1. 제네릭 - Generic1

#1.인강/0.자바/4.자바-중급2편

- /프로젝트 환경 구성
- /제네릭이 필요한 이유
- /다형성을 통한 중복 해결 시도
- /제네릭 적용
- /제네릭 용어와 관례
- /제네릭 활용 예제
- /문제와 풀이1

프로젝트 환경 구성

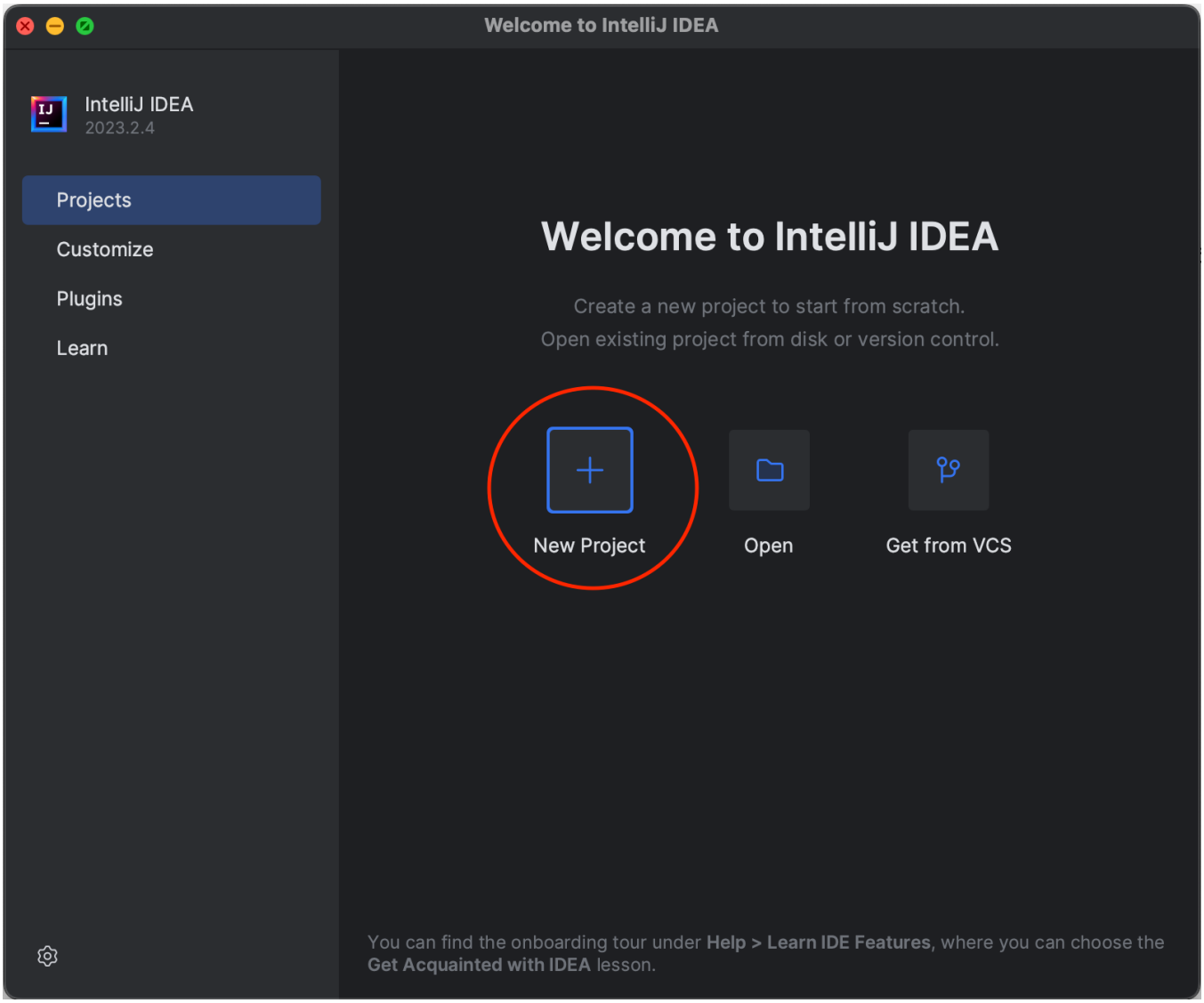
자바 입문편에서 인텔리제이 설치, 선택 이유 설명

프로젝트 환경 구성에 대한 자세한 내용은 자바 입문편 참고

여기서는 입문편을 들었다는 가정하에 설정 진행

인텔리제이 실행하기

New Project



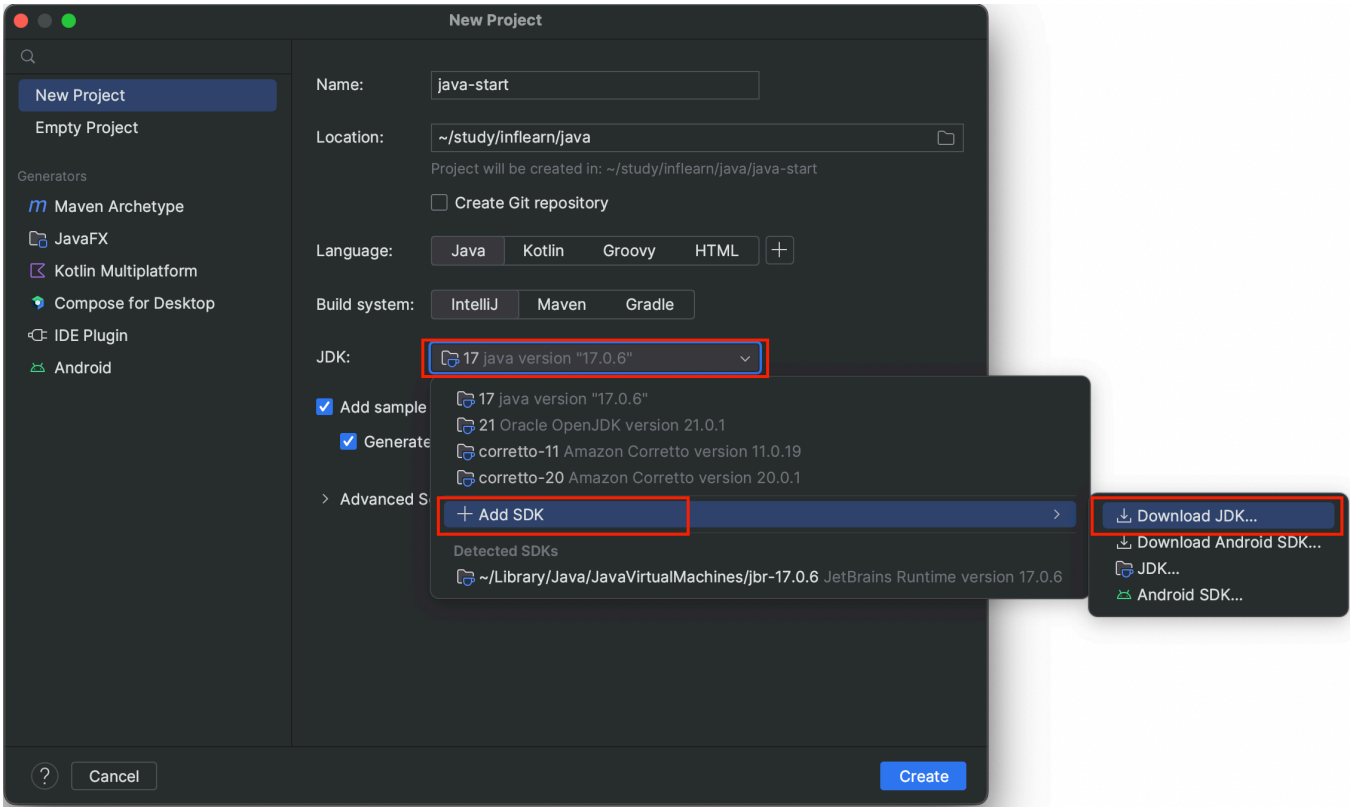
- New Project를 선택해서 새로운 프로젝트를 만들자

New Project 화면

- Name:
  - 자바 입문편 강의: java-start
  - 자바 기본편 강의: java-basic
  - 자바 중급1편 강의: java-mid1
  - 자바 중급2편 강의: **java-mid2**
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: 자바 버전 17 이상
- `Add sample code` 선택

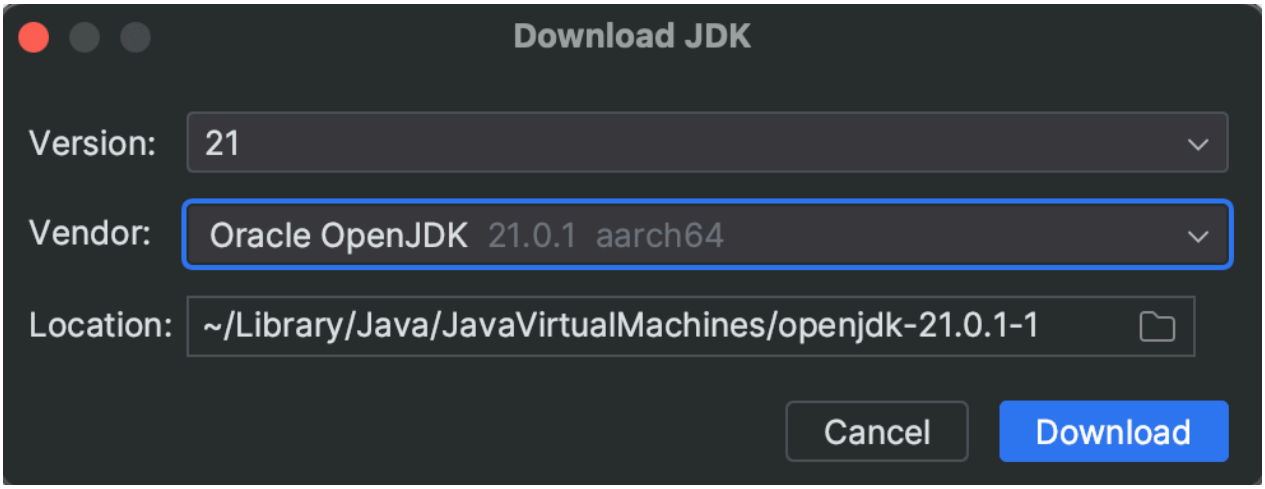
JDK 다운로드 화면 이동 방법

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



- **Name:**
  - 자바 입문편 강의: java-start
  - 자바 기본편 강의: java-basic
  - 자바 중급1편 강의: java-mid1
  - 자바 중급2편 강의: **java-mid2**

JDK 다운로드 화면



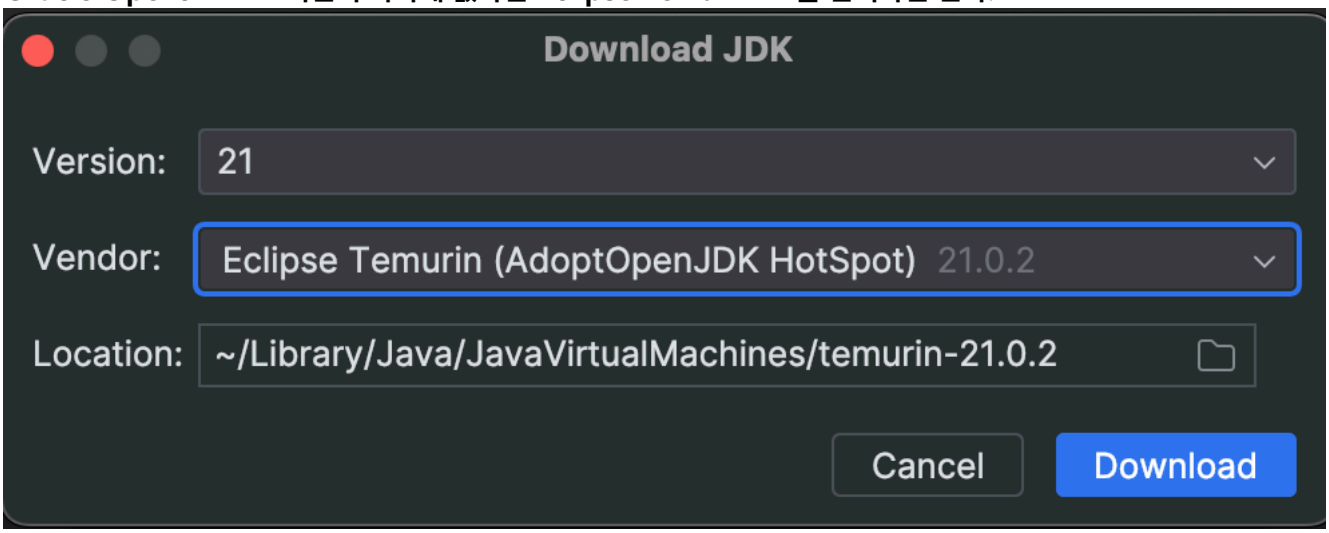
- Version: 21을 선택하자.
- Vendor: Oracle OpenJDK를 선택하자. 없다면 다른 것을 선택해도 된다.
  - aarch64: 애플 M1, M2, M3 CPU 사용시 선택, 나머지는 뒤에 이런 코드가 붙지 않은 JDK를 선택하면 된

다.

- Location: JDK 설치 위치, 기본값을 사용하자.

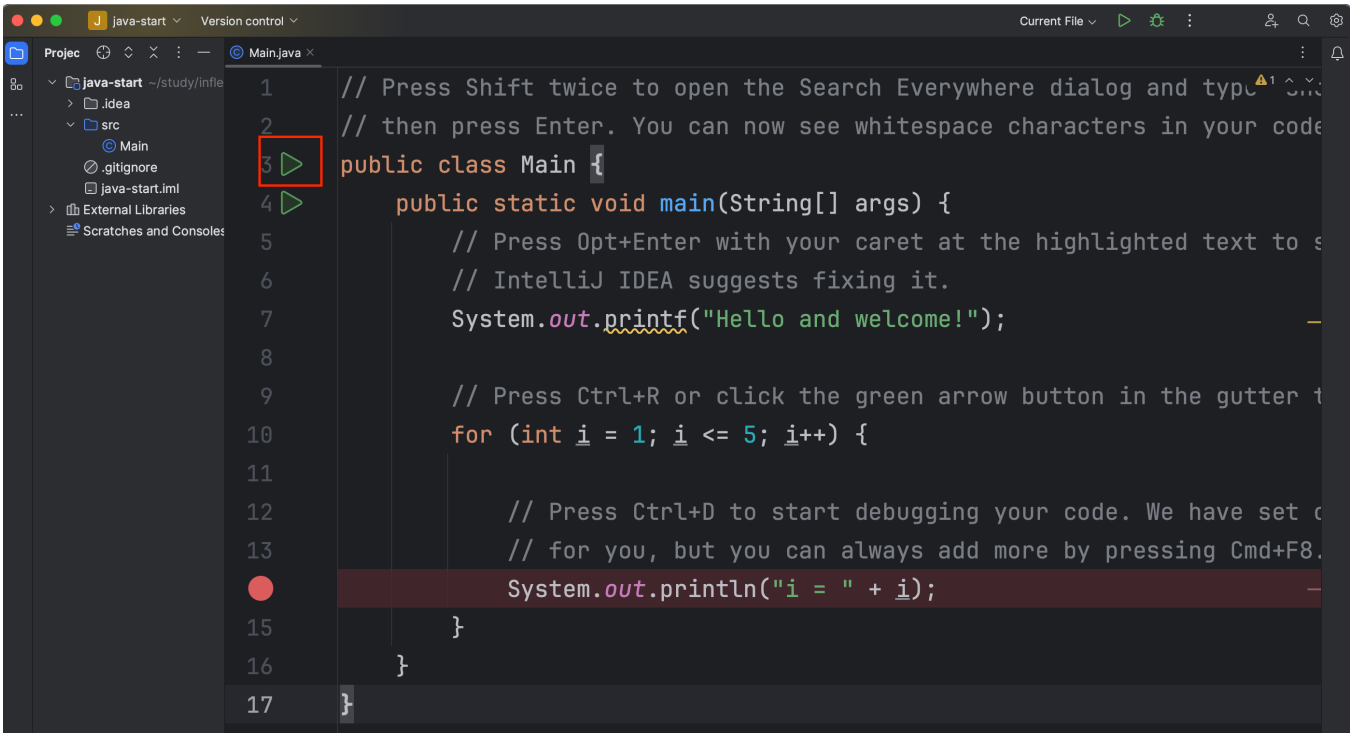
주의 - 변경 사항

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



`Download` 버튼을 통해서 다운로드 JDK를 다운로드 받는다.
다운로드가 완료 되고 이전 화면으로 돌아가면 `Create` 버튼 선택하자. 그러면 다음 IntelliJ 메인 화면으로 넘어간다.

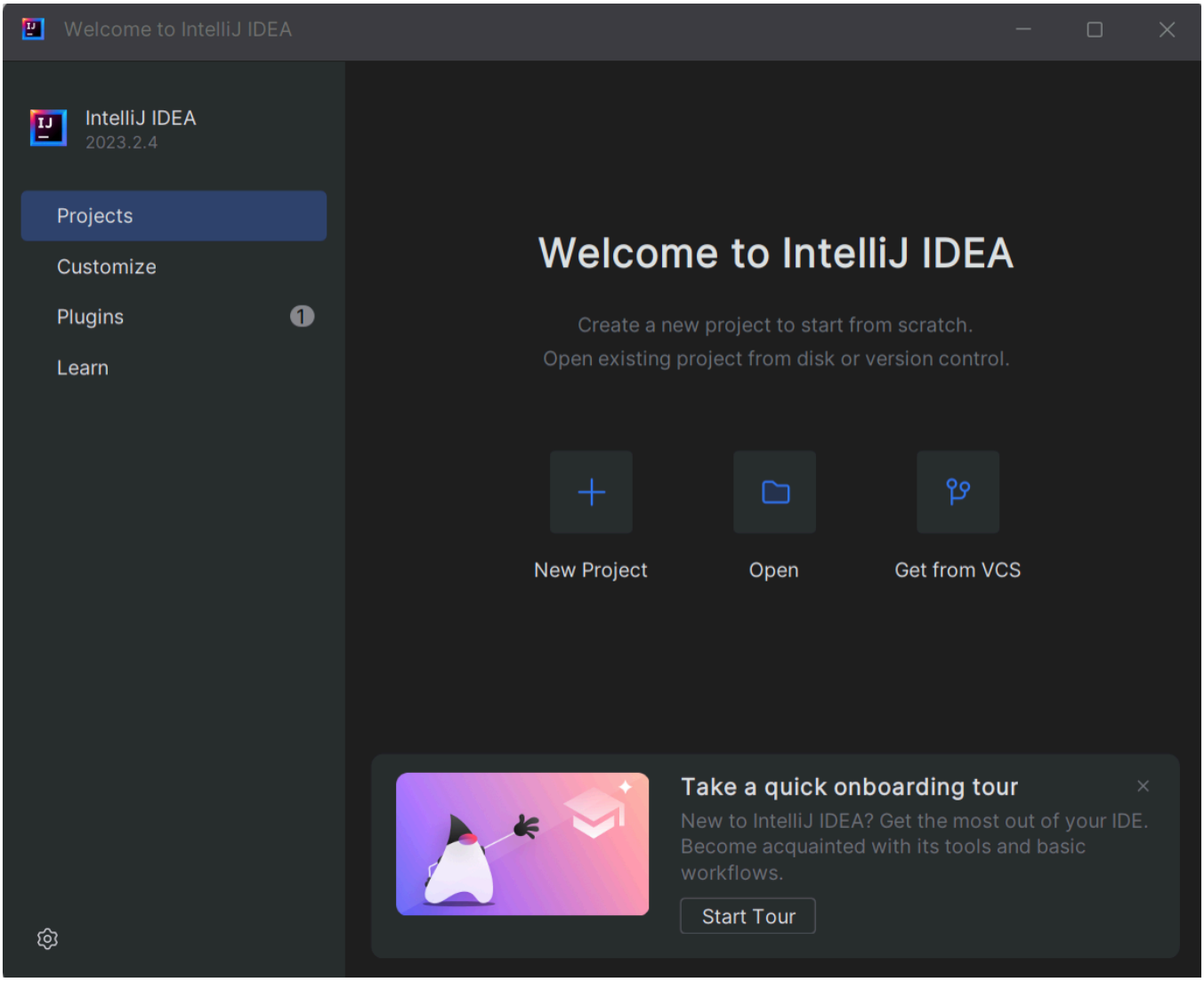
IntelliJ 메인 화면



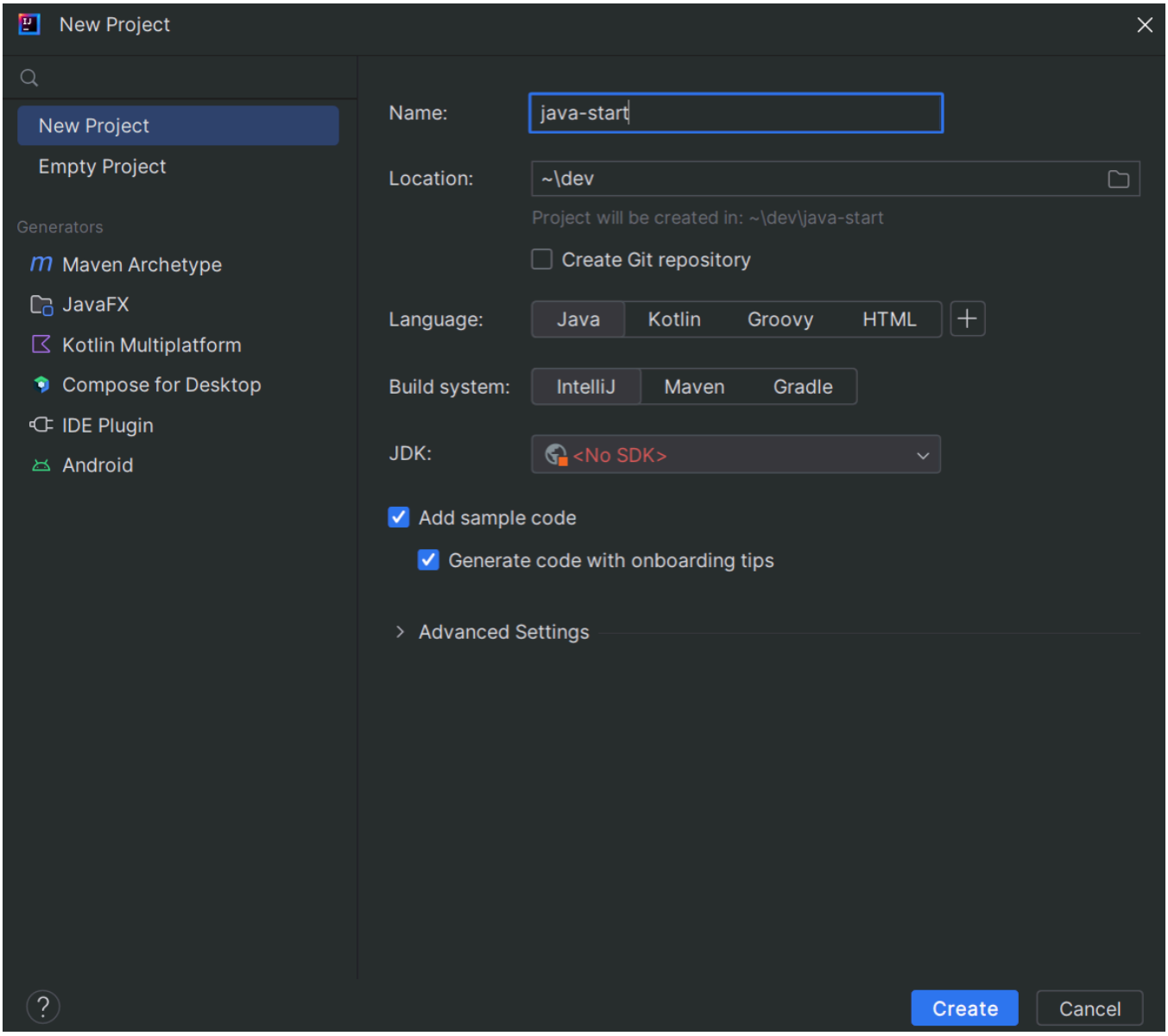
- 앞서 `Add sample code` 선택해서 샘플 코드가 만들어져 있다.
- 위쪽에 빨간색으로 강조한 초록색 화살표 버튼을 선택하고 `Run 'Main.main()` 버튼을 선택하면 프로그램이 실행된다.

윈도우 사용자 추가 설명서

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

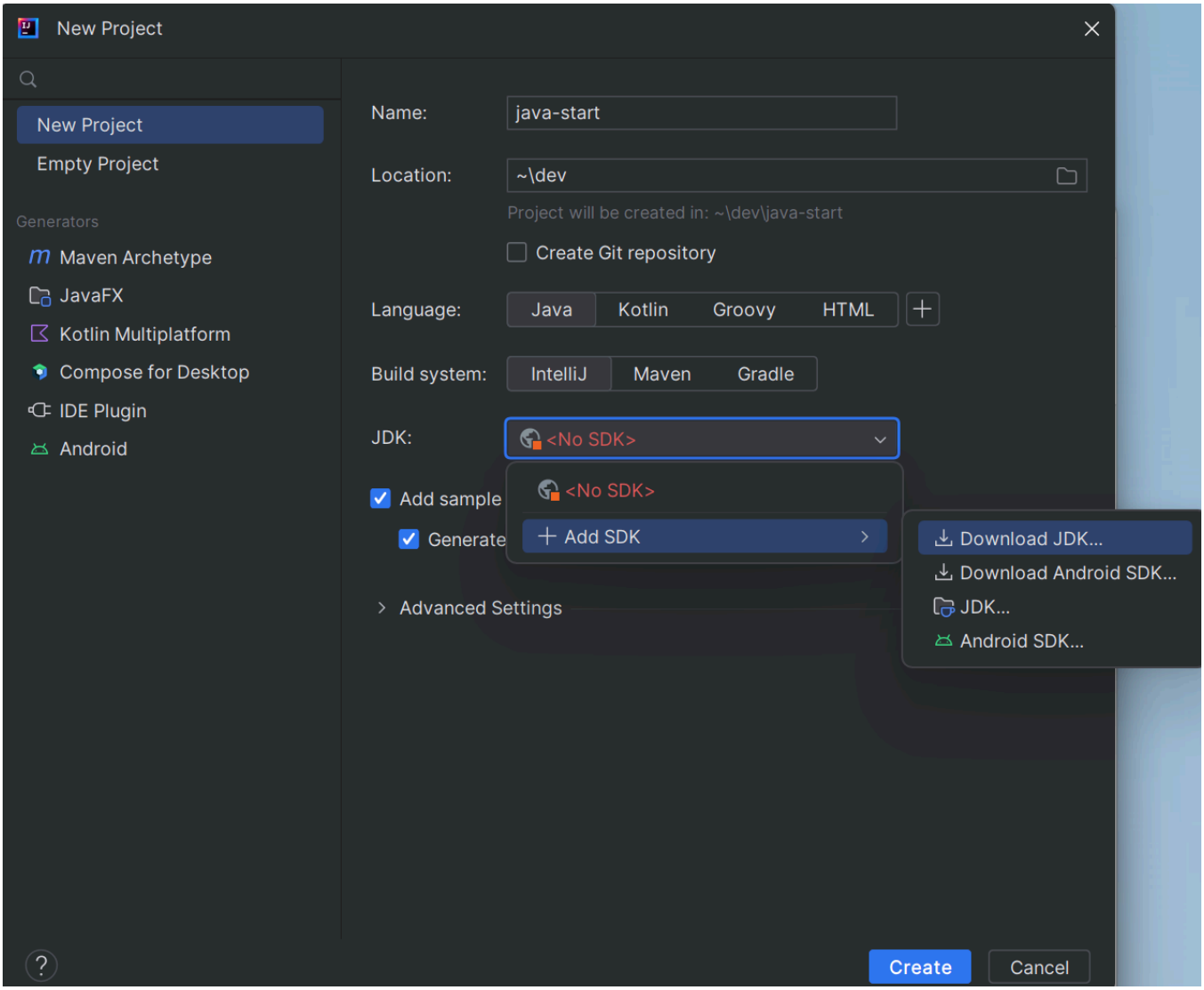


- 프로그램 시작 화면
- New Project 선택

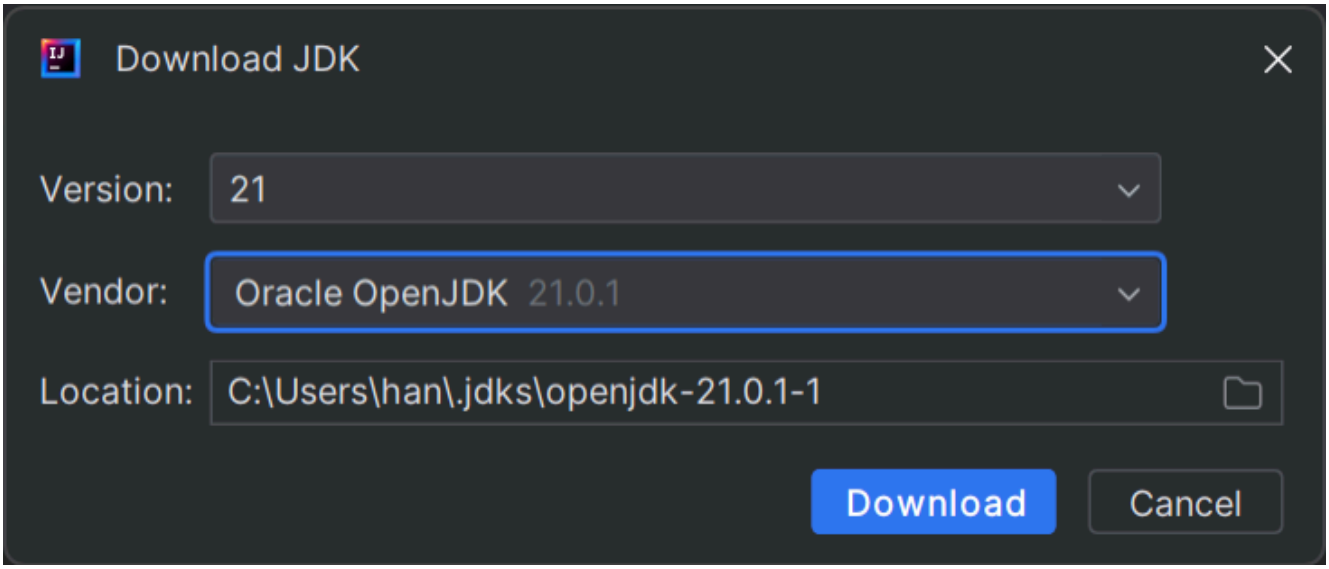


New Project 화면

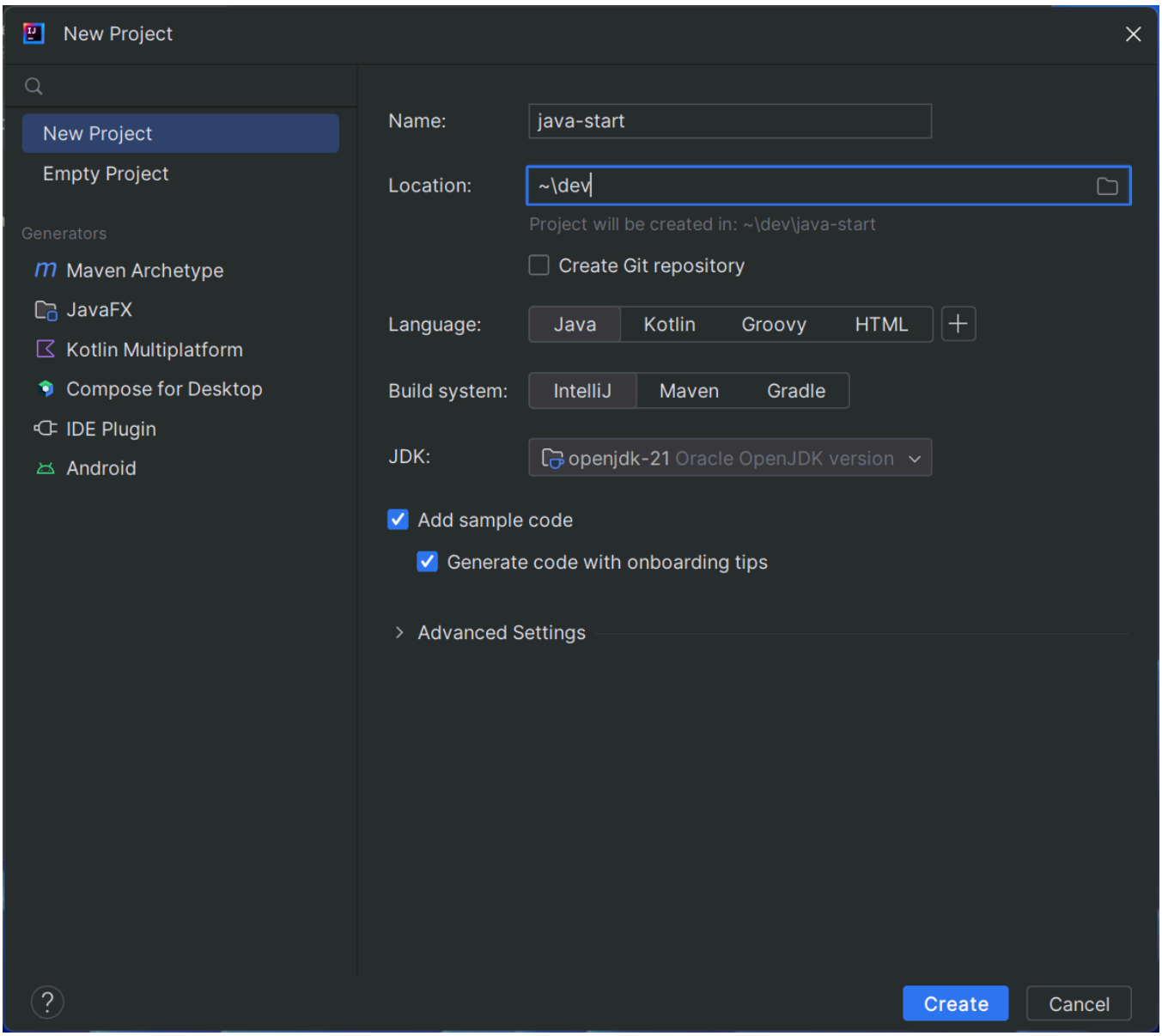
- **Name:**
  - 자바 입문편 강의: java-start
  - 자바 기본편 강의: java-basic
  - 자바 중급1편 강의: java-mid1
  - 자바 중급2편 강의: **java-mid2**
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: 자바 버전 17 이상
- `Add sample code` 선택



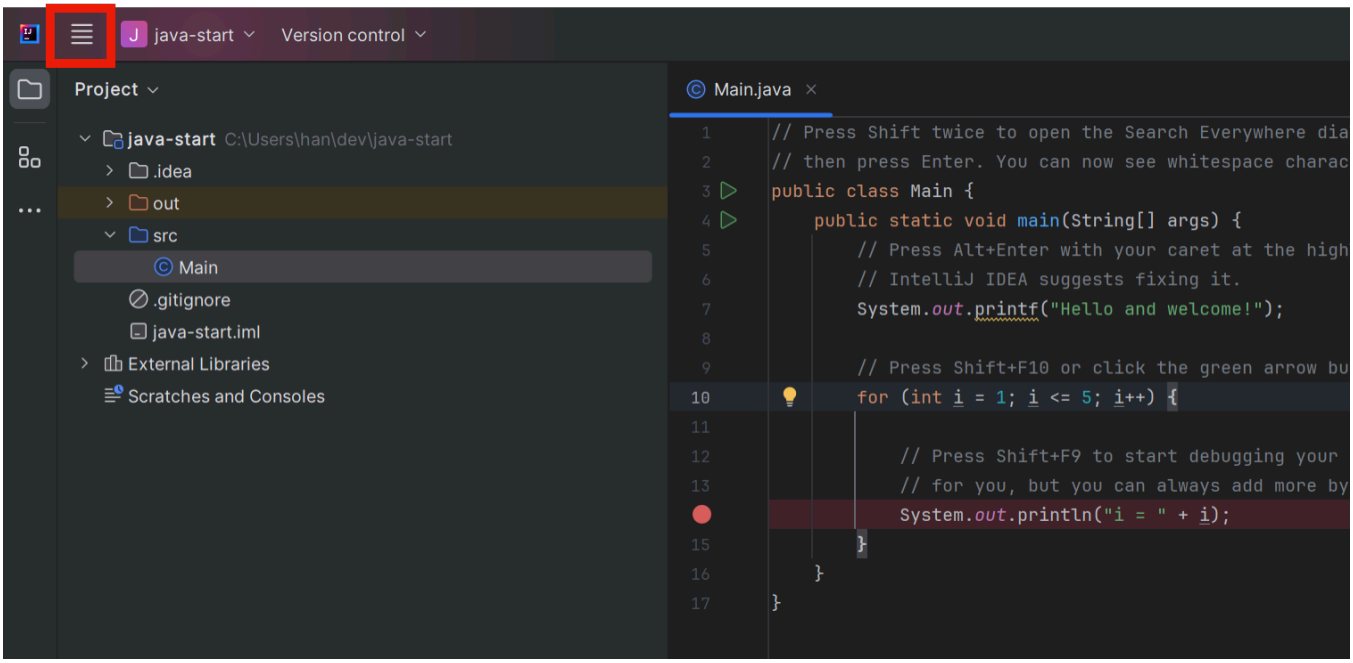
JDK 설치는 Mac과 동일하다.



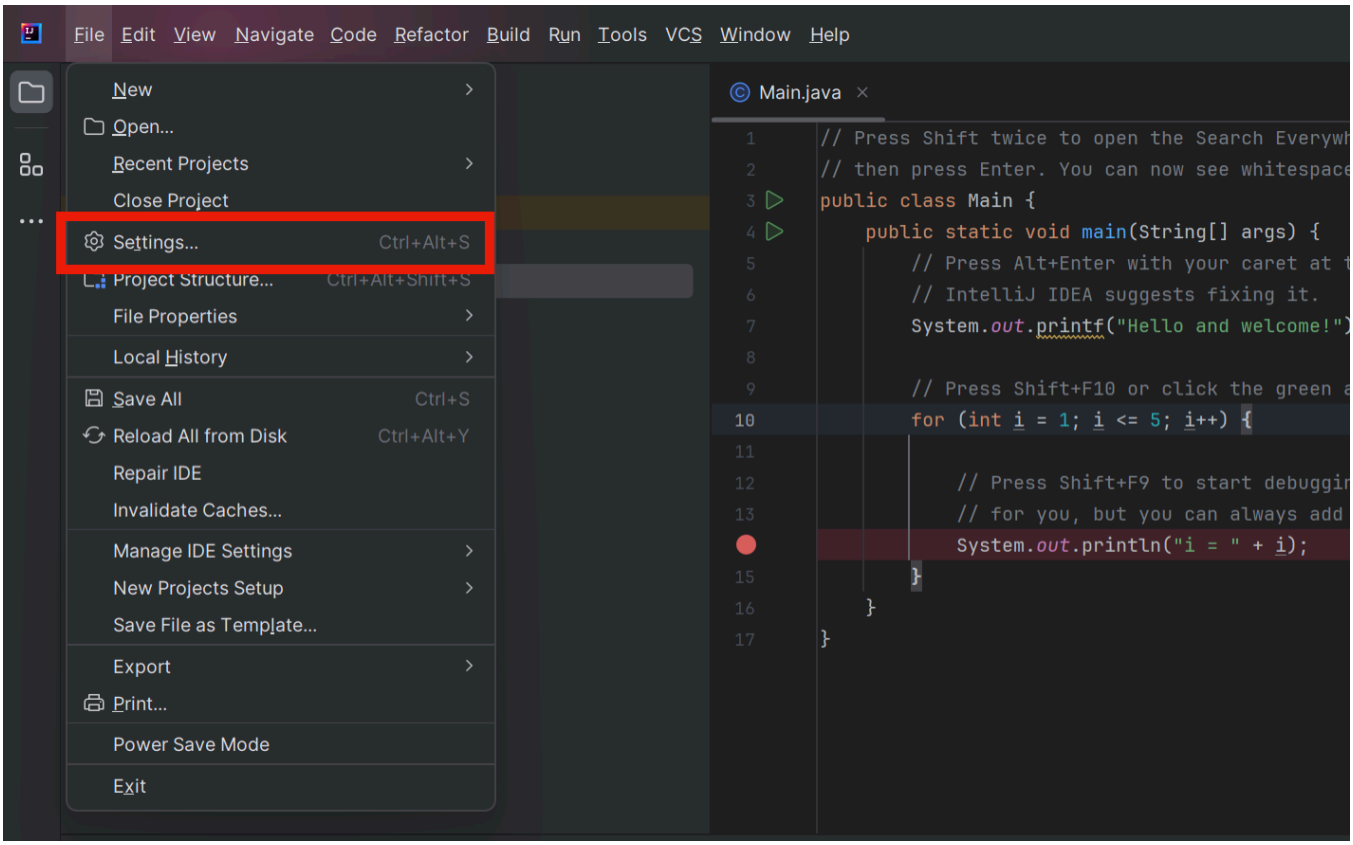
- Version: 21
- Vendor: Oracle OpenJDK
- Location은 가급적 변경하지 말자.



- New Project 완료 화면



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



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

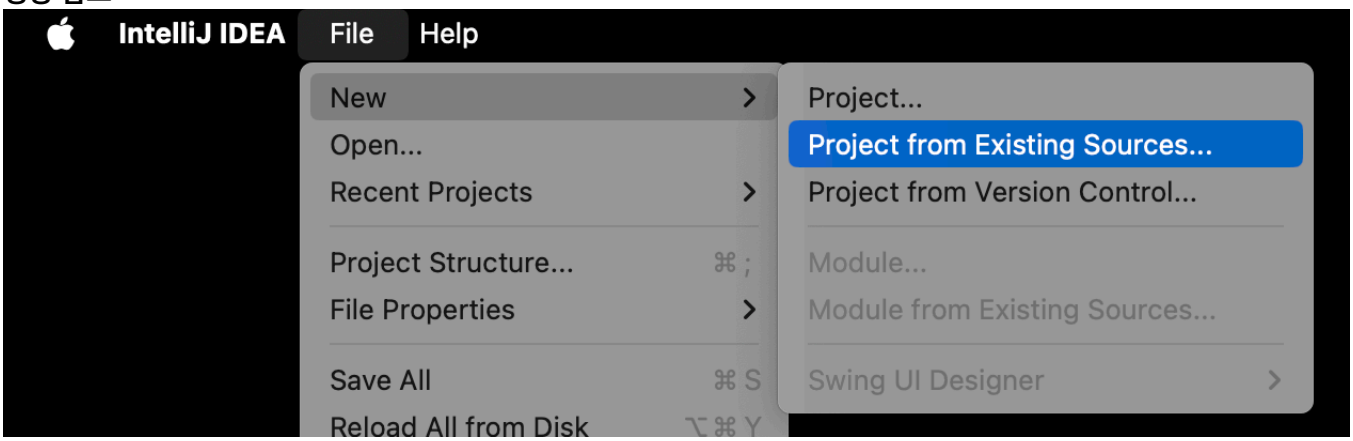
한글 언어팩 → 영어로 변경

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

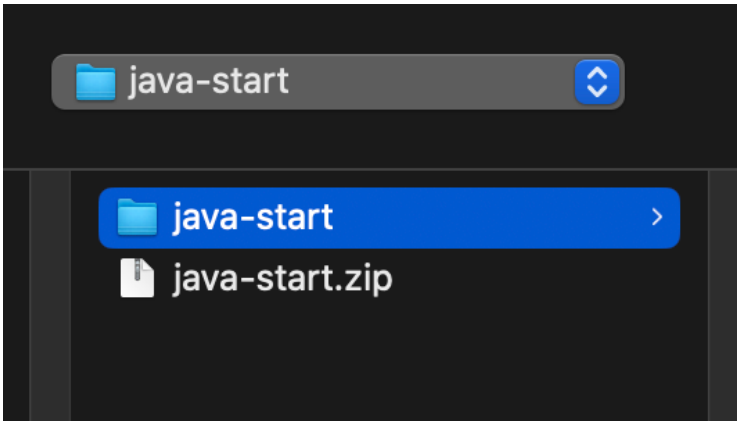
- 만약 한글로 나온다면 다음과 같이 영문으로 변경하자.
- **Mac**: IntelliJ IDEA(메뉴) ➜ Settings... ➜ Plugins ➜ Installed
- **윈도우**: File ➜ Settings... ➜ Plugins ➜ Installed
  - Korean Language Pack 체크 해제
  - OK 선택후 IntelliJ 다시 시작

다운로드 소스 코드 실행 방법

영상 참고

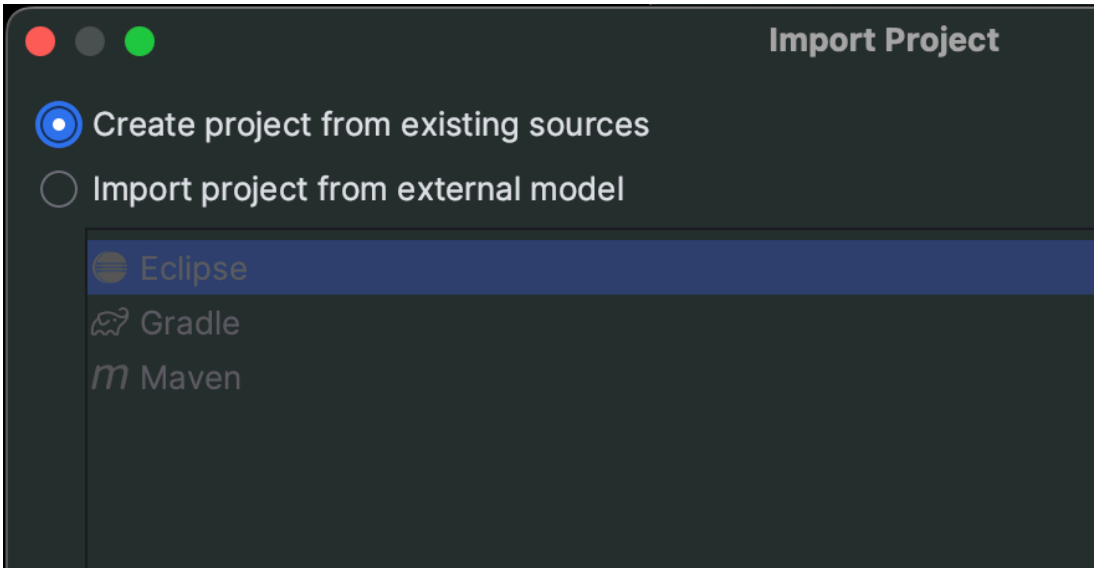


`File -> New -> Project from Existing Sources...` 선택



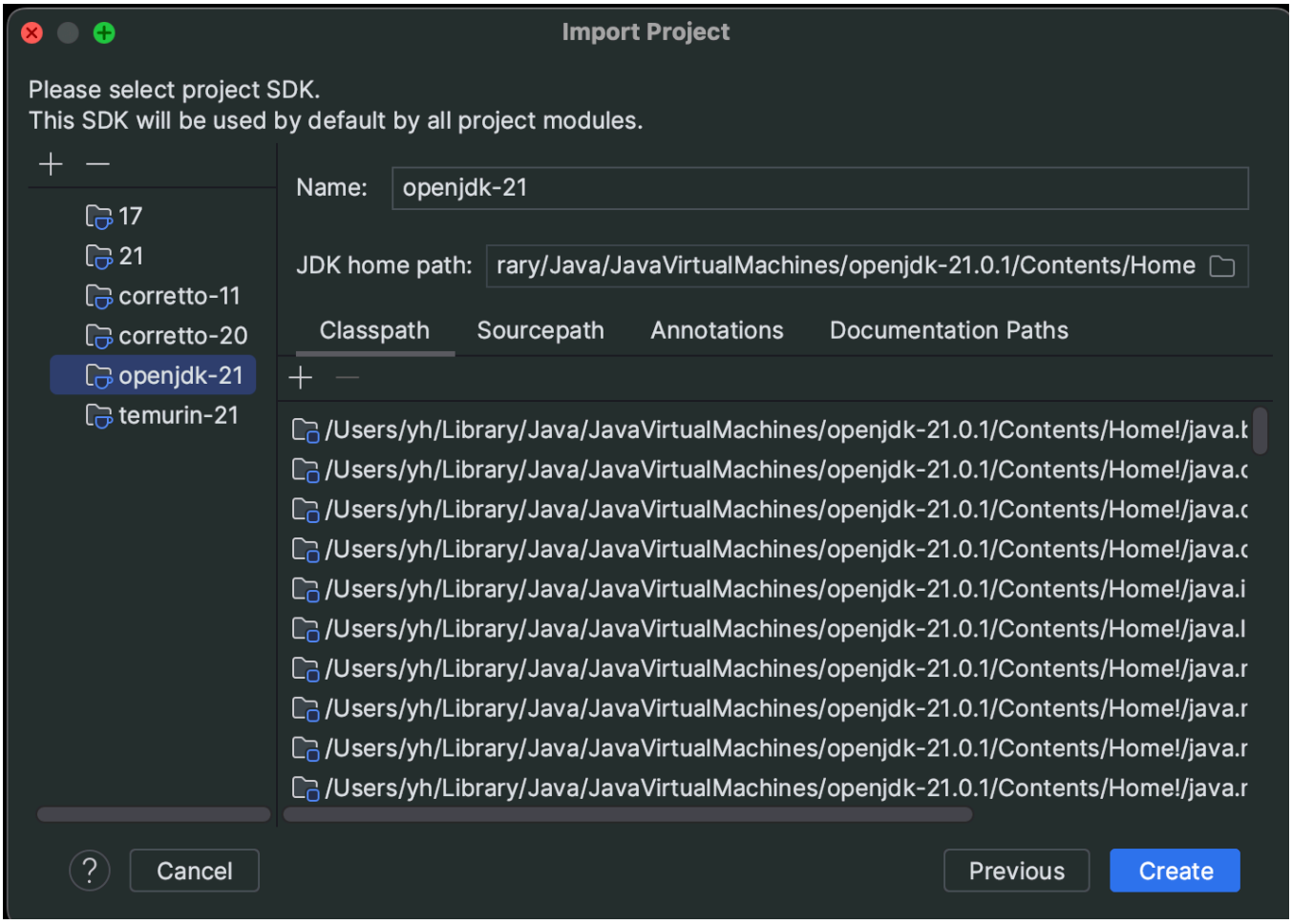
압축을 푼 프로젝트 폴더 선택

- 자바 입문 강의 폴더: java-start
- 자바 기본 강의 폴더: **java-basic**
- 자바 중급1편 강의 폴더: **java-mid1**
- 자바 중급2편 강의 폴더: **java-mid2**

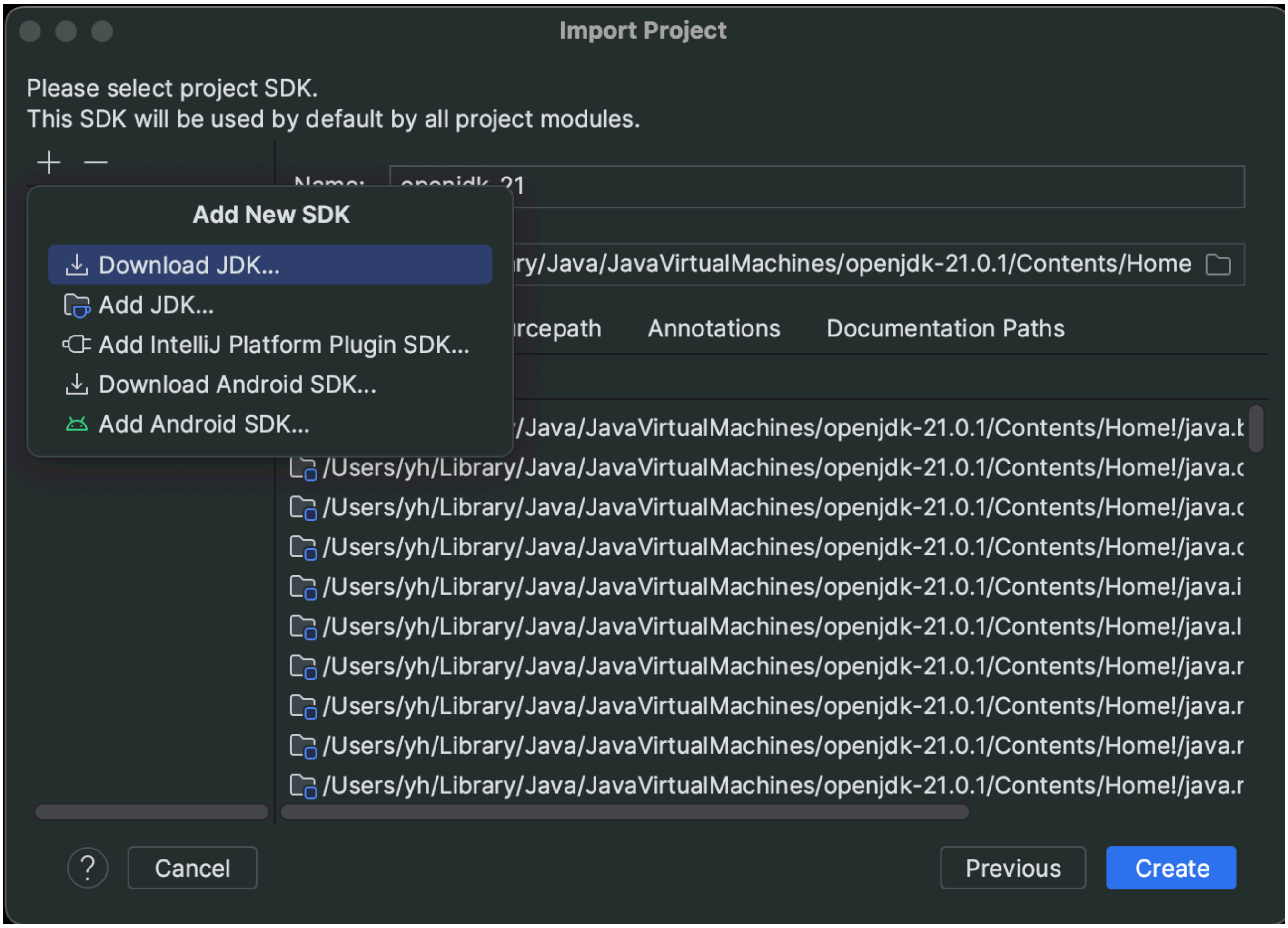


`Create project from existing sources` 선택

이후 계속 `Next` 선택



openjdk-21 선택



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

이후 `Create` 버튼 선택

제네릭이 필요한 이유

대부분의 최신 프로그래밍 언어는 제네릭(Generic) 개념을 제공한다.

처음 제네릭을 접하는 경우 내용을 이해하기 쉽지 않을 수 있다. 따라서 단계별로 천천히 진행하겠다.

제네릭이 왜 필요한지 지금부터 코드로 알아보자.

```
package generic.ex1;

public class IntegerBox {

    private Integer value;
```

```
    public void set(Integer value) {
        this.value = value;
    }

    public Integer get() {
        return value;
    }
}
```

- 숫자를 보관하고 꺼낼 수 있는 단순한 기능을 제공한다.

```
package generic.ex1;

public class StringBox {

    private String value;

    public void set(String object) {
        this.value = object;
    }

    public String get() {
        return value;
    }
}
```

- 문자열을 보관하고 꺼낼 수 있는 단순한 기능을 제공한다.

```
package generic.ex1;

public class BoxMain1 {

    public static void main(String[] args) {
        IntegerBox integerBox = new IntegerBox();
        integerBox.set(10); //오토 박싱
        Integer integer = integerBox.get();
        System.out.println("integer = " + integer);

        StringBox stringBox = new StringBox();
        stringBox.set("hello");
```

```
        String str = stringBox.get();
        System.out.println("str = " + str);
    }

}
```

실행 결과

```
integer = 10
str = hello
```

코드를 보면 먼저 숫자를 보관하는 `IntegerBox`를 생성하고, 그곳에 숫자 `10`을 보관하고, 꺼낸 다음에 출력했다.
참고로 오토 박싱에 의해 `int`가 `Integer`로 자동 변환된다.
다음으로 문자열을 보관하는 `StringBox`를 생성하고 그곳에 문자열 `"hello"`를 보관하고, 꺼낸 다음에 출력했다.

문제
이후에 `Double`, `Boolean`을 포함한 다양한 타입을 담는 박스가 필요하다면 각각의 타입별로 `DoubleBox`, `BooleanBox`와 같이 클래스를 새로 만들어야 한다. 담는 타입이 수십개라면, 수십개의 `XxxBox` 클래스를 만들어야 한다.
이 문제를 어떻게 해결할 수 있을까?

다형성을 통한 중복 해결 시도

`Object`는 모든 타입의 부모이다. 따라서 다형성(다형적 참조)를 사용해서 이 문제를 간단히 해결할 수 있을 것 같다.

```
package generic.ex1;

public class ObjectBox {

    private Object value;

    public void set(Object object) {
        this.value = object;
```

```
    }

    public Object get() {
        return value;
    }
}
```

- 내부에 `Object value`를 가지고 있다. `Object`는 모든 타입의 부모이다. 부모는 자식을 담을 수 있으므로 세상의 모든 타입을 `ObjectBox`에 보관할 수 있다.

```
package generic.ex1;

public class BoxMain2 {

    public static void main(String[] args) {
        ObjectBox integerBox = new ObjectBox();
        integerBox.set(10);
        Integer integer = (Integer) integerBox.get(); //Object -> Integer 캐스팅
        System.out.println("integer = " + integer);

        ObjectBox stringBox = new ObjectBox();
        stringBox.set("hello");
        String str = (String) stringBox.get(); //Object -> String 캐스팅
        System.out.println("str = " + str);

        //잘못된 타입의 인수 전달시
        integerBox.set("문자100");
        Integer result = (Integer) integerBox.get(); // String -> Integer 캐스팅
예외
        System.out.println("result = " + result);
    }

}
```

실행 결과

```
integer = 10
str = hello
Exception in thread "main" java.lang.ClassCastException: class
java.lang.String cannot be cast to class java.lang.Integer (java.lang.String
```

```
and java.lang.Integer are in module java.base of loader 'bootstrap')
    at generic.ex1.BoxMain2.main(BoxMain2.java:24)
```

잘 동작하는 것 같지만 몇 가지 문제가 있다.

반환 타입이 맞지 않는 문제

먼저 `integerBox`를 만들어서 숫자 `10`을 보관했다. 숫자를 입력하는 부분에는 문제가 없어 보이지만, `integerBox.get()`을 호출할 때 문제가 나타난다.

`integerBox.get()`의 반환 타입은 `Object`이다.

```
Object obj = integerBox.get();
```

`Integer = Object`는 성립하지 않는다. 자식은 부모를 담을 수 없다. 따라서 다음과 같이 `(Integer)` 타입 캐스팅 코드를 넣어서 `Object` 타입을 `Integer` 타입으로 직접 다운 캐스팅해야 한다.

```
Integer integer = (Integer) integerBox.get() //1
Integer integer = (Integer) (Object)value //2
Integer integer = (Integer)value //3
```

`stringBox`의 경우도 마찬가지이다. `stringBox.get()`이 `Object`를 반환하므로 다음과 같이 다운 캐스팅해야 한다.

```
String str = (String) stringBox.get()
```

잘못된 타입의 인수 전달 문제

```
integerBox.set("문자100");
```

개발자의 의도는 `integerBox`에는 변수 이름과 같이 숫자 타입이 입력되기를 기대했다.
하지만 `set(Object ..)` 메서드는 모든 타입의 부모인 `Object`를 매개변수로 받기 때문에 세상의 어떤 데이터도 입력받을 수 있다. 따라서 이렇게 문자열을 입력해도 자바 언어 입장에서는 아무런 문제가 되지 않는다.

잘못된 타입의 값을 전달하면 값을 꺼낼 때 문제가 발생한다.

```
Integer result = (Integer) integerBox.get(); //1
Integer result = (Integer) "문자100"; //2
Integer result = (Integer) "문자100"; //3. 예외 발생 String을 Integer로 캐스팅할 수 없다.
```

숫자가 들어가 있을 것으로 예상한 박스에는 문자열이 들어가 있었다. 결과적으로 다운 캐스팅시에 `String`을 `Integer`로 캐스팅 할 수 없다는 예외가 발생하고 프로그램이 종료된다.

정리

다형성을 활용한 덕분에 코드의 중복을 제거하고, 기존 코드를 재사용할 수 있게 되었다. 하지만 입력할 때 실수로 원하지 않는 타입이 들어갈 수 있는 타입 안전성 문제가 발생한다. 예를 들어서 `integerBox`에는 숫자만 넣어야 하고, `stringBox`에는 문자열만 입력할 수 있어야 한다. 하지만 박스에 값을 보관하는 `set()`의 매개변수가 `Object`이기 때문에 다른 타입의 값을 입력할 수 있다. 그리고 반환 시점에도 `Object`를 반환하기 때문에 원하는 타입을 정확하게 받을 수 없고, 항상 위험한 다운 캐스팅을 시도해야한다. 결과적으로 이 방식은 타입 안전성이 떨어진다.

지금까지 개발한 프로그램은 코드 재사용과 타입 안전성이라는 2마리 토끼를 한번에 잡을 수 없다. 코드 재사용을 늘리기 위해 `Object`와 다형성을 사용하면 타입 안전성이 떨어지는 문제가 발생한다.

- `BoxMain1` : 각각의 타입별로 `IntegerBox`, `StringBox`와 같은 클래스를 모두 정의
  - 코드 재사용X
  - 타입 안전성O
- `BoxMain2` : `ObjectBox`를 사용해서 다형성으로 하나의 클래스만 정의
  - 코드 재사용O
  - 타입 안전성X

제네릭 적용

제네릭을 사용하면 코드 재사용과 타입 안전성이라는 두 마리 토끼를 한 번에 잡을 수 있다.
제네릭을 사용해서 문제를 해결해보자.

제네릭 적용 예제

```
package generic.ex1;

public class GenericBox<T> {
```

```
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}
```

- `<>`를 사용한 클래스를 제네릭 클래스라 한다. 이 기호(`<>`)를 보통 다이아몬드라 한다.
- 제네릭 클래스를 사용할 때는 `Integer`, `String` 같은 타입을 미리 결정하지 않는다.
- 대신에 클래스명 오른쪽에 `<T>`와 같이 선언하면 제네릭 클래스가 된다. 여기서 `T`를 **타입 매개변수**라 한다. 이 타입 매개변수는 이후에 `Integer`, `String`으로 변할 수 있다.
- 그리고 클래스 내부에 `T` 타입이 필요한 곳에 `T value`와 같이 타입 매개변수를 적어두면 된다.

```
package generic.ex1;

public class BoxMain3 {

    public static void main(String[] args) {
        GenericBox<Integer> integerBox = new GenericBox<Integer>(); //생성 시점에
T의 타입 결정
        integerBox.set(10);
        //integerBox.set("문자100"); // Integer 타입만 허용, 컴파일 오류
        Integer integer = integerBox.get(); // Integer 타입 반환 (캐스팅 X)
        System.out.println("integer = " + integer);

        GenericBox<String> stringBox = new GenericBox<String>();
        stringBox.set("hello"); // String 타입만 허용
        String str = stringBox.get(); // String 타입만 반환
        System.out.println("str = " + str);

        //원하는 모든 타입 사용 가능
        GenericBox<Double> doubleBox = new GenericBox<Double>();
        doubleBox.set(10.5);
        Double doubleValue = doubleBox.get();
        System.out.println("doubleValue = " + doubleValue);
```

```
        //타입 추론: 생성하는 제네릭 타입 생략 가능
        GenericBox<Integer> integerBox2 = new GenericBox<>();
    }
}
```

실행 결과

```
integer = 10
str = hello
doubleValue = 10.5
```

생성 시점에 원하는 타입 지정

제네릭 클래스는 다음과 정의한다. `<>` (다이아몬드 기호)안에 타입 매개변수를 정의하면 된다.

```
class GenericBox<T>
```

제네릭 클래스는 생성하는 시점에 `<>` 사이에 원하는 타입을 지정한다.

```
new GenericBox<Integer>()
```

이렇게 하면 앞서 정의한 `GenericBox`의 `T`가 다음과 같이 지정한 타입으로 변한 다음 생성된다.

T에 Integer를 적용한 GenericBox 클래스

```
public class GenericBox<Integer> {

    private Integer value;

    public void set(Integer value) {
        this.value = value;
    }

    public Integer get() {
        return value;
    }
}
```

`T`가 모두 `Integer`로 변한다. 따라서 `Integer` 타입을 입력하고 조회할 수 있다.

이제 `set(Integer value)`이므로 이 메서드에는 `Integer` 숫자만 담을 수 있다.

```
integerBox.set(10); //성공
integerBox.set("문자100"); // Integer 타입만 허용, 컴파일 오류
```

`get()`의 경우에도 `Integer`를 반환하기 때문에 타입 캐스팅 없이 숫자 타입으로 조회할 수 있다.

```
Integer integer = integerBox.get(); // Integer 타입 반환 (캐스팅 X)
```

`String`를 사용하면 다음과 같다.

```
new GenericBox<String>()
```

T에 String를 적용한 GenericBox 클래스

```
public class GenericBox<String> {

    private String value;

    public void set(String value) {
        this.value = value;
    }

    public String get() {
        return value;
    }
}
```

`T`가 모두 `String`으로 변한다. 따라서 문자열을 입력하고, 문자열을 그대로 조회할 수 있다.

원하는 모든 타입 사용 가능

제네릭 클래스를 사용하면 다음과 같이 `GenericBox` 객체를 생성하는 시점에 원하는 타입을 마음껏 지정할 수 있다.

```
new GenericBox<Double>()
new GenericBox<Boolean>()
new GenericBox<MyClass>()
```

참고로 제네릭을 도입한다고 해서 앞서 설명한 `GenericBox<String>`, `GenericBox<Integer>`와 같은 코드가 실제 만들어지는 것은 아니다. 대신에 자바 컴파일러가 우리가 입력한 타입 정보를 기반으로 이런 코드가 있다고 가정하고 컴파일 과정에 타입 정보를 반영한다. 이 과정에서 타입이 맞지 않으면 컴파일 오류가 발생한다. 더 자세한 내용은 뒤에서 설명한다.

타입 추론

```
GenericBox<Integer> integerBox = new GenericBox<Integer>() // 타입 직접 입력
GenericBox<Integer> integerBox2 = new GenericBox<>() // 타입 추론
```

첫번째 줄의 코드를 보면 변수를 선언할 때와 객체를 생성할 때 `<Integer>`가 두 번 나온다. 자바는 왼쪽에 있는 변수를 선언할 때의 `<Integer>`를 보고 오른쪽에 있는 객체를 생성할 때 필요한 타입 정보를 얻을 수 있다. 따라서 두 번째 줄의 오른쪽 코드 `new GenericBox<>()`와 같이 타입 정보를 생략할 수 있다. 이렇게 자바가 스스로 타입 정보를 추론해서 개발자가 타입 정보를 생략할 수 있는 것을 타입 추론이라 한다.
참고로 타입 추론이 그냥 되는 것은 아니고, 자바 컴파일러가 타입을 추론할 수 있는 상황에만 가능하다. 쉽게 이야기해서 읽을 수 있는 타입 정보가 주변에 있어야 추론할 수 있다.

정리
제네릭을 사용한 덕분에 코드 재사용과 타입 안전성이라는 두 마리 토끼를 모두 잡을 수 있었다.

제네릭 용어와 관례

제네릭의 핵심은 **사용할 타입을 미리 결정하지 않는다는 점**이다. 클래스 내부에서 사용하는 타입을 클래스를 정의하는 시점에 결정하는 것이 아니라 실제 사용하는 생성 시점에 타입을 결정하는 것이다.
이것을 쉽게 비유하자면 메서드의 매개변수와 인자의 관계와 비슷하다.

메서드에 필요한 값을 메서드 정의 시점에 미리 결정

```
void method1() {
```

```
    println("hello");
}
```

- 메서드에 필요한 값을 이렇게 메서드 정의 시점에 미리 결정하게 되면, 이 메서드는 오직 `"hello"` 라는 값만 출력할 수 있다. 따라서 재사용성이 떨어진다.

메서드에 필요한 값을 인자를 통해 매개변수로 전달해서 결정

```
void method2(String param) {
    println(param);
}

void main() {
    method2("hello");
    method2("hi");
}
```

- 메서드에 필요한 값을 메서드를 정의하는 시점에 미리 결정하는 것이 아니라, 메서드를 실제 사용하는 시점으로 미룰 수 있다.
- 메서드에 매개변수(`String param`)를 지정하고, 매서드를 사용할 때 원하는 값을 인자(`"hello"`, `"hi"`)로 전달하면 된다.

다양한 값을 처리하는 메서드

```
//method2("hello") 호출 예
void method2(String param="hello") {
    println(param);
}

//method2("hi") 호출 예
void method2(String param="hi") {
    println(param);
}
```

매개변수를 정의하고, 실행 시점에 인자를 통해 원하는 값을 매개변수에 전달했다.
이렇게 하면 이 메서드는 실행 시점에 얼마든지 다른 값을 받아서 처리할 수 있다. 따라서 재사용성이 크게 늘어난다.

메서드의 매개변수와 인자

```
void method(String param) //매개변수
```

```
void main() {
  String arg = "hello";
  method(arg) //인수 전달
}
```

- 매개변수(Parameter): `String param`
- 인자, 인수(Argument): `arg`

메서드의 매개변수에 인자를 전달해서 메서드의 사용 값을 결정한다.

제네릭의 타입 매개변수와 타입 인자

제네릭도 앞서 설명한 메서드의 매개변수와 인자의 관계와 비슷하게 작동한다.

제네릭 클래스를 정의할 때 내부에서 사용할 타입을 미리 결정하는 것이 아니라, 해당 클래스를 실제 사용하는 생성 시점에 내부에서 사용할 타입을 결정하는 것이다. 차이가 있다면 **메서드의 매개변수는 사용할 값에 대한 결정을 나중으로 미루는 것**이고, **제네릭의 타입 매개변수는 사용할 타입에 대한 결정을 나중으로 미루는 것**이다.

정리하면 다음과 같다.

- 메서드는 **매개변수**에 **인자**를 전달해서 사용할 값을 결정한다.
- 제네릭 클래스는 **타입 매개변수**에 **타입 인자**를 전달해서 사용할 타입을 결정한다.

제네릭에서 사용하는 용어도 매개변수, 인자의 용어를 그대로 가져다 사용한다. 다만 값이 아니라 **타입을 결정**하는 것이기 때문에 앞에 타입을 붙인다.

- 타입 매개변수: `GenericBox<T>` 에서 `T`
- 타입 인자:
  - `GenericBox<Integer>` 에서 `Integer`
  - `GenericBox<String>` 에서 `String`

제네릭 타입의 타입 매개변수 `<T>` 에 타입 인자를 전달해서 제네릭의 사용 타입을 결정한다.

`GenericBox<T>`

- `String` ➜ `GenericBox<String>`
- `Integer` ➜ `GenericBox<Integer>`

용어 정리

- **제네릭(Generic) 단어**
  - 제네릭이라는 단어는 일반적인, 범용적인이라는 영어 단어 뜻이다.
  - 풀어보면 특정 타입에 속한 것이 아니라 일반적으로, 범용적으로 사용할 수 있다는 뜻이다.
- **제네릭 타입 (Generic Type)**

    - 클래스나 인터페이스를 정의할 때 타입 매개변수를 사용하는 것을 말한다.
    - 제네릭 클래스, 제네릭 인터페이스를 모두 합쳐서 제네릭 타입이라 한다.
        - 타입은 클래스, 인터페이스, 기본형(`int` 등)을 모두 합쳐서 부르는 말이다.
    - 예: `class GenericBox<T> { private T t; }`
    - 여기에서 `GenericBox<T>`를 제네릭 타입이라 한다.
- **타입 매개변수 (Type Parameter)**
    - 제네릭 타입이나 메서드에서 사용되는 변수로, 실제 타입으로 대체된다.
    - 예: `GenericBox<T>`
    - 여기에서 `T`를 타입 매개변수라 한다.
- **타입 인자 (Type Argument)**
    - 제네릭 타입을 사용할 때 제공되는 실제 타입이다.
    - 예: `GenericBox<Integer>`
    - 여기에서 `Integer`를 타입 인자라 한다.

제네릭 명명 관례

타입 매개변수는 일반적인 변수명처럼 소문자로 사용해도 문제는 없다.
하지만 일반적으로 대문자를 사용하고 용도에 맞는 단어의 첫글자를 사용하는 관례를 따른다.

주로 사용하는 키워드는 다음과 같다.

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

제네릭 기타

다음과 같이 한번에 여러 타입 매개변수를 선언할 수 있다.

```
class Data<K, V> {}
```

타입 인자로 기본형은 사용할 수 없다

제네릭의 타입 인자로 기본형(`int`, `double` ..)은 사용할 수 없다. 대신에 래퍼 클래스(`Integer`, `Double`)를 사용하면 된다.

로 타입 - raw type

```
package generic.ex1;

public class RawTypeMain {

    public static void main(String[] args) {
        GenericBox integerBox = new GenericBox();
        //GenericBox<Object> integerBox = new GenericBox<>(); // 권장
        integerBox.set(10);
        Integer result = (Integer) integerBox.get();
        System.out.println("result = " + result);
    }
}
```

> 참고: 영상에 `RowTypeMain`으로 오타가 있는데 Row ➡ Raw로 정정합니다.

실행 결과

```
result = 10
```

제네릭 타입을 사용할 때는 항상 `<>`를 사용해서 사용시점에 원하는 타입을 지정해야 한다.
그런데 다음과 같이 `<>`을 지정하지 않을 수 있는데, 이런 것을 로 타입(raw type), 또는 원시 타입이라한다.

```
GenericBox integerBox = new GenericBox();
```

원시 타입을 사용하면 내부의 타입 매개변수가 `Object`로 사용된다고 이해하면 된다.

제네릭 타입을 사용할 때는 항상 `<>`를 사용해서 사용시점에 타입을 지정해야 한다. 그런데 왜 이런 로 타입을 지원하는 것일까?
자바의 제네릭이 자바가 처음 등장할 때 부터 있었던 것이 아니라 자바가 오랜기간 사용된 이후에 등장했기 때문에 제네릭이 없던 시절의 과거 코드와의 하위 호환이 필요했다. 그래서 어쩔 수 없이 이런 로 타입을 지원한다.

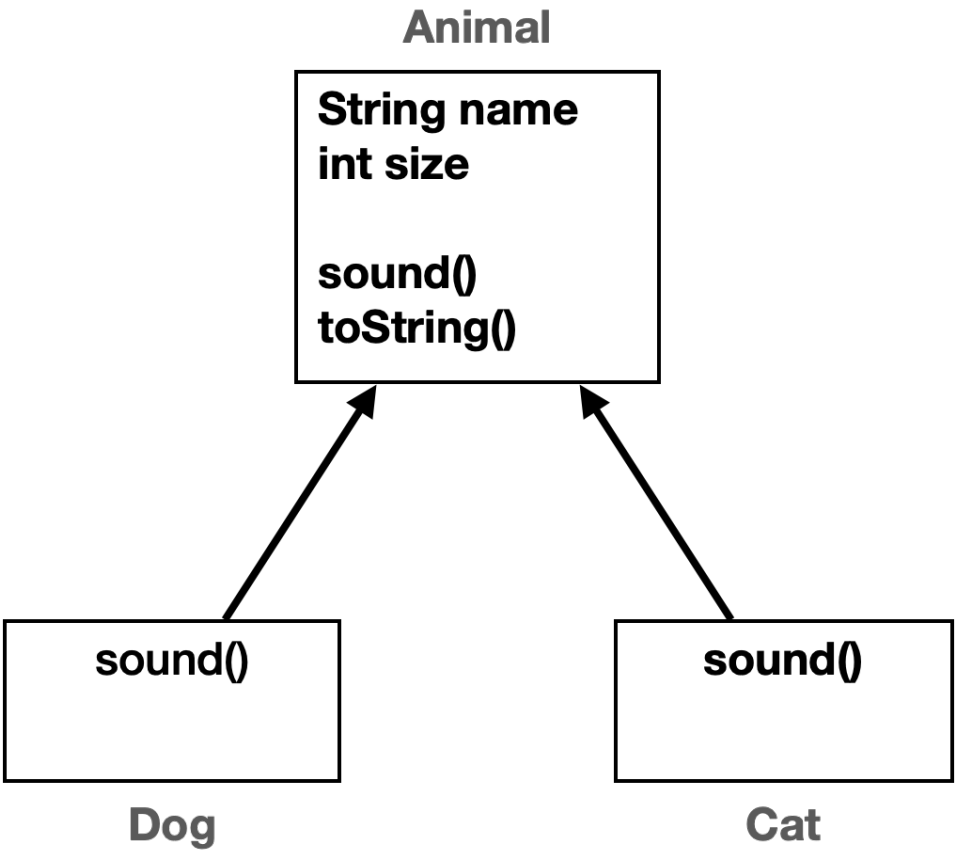
정리하면 로 타입을 사용하지 않아야 한다.
만약에 `Object` 타입을 사용해야 한다면 다음과 같이 타입 인자로 `Object`를 지정해서 사용하면 된다.

```
GenericBox<Object> integerBox = new GenericBox<>();
```

제네릭 활용 예제

이번에는 직접 클래스를 만들고, 제네릭도 도입해보자.

지금부터 사용할 `Animal` 관련 클래스들은 이후 예제에서도 사용하므로 `generic.animal`이라는 별도의 패키지에서 관리하겠다.




```
package generic.animal;

public class Animal {

    private String name;
    private int size;

    public Animal(String name, int size) {
```

```
        this.name = name;
        this.size = size;
    }

    public String getName() {
        return name;
    }

    public int getSize() {
        return size;
    }

    public void sound() {
        System.out.println("동물 울음 소리");
    }

    @Override
    public String toString() {
        return "Animal{" +
                "name='" + name + '\'' +
                ", size=" + size +
                '}';
    }
}
```

- 이름(`name`), 크기(`size`) 정보를 가지는 부모 클래스이다.
- `toString()`을 IDE를 통해서 오버라이딩 했다.

```
package generic.animal;

public class Dog extends Animal {

    public Dog(String name, int size) {
        super(name, size);
    }

    @Override
    public void sound() {
        System.out.println("멍멍");
    }

}
```

- `Animal`을 상속 받는다.
- 부모 클래스에 정의된 생성자가 있기 때문에 맞추어 `super(name, size)`를 호출한다.

```
package generic.animal;

public class Cat extends Animal {

    public Cat(String name, int size) {
        super(name, size);
    }

    @Override
    public void sound() {
        System.out.println("냐옹");
    }
}
```

여기서부터 패키지가 달라지므로 주의하자.

```
package generic.ex2;

public class Box<T> {

    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }

}
```

- 객체를 보관할 수 있는 제네릭 클래스다.

```java
package generic.ex2;

import generic.animal.Animal;
import generic.animal.Cat;
import generic.animal.Dog;

public class AnimalMain1 {

    public static void main(String[] args) {
        Animal animal = new Animal("동물", 0);
        Dog dog = new Dog("멍멍이", 100);
        Cat cat = new Cat("냐옹이", 50);

        Box<Dog> dogBox = new Box<>();
        dogBox.set(dog);
        Dog findDog = dogBox.get();
        System.out.println("findDog = " + findDog);

        Box<Cat> catBox = new Box<>();
        catBox.set(cat);
        Cat findCat = catBox.get();
        System.out.println("findCat = " + findCat);

        Box<Animal> animalBox = new Box<>();
        animalBox.set(animal);
        Animal findAnimal = animalBox.get();
        System.out.println("findAnimal = " + findAnimal);
    }
}
```

실행 결과

```
findDog = Animal{name='멍멍이', size=100}
findCat = Animal{name='냐옹이', size=50}
findAnimal = Animal{name='동물', size=0}
```

- `Box` 제네릭 클래스에 각각의 타입에 맞는 동물을 보관하고 꺼낸다.
- `Box<Dog> dogBox` : `Dog` 타입을 보관할 수 있다.
- `Box<Cat> catBox` : `Cat` 타입을 보관할 수 있다.
- `Box<Animal> animalBox` : `Animal` 타입을 보관할 수 있다.

여기서 `Box<Animal>`의 경우 타입 매개변수 `T`에 타입 인자 `Animal`을 대입하면 다음 코드와 같다.

```
public class Box<Animal> {

    private Animal value;

    public void set(Animal value) {
        this.value = value;
    }

    public Animal get() {
        return value;
    }

}
```

- 따라서 `set(Animal value)`이므로 `set()`에 `Animal`의 하위 타입인 `Dog`, `Cat`도 전달할 수 있다.
- 물론 이 경우 꺼낼 때는 `Animal` 타입으로만 꺼낼 수 있다.

```
package generic.ex2;

import generic.animal.Animal;
import generic.animal.Cat;
import generic.animal.Dog;

public class AnimalMain2 {

    public static void main(String[] args) {
        Animal animal = new Animal("동물", 0);
        Dog dog = new Dog("멍멍이", 100);
        Cat cat = new Cat("냐옹이", 50);

        Box<Animal> animalBox = new Box<>();
        animalBox.set(animal);
        animalBox.set(dog); // Animal = Dog
        animalBox.set(cat); // Animal = Cat
        Animal findAnimal = animalBox.get();
        System.out.println("findAnimal = " + findAnimal);
    }
}
```

실행 결과

```
findAnimal = Animal{name='냐옹이', size=50}
```

문제와 풀이1

문제1 - 제네릭 기본1

문제 설명

- 다음 코드와 실행 결과를 참고해서 `Container` 클래스를 만들어라.
- `Container` 클래스는 제네릭을 사용해야 한다.

```
package generic.test.ex1;

public class ContainerTest {
    public static void main(String[] args) {
        Container<String> stringContainer = new Container<>();
        System.out.println("빈값 확인1: " + stringContainer.isEmpty());

        stringContainer.setItem("data1");
        System.out.println("저장 데이터: " + stringContainer.getItem());
        System.out.println("빈값 확인2: " + stringContainer.isEmpty());

        Container<Integer> integerContainer = new Container<>();
        integerContainer.setItem(10);
        System.out.println("저장 데이터: " + integerContainer.getItem());
    }
}
```

실행 결과

```
빈값 확인1: true
저장 데이터: data1
```

```
빈값 확인2: false
저장 데이터: 10
```

정답 - Container 클래스

```
package generic.test.ex1;

public class Container<T> {

    private T item;

    public void setItem(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }

    public boolean isEmpty() {
        return item == null;
    }
}
```

문제2 - 제네릭 기본2

문제 설명

- 다음 코드와 실행 결과를 참고해서 `Pair` 클래스를 만들어라.
- `Pair` 클래스는 제네릭을 사용해야 한다.

```
package generic.test.ex2;

public class PairTest {
    public static void main(String[] args) {
        Pair<Integer, String> pair1 = new Pair<>();
        pair1.setFirst(1);
        pair1.setSecond("data");
```

```
        System.out.println(pair1.getFirst());
        System.out.println(pair1.getSecond());
        System.out.println("pair1 = " + pair1);

        Pair<String, String> pair2 = new Pair<>();
        pair2.setFirst("key");
        pair2.setSecond("value");
        System.out.println(pair2.getFirst());
        System.out.println(pair2.getSecond());
        System.out.println("pair2 = " + pair2);
    }
}
```

실행 결과

```
1
data
pair1 = Pair{first=1, second=data}
key
value
pair2 = Pair{first=key, second=value}
```

정답 - Pair 클래스

```
package generic.test.ex2;

public class Pair<T1, T2> {

    private T1 first;
    private T2 second;

    public void setFirst(T1 first) {
        this.first = first;
    }

    public void setSecond(T2 second) {
        this.second = second;
    }
```

```
    public T1 getFirst() {
        return first;
    }

    public T2 getSecond() {
        return second;
    }

    @Override
    public String toString() {
        return "Pair{" +
                "first=" + first +
                ", second=" + second +
                '}';
    }
}
```