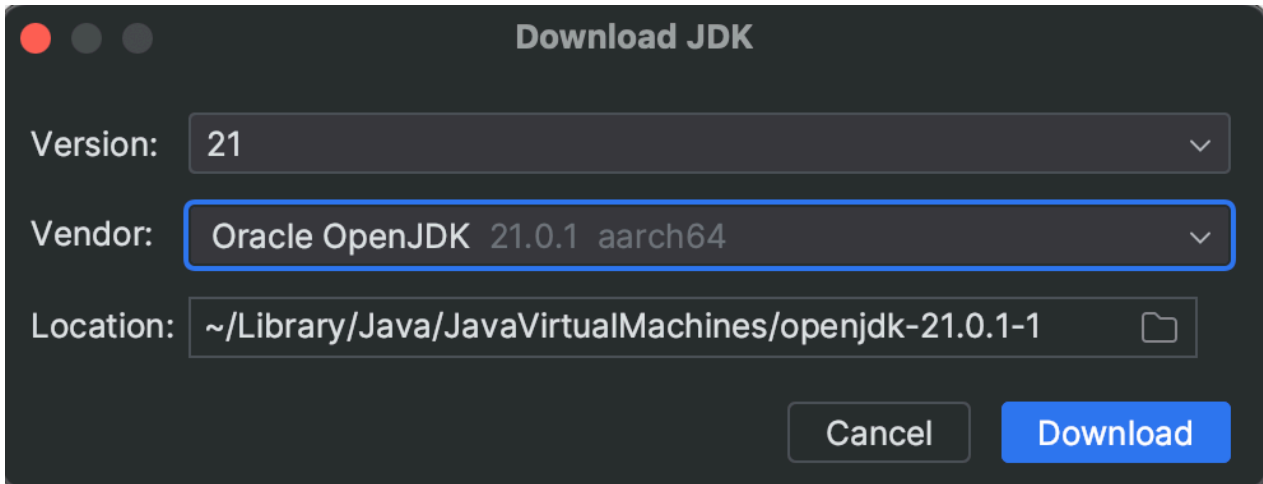
JDK 다운로드 화면 이동 방법

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



- Name:
 - 자바 입문편 강의: java-start
 - 자바 기본편 강의: java-basic
 - 자바 중급1편 강의: java-mid1
 - 자바 중급2편 강의: java-mid2
 - 자바 고급1편 강의: java-adv1
 - 자바 고급2편 강의: java-adv2
 - 자바 고급3편 강의: **java-adv3**

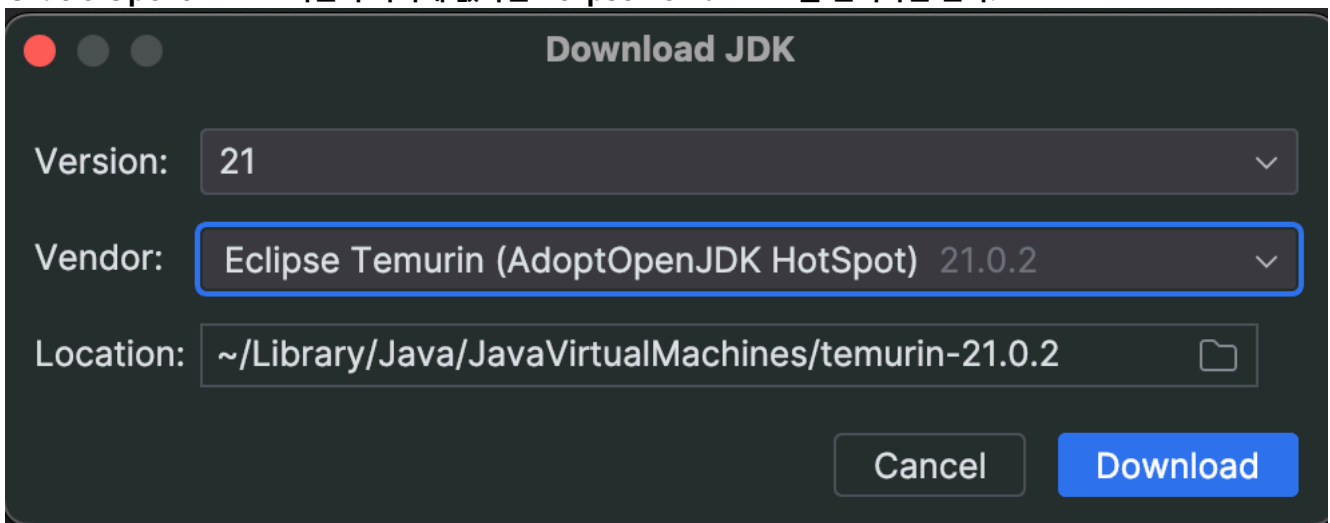
JDK 다운로드 화면



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

주의 - 변경 사항

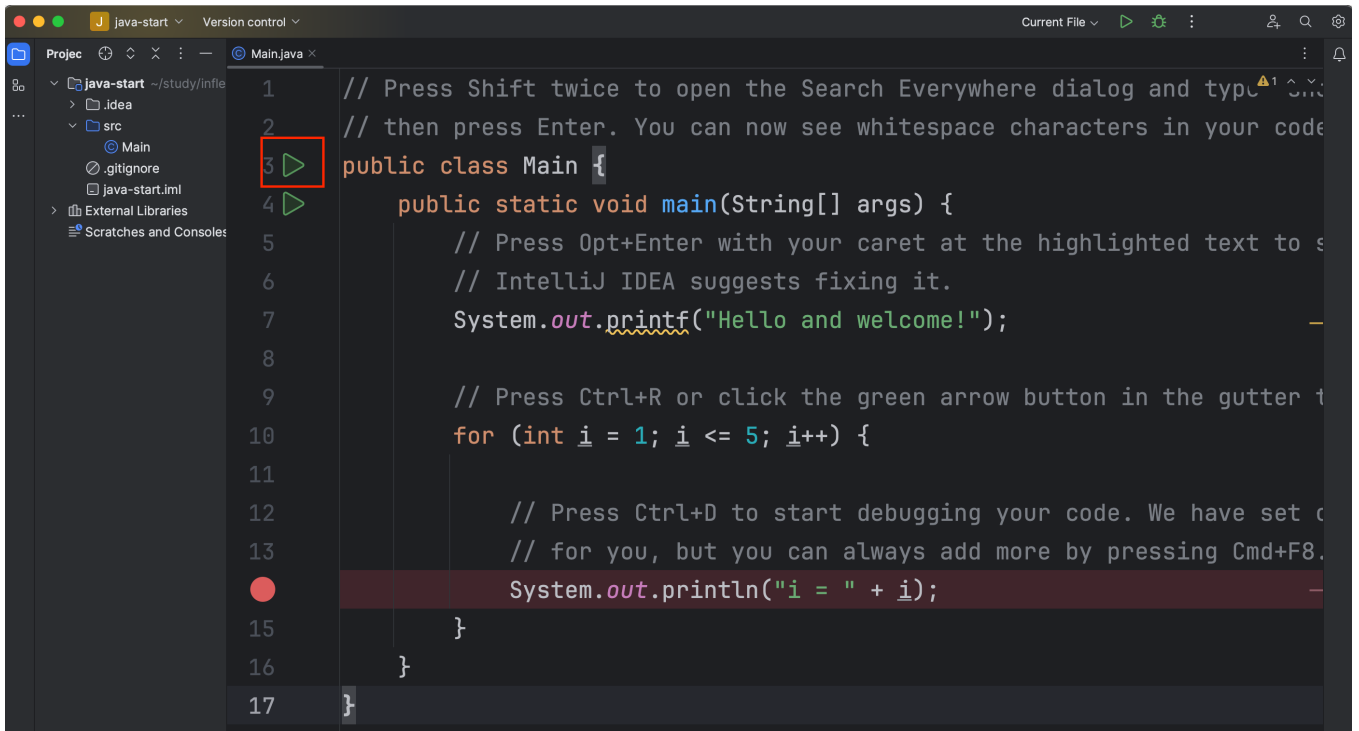
Oracle OpenJDK 21 버전이 목록에 없다면 Eclipse Temurin 21을 선택하면 된다.



Download 버튼을 통해서 JDK를 다운로드 받는다.

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

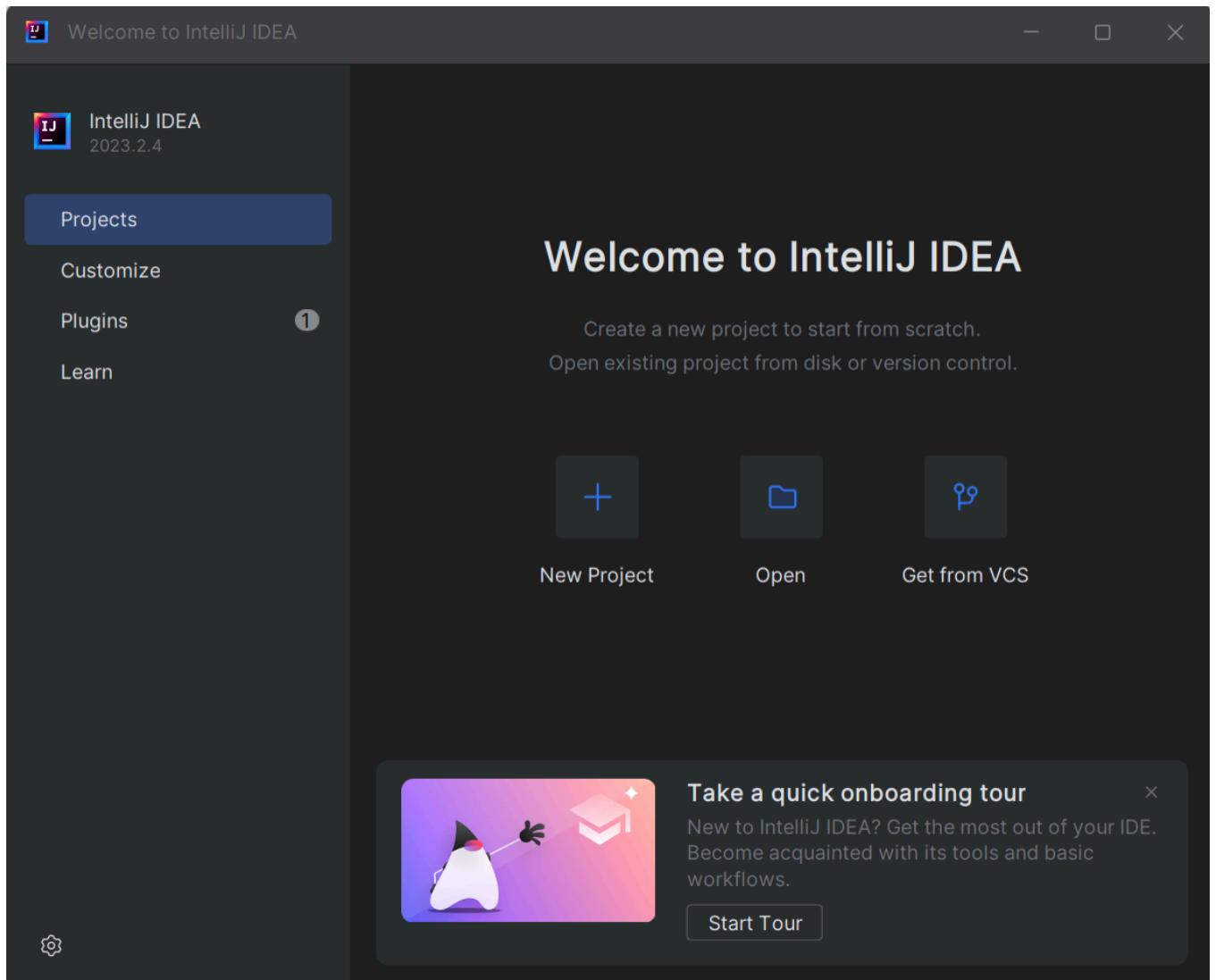
IntelliJ 메인 화면



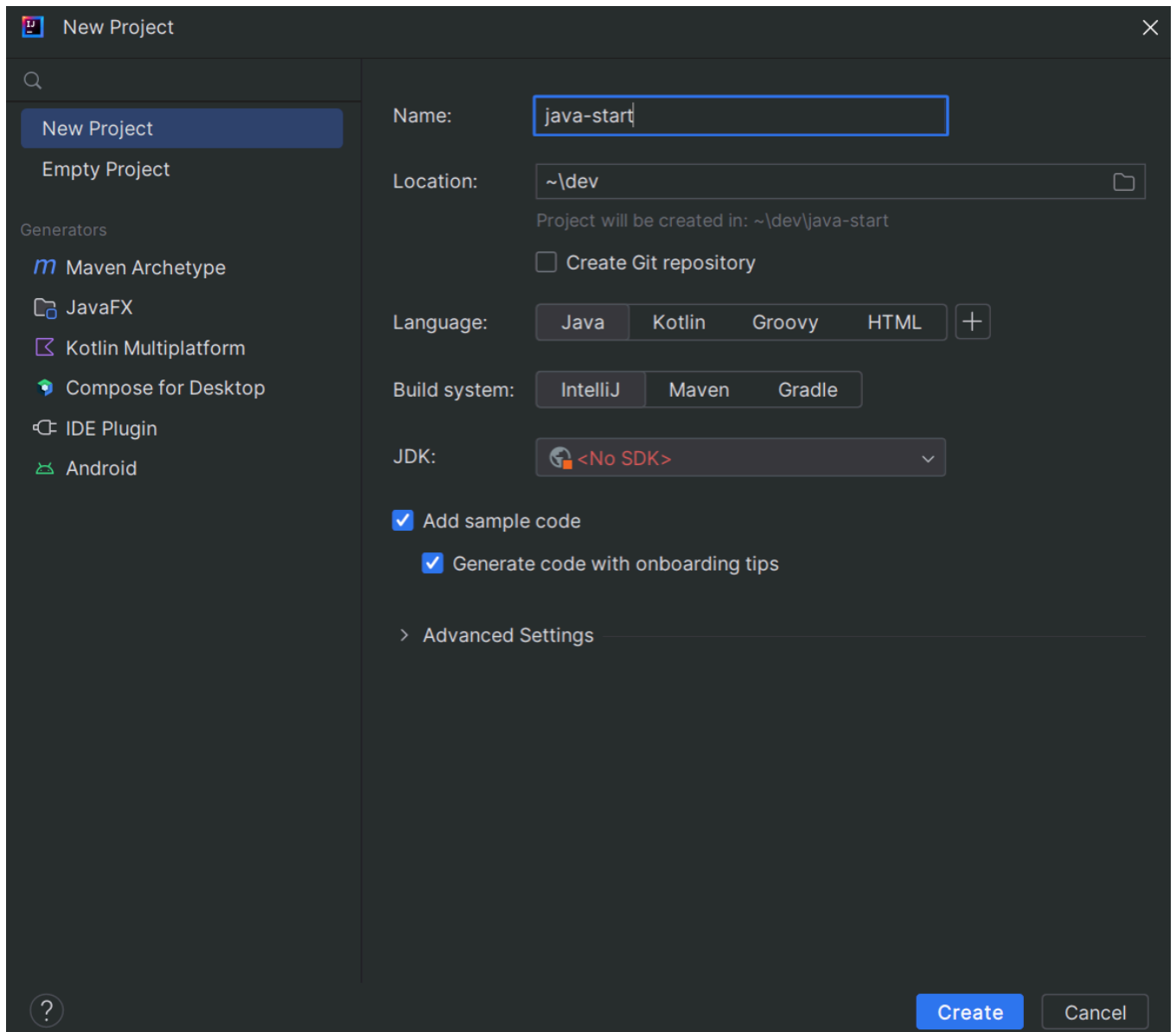
- 앞서 Add sample code 선택해서 샘플 코드가 만들어져 있다.
- 위쪽에 빨간색으로 강조한 초록색 화살표 버튼을 선택하고 Run 'Main.main()' 버튼을 선택하면 프로그램이 실행된다.

윈도우 사용자 추가 설명서

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



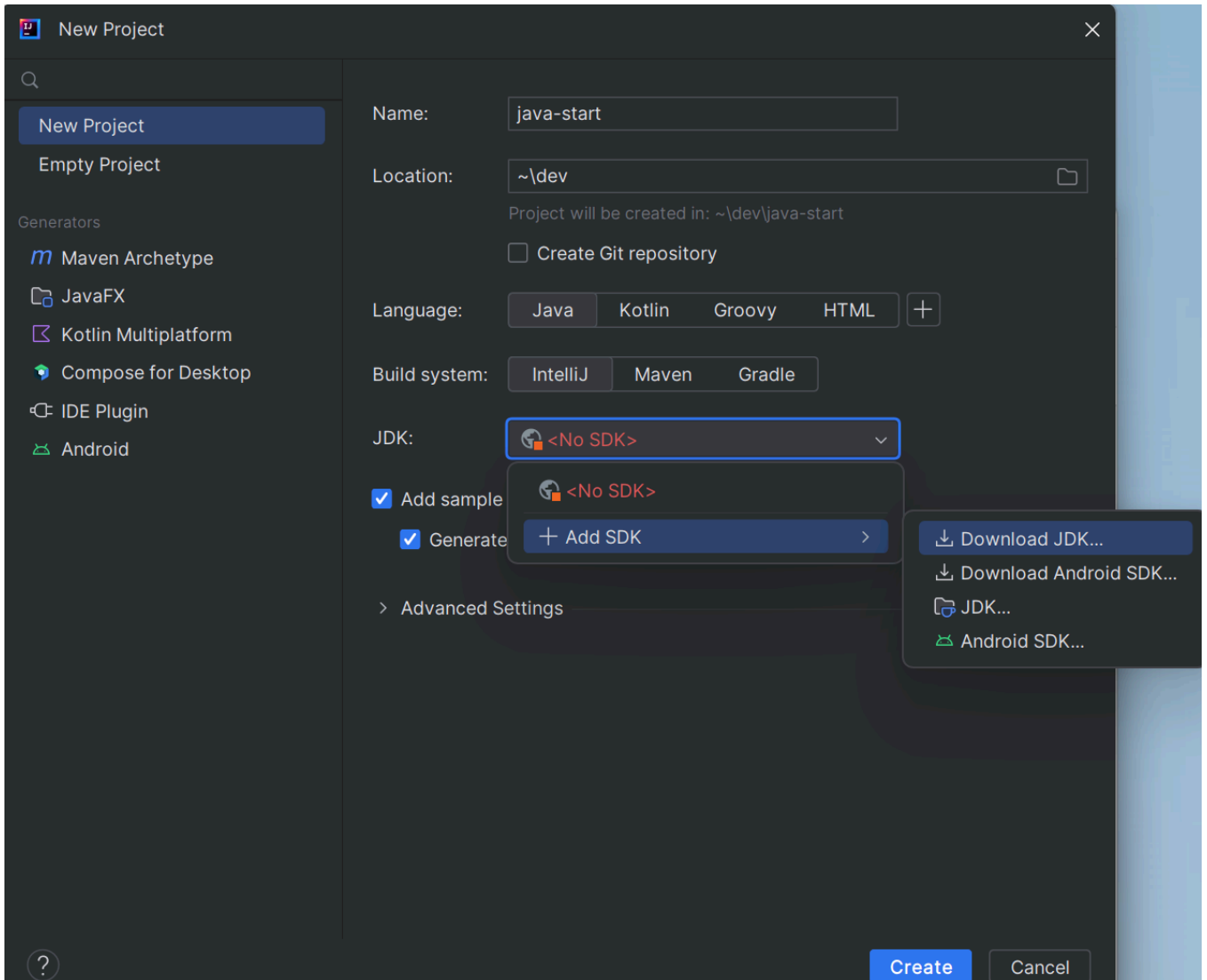
- 프로그램 시작 화면
- New Project 선택



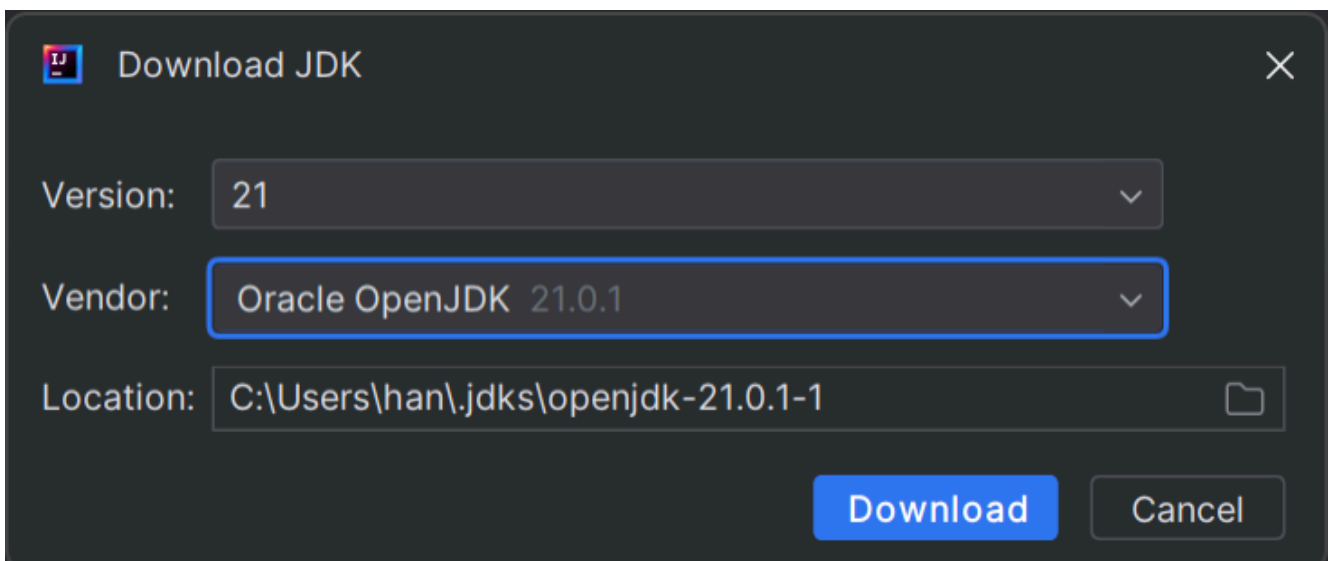
New Project 화면

- **Name:**
 - 자바 입문편 강의: java-start
 - 자바 기본편 강의: java-basic
 - 자바 중급1편 강의: java-mid1
 - 자바 중급2편 강의: java-mid2
 - 자바 고급1편 강의: java-adv1
 - 자바 고급2편 강의: java-adv2
 - 자바 고급3편 강의: **java-adv3**
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ

- JDK: **자바 버전 21 이상**
- Add sample code 선택



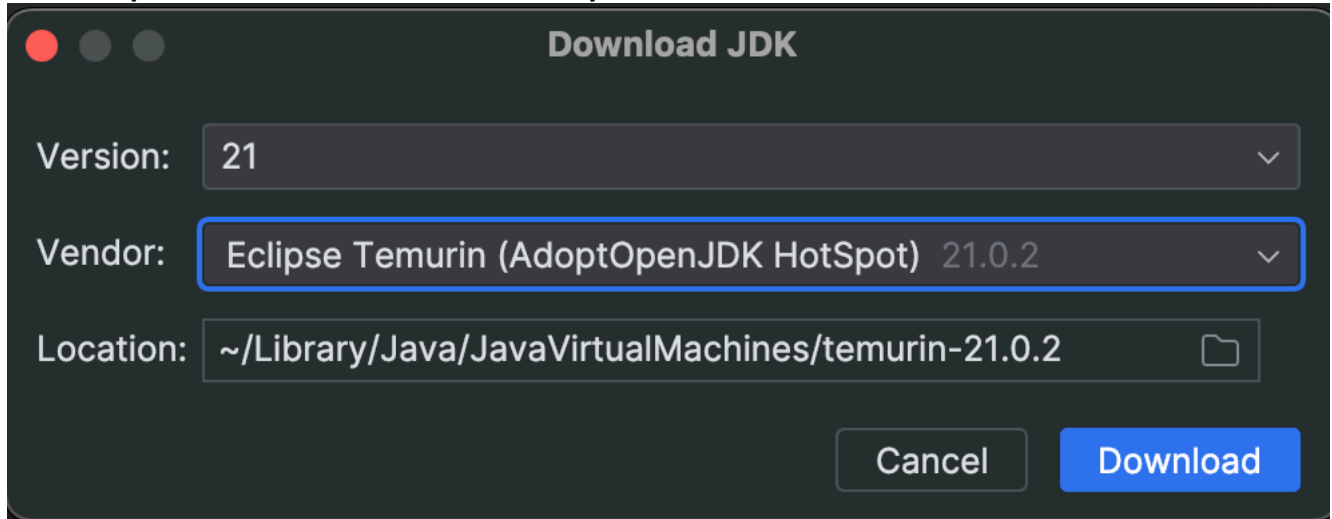
JDK 설치는 Mac과 동일하다.



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

주의 - 변경 사항

Oracle OpenJDK 21 버전이 목록에 없다면 Eclipse Temurin 21을 선택하면 된다.



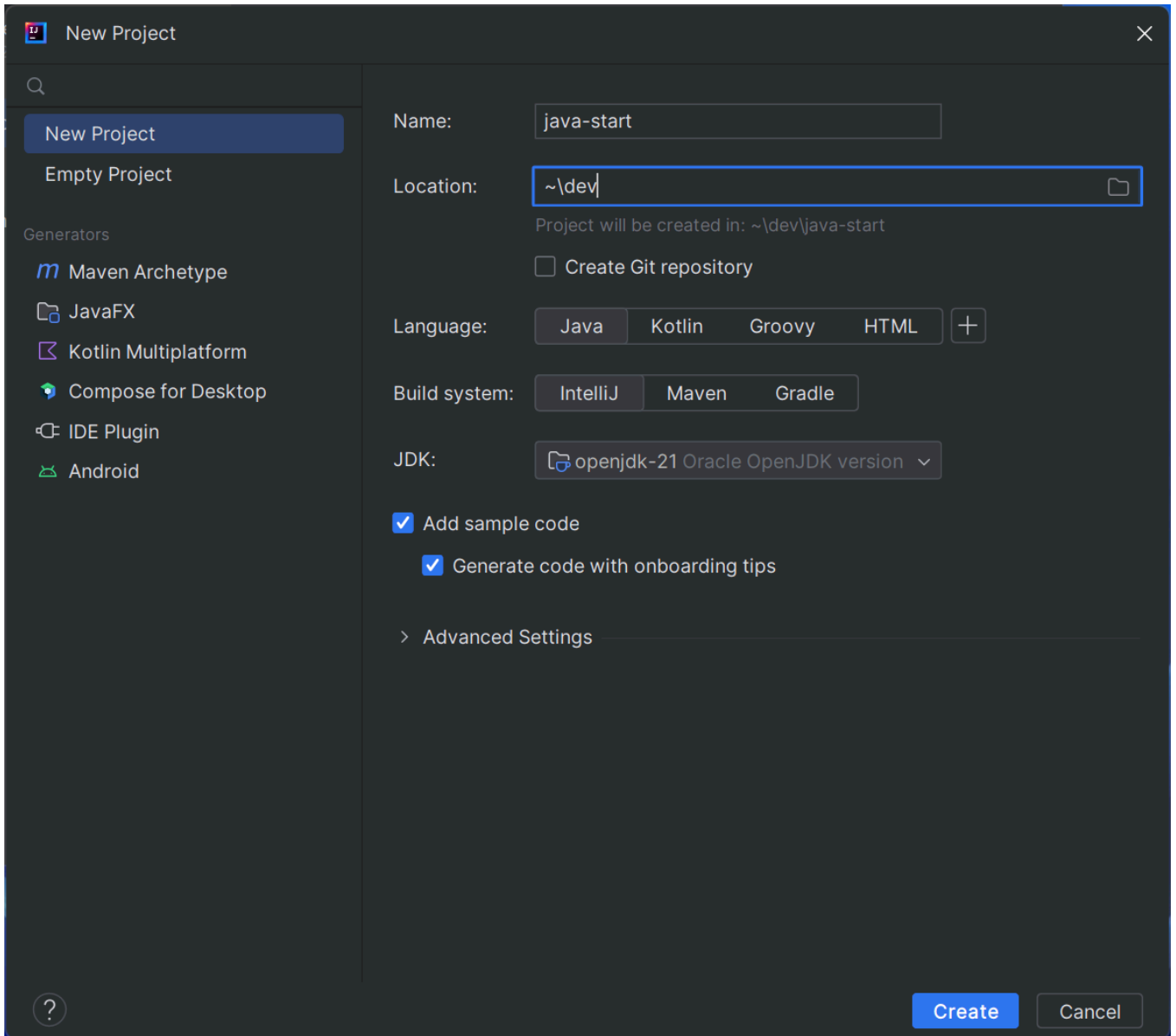
Download JDK

Version: 21

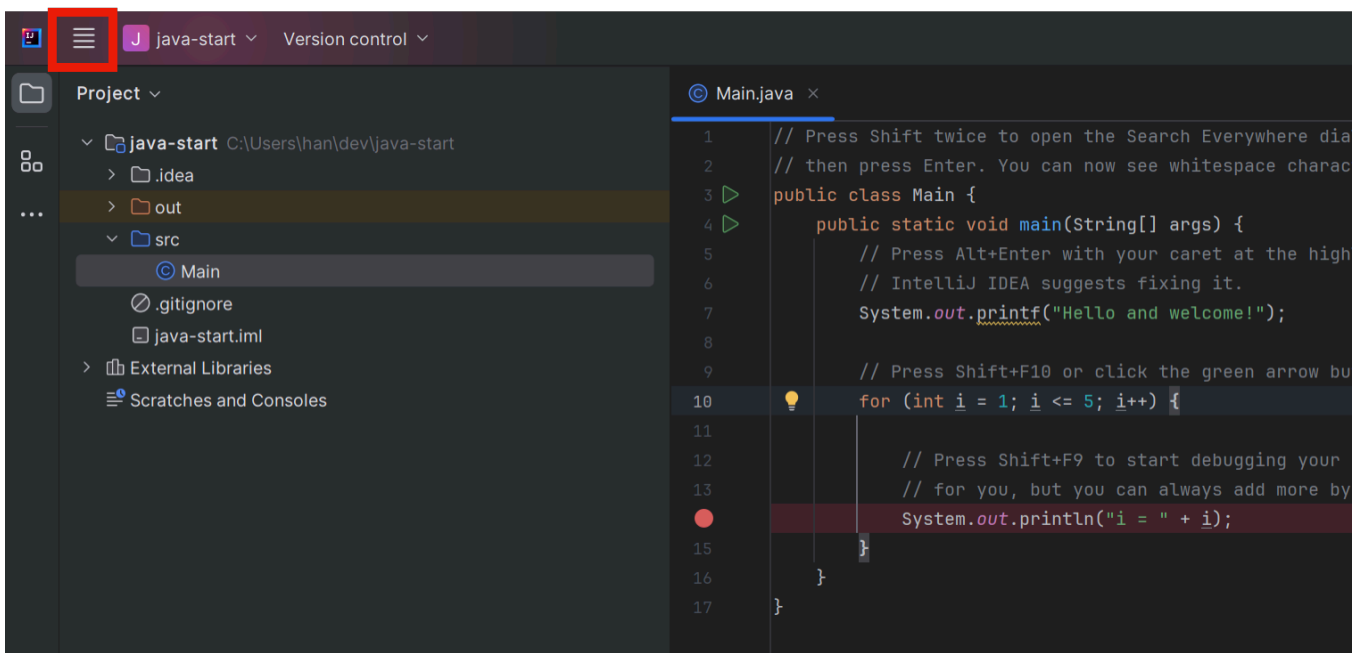
Vendor: Eclipse Temurin (AdoptOpenJDK HotSpot) 21.0.2

Location: ~/Library/Java/JavaVirtualMachines/temurin-21.0.2

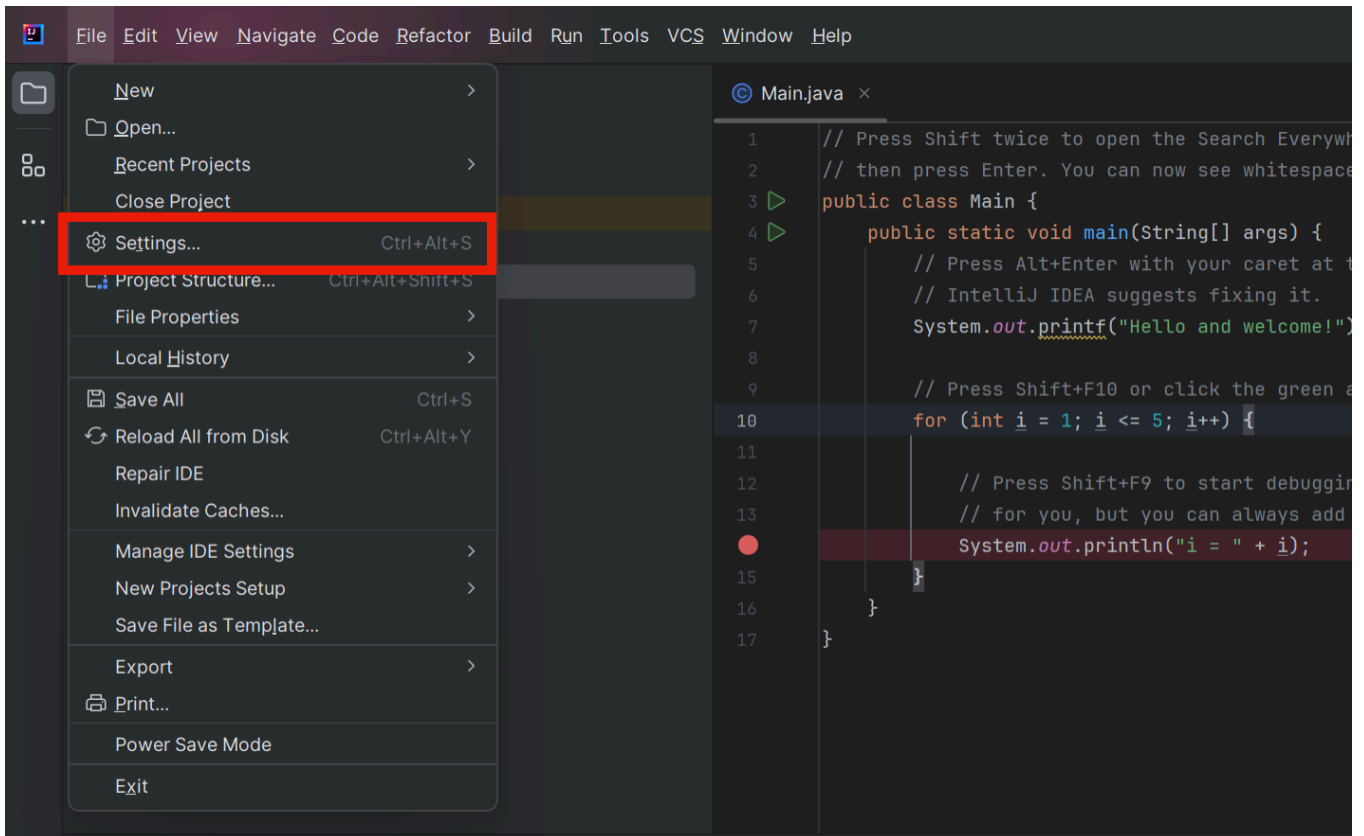
Cancel Download



- New Project 완료 화면



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



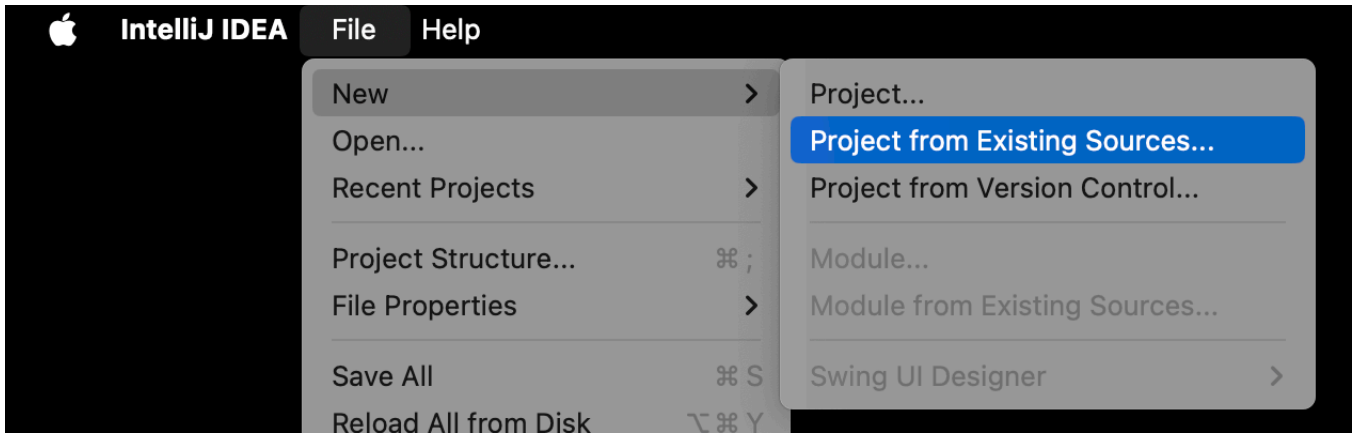
- Mac과 다르게 `Settings...` 메뉴가 `File`에 있다. 이 부분이 Mac과 다르므로 유의하자.

한글 언어팩을 영어로 변경

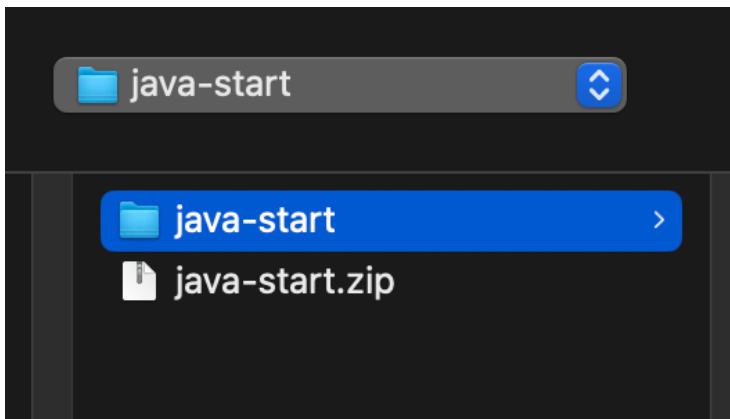
- IntelliJ는 가급적 한글 버전 대신, 영문 버전을 사용하자. 개발하면서 필요한 기능들을 검색하게 되는데, 영문으로 된 자료가 많다. 이번 강의도 영문을 기준으로 진행한다.
- 만약 한글로 나온다면 다음과 같이 영문으로 변경하자.
- **Mac:** IntelliJ IDEA(메뉴) → Settings... → Plugins → Installed
- **윈도우:** File → Settings... → Plugins → Installed
 - Korean Language Pack 체크 해제
 - OK 선택후 IntelliJ 다시 시작

다운로드 소스 코드 실행 방법

영상 참고

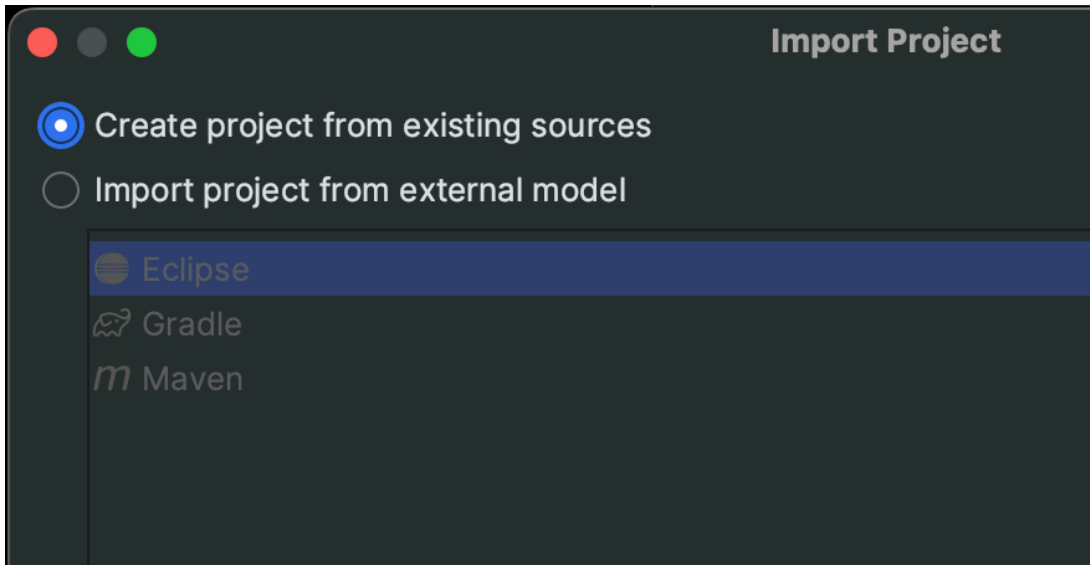


File -> New -> Project from Existing Sources... 선택



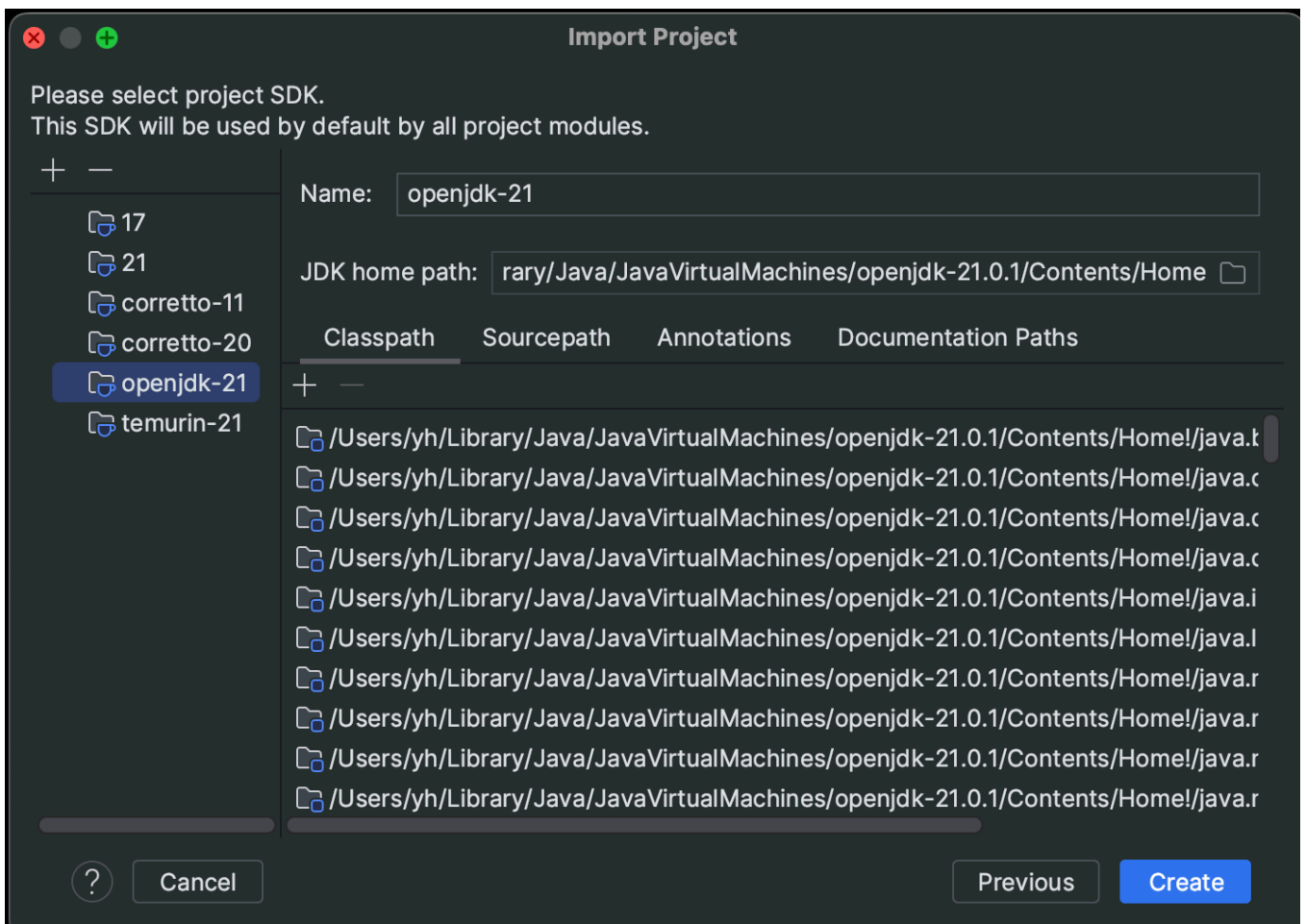
압축을 푼 프로젝트 폴더 선택

- 자바 입문 강의 폴더: java-start
- 자바 기본 강의 폴더: java-basic
- 자바 중급1편 강의 폴더: java-mid1
- 자바 중급2편 강의 폴더: java-mid2
- 자바 고급1편 강의: java-adv1
- 자바 고급2편 강의: java-adv2
- 자바 고급3편 강의: **java-adv3**

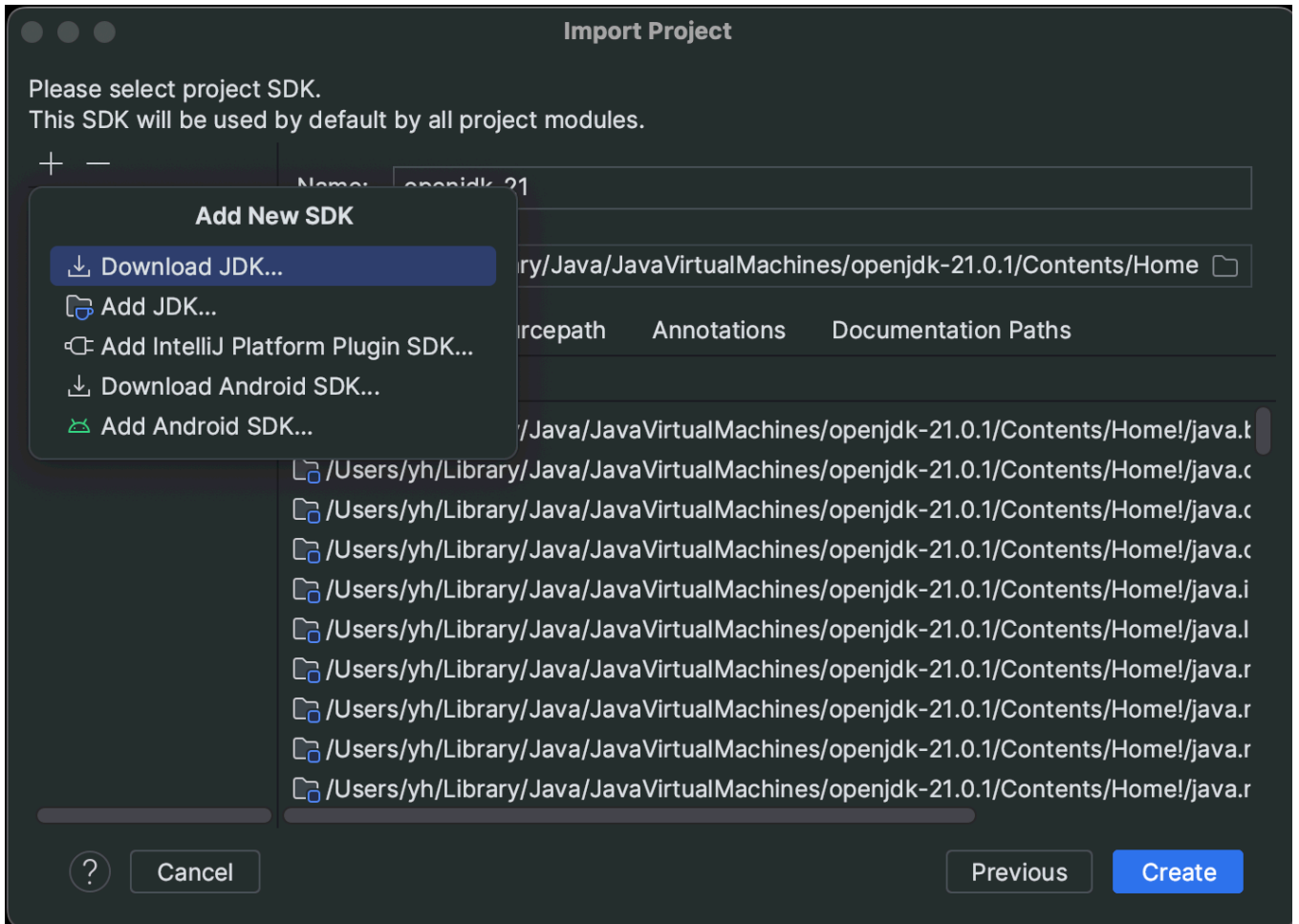


Create project from existing sources 선택

이후 계속 Next 선택



openjdk-21 선택



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

Oracle OpenJDK 21 버전이 목록에 없다면 Eclipse Temurin 21을 선택하면 된다.

이후 Create 버튼 선택

람다가 필요한 이유1

람다를 본격적으로 학습하기 전에, 먼저 람다가 필요한 이유에 대해서 알아보자. 람다를 이해하려면 먼저 내부 클래스에 대한 개념을 확실히 알아두어야 한다.

학습 전 체크 사항

자바 람다에 대해 제대로 이해하려면 먼저 내부 클래스에 대해 확실한 이해가 필요하다.

내부 클래스에 대한 개념이 약하다면 중급 1편 - 중첩 클래스, 내부 클래스를 먼저 복습하자.

지금부터 설명하는 내용은 중급 1편의 익명 클래스 활용을 일부 심화한 내용이다. 이 부분은 기존에 이해를 잘 했어도 복습한다 생각하고, 코드로 꼭 따라해보자.

람다에 대해서 알아보기 전에 간단한 문제를 하나 풀어보자.

리팩토링 전

```
package lambda.start;

public class Ex0Main {

    public static void helloJava() {
        System.out.println("프로그램 시작");
        System.out.println("Hello Java");
        System.out.println("프로그램 종료");
    }

    public static void helloSpring() {
        System.out.println("프로그램 시작");
        System.out.println("Hello Spring");
        System.out.println("프로그램 종료");
    }

    public static void main(String[] args) {
        helloJava();
        helloSpring();
    }
}
```

```
프로그램 시작
Hello Java
프로그램 종료
프로그램 시작
Hello Spring
프로그램 종료
```

코드의 중복이 보일 것이다. 코드를 리팩토링해서 코드의 중복을 제거해보자.

`helloJava()`, `helloSpring()` 메서드를 하나로 통합하면 된다.

리팩토링 후

```

package lambda.start;

public class Ex0RefMain {

    public static void hello() {
        System.out.println("프로그램 시작"); // 변하지 않는 부분

        //변하는 부분 시작
        System.out.println("Hello Java");
        System.out.println("Hello Spring");
        //변하는 부분 종료

        System.out.println("프로그램 종료"); // 변하지 않는 부분
    }

    public static void hello(String str) {
        System.out.println("프로그램 시작");
        System.out.println(str);
        System.out.println("프로그램 종료");
    }

    public static void main(String[] args) {
        hello("hello Java");
        hello("hello Spring");
    }
}

```

실행 결과

```

프로그램 시작
hello Java
프로그램 종료
프로그램 시작
hello Spring
프로그램 종료

```

코드를 분석해보자.

기존 코드에서 변하는 부분과 변하지 않는 부분을 분리해야 한다.


```

public static void helloJava() {
    System.out.println("프로그램 시작"); // 변하지 않는 부분
    System.out.println("Hello Java"); // 변하는 부분
    System.out.println("프로그램 종료"); // 변하지 않는 부분
}

public static void helloSpring() {
    System.out.println("프로그램 시작"); // 변하지 않는 부분
    System.out.println("Hello Spring"); // 변하는 부분
    System.out.println("프로그램 종료"); // 변하지 않는 부분
}

```

여기서 핵심은 **변하는 부분**과 **변하지 않는 부분**을 분리하는 것이다.

변하는 부분은 그대로 유지하고 변하지 않는 부분을 어떻게 해결할 것 인가에 집중하면 된다.

이해를 돕기 위해 다음과 같은 코드를 먼저 작성해보자.

```

public static void hello() {
    System.out.println("프로그램 시작"); // 변하지 않는 부분

    //변하는 부분 시작
    System.out.println("Hello Java");
    System.out.println("Hello Spring");
    //변하는 부분 종료

    System.out.println("프로그램 종료"); // 변하지 않는 부분
}

```

여기서 "Hello Java", "Hello Spring" 와 같은 문자열은 상황에 따라서 변한다.

여기서는 상황에 따라 변하는 문자열 데이터를 다음과 같이 매개변수(파라미터)를 통해 외부에서 전달 받아서 출력하면 된다.

```

public static void hello(String str) {
    System.out.println("프로그램 시작"); // 변하지 않는 부분
    System.out.println(str); // str: 변하는 부분
    System.out.println("프로그램 종료"); // 변하지 않는 부분
}

```

- 변하지 않는 부분은 그대로 유지하고, 변하는 부분은 외부에서 전달 받아서 처리한다.

단순한 문제였지만 **프로그래밍에서 중복을 제거하고, 좋은 코드를 유지하는 핵심은 변하는 부분과 변하지 않는 부분을 분리하는 것이다**. 여기서 변하지 않는 "프로그램 시작", "프로그램 종료"를 출력하는 부분은 그대로 유지하고, 상황에 따라 변화가 필요한 문자열은 외부에서 전달 받아서 처리했다.

이렇게 변하는 부분과 변하지 않는 부분을 분리하고, 변하는 부분을 외부에서 전달 받으면, 메서드(함수)의 재사용성을 높일 수 있다.

리팩토링 전과 후를 비교해보자. `hello(String str)` 메서드의 재사용성은 매우 높아졌다.

여기서 핵심은 변하는 부분을 메서드 내부에서 가지고 있는 것이 아니라, 외부에서 전달 받는다는 점이다.

값 매개변수화(Value Parameterization)

여기서 변하는 부분은 `"Hello Java"`, `"Hello Spring"` 같은 문자값(Value)이다.

```
System.out.println("Hello Java");
System.out.println("Hello Spring");
```

이번 예제에서는 `String str` 매개변수(파라미터)를 사용해서 문자값을 매개변수로 만들었다.

```
// str = Hello Java, Hello Spring 등을 전달
public static void hello(String str) {
    System.out.println("프로그램 시작");
    System.out.println(str); // str: 변하는 부분
    System.out.println("프로그램 종료");
}
```

문자값(Value), 숫자값(Value)처럼 구체적인 값을 메서드(함수) 안에 두는 것이 아니라, **매개변수(파라미터)**를 통해 외부에서 전달 받도록 해서, 메서드의 동작을 달리하고, 재사용성을 높이는 방법을 **값 매개변수화(Value Parameterization)**라 한다.

람다가 필요한 이유2

이번에는 비슷한 다른 문제를 한번 풀어보자.

리팩토링 전

```

package lambda.start;

import java.util.Random;

public class Ex1Main {

    public static void helloDice() {
        long startNs = System.nanoTime();

        //코드 조각 시작
        int randomValue = new Random().nextInt(6) + 1;
        System.out.println("주사위 = " + randomValue);
        //코드 조각 종료

        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }

    public static void helloSum() {
        long startNs = System.nanoTime();

        //코드 조각 시작
        for (int i = 1; i <= 3; i++) {
            System.out.println("i = " + i);
        }
        //코드 조각 종료

        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }

    public static void main(String[] args) {
        helloDice();
        helloSum();
    }
}

```

실행 결과

주사위 = 2

실행 시간: 2882959ns

i = 1

i = 2

i = 3

실행 시간: 191083ns

이 코드를 이전에 리팩토링 한 예와 같이 **하나의 메서드에서 실행**할 수 있도록 리팩토링 해보자.

참고로 이전 문제는 변하는 문자열 값을 매개변수화 해서 외부에서 전달하면 되었다. 이번에는 문자열 같은 단순한 값이 아니라 코드 조각을 전달해야 한다.

리팩토링 후

```
package lambda;

public interface Procedure {
    void run();
}
```

- 클래스 이름과 패키지 위치에 주의하자!
- `Procedure` 인터페이스는 이후에 다른 곳에서도 함께 사용할 예정이어서 패키지 위치를 다른 곳으로 지정했다.

```
package lambda.start;

import lambda.Procedure;

import java.util.Random;

// 정적 중첩 클래스 사용
public class Ex1RefMainV1 {

    public static void hello(Procedure procedure) {
        long startNs = System.nanoTime();
        //코드 조각 시작
        procedure.run();
        //코드 조각 종료
        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }
}
```

```

static class Dice implements Procedure {

    @Override
    public void run() {
        int randomValue = new Random().nextInt(6) + 1;
        System.out.println("주사위 = " + randomValue);
    }
}

static class Sum implements Procedure {

    @Override
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println("i = " + i);
        }
    }
}

public static void main(String[] args) {
    Procedure dice = new Dice();
    Procedure sum = new Sum();

    hello(dice);
    hello(sum);
}
}

```

실행 결과

```

주사위 = 6
실행 시간: 3245875ns
i = 1
i = 2
i = 3
실행 시간: 259125ns

```

코드를 분석해보자.

여기서는 단순히 데이터를 전달하는 수준을 넘어서, 코드 조각을 전달해야 한다.

리팩토링 전

```
public static void helloDice() {
    long startNs = System.nanoTime(); // 변하지 않는 부분

    //코드 조각 시작
    int randomValue = new Random().nextInt(6) + 1;
    System.out.println("주사위 = " + randomValue);
    //코드 조각 종료

    long endNs = System.nanoTime(); // 변하지 않는 부분
    System.out.println("실행 시간: " + (endNs - startNs) + "ns"); // 변하지 않는 부분
}

public static void helloSum() {
    long startNs = System.nanoTime(); // 변하지 않는 부분

    //코드 조각 시작
    for (int i = 1; i <= 3; i++) {
        System.out.println("i = " + i);
    }
    //코드 조각 종료

    long endNs = System.nanoTime(); // 변하지 않는 부분
    System.out.println("실행 시간: " + (endNs - startNs) + "ns"); // 변하지 않는 부분
}
```

- 두 메서드에서 시간을 측정하고, 시간을 출력하는 부분은 변하지 않는 부분이다.
- 코드 조각을 시작하고 종료하는 부분은 변하는 부분이다.
- 중복을 제거하고 재사용성을 늘리려면 결국 코드 조각을 시작하고 종료하는 부분을 외부에서 전달 받아야 한다. 이것은 단순히 문자열, 숫자 같은 값 데이터를 전달 받는 것과는 차원이 다른 문제이다.

어떻게 외부에서 코드 조각을 전달할 수 있을까?

코드 조각은 보통 메서드(함수)에 정의한다. 따라서 코드 조각을 전달하기 위해서는 메서드가 필요하다.

그런데 지금까지 학습한 내용으로는 메서드만 전달할 수 있는 방법이 없다. 대신에 인스턴스를 전달하고, 인스턴스에 있는 메서드를 호출하면 된다.

이 문제를 해결하기 위해 인터페이스를 정의하고 구현 클래스를 만들었다.

```
package lambda;

public interface Procedure {
    void run();
}
```

```
static class Dice implements Procedure {

    @Override
    public void run() {
        int randomValue = new Random().nextInt(6) + 1;
        System.out.println("주사위 = " + randomValue);
    }
}

static class Sum implements Procedure {

    @Override
    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println("i = " + i);
        }
    }
}
```

- Dice, Sum 각각의 클래스는 Procedure 인터페이스를 구현하고 run() 메서드에 필요한 코드 조각을 구현한다.
- 여기서는 정적 중첩 클래스를 사용했다. 물론 정적 중첩 클래스가 아니라 외부에 따로 클래스를 만들어도 된다.

리팩토링 완료

```
public static void hello(Procedure procedure) {
    long startNs = System.nanoTime();
    //코드 조각 시작
    procedure.run();
    //코드 조각 종료
    long endNs = System.nanoTime();
    System.out.println("실행 시간: " + (endNs - startNs) + "ns");
}
```

- 리팩토링한 `hello()` 메서드에는 `Procedure procedure` 매개변수(파라미터)를 통해 인스턴스를 전달할 수 있다. 이 인스턴스의 `run()` 메서드를 실행하면 필요한 코드 조각을 실행할 수 있다.
- 이때 다형성을 활용해서 외부에서 전달되는 인스턴스에 따라 각각 다른 코드 조각이 실행된다.

```
public static void main(String[] args) {
    Procedure dice = new Dice();
    Procedure sum = new Sum();

    hello(dice);
    hello(sum);
}
```

- `hello()` 를 호출할 때 어떤 인스턴스를 전달하는가에 따라서 다른 결과가 실행된다.
- `hello(dice)` 를 호출하면 주사위 로직이, `hello(sum)` 을 호출하면 계산 로직이 수행된다.

실행 결과

```
주사위 = 4
실행 시간: 3570583ns
i = 1
i = 2
i = 3
실행 시간: 172542ns
```

정리

- 문자열, 숫자 같은 값 데이터를 메서드에 전달할 때는 `String`, `int` 와 같은 각 데이터에 맞는 값을 전달하면 된다.
- 코드 조각을 메서드에 전달할 때는 인스턴스를 전달하고 해당 인스턴스에 있는 메서드를 호출하면 된다.

동작 매개변수화(Behavior Parameterization)

값 매개변수화(Value Parameterization)

- 문자값(**Value**), 숫자값(**Value**)처럼 구체적인 값을 메서드(함수) 안에 두는 것이 아니라, **매개변수**(파라미터)를 통해 외부에서 전달 받도록 해서, 메서드의 동작을 달리하고, 재사용성을 높이는 방법을 **값 매개변수화**라 한다.
- 값 매개변수화, 값 파라미터화 등으로 부른다.

동작 매개변수화(Behavior Parameterization)

- 코드 조각(코드의 동작 방법, 로직, **Behavior**)을 메서드(함수) 안에 두는 것이 아니라, **매개변수**(파라미터)를 통해서 외부에서 전달 받도록 해서, 메서드의 동작을 달리하고, 재사용성을 높이는 방법을 동작 매개변수화라 한다.
- 동작 매개변수화, 동작 파라미터화, 행동 매개변수화(파라미터화), 행위 파라미터화 등으로 부른다.

정리하면 다음과 같다.

- **값 매개변수화**: 값(숫자, 문자열 등)을 바꿔가며 메서드(함수)의 동작을 달리 함
- **동작 매개변수화**: 어떤 동작(로직)을 수행할지를 메서드(함수)에 전달(인스턴스 참조, 람다 등)

자바에서 동작 매개변수화를 하려면 클래스를 정의하고 해당 클래스를 인스턴스로 만들어서 전달해야 한다.

자바8에서 등장한 람다를 사용하면 코드 조각을 매우 편리하게 전달 할 수 있는데, 람다를 알아보기 전에 기존에 자바로 할 수 있는 다양한 방법들을 먼저 알아보자.

람다가 필요한 이유3

익명 클래스 사용1

이번에는 익명 클래스를 사용해서 같은 기능을 구현해보자.

```
package lambda.start;

import lambda.Procedure;

import java.util.Random;

// 익명 클래스 사용
public class Ex1RefMainV2 {

    public static void hello(Procedure procedure) {
        long startNs = System.nanoTime();
        //코드 조각 시작
        procedure.run();
        //코드 조각 종료
        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }
}
```

```

public static void main(String[] args) {
    Procedure dice = new Procedure() {
        @Override
        public void run() {
            int randomValue = new Random().nextInt(6) + 1;
            System.out.println("주사위 = " + randomValue);
        }
    };

    Procedure sum = new Procedure() {
        @Override
        public void run() {
            for (int i = 1; i <= 3; i++) {
                System.out.println("i = " + i);
            }
        }
    };

    hello(dice);
    hello(sum);
}

```

실행 결과는 기존과 같다.

익명 클래스 사용2 - 참조값 직접 전달

익명 클래스의 참조값을 지역 변수에 담아둘 필요 없이, 매개변수에 직접 전달해보자.

```

package lambda.start;

import lambda.Procedure;

import java.util.Random;

// 익명 클래스 사용, 변수 제거, 익명 클래스의 참조값을 매개변수(파라미터)에 직접 전달
public class Ex1RefMainV3 {

    public static void hello(Procedure procedure) {
        long startNs = System.nanoTime();
        //코드 조각 시작
        procedure.run();
    }
}

```

```

//코드 조각 종료
long endNs = System.nanoTime();
System.out.println("실행 시간: " + (endNs - startNs) + "ns");
}

public static void main(String[] args) {
    hello(new Procedure() {
        @Override
        public void run() {
            int randomValue = new Random().nextInt(6) + 1;
            System.out.println("주사위 = " + randomValue);
        }
    });

    hello(new Procedure() {
        @Override
        public void run() {
            for (int i = 1; i <= 3; i++) {
                System.out.println("i = " + i);
            }
        }
    });
}
}

```

실행 결과는 기존과 같다.

람다(lambda)

자바에서 메서드의 매개변수에 인수로 전달할 수 있는 것은 크게 2가지이다.

- `int`, `double` 과 같은 기본형 타입
- `Procedure` `Member` 와 같은 참조형 타입(인스턴스)

결국 메서드에 인수로 전달할 수 있는 것은 간단한 값이나, 인스턴스의 참조이다.

지금처럼 코드 조각을 전달하기 위해 클래스를 정의하고 메서드를 만들고 또 인스턴스까지 생성하는 복잡한 과정을 거쳐야 할까?

생각해보면 클래스나 인스턴스와 관계 없이 다음과 같이 직접 코드 블록을 전달할 수 있다면 더 간단하지 않을까?

```

public static void main(String[] args) {
    hello({ // 랜덤 값을 출력하는 코드 블록

```

```

        int randomValue = new Random().nextInt(6) + 1;
        System.out.println("주사위 = " + randomValue);
    }
};

hello({    // 1 ~ 3 출력하는 코드 블록
    for (int i = 1; i <= 3; i++) {
        System.out.println("i = " + i);
    }
});
}

```

자바8에 들어서면서 큰 변화가 있었는데, 바로 람다를 통해 코드 블록을 인수로 전달할 수 있게 되었다. 다음 코드로 확인해보자.

리팩토링 - 람다

```

package lambda.start;

import lambda.Procedure;

import java.util.Random;

// 람다 사용
public class Ex1RefMainV4 {

    public static void hello(Procedure procedure) {
        long startNs = System.nanoTime();
        //코드 조각 시작
        procedure.run();
        //코드 조각 종료
        long endNs = System.nanoTime();
        System.out.println("실행 시간: " + (endNs - startNs) + "ns");
    }

    public static void main(String[] args) {
        hello(() -> {
            int randomValue = new Random().nextInt(6) + 1;
            System.out.println("주사위 = " + randomValue);
        });

        hello(() -> {

```

```

        for (int i = 1; i <= 3; i++) {
            System.out.println("i = " + i);
        }
    });
}
}

```

- 실행 결과는 기존과 같다.
- `() -> {...}` 부분이 람다를 사용한 코드이다.
- 람다를 사용한 코드를 보면 클래스나 인스턴스를 정의하지 않고, 매우 간편하게 코드 블록을 직접 정의하고, 전달하는 것을 확인할 수 있다.

이제 본격적으로 람다를 알아보자. 람다는 함수이다. 따라서 람다를 제대로 이해하기 위해서는 먼저 함수에 대해서 알아야 한다. 함수와 메서드의 차이를 먼저 간단히 알아보자.

함수 vs 메서드

함수(Function)와 메서드(Method)는 둘 다 어떤 작업(로직)을 수행하는 코드의 묶음이다. 하지만 일반적으로 **객체지향 프로그래밍(OOP)** 관점에서 다음과 같은 차이가 있다. 먼저 간단한 예시를 알아보자.

예시

C 언어

```

// C에서는 클래스나 객체가 없으므로, 모든 것이 함수
int add(int x, int y) {
    return x + y;
}

```

Java

```

// 자바에서는 클래스 내부에 함수를 정의 -> 메서드
public class Calculator {
    // 인스턴스 메서드
    public int add(int x, int y) {
        return x + y;
    }
}

```

```

    }
}

// 사용 예
Calculator cal = new Calculator();
int result = cal.add(2, 3); // 'add'는 메서드

```

Python

```

# 함수: 클래스 밖에서 독립적으로 정의
def add(x, y):
    return x + y

# 메서드: 클래스(객체) 내부에 정의
class Calculator:
    def add(self, x, y):
        return x + y

# 사용 예
print(add(2, 3)) # 함수 호출

cal = Calculator()
print(cal.add(2, 3)) # 메서드 호출

```

객체(클래스)와의 관계

- **함수(Function)**
 - 독립적으로 존재하며, 클래스(객체)와 직접적인 연관이 없다.
 - 객체지향 언어가 아닌 C 등의 절차적 언어에서는 모든 로직이 함수 단위로 구성된다.
 - 객체지향 언어라 하더라도, 예를 들어 Python이나 JavaScript처럼 클래스 밖에서도 정의할 수 있는 "함수" 개념을 지원하는 경우, 이를 그냥 함수라고 부른다.
- **메서드(Method)**
 - 클래스(또는 객체)에 속해 있는 "함수"이다.
 - 객체의 상태(필드, 프로퍼티 등)에 직접 접근하거나, 객체가 제공해야 할 기능을 구현할 수 있다.
 - Java, C++, C#, Python 등 대부분의 객체지향 언어에서 클래스 내부에 정의된 함수는 보통 "메서드"라고 부른다.

호출 방식과 스코프

- **함수(Function)**

- 호출 시에 객체 인스턴스가 필요 없다.
- 보통 이름(매개변수) 형태로 호출된다.
- 지역 변수, 전역 변수 등과 함께 동작하며, 클래스나 객체 특유의 속성(인스턴스 변수 등)은 다루지 못한다.
- **메서드(Method)**
 - 보통 객체(인스턴스).메서드이름(매개변수) 형태로 호출한다.
 - 호출될 때, 해당 객체의 필드(속성)나 다른 메서드에 접근 가능하며, 이를 이용해 로직을 수행한다.
 - 인스턴스 메서드, 클래스(정적) 메서드, 추상 메서드 등 다양한 형태가 있을 수 있다.

정리

- 메서드는 기본적으로 클래스(객체) 내부의 함수를 가리키며, 객체의 상태와 밀접한 관련이 있다.
- 함수는 클래스(객체)와 상관없이, 독립적으로 호출 가능한 로직의 단위이다.
- 메서드는 객체지향에서 클래스 안에 정의하는 특별한 함수라고 생각하면 된다.

따라서 함수와 메서드는 수행하는 역할 자체는 같지만, 소속(클래스 or 독립)과 호출 방식에서 차이가 난다고 이해하면 된다.

람다 시작

이제 본격적으로 람다에 대해서 알아보자.

이론적인 부분을 알아보기 전에 간단히 익명 클래스와 람다를 코드로 만들어보자.

익명 클래스를 람다로1

매개변수가 없는 간단한 익명 클래스를 하나 만들고 실행해보자.

```
package lambda.lambda1;

import lambda.Procedure;

public class ProcedureMain1 {

    public static void main(String[] args) {
        Procedure procedure = new Procedure() {
            @Override
            public void run() {
                System.out.println("hello! lambda");
            }
        };
    }
}
```

```

        }
    };

    procedure.run();
}
}

```

- `run()` 메서드는 매개변수가 없고, 반환 값이 없다.

실행 결과

```
hello! lambda
```

앞서 만든 매개변수가 없는 익명 클래스 예제는 다음과 같이 람다로 작성할 수 있다.

```

package lambda.lambda1;

import lambda.Procedure;

public class ProcedureMain2 {

    public static void main(String[] args) {
        Procedure procedure = () -> {
            System.out.println("hello! lambda");
        };

        procedure.run();
    }
}

```

- 람다는 `() -> {}` 와 같이 표현한다. `()` 부분이 메서드의 매개변수라 생각하면 되고, `{}` 부분이 코드 조각이 들어가는 본문이다.
- 람다를 사용할 때는 이름, 반환 타입은 생략하고, 매개변수와 본문만 간단하게 적으면 된다.
 - `(매개변수) -> { 본문 }`, 여기서는 매개변수가 없으므로 `() -> {본문}`
- 쉽게 이야기해서 익명 클래스를 만들기 위한 모든 부분을 생략하고, 꼭 필요한 매개변수와 본문만 작성하면 된다.
- 익명 클래스를 사용하는 것 보다 람다를 사용할 때 코드가 훨씬 간결한 것을 확인할 수 있다.

실행 결과


```
hello! lambda
```

익명 클래스를 람다로2

이번에는 매개변수가 있는 간단한 익명 클래스를 하나 만들고 실행해보자.

```
package lambda;

public interface MyFunction {
    int apply(int a, int b);
}
```

- `MyFunction`은 앞으로 여러 곳에서 사용할 예정이다. 따라서 **패키지 위치가 다르다**. 주의하자

```
package lambda.lambda1;

import lambda.MyFunction;

public class MyFunctionMain1 {

    public static void main(String[] args) {
        MyFunction myFunction = new MyFunction() {
            @Override
            public int apply(int a, int b) {
                return a + b;
            }
        };

        int result = myFunction.apply(1, 2);
        System.out.println("result = " + result);
    }
}
```

- `int apply(int a, int b)` 메서드는 `int a, int b` 매개변수가 2개 있고, `int`를 반환한다.

실행 결과

```
result = 3
```

앞서 만든 매개변수가 있는 익명 클래스 예제는 다음과 같이 람다로 작성할 수 있다.

```
package lambda.lambda1;

import lambda.MyFunction;

public class MyFunctionMain2 {

    public static void main(String[] args) {
        MyFunction myFunction = (int a, int b) -> {
            return a + b;
        };

        int result = myFunction.apply(1, 2);
        System.out.println("result = " + result);
    }
}
```

- 람다는 () -> {} 와 같이 표현한다.
- 람다를 사용할 때는 이름, 반환 타입은 생략하고, 매개변수와 본문만 간단하게 적으면 된다.
- 이번에는 매개변수가 있으므로 (int a, int b) -> {본문} 과 같이 작성하면 된다.

실행 결과

```
result = 3
```