1. Object 클래스

#0.강의/1.자바로드맵/3.자바-중급1편

프로젝트 환경 구성

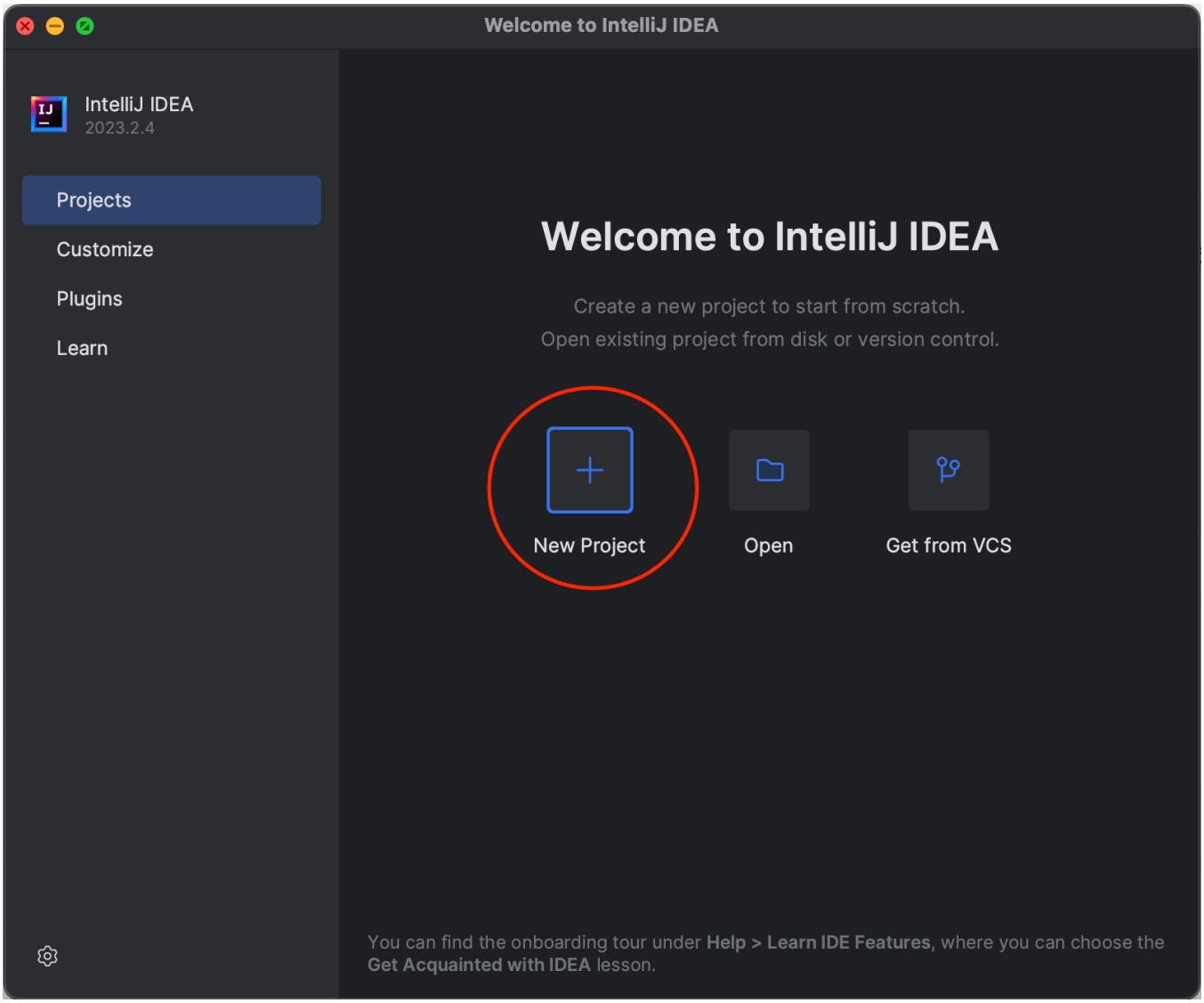
자바 입문편에서 인텔리제이 설치, 선택 이유 설명

프로젝트 환경 구성에 대한 자세한 내용은 자바 입문편 참고

여기서는 입문편을 들었다는 가정하에 설정 진행

인텔리제이 실행하기

New Project



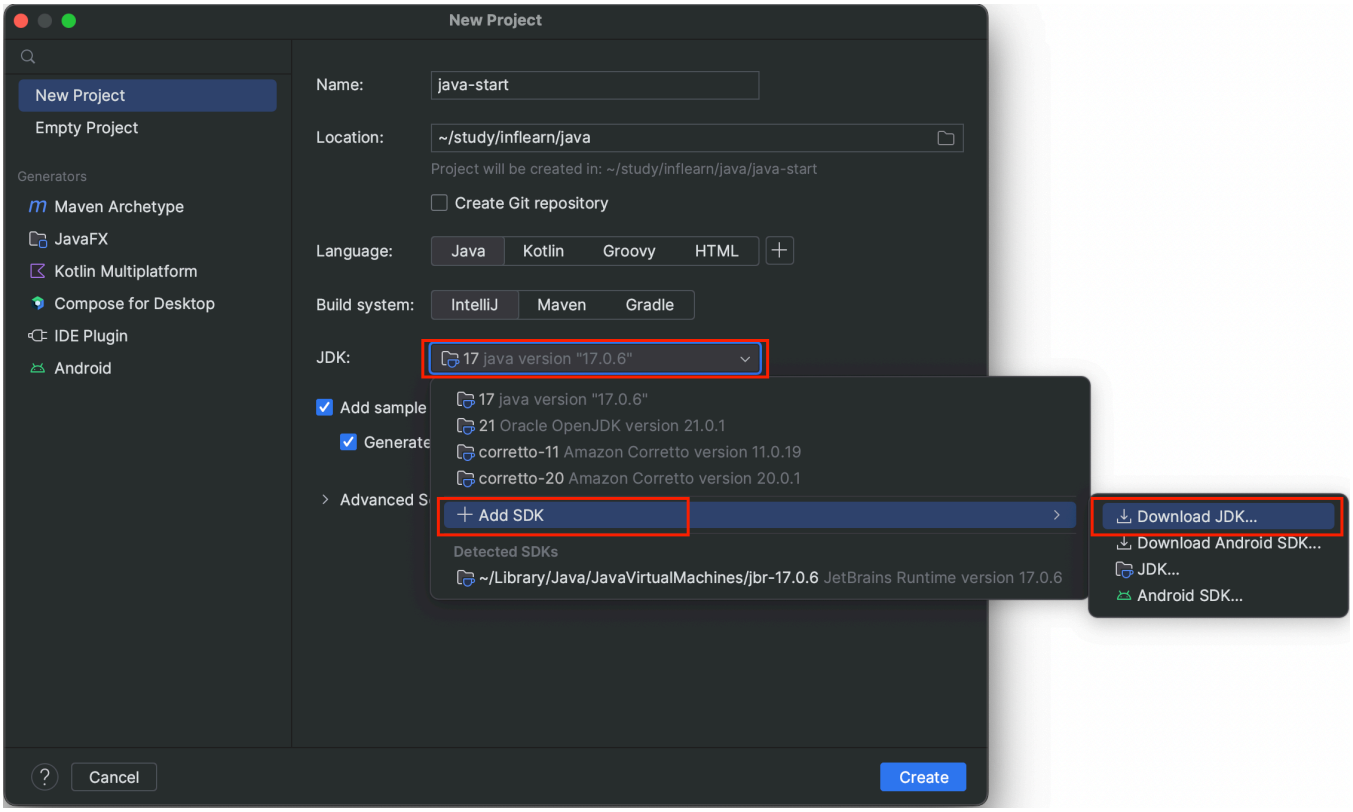
- New Project를 선택해서 새로운 프로젝트를 만들자

New Project 화면

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

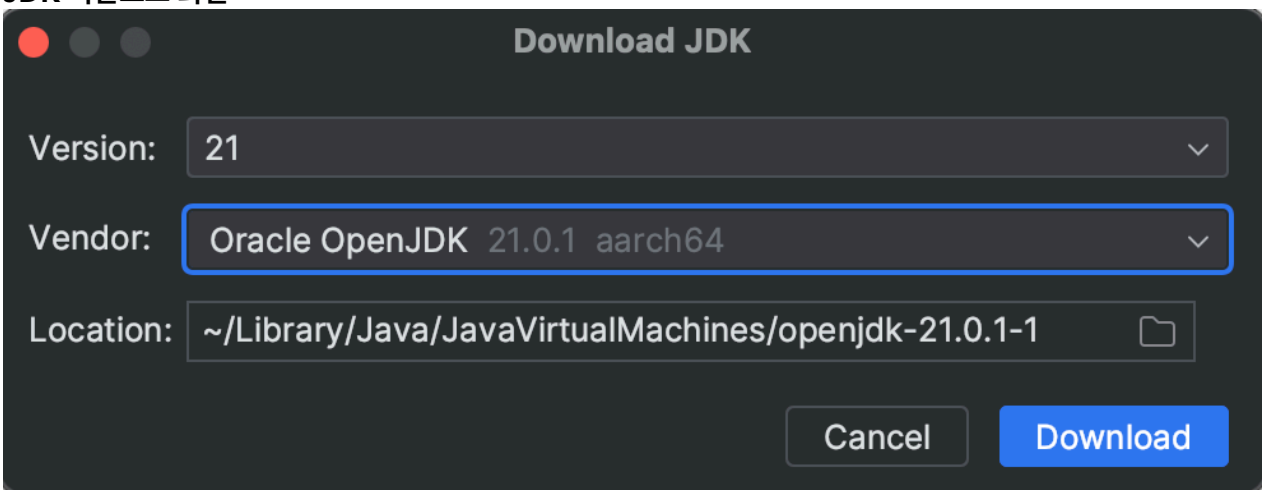
JDK 다운로드 화면 이동 방법

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**

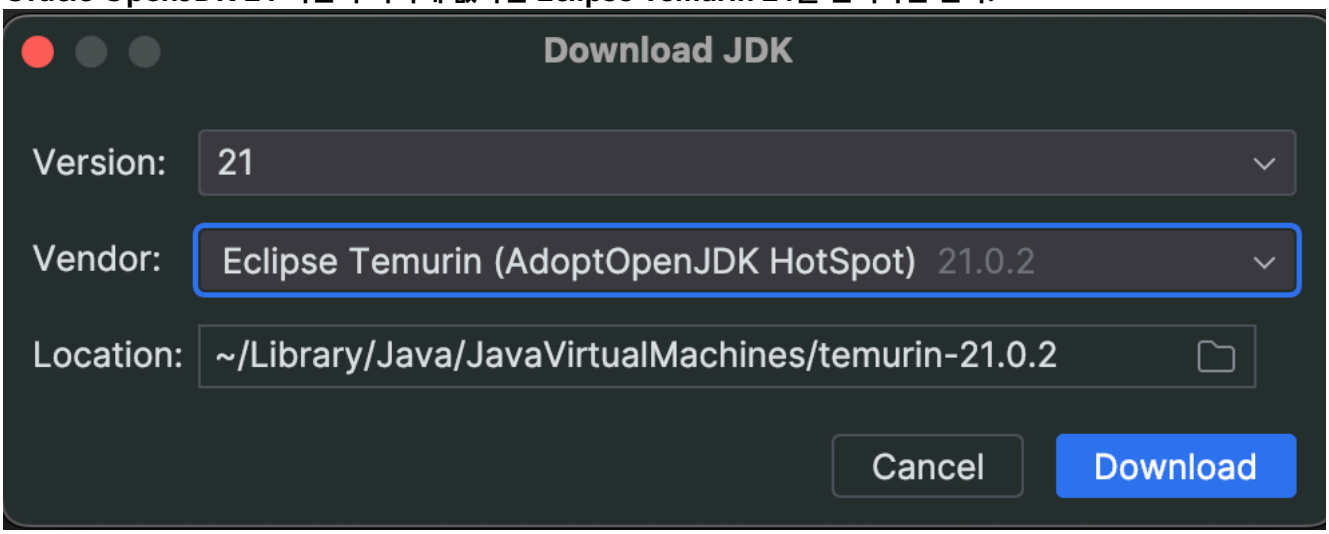
JDK 다운로드 화면



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

주의 - 변경 사항

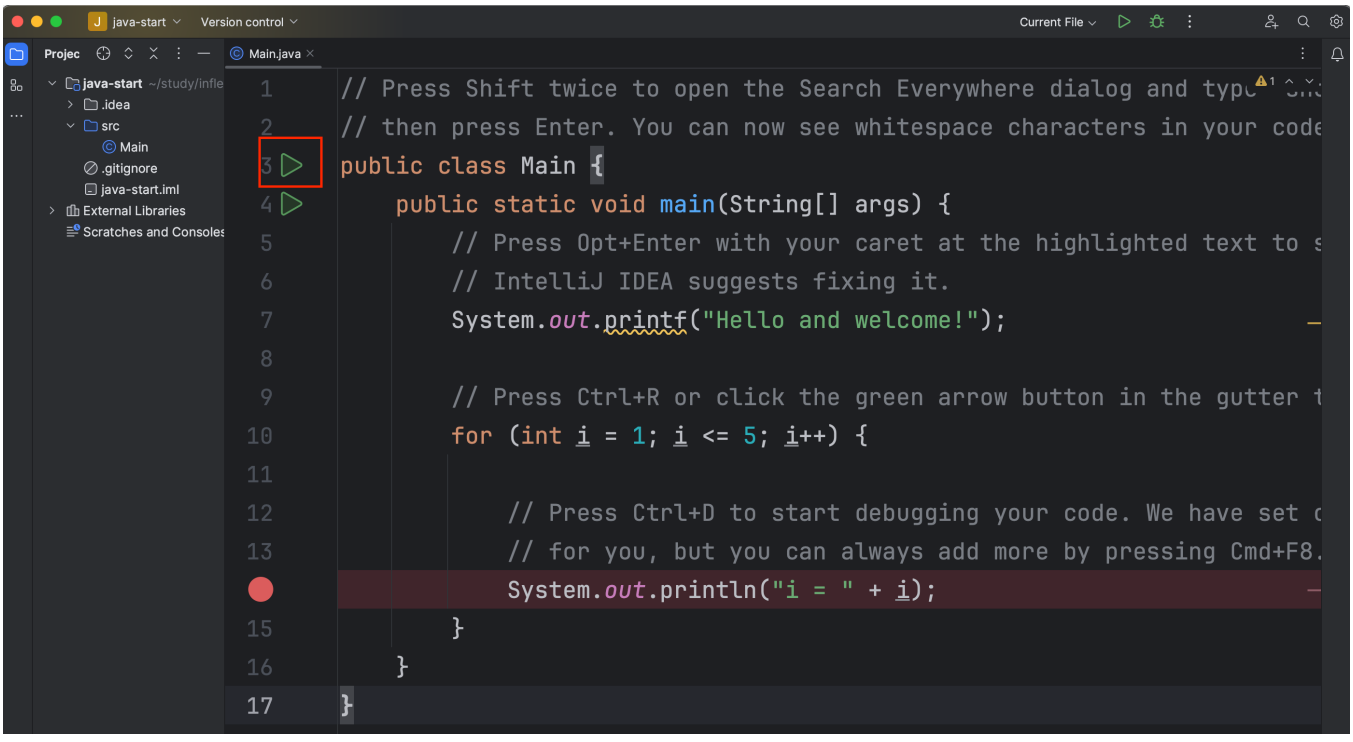
Oracle OpenJDK 21 버전이 목록에 없다면 Eclipse Temurin 21을 선택하면 된다.



`Download` 버튼을 통해서 다운로드 JDK를 다운로드 받는다.

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

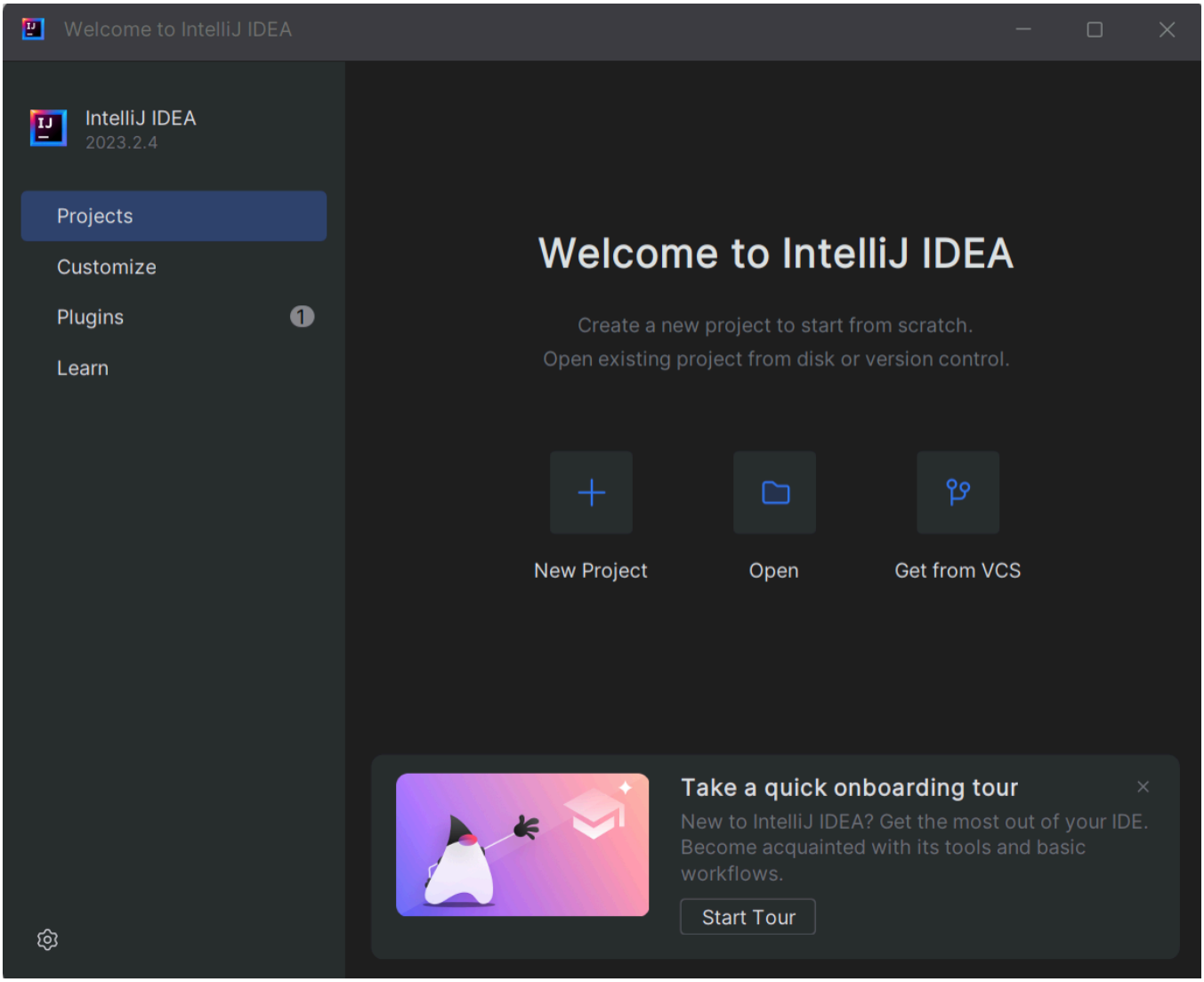
IntelliJ 메인 화면



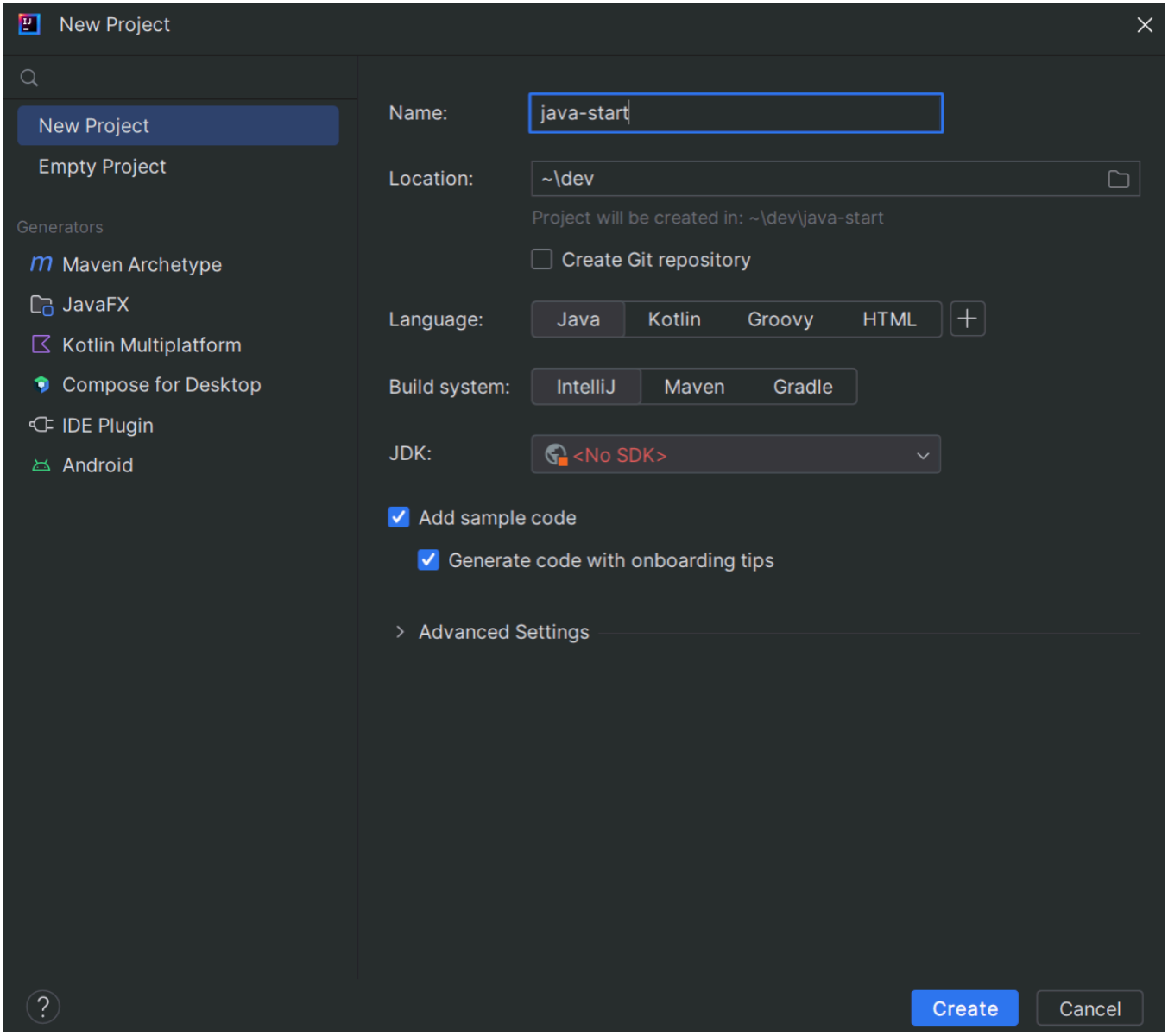
- 앞서 `Add sample code` 선택해서 샘플 코드가 만들어져 있다.
- 위쪽에 빨간색으로 강조한 초록색 화살표 버튼을 선택하고 `Run 'Main.main()` 버튼을 선택하면 프로그램이 실행된다.

윈도우 사용자 추가 설명서

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

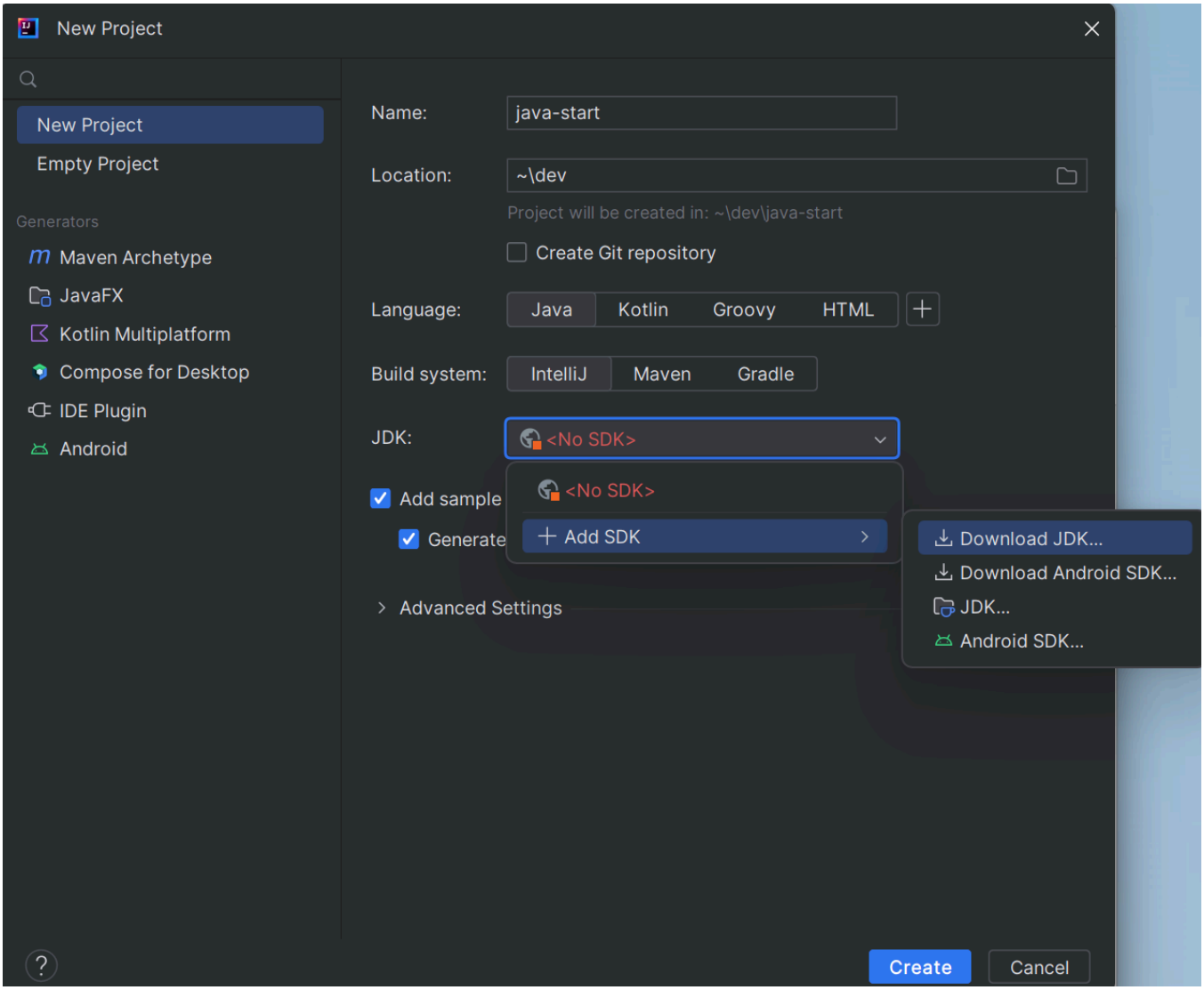


- 프로그램 시작 화면
- New Project 선택

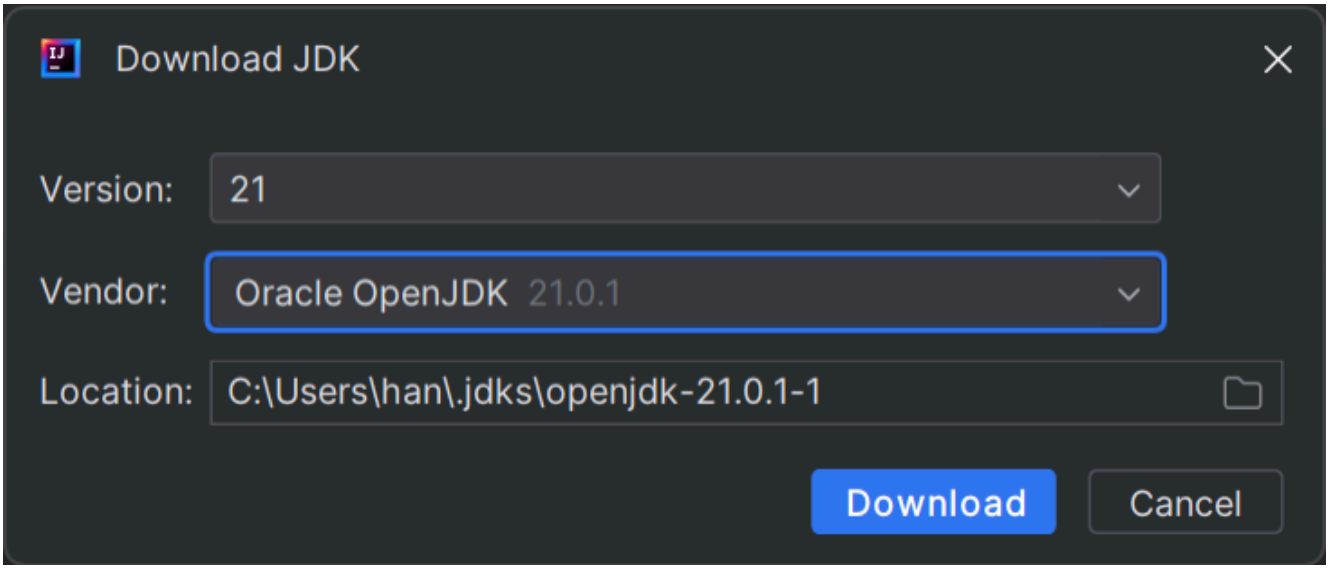


New Project 화면

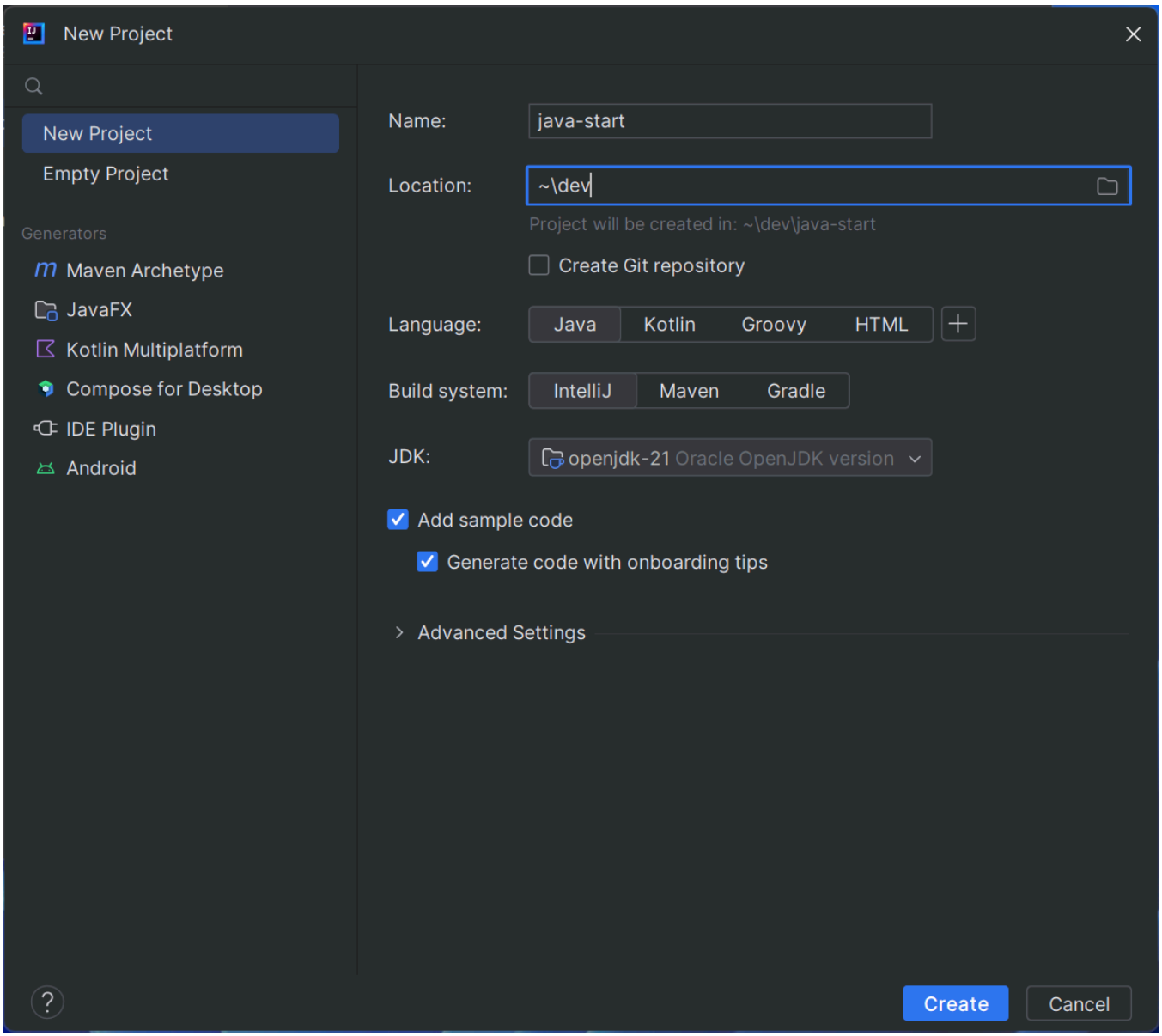
- **Name:**
  - 자바 입문편 강의: java-start
  - 자바 기본편 강의: java-basic
  - 자바 중급1편 강의: **java-mid1**
- Location: 프로젝트 위치, 임의 선택
- Create Git repository 선택하지 않음
- Language: Java
- Build system: IntelliJ
- JDK: 자바 버전 17 이상
- `Add sample code` 선택



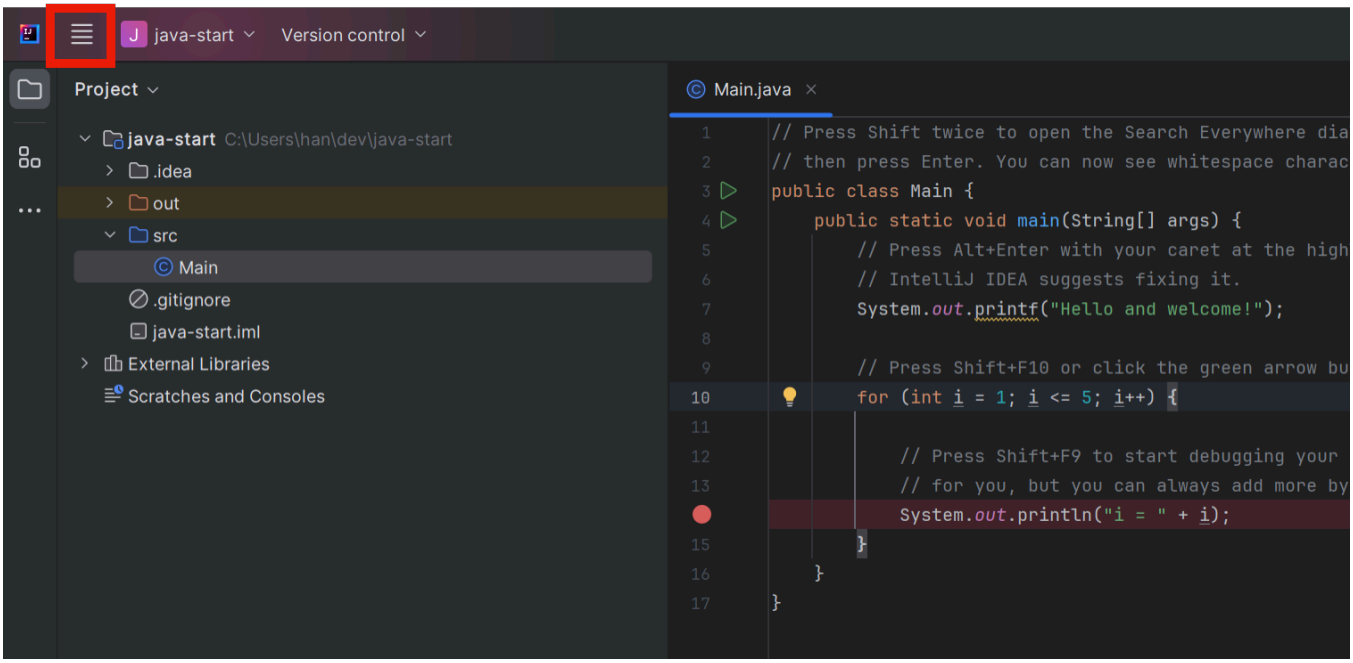
JDK 설치는 Mac과 동일하다.



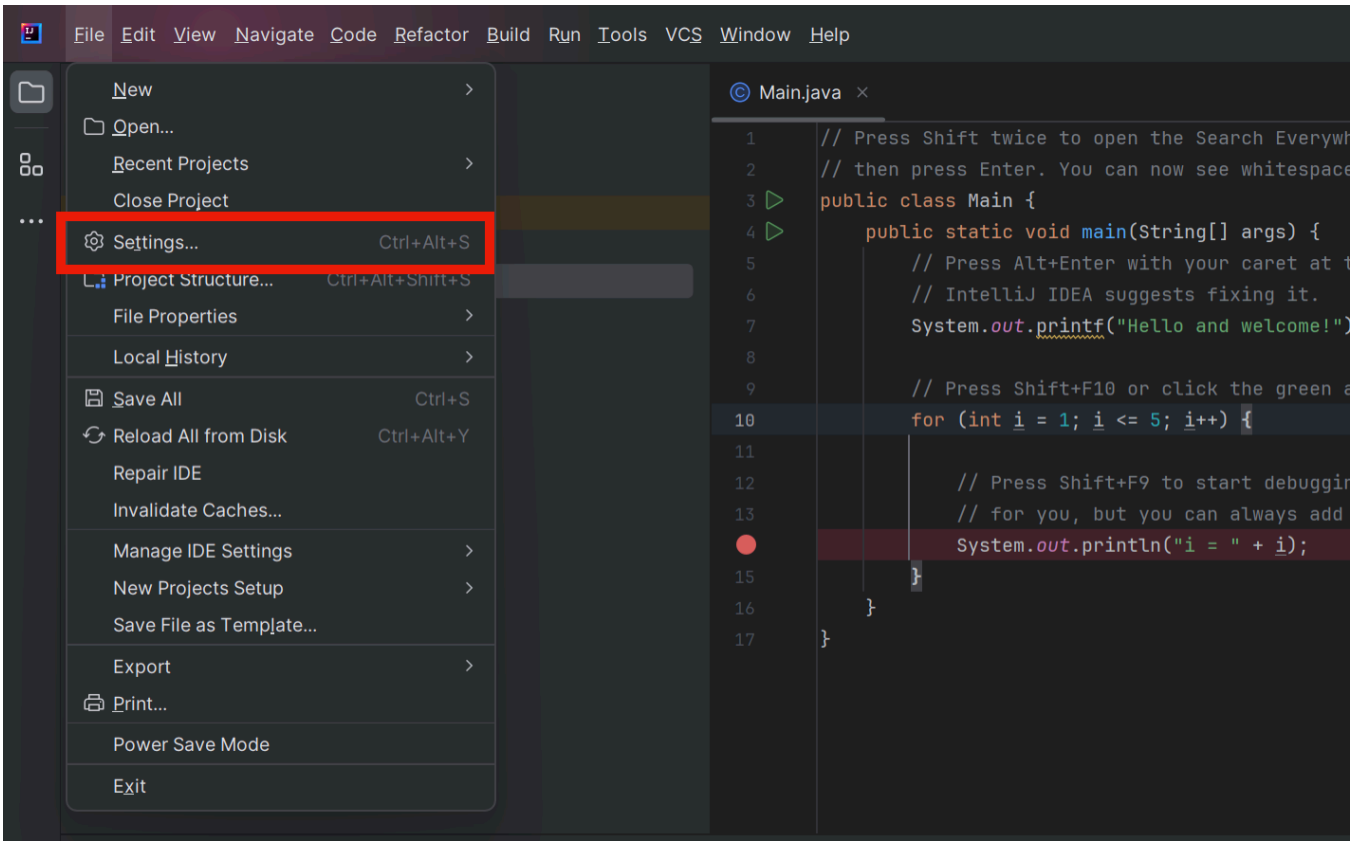
- Version: 21
- Vendor: Oracle OpenJDK
- Location은 가급적 변경하지 말자.



- 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.

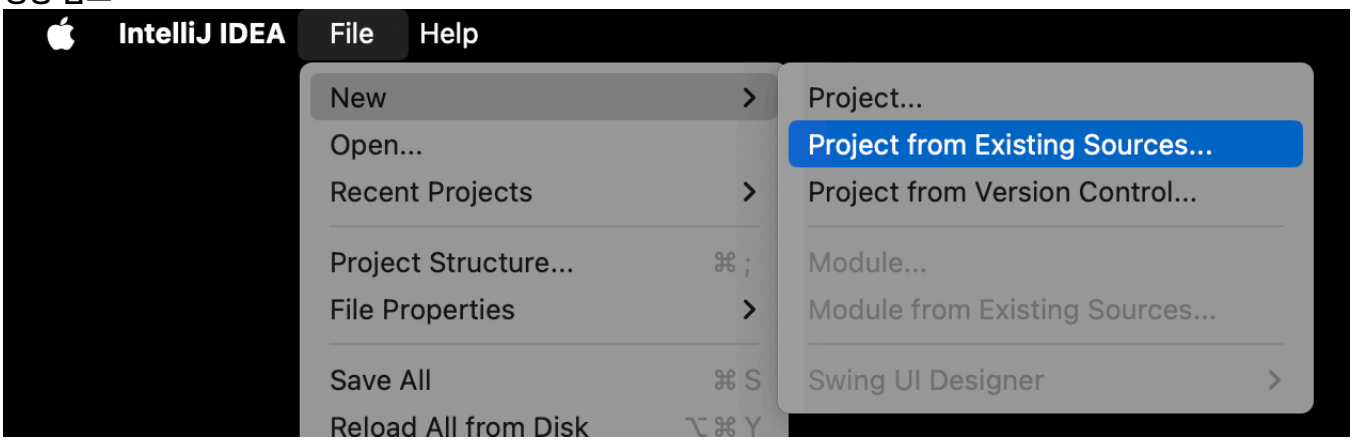
한글 언어팩 → 영어로 변경

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

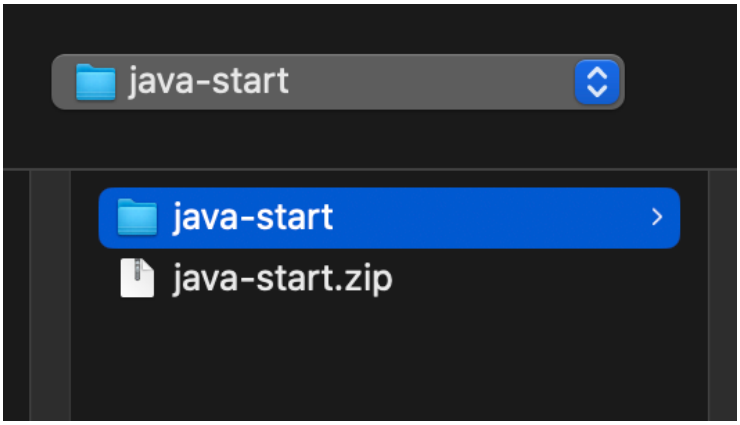
- 만약 한글로 나온다면 다음과 같이 영문으로 변경하자.
- **Mac**: IntelliJ IDEA(메뉴) ➜ Settings... ➜ Plugins ➜ Installed
- **윈도우**: File ➜ Settings... ➜ Plugins ➜ Installed
  - Korean Language Pack 체크 해제
  - OK 선택후 IntelliJ 다시 시작

다운로드 소스 코드 실행 방법

영상 참고

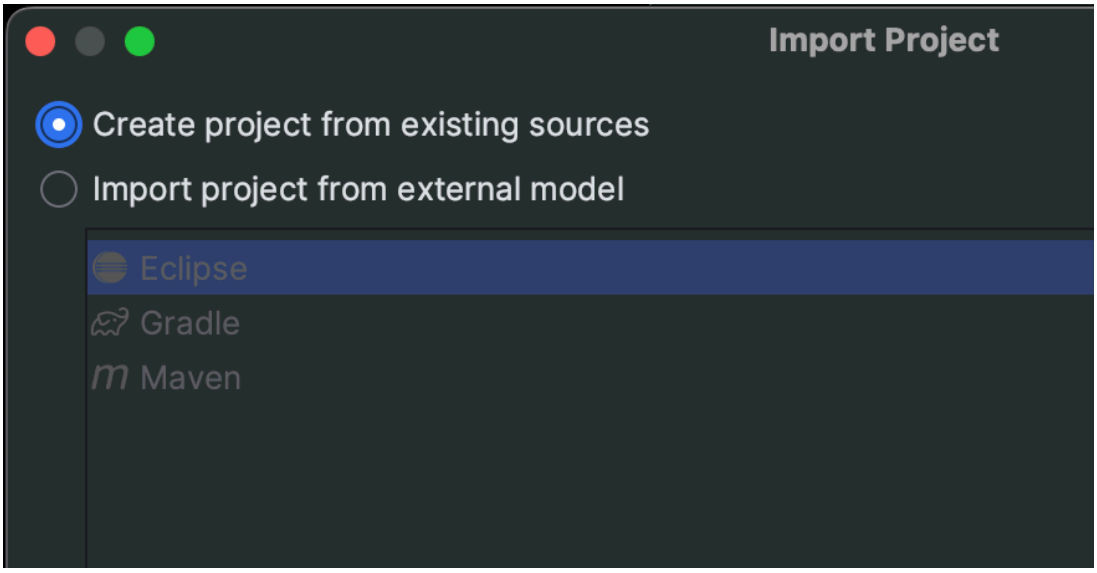


`File -> New -> Project from Existing Sources...` 선택



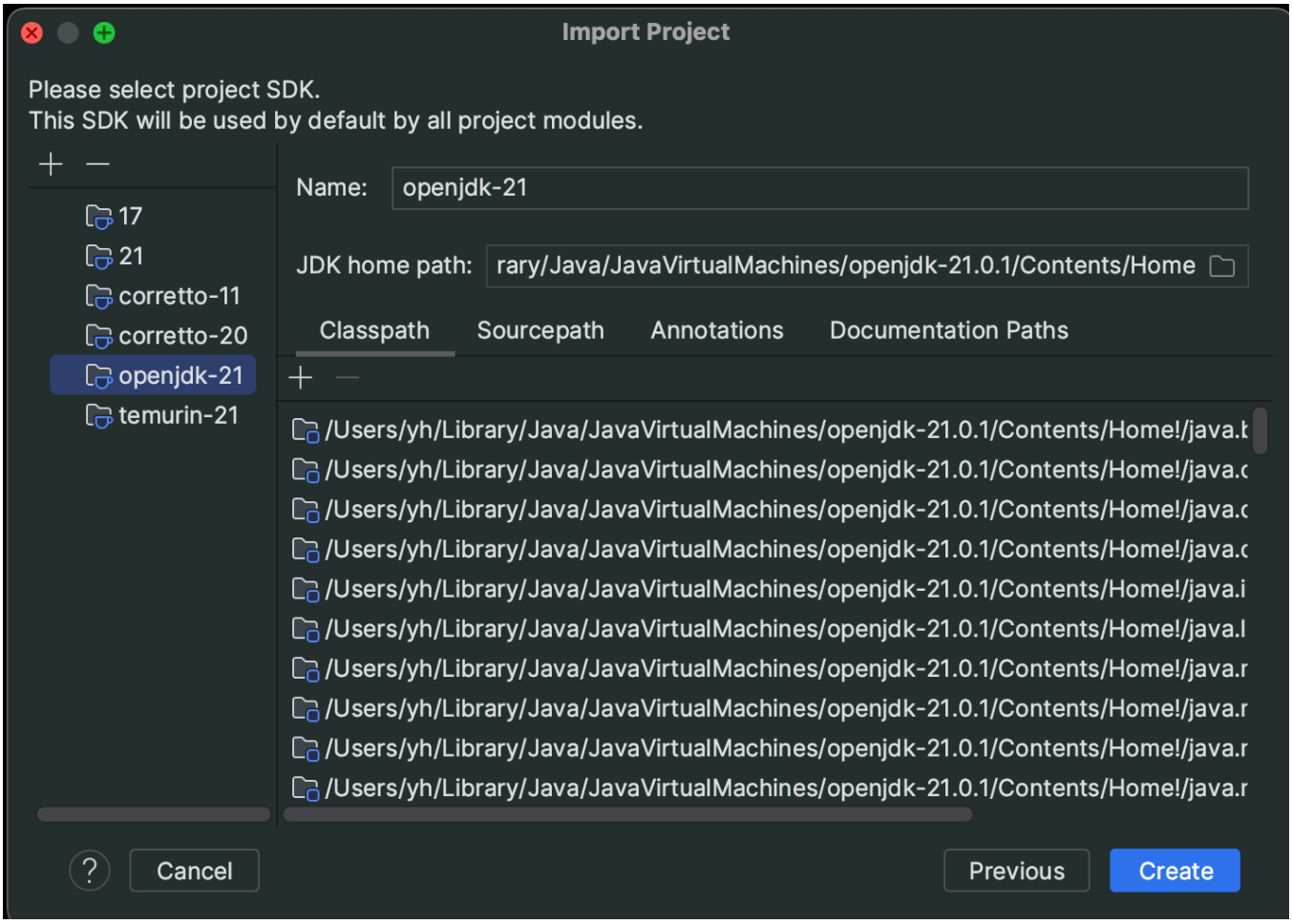
압축을 푼 프로젝트 폴더 선택

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

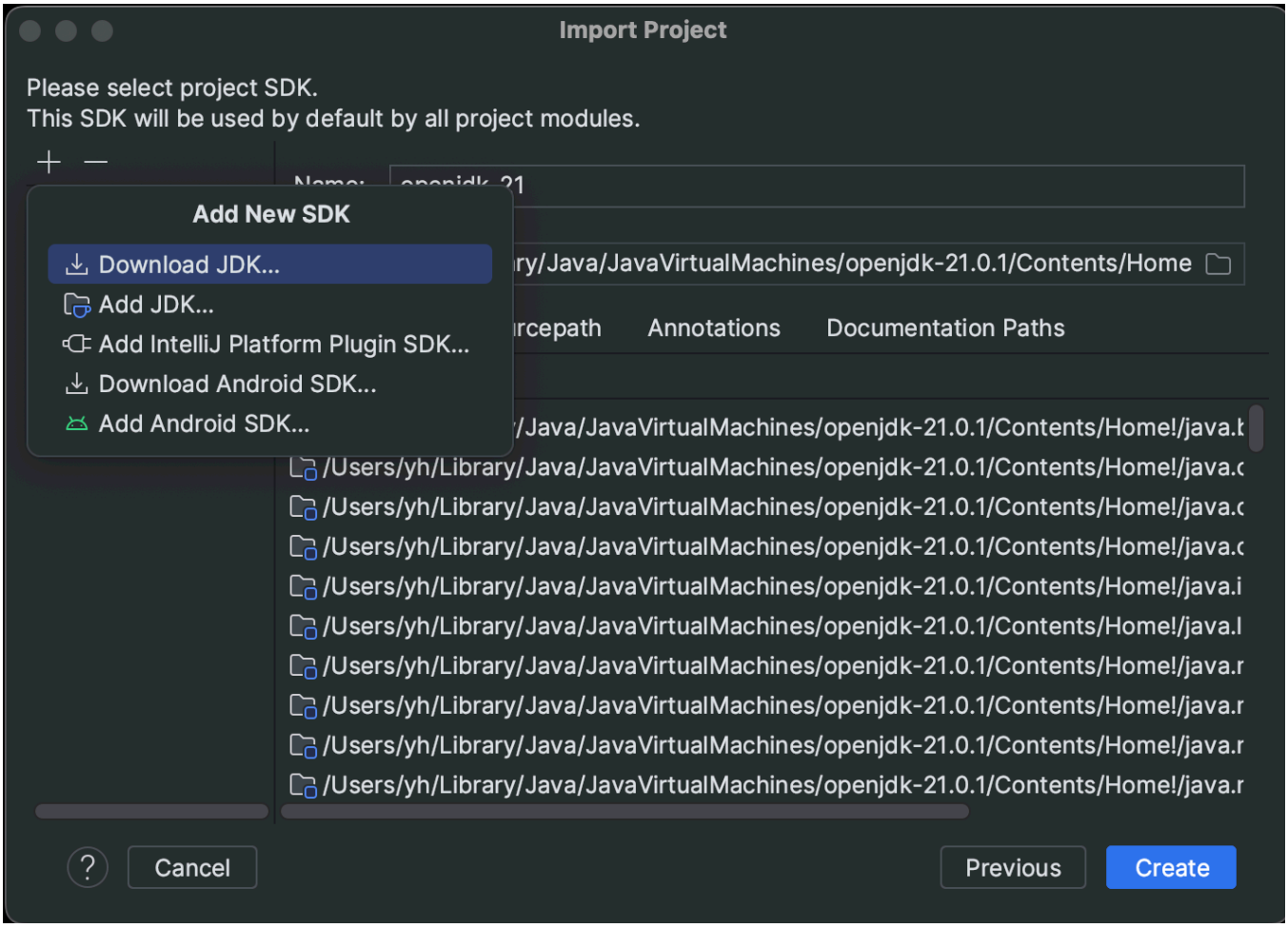


`Create project from existing sources` 선택

이후 계속 `Next` 선택



openjdk-21 선택



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

이후 `Create` 버튼 선택

java.lang 패키지 소개

자바가 기본으로 제공하는 라이브러리(클래스 모음) 중에 가장 기본이 되는 것이 바로 `java.lang` 패키지이다.
여기서 `lang`은 `Language`(언어)의 줄임말이다. 쉽게 이야기해서 자바 언어를 이루는 가장 기본이 되는 클래스들을 보관하는 패키지를 뜻한다.

java.lang 패키지의 대표적인 클래스들

- `Object`: 모든 자바 객체의 부모 클래스
- `String`: 문자열
- `Integer`, `Long`, `Double`: 래퍼 타입, 기본형 데이터 타입을 객체로 만든 것
- `Class`: 클래스 메타 정보
- `System`: 시스템과 관련된 기본 기능들을 제공

여기 나열한 클래스들은 자바 언어의 기본을 이루기 때문에 반드시 잘 알아두어야 한다.

import 생략 가능

`java.lang` 패키지는 모든 자바 애플리케이션에 자동으로 임포트(`import`)된다. 따라서 임포트 구문을 사용하지 않아도 된다.

다른 패키지에 있는 클래스를 사용하려면 다음과 같이 임포트를 사용해야 한다.

```
package lang;

import java.lang.System;

public class LangMain {

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

`System` 클래스는 `java.lang` 패키지 소속이다. 따라서 다음과 같이 임포트를 생략할 수 있다.

```
package lang;

public class LangMain {

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

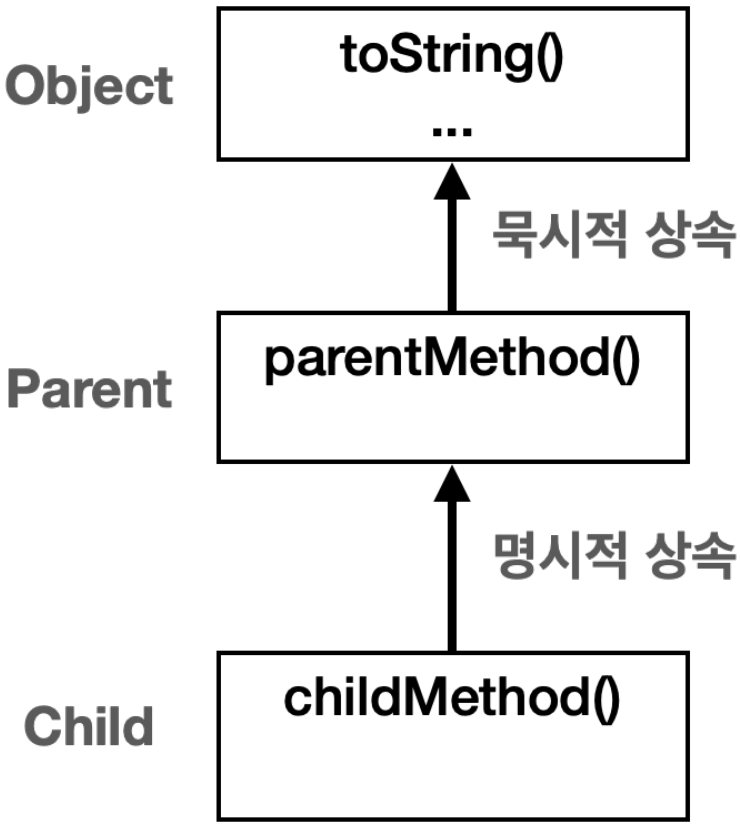
- `import java.lang.System;` 코드를 삭제해도 정상 동작한다.

이제 본격적으로 `java.lang` 패키지가 제공하는 기능들을 하나씩 알아보자.

Object 클래스

자바에서 모든 클래스의 최상위 부모 클래스는 항상 `Object` 클래스이다.

다음 그림과 예제 코드를 보자.




```
package lang.object;

//부모가 없으면 묵시적으로 Object 클래스를 상속받는다.
public class Parent {

    public void parentMethod() {
        System.out.println("Parent.parentMethod");
    }

}
```

앞의 코드는 다음 코드와 같다.

```
```

```java
package lang.object;

//extends Object 추가
public class Parent extends Object {

    public void parentMethod() {
        System.out.println("Parent.parentMethod");
    }

}
```

- 클래스에 상속 받을 부모 클래스가 없으면 묵시적으로 `Object` 클래스를 상속 받는다.
  - 쉽게 이야기해서 자바가 `extends Object` 코드를 넣어준다.
  - 따라서 `extends Object`는 생략하는 것을 권장한다.

```java
package lang.object;

public class Child extends Parent {

   public void childMethod() {
       System.out.println("Child.childMethod");
   }

}
```

- 클래스에 상속 받을 부모 클래스를 명시적으로 지정하면 `Object`를 상속 받지 않는다.
  - 쉽게 이야기해서 이미 명시적으로 상속했기 때문에 자바가 `extends Object` 코드를 넣지 않는다.

> **묵시적(Implicit) vs 명시적(Explicit)**
> 묵시적: 개발자가 코드에 직접 기술하지 않아도 시스템 또는 컴파일러에 의해 자동으로 수행되는 것을 의미
> 명시적: 개발자가 코드에 직접 기술해서 작동하는 것을 의미

```java
package lang.object;

public class ObjectMain {
    public static void main(String[] args) {
        Child child = new Child();
        child.childMethod();
```

```
        child.parentMethod();

        // toString()은 Object 클래스의 메서드
        String string = child.toString();
        System.out.println(string);
    }
}
```

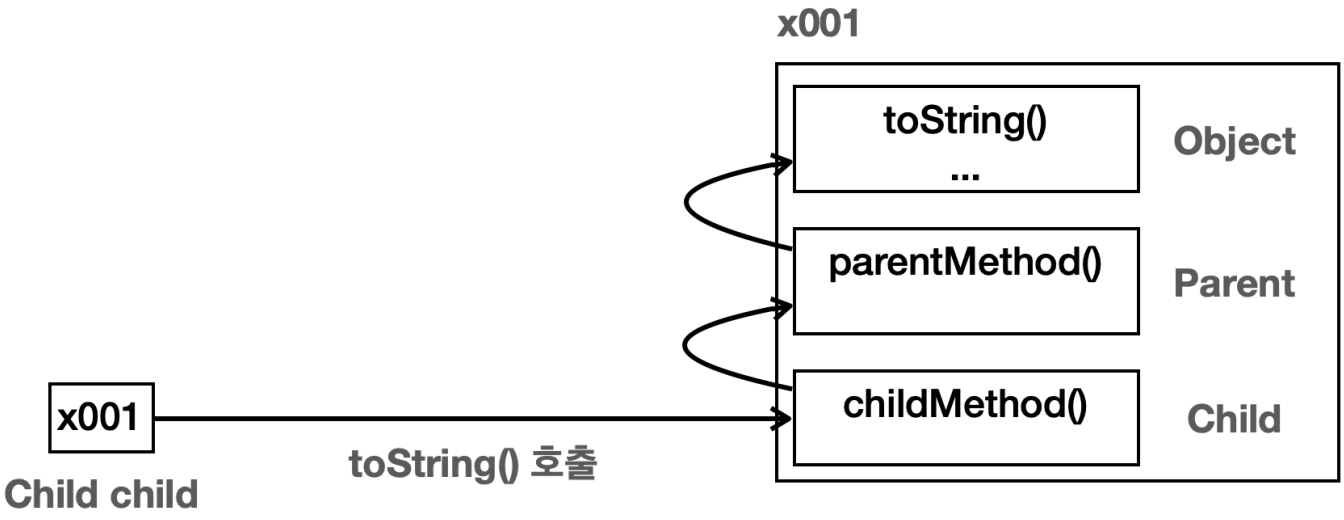
- 예제에서 `toString()`은 `Object` 클래스의 메서드이다. 이 메서드는 객체의 정보를 제공한다.

실행 결과

```
Child.childMethod
Parent.parentMethod
lang.object.Child@X001
```

실행 결과 그림

`Parent`는 `Object`를 묵시적으로 상속 받았기 때문에 메모리에도 함께 생성된다.



1. `child.toString()`을 호출한다.
2. 먼저 본인의 타입인 `Child`에서 `toString()`을 찾는다. 없으므로 부모 타입으로 올라가서 찾는다.
3. 부모 타입인 `Parent`에서 찾는다. 없으므로 부모 타입으로 올라가서 찾는다.
4. 부모 타입인 `Object`에서 찾는다. `Object`에 `toString()`이 있으므로 이 메서드를 호출한다.

정리

자바에서 모든 객체의 최종 부모는 `Object`다.

자바에서 Object 클래스가 최상위 부모 클래스인 이유

모든 클래스가 `Object` 클래스를 상속 받는 이유는 다음과 같다.

- 공통 기능 제공
- 다형성의 기본 구현

공통 기능 제공

객체의 정보를 제공하고, 이 객체가 다른 객체와 같은지 비교하고, 객체가 어떤 클래스로 만들어졌는지 확인하는 기능은 모든 객체에게 필요한 기본 기능이다. 이런 기능을 객체를 만들 때 마다 항상 새로운 메서드를 정의해서 만들어야 한다면 상당히 번거로울 것이다.
그리고 막상 만든다고 해도 개발자마다 서로 다른 이름의 메서드를 만들어서 일관성이 없을 것이다. 예를 들어서 객체의 정보를 제공하는 기능을 만든다고 하면 어떤 개발자는 `toString()`으로 또 어떤 개발자는 `objectInfo()`와 같이 서로 다른 이름으로 만들 수 있다. 객체를 비교하는 기능을 만들 때도 어떤 개발자는 `equals()`로 어떤 개발자는 `same()`으로 만들 수 있다.

`Object`는 모든 객체에 필요한 공통 기능을 제공한다. `Object`는 최상위 부모 클래스이기 때문에 모든 객체는 공통 기능을 편리하게 제공(상속) 받을 수 있다.

`Object`가 제공하는 기능은 다음과 같다.

- 객체의 정보를 제공하는 `toString()`
- 객체의 같음을 비교하는 `equals()`
- 객체의 클래스 정보를 제공하는 `getClass()`
- 기타 여러가지 기능

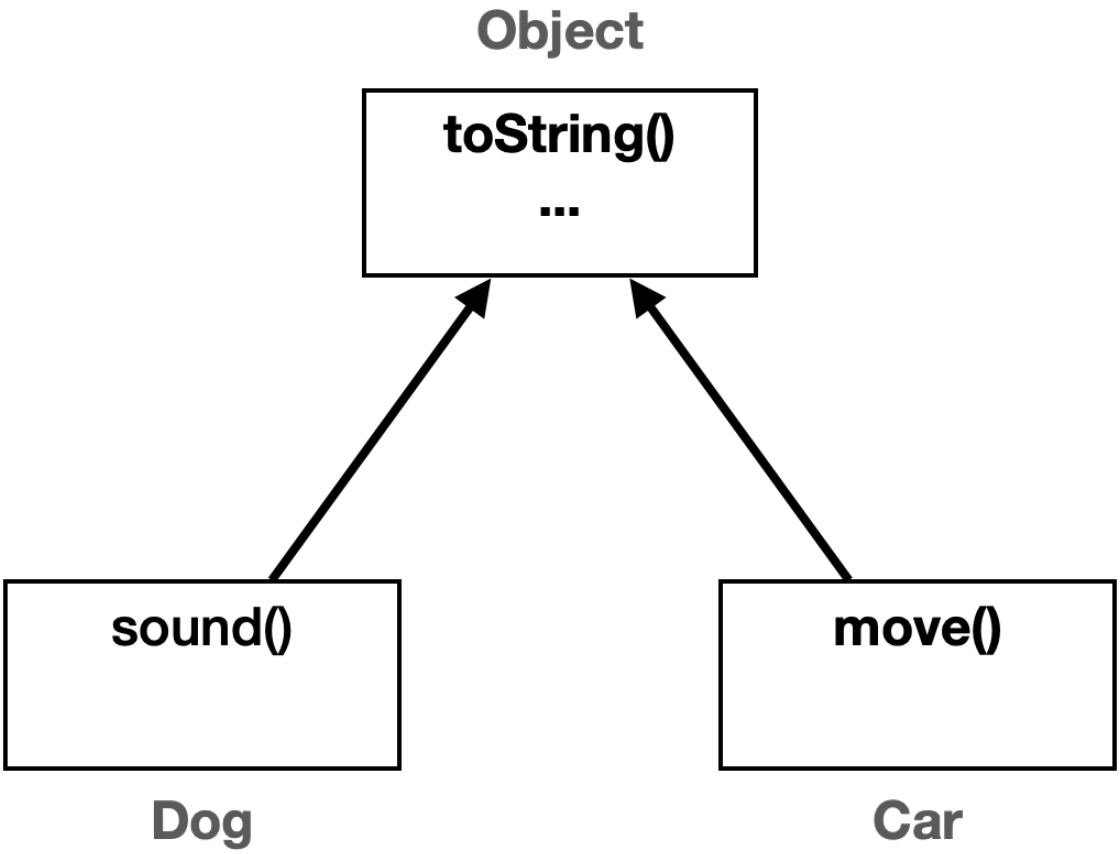
개발자는 모든 객체가 앞서 설명한 메서드를 지원한다는 것을 알고 있다. 따라서 프로그래밍이 단순화되고, 일관성을 가진다.
각각의 기능에 대한 자세한 내용은 이후에 하나씩 알아보자.

다형성의 기본 구현

부모는 자식을 담을 수 있다. `Object`는 모든 클래스의 부모 클래스이다. 따라서 모든 객체를 참조할 수 있다.
`Object` 클래스는 다형성을 지원하는 기본적인 메커니즘을 제공한다. 모든 자바 객체는 `Object` 타입으로 처리될 수 있으며, 이는 다양한 타입의 객체를 통합적으로 처리할 수 있게 해준다.
쉽게 이야기해서 `Object`는 모든 객체를 다 담을 수 있다. 타입이 다른 객체들을 어딘가에 보관해야 한다면 바로 `Object`에 보관하면 된다.

Object 다형성

`Object`는 모든 클래스의 부모 클래스이다. 따라서 `Object`는 모든 객체를 참조할 수 있다.
예제를 통해서 `Object`의 다형성에 대해 알아보자.



`Dog`와 `Car`은 서로 아무런 관련이 없는 클래스이다. 둘다 부모가 없으므로 `Object`를 자동으로 상속 받는다.

```
package lang.object.poly;

class Car {
    public void move() {
        System.out.println("자동차 이동");
    }
}
```

```
package lang.object.poly;

class Dog {
    public void sound() {
        System.out.println("멍멍");
```

```
    }
}
```

```
package lang.object.poly;

public class ObjectPolyExample1 {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Car car = new Car();

        action(dog);
        action(car);
    }

    private static void action(Object obj) {
        //obj.sound(); //컴파일 오류, Object는 sound()가 없다.
        //obj.move(); //컴파일 오류, Object는 move()가 없다.

        //객체에 맞는 다운캐스팅 필요
        if (obj instanceof Dog dog) {
            dog.sound();
        } else if (obj instanceof Car car) {
            car.move();
        }
    }
}
```

실행 결과

```
멍멍
자동차 이동
```

`Object`는 모든 타입의 부모다. 부모는 자식을 담을 수 있으므로 앞의 코드를 다음과 같이 변경해도 된다.

```
Object dog = new Dog(); //Dog -> Object
Object car = new Car(); //Car -> Object
```

Object 다형성의 장점

`action(Object obj)` 메서드를 분석해보자.

이 메서드는 `Object` 타입의 매개변수를 사용한다. 그런데 `Object`는 모든 객체의 부모다. 따라서 어떤 객체든지 인자로 전달할 수 있다.

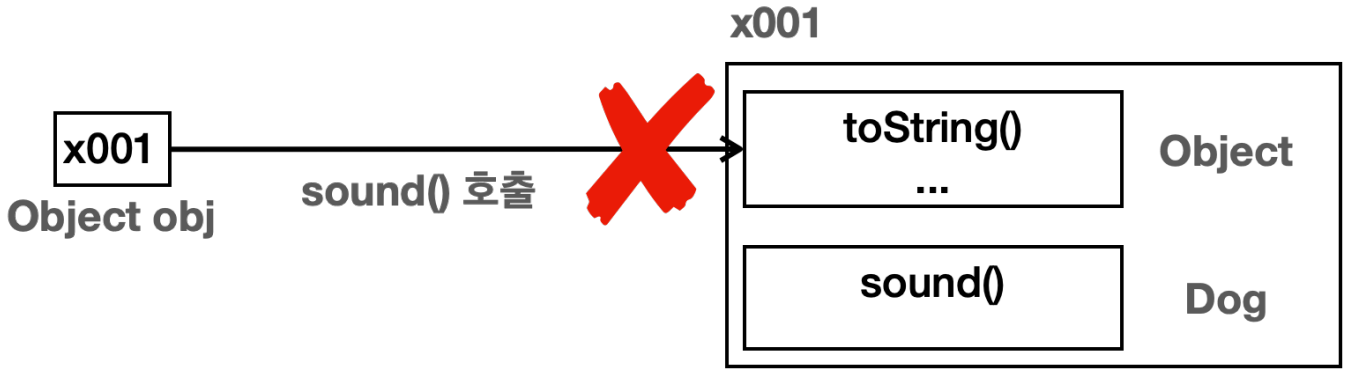
```
action(dog) //main에서 dog 전달
void action(Object obj = dog(Dog)) //Object는 자식인 Dog 타입을 참조할 수 있다.
```

```
action(car) //main에서 car 전달
void action(Object obj = car(Car)) //Object는 자식인 Car 타입을 참조할 수 있다.
```

Object 다형성의 한계

```
action(dog) //main에서 dog 전달
private static void action(Object obj) {
    obj.sound(); //컴파일 오류, Object는 sound()가 없다.
}
```

`action()` 메서드안에서 `obj.sound()`를 호출하면 오류가 발생한다. 왜냐하면 매개변수인 `obj`는 `Object` 타입이기 때문이다. `Object`에는 `sound()` 메서드가 없다.



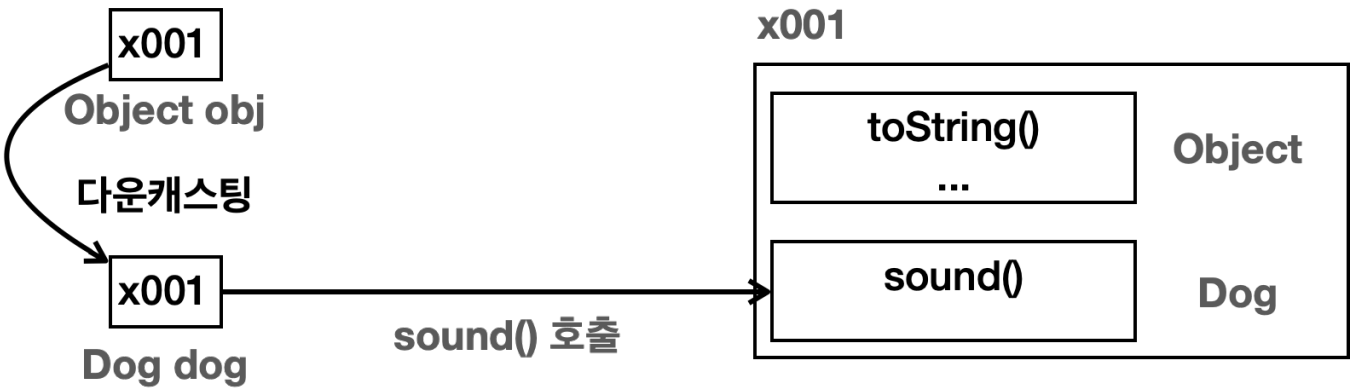
obj.sound() 호출

- `obj.sound()`를 호출한다.
- `obj`는 `Object` 타입이므로 `Object` 타입에서 `sound()`를 찾는다.

- `Object`에서 `sound()`를 찾을 수 없다. `Object`는 최종 부모이므로 더는 올라가서 찾을 수 없다. 따라서 오류가 발생한다.

`Dog` 인스턴스의 `sound()`를 호출하려면 다음과 같이 다운캐스팅을 해야한다.

```
if (obj instanceof Dog dog) {
    dog.sound();
}
```



- `Object obj`의 참조값을 `Dog dog`로 다운캐스팅 하면서 전달한다.
- `dog.sound()`를 호출하면 `Dog` 타입에서 `sound()`를 찾아서 호출한다.

Object를 활용한 다형성의 한계

- `Object`는 모든 객체를 대상으로 다형적 참조를 할 수 있다.
  - 쉽게 이야기해서 `Object`는 모든 객체의 부모이므로 모든 객체를 담을 수 있다.
- `Object`를 통해 전달 받은 객체를 호출하려면 각 객체에 맞는 다운캐스팅 과정이 필요하다.
  - `Object`가 세상의 모든 메서드를 알고 있는 것이 아니다.

다형성을 제대로 활용하려면 자바 기본편에서 배운 것 처럼 다형적 참조 + 메서드 오버라이딩을 함께 사용해야 한다. 그런면에서 `Object`를 사용한 다형성에는 한계가 있다.
`Object`는 모든 객체의 부모이므로 모든 객체를 대상으로 다형적 참조를 할 수 있다. 하지만 `Object`에는 `Dog.sound()`, `Car.move()`와 같은 다른 객체의 메서드가 정의되어 있지 않다. 따라서 메서드 오버라이딩을 활용할 수 없다. 결국 각 객체의 기능을 호출하려면 다운캐스팅을 해야 한다.
참고로 `Object` 본인이 보유한 `toString()` 같은 메서드는 당연히 자식 클래스에서 오버라이딩 할 수 있다. 여기서 이야기하는 것은 앞서 설명한 `Dog.sound()`, `Car.move()` 같은 `Object`에 속하지 않은 메서드를 말한다.

결과적으로 다형적 참조는 가능하지만, 메서드 오버라이딩이 안되기 때문에 다형성을 활용하기에는 한계가 있다.
그렇다면 `Object`를 언제 활용하면 좋을까? 지금부터 하나씩 알아보자.

Object 배열

이번에는 `Object` 배열을 알아보자.

`Object`는 모든 타입의 객체를 담을 수 있다. 따라서 `Object[]`을 만들면 세상의 모든 객체를 담을 수 있는 배열을 만들 수 있다.

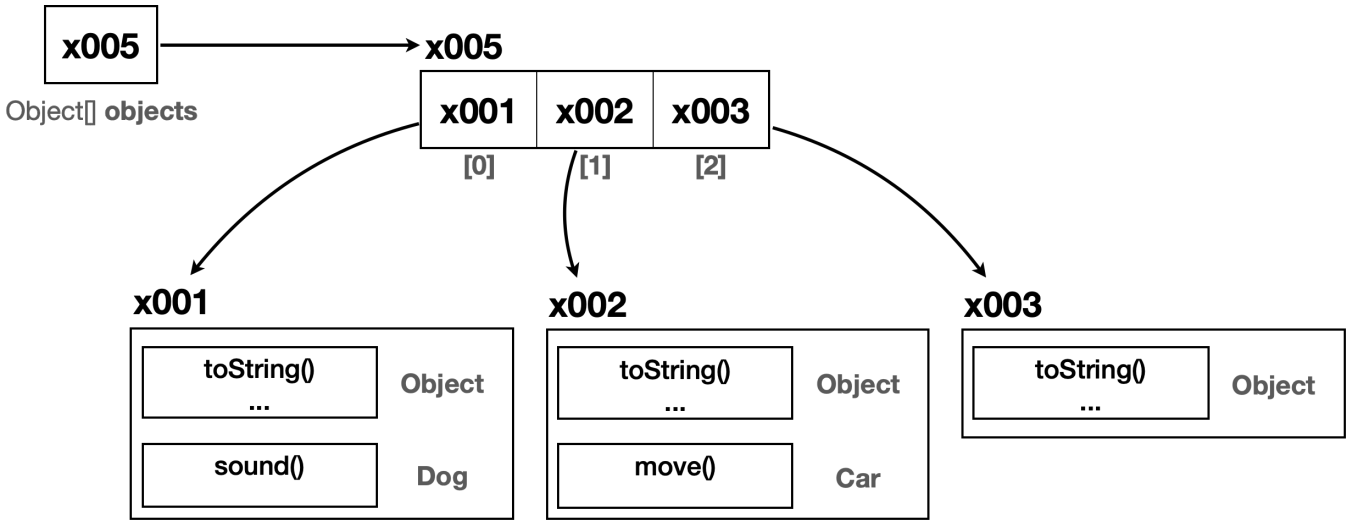
```
package lang.object.poly;

public class ObjectPolyExample2 {

    public static void main(String[] args) {
        Dog dog = new Dog();
        Car car = new Car();
        Object object = new Object(); //Object 인스턴스도 만들 수 있다.

        Object[] objects = {dog, car, object};

        size(objects);
    }

    private static void size(Object[] objects) {
        System.out.println("전달된 객체의 수는: " + objects.length);
    }
}
```

실행 결과

```
전달된 객체의 수는: 3
```

```
Object[] objects = {dog, car, object};
//쉽게 풀어서 설명하면 다음과 같다.
Object objects[0] = new Dog();
Object objects[1] = new Car();
Object objects[2] = new Object();
```

`Object` 타입을 사용한 덕분에 세상의 모든 객체를 담을 수 있는 배열을 만들 수 있었다.



size() 메서드

`size(Object[] objects)` 메서드는 배열에 담긴 객체의 수를 세는 역할을 담당한다.

이 메서드는 `Object` 타입만 사용한다. `Object` 타입의 배열은 세상의 모든 객체를 담을 수 있기 때문에, 새로운 클래스가 추가되거나 변경되어도 이 메서드를 수정하지 않아도 된다. 지금 만든 `size()` 메서드는 자바를 사용하는 곳이라면 어디든지 사용될 수 있다.

Object가 없다면?

만약 `Object`와 같은 개념이 없다면 어떻게 될까?

- `void action(Object obj)`과 같이 모든 객체를 받을 수 있는 메서드를 만들 수 없다.
- `Object[] objects`처럼 모든 객체를 저장할 수 있는 배열을 만들 수 없다.

물론 `Object`가 없어도 직접 `MyObject`와 같은 클래스를 만들고 모든 클래스에서 직접 정의한 `MyObject`를 상속받으면 된다. 하지만 하나의 프로젝트를 넘어서 전세계 모든 개발자가 비슷한 클래스를 만들 것이고, 서로 호환되지 않는 수 많은 `XxxObject`들이 넘쳐날 것이다.

toString()

`Object.toString()` 메서드는 객체의 정보를 문자열 형태로 제공한다. 그래서 디버깅과 로깅에 유용하게 사용된다.

이 메서드는 `Object` 클래스에 정의되므로 모든 클래스에서 상속받아 사용할 수 있다.

코드로 확인해보자.

```
package lang.object.tostring;

public class ToStringMain1 {

    public static void main(String[] args) {
        Object object = new Object();
        String string = object.toString();

        //toString() 반환값 출력
        System.out.println(string);

        //object 직접 출력
        System.out.println(object);
    }
}
```

실행 결과

```
java.lang.Object@a09ee92
java.lang.Object@a09ee92
```

Object.toString()

```
public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

- `Object`가 제공하는 `toString()` 메서드는 기본적으로 패키지를 포함한 객체의 이름과 객체의 참조값(해시코드)를 16진수로 제공한다.

> **참고**: 해시코드(`hashCode()`)에 대한 정확한 내용은 이후에 별도로 다룬다. 지금은 객체의 참조값 정도로 생각하면 된다.

println()과 toString()

그런데 `toString()`의 결과를 출력한 코드와 `object`를 `println()`에 직접 출력한 코드의 결과가 완전히 같다.

```
//toString() 반환값 출력
String string = object.toString();
System.out.println(string);

//object 직접 출력
System.out.println(object);
```

`System.out.println()` 메서드는 사실 내부에서 `toString()`을 호출한다.
`Object` 타입(자식 포함)이 `println()`에 인수로 전달되면 내부에서 `obj.toString()` 메서드를 호출해서 결과를 출력한다.

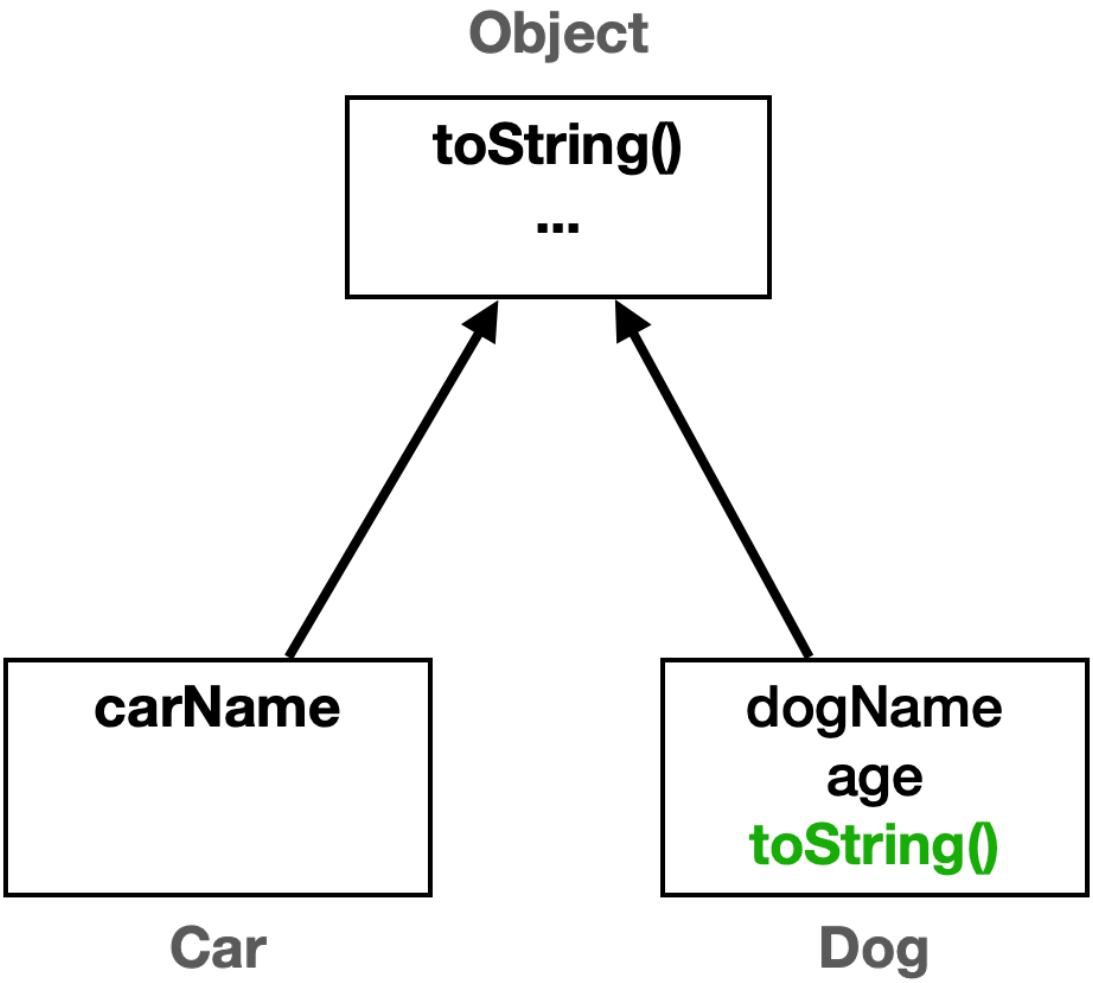
```
public void println(Object x) {
    String s = String.valueOf(x);
    //...
}
```

```
public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}
```

따라서 `println()`을 사용할 때, `toString()`을 직접 호출할 필요 없이 객체를 바로 전달하면 객체의 정보를 출력할 수 있다.

toString() 오버라이딩

`Object.toString()` 메서드가 클래스 정보와 참조값을 제공하지만 이 정보만으로는 객체의 상태를 적절히 나타내지 못한다. 그래서 보통 `toString()`을 재정의(오버라이딩)해서 보다 유용한 정보를 제공하는 것이 일반적이다.




```
package lang.object.tostring;

public class Car {

    private String carName;

    public Car(String carName) {
        this.carName = carName;
    }

}
```

- `Car`는 `toString()`을 재정의하지 않는다.

```
package lang.object.tostring;

public class Dog {
```

```
    private String dogName;
    private int age;

    public Dog(String dogName, int age) {
        this.dogName = dogName;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Dog{" +
                "dogName='" + dogName + '\'' +
                ", age=" + age +
                '}';
    }
}
```

- `Dog`는 `toString()`을 재정의했다.
- `toString()` 메서드는 IDE의 도움을 받아서 작성하는 것이 매우 편리하다.
  - generator 단축키: `⌘N` (macOS) / `Alt+Insert` (Windows/Linux)

```
package lang.object.tostring;

public class ObjectPrinter {
    public static void print(Object obj) {
        String string = "객체 정보 출력: " + obj.toString();
        System.out.println(string);
    }
}
```

- "객체 정보 출력:" 이라는 문자와 객체의 `toString()` 결과를 합해서 출력하는 단순한 기능을 제공한다.

```
package lang.object.tostring;

public class ToStringMain2 {

    public static void main(String[] args) {
        Car car = new Car("ModelY");
        Dog dog1 = new Dog("멍멍이1", 2);
```

```
        Dog dog2 = new Dog("멍멍이2", 5);

        System.out.println("1. 단순 toString 호출");
        System.out.println(car.toString());
        System.out.println(dog1.toString());
        System.out.println(dog2.toString());

        System.out.println("2. println 내부에서 toString 호출");
        //println 내부에서 toString 호출
        System.out.println(car);
        System.out.println(dog1);
        System.out.println(dog2);

        System.out.println("3. Object 다형성 활용");
        ObjectPrinter.print(car);
        ObjectPrinter.print(dog1);
        ObjectPrinter.print(dog2);
    }
}
```

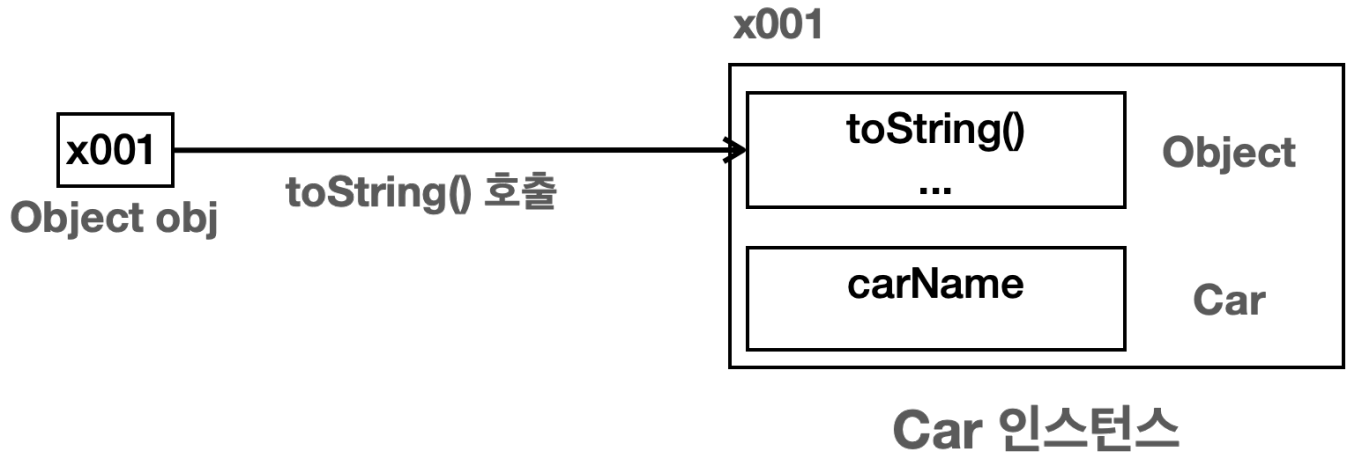
실행 결과

```
1. 단순 toString 호출
lang.object.tostring.Car@452b3a41
Dog{dogName='멍멍이1', age=2}
Dog{dogName='멍멍이2', age=5}

2. println 내부에서 toString 호출
lang.object.tostring.Car@452b3a41
Dog{dogName='멍멍이1', age=2}
Dog{dogName='멍멍이2', age=5}

3. Object 다형성 활용
객체 정보 출력: lang.object.tostring.Car@452b3a41
객체 정보 출력: Dog{dogName='멍멍이1', age=2}
객체 정보 출력: Dog{dogName='멍멍이2', age=5}
```

`Car` 인스턴스는 `toString()`을 재정의 하지 않았다. 따라서 `Object`가 제공하는 기본 `toString()` 메서드를 사용한다.

`Dog` 인스턴스는 `toString()`을 재정의 한 덕분에 객체의 상태를 명확하게 확인할 수 있다.

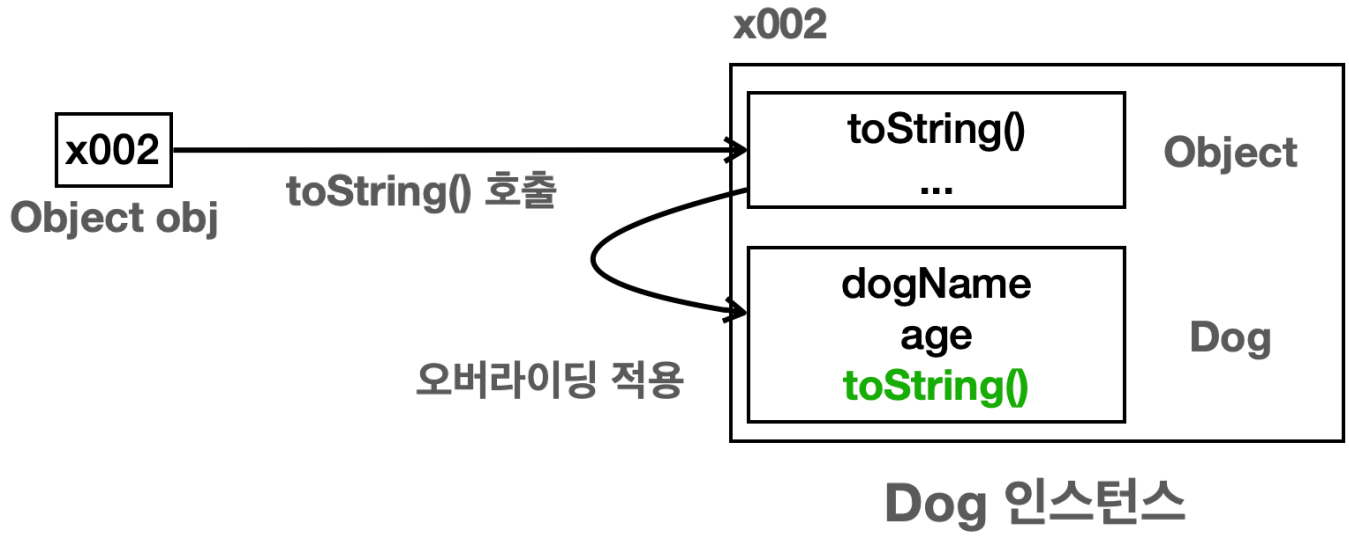
ObjectPrinter.print(Object obj) 분석 - Car 인스턴스




```
ObjectPrinter.print(car) //main에서 호출

void print(Object obj = car(Car)) { //인수 전달
    String string = "객체 정보 출력: " + obj.toString();
}
```

- `Object obj`의 인수로 `car(Car)`가 전달 된다.
- 메서드 내부에서 `obj.toString()`을 호출한다.
- `obj`는 `Object` 타입이다. 따라서 `Object`에 있는 `toString()`을 찾는다.
- 이때 자식에 재정의(오버라이딩)된 메서드가 있는지 찾아본다. 재정의된 메서드가 없다.
- `Object.toString()`을 실행한다.

ObjectPrinter.print(Object obj) 분석 - Dog 인스턴스



```
ObjectPrinter.print(dog) //main에서 호출
void print(Object obj = dog(Dog)) { //인수 전달
    String string = "객체 정보 출력: " + obj.toString();
}
```

- `Object obj`의 인수로 `dog(Dog)`가 전달 된다.
- 메서드 내부에서 `obj.toString()`을 호출한다.
- `obj`는 `Object` 타입이다. 따라서 `Object`에 있는 `toString()`을 찾는다.
- 이때 자식에 재정의(오버라이딩)된 메서드가 있는지 찾아본다. `Dog`에 재정의된 메서드가 있다.
- `Dog.toString()`을 실행한다.

참고 - 객체의 참조값 직접 출력

`toString()`은 기본으로 객체의 참조값을 출력한다. 그런데 `toString()`이나 `hashCode()`를 재정의하면 객체의 참조값을 출력할 수 없다. 이때는 다음 코드를 사용하면 객체의 참조값을 출력할 수 있다.

```
String refValue = Integer.toHexString(System.identityHashCode(dog1));
System.out.println("refValue = " + refValue);
```

실행 결과

```
refValue = 72ea2f77
```

Object와 OCP

만약 `Object`가 없고, 또 `Object`가 제공하는 `toString()`이 없다면 서로 아무 관계가 없는 객체의 정보를 출력하기 어려울 것이다. 여기서 아무 관계가 없다는 것은 공통의 부모가 없다는 뜻이다. 아마도 다음의 `BadObjectPrinter` 클래스와 같이 각각의 클래스마다 별도의 메서드를 작성해야 할 것이다.

BadObjectPrinter

```
public class BadObjectPrinter {
    public static void print(Car car) { //Car 전용 메서드
        String string = "객체 정보 출력: " + car.carInfo(); //carInfo() 메서드 만듬
        System.out.println(string);
```

```
    }

    public static void print(Dog dog) { //Dog 전용 메서드
        String string = "객체 정보 출력: " + dog.dogInfo(); //dogInfo() 메서드 만듬
        System.out.println(string);
    }
}
```

구체적인 것에 의존

`BadObjectPrinter`는 구체적인 타입인 `Car`, `Dog`를 사용한다. 따라서 이후에 출력해야 할 구체적인 클래스가 10개로 늘어나면 구체적인 클래스에 맞추어 메서드도 10개로 계속 늘어나게 된다. 이렇게 `BadObjectPrinter` 클래스가 구체적인 특정 클래스인 `Car`, `Dog`를 사용하는 것을 `BadObjectPrinter`는 `Car`, `Dog`에 의존한다고 표현한다.

다행히도 자바에는 **객체의 정보를 사용할 때**, 다형적 참조 문제를 해결해줄 `Object` 클래스와 메서드 오버라이딩 문제를 해결해줄 `Object.toString()` 메서드가 있다. (물론 직접 `Object`와 비슷한 공통의 부모 클래스를 만들어서 해결할 수도 있다.)

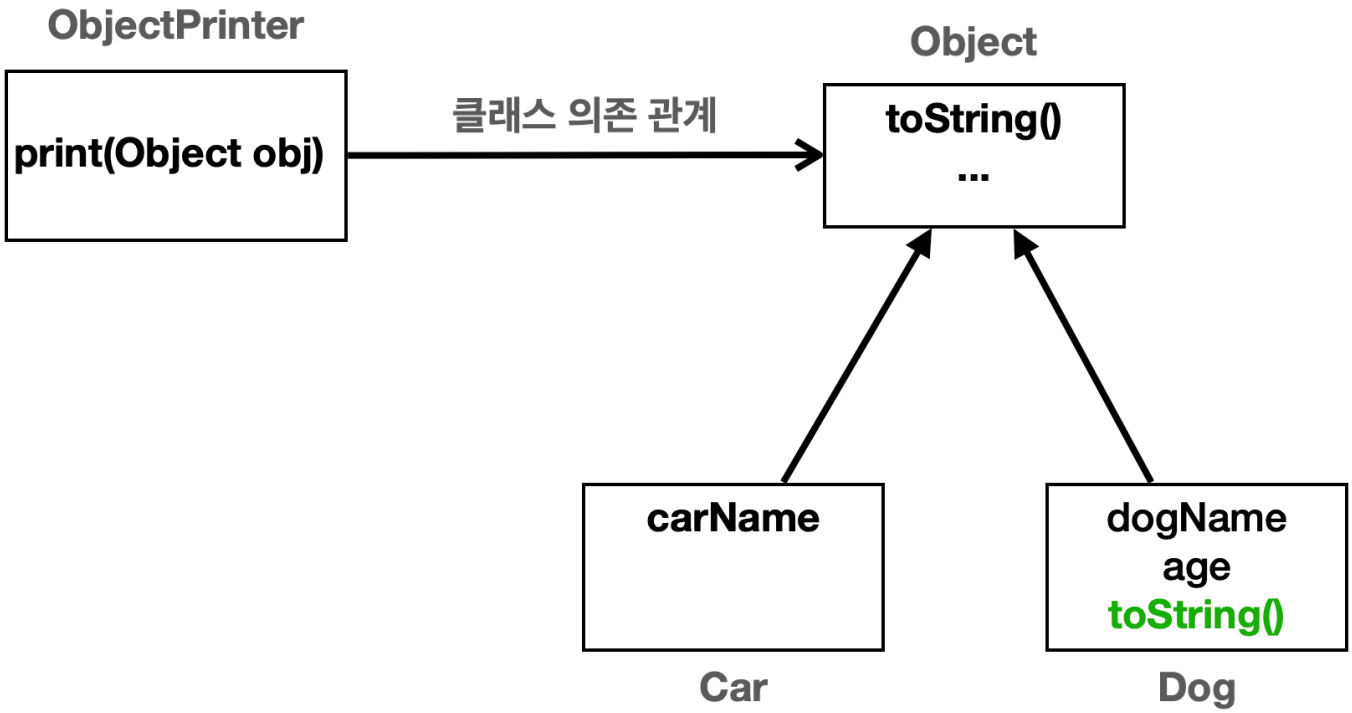
추상적인 것에 의존

우리가 앞서 만든 `ObjectPrinter` 클래스는 `Car`, `Dog` 같은 구체적인 클래스를 사용하는 것이 아니라, 추상적인 `Object` 클래스를 사용한다. 이렇게 `ObjectPrinter` 클래스가 `Object` 클래스를 사용하는 것을 `ObjectPrinter` 클래스가 `Object`에 클래스에 의존한다고 표현한다.

```
public class ObjectPrinter {
    public static void print(Object obj) {
        String string = "객체 정보 출력: " + obj.toString();
        System.out.println(string);
    }
}
```

`ObjectPrinter`는 구체적인 것에 의존하는 것이 아니라 추상적인 것에 의존한다.

> **추상적**: 여기서 말하는 추상적이라는 뜻은 단순히 추상 클래스나 인터페이스만 뜻하는 것은 아니다. `Animal`과 `Dog`, `Cat`의 관계를 떠올려보자. `Animal` 같은 부모 타입으로 올라갈수록 개념은 더 추상적이게 되고, `Dog`, `Cat`과 같이 하위 타입으로 내려갈 수록 개념은 더 구체적이게 된다.



`ObjectPrinter`와 `Object`를 사용하는 구조는 다형성을 매우 잘 활용하고 있다. 다형성을 잘 활용한다는 것은 다형적 참조와 메서드 오버라이딩을 적절하게 사용한다는 뜻이다.

`ObjectPrinter`의 `print()` 메서드와 전체 구조를 분석해보자.

- **다형적 참조**: `print(Object obj)`, `Object` 타입을 매개변수로 사용해서 다형적 참조를 사용한다. `Car`, `Dog` 인스턴스를 포함한 세상의 모든 객체 인스턴스를 인수로 받을 수 있다.
- **메서드 오버라이딩**: `Object`는 모든 클래스의 부모이다. 따라서 `Dog`, `Car`와 같은 구체적인 클래스는 `Object`가 가지고 있는 `toString()` 메서드를 오버라이딩 할 수 있다. 따라서 `print(Object obj)` 메서드는 `Dog`, `Car`와 같은 구체적인 타입에 의존(사용)하지 않고, 추상적인 `Object` 타입에 의존하면서 런타임에 각 인스턴스의 `toString()`을 호출할 수 있다.

OCP 원칙

기본편에서 학습한 OCP 원칙을 떠올려보자.

- **Open**: 새로운 클래스를 추가하고, `toString()`을 오버라이딩해서 기능을 확장할 수 있다.
- **Closed**: 새로운 클래스를 추가해도 `Object`와 `toString()`을 사용하는 클라이언트 코드인 `ObjectPrinter`는 변경하지 않아도 된다.

다형적 참조, 메서드 오버라이딩, 그리고 클라이언트 코드가 구체적인 `Car`, `Dog`에 의존하는 것이 아니라 추상적인 `Object`에 의존하면서 OCP 원칙을 지킬 수 있었다. 덕분에 새로운 클래스를 추가하고 `toString()` 메서드를 새롭게 오버라이딩해서 기능을 확장할 수 있다. 그리고 이러한 변화에도 불구하고 클라이언트 코드인 `ObjectPrinter`는 변경할 필요가 없다.

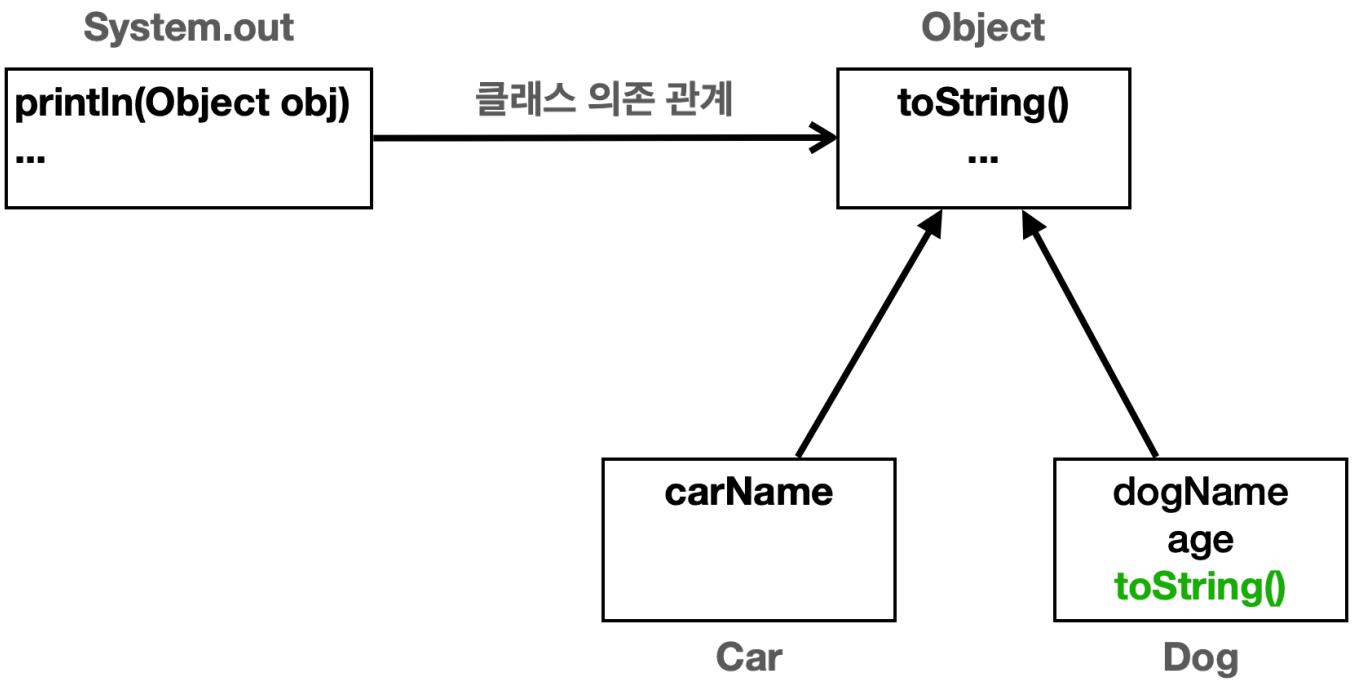
`ObjectPrinter`는 모든 타입의 부모인 `Object`를 사용하고, `Object`가 제공하는 `toString()` 메서드만 사용

한다. 따라서 `ObjectPrinter`를 사용하면 세상의 모든 객체의 정보(`toString()`)를 편리하게 출력할 수 있다.

System.out.println()

지금까지 설명한 `ObjectPrinter.print()`는 사실 `System.out.println()`의 작동 방식을 설명하기 위해 만든 것이다.

`System.out.println()` 메서드도 `Object` 매개변수를 사용하고 내부에서 `toString()`을 호출한다. 따라서 `System.out.println()`를 사용하면 세상의 모든 객체의 정보(`toString()`)를 편리하게 출력할 수 있다.



자바 언어는 객체지향 언어 답게 언어 스스로도 객체지향의 특징을 매우 잘 활용한다.

우리가 지금까지 배운 `toString()` 메서드와 같이, 자바 언어가 기본으로 제공하는 다양한 메서드들은 개발자가 필요에 따라 오버라이딩해서 사용할 수 있도록 설계되어 있다.

참고 - 정적 의존관계 vs 동적 의존관계

- 정적 의존관계는 컴파일 시간에 결정되며, 주로 클래스 간의 관계를 의미한다. 앞서 보여준 클래스 의존 관계 그림이 바로 정적 의존관계이다. 쉽게 이야기해서 프로그램을 실행하지 않고, 클래스 내에서 사용하는 타입들만 보면 쉽게 의존관계를 파악할 수 있다.
- 동적 의존관계는 프로그램을 실행하는 런타임에 확인할 수 있는 의존관계이다. 앞서 `ObjectPrinter.print(Object obj)`에 인자로 어떤 객체가 전달 될 지는 프로그램을 실행해봐야 알 수 있다. 어떤 경우에는 `Car` 인스턴스가 넘어오고, 어떤 경우에는 `Dog` 인스턴스가 넘어온다. 이렇게 런타임에 어떤 인스턴스를 사용하는지를 나타내는 것이 동적 의존관계이다.
- 참고로 단순히 의존관계 또는 어디에 의존한다고 하면 주로 정적 의존관계를 뜻한다.

- 예) `ObjectPrinter`는 `Object`에 의존한다.

equals() - 1. 동일성과 동등성

`Object`는 동등성 비교를 위한 `equals()` 메서드를 제공한다.

자바는 두 객체가 같다라는 표현을 2가지로 분리해서 제공한다.

- **동일성(Identity)**: `==` 연산자를 사용해서 두 객체의 참조가 동일한 객체를 가리키고 있는지 확인
- **동등성(Equality)**: `equals()` 메서드를 사용하여 두 객체가 논리적으로 동등한지 확인

단어 정리

"동일"은 완전히 같음을 의미한다. 반면 "동등"은 같은 가치나 수준을 의미하지만 그 형태나 외관 등이 완전히 같지는 않을 수 있다.

쉽게 이야기해서 동일성은 물리적으로 같은 메모리에 있는 객체 인스턴스인지 참조값을 확인하는 것이고, 동등성은 논리적으로 같은지 확인하는 것이다.
동일성은 자바 머신 기준이고 메모리의 참조가 기준이므로 물리적이다. 반면 동등성은 보통 사람이 생각하는 논리적인 기준에 맞추어 비교한다.

예를 들어 같은 회원 번호를 가진 회원 객체가 2개 있다고 가정해보자.

```
User a = new User("id-100") //참조 x001
User b = new User("id-100") //참조 x002
```

이 경우 물리적으로 다른 메모리에 있는 다른 객체이지만, 회원 번호를 기준으로 생각해보면 논리적으로는 같은 회원으로 볼 수 있다.
(주민등록번호가 같다고 가정해도 된다.)

따라서 동일성은 다르지만, 동등성은 같다.

문자의 경우도 마찬가지이다.

```
String s1 = "hello";
String s2 = "hello";
```

이 경우 물리적으로는 각각의 "hello" 문자열이 다른 메모리에 존재할 수 있지만, 논리적으로는 같은 `"hello"` 라는 문자열이다.
(사실 이 경우 자바가 같은 메모리를 사용하도록 최적화 한다. 이 부분은 뒤에서 다룬다.)

예제를 통해서 동일성과 동등성을 비교해보자.

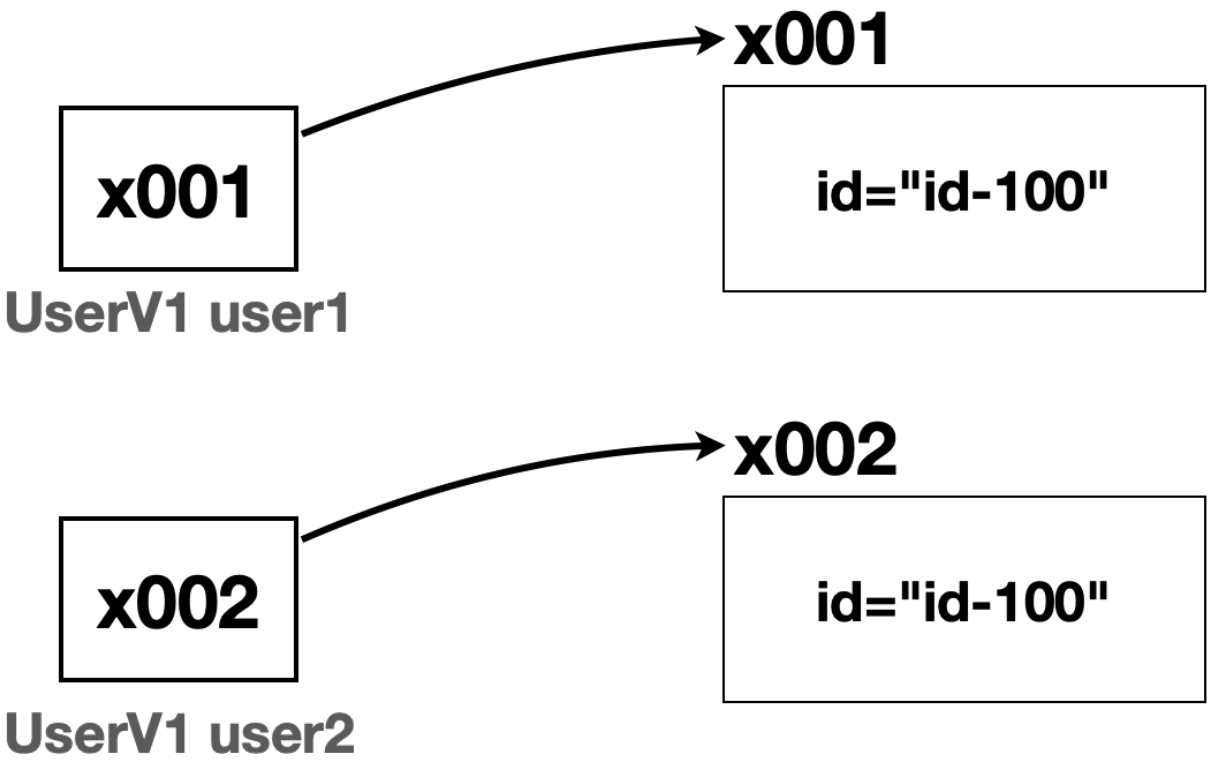
UserV1 예제

```
package lang.object.equals;

public class UserV1 {

    private String id;

    public UserV1(String id) {
        this.id = id;
    }
}
```

```
package lang.object.equals;

public class EqualsMainV1 {

    public static void main(String[] args) {
        UserV1 user1 = new UserV1("id-100");
        UserV1 user2 = new UserV1("id-100");

        System.out.println("identity = " + (user1 == user2));
        System.out.println("equality = " + user1.equals(user2));
    }
}
```

실행 결과

```
identity = false
equality = false
```



동일성 비교

```
user1 == user2
x001 == x002
false //결과
```

동등성 비교

Object.equals() 메서드

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

`Object`가 기본으로 제공하는 `equals()`는 `==`으로 동일성 비교를 제공한다.

equals 실행 순서 예시

```
user1.equals(user2)
return (user1 == user2) //Object.equals 메서드 안
```

```
return (x001 == x002) //Object.equals 메서드 안
return false
false
```

동등성이라는 개념은 각각의 클래스 마다 다르다. 어떤 클래스는 주민등록번호를 기반으로 동등성을 처리할 수 있고, 어떤 클래스는 고객의 연락처를 기반으로 동등성을 처리할 수 있다. 어떤 클래스는 회원 번호를 기반으로 동등성을 처리할 수 있다.
따라서 동등성 비교를 사용하고 싶으면 `equals()` 메서드를 재정의해야 한다. 그렇지 않으면 `Object`는 동일성 비교를 기본으로 제공한다.

equals() - 2. 구현

UserV2 예제

`UserV2`는 `id`(고객번호)가 같으면 논리적으로 같은 객체로 정의하겠다.

```
package lang.object.equals;

public class UserV2 {
    private String id;

    public UserV2(String id) {
        this.id = id;
    }

    @Override
    public boolean equals(Object obj) {
        UserV2 user = (UserV2) obj;
        return id.equals(user.id);
    }

}
```

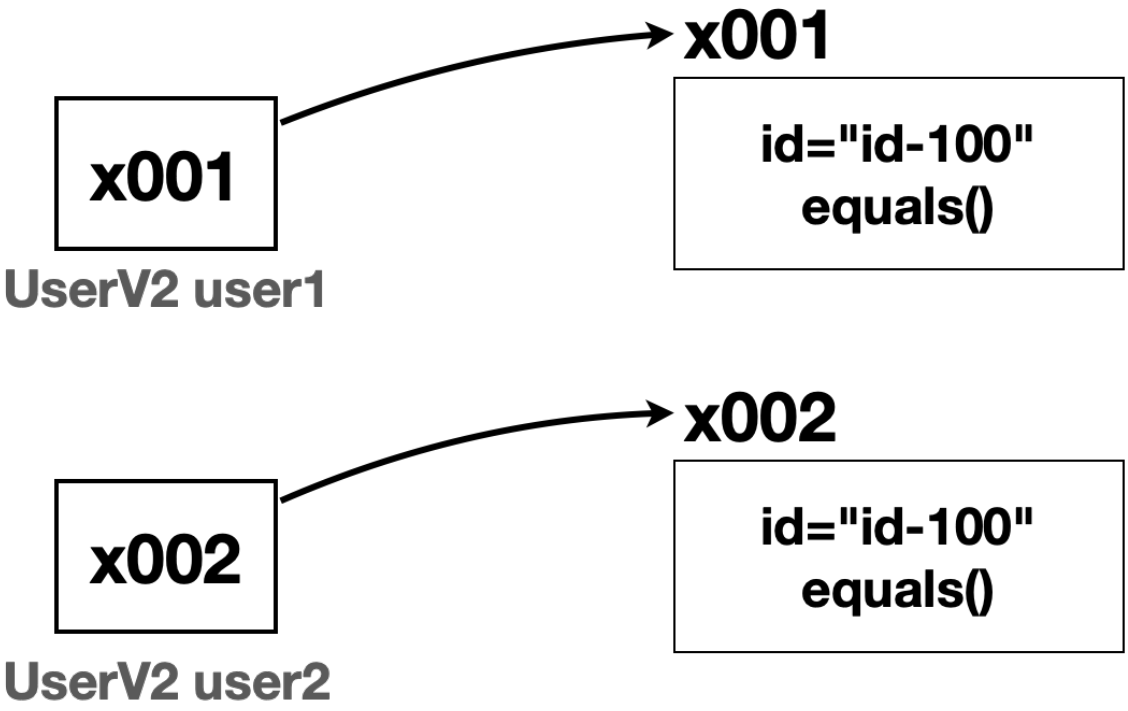
- `Object`의 `equals()` 메서드를 재정의했다.
- `UserV2`의 동등성은 `id`(고객번호)로 비교한다.

- `equals()`는 `Object` 타입을 매개변수로 사용한다. 따라서 객체의 특정 값을 사용하려면 다운캐스팅이 필요하다.
- 여기서는 현재 인스턴스(`this`)에 있는 `id` 문자열과 비교 대상으로 넘어온 객체의 `id` 문자열을 비교한다.
- `UserV2`에 있는 `id`는 `String`이다. 문자열 비교는 `==`이 아니라 `equals()`를 사용해야 한다.

```
package lang.object.equals;

public class EqualsMainV2 {

    public static void main(String[] args) {
        UserV2 user1 = new UserV2("id-100");
        UserV2 user2 = new UserV2("id-100");

        System.out.println("identity = " + (user1 == user2));
        System.out.println("equality = " + user1.equals(user2));
    }
}
```

실행 결과

```
identity = false
equality = true
```



- **동일성(Identity)**: 객체의 참조가 다르므로 동일성은 다르다.
- **동등성(Equality)**: `user1`, `user2`는 서로 다른 객체이지만 둘다 같은 `id`(고객번호)를 가지고 있다. 따라서 동등하다.

정확한 equals() 구현

앞서 `UserV2`에서 구현한 `equals()`는 이해를 돕기 위해 매우 간단히 만든 버전이고, 실제로 정확하게 동작하려면 다음과 같이 구현해야 한다. 정확한 `equals()` 메서드를 구현하는 것은 생각보다 쉽지 않다.

IntelliJ를 포함한 대부분의 IDE는 정확한 `equals()` 코드를 자동으로 만들어준다.

- generator 단축키: `⌘N` (macOS) / `Alt+Insert` (Windows/Linux)

```
//변경 - 정확한 equals 구현, IDE 자동 생성
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;
    User user = (User) o;
    return Objects.equals(id, user.id);
}
```

> **참고**: 최신 IntelliJ에서는 equals 구현에서 다음 첫 줄 코드가 제거된다.

```
if (this == o) return true;
```

- 이유: 내부 분석 결과, 대부분의 실무 코드에서는 자기 자신을 비교하는 경우가 거의 없다. 따라서 분기 예측 미스가 더 큰 비용을 유발한다는 리포트(IDEA-357686)가 반영되었다.

equals() 메서드를 구현할 때 지켜야 하는 규칙

- **반사성(Reflexivity)**: 객체는 자기 자신과 동등해야 한다. (`x.equals(x)`는 항상 `true`).
- **대칭성(Symmetry)**: 두 객체가 서로에 대해 동일하다고 판단하면, 이는 양방향으로 동일해야 한다. (`x.equals(y)`가 `true`이면 `y.equals(x)`도 `true`).
- **추이성(Transitivity)**: 만약 한 객체가 두 번째 객체와 동일하고, 두 번째 객체가 세 번째 객체와 동일하다면, 첫 번째 객체는 세 번째 객체와도 동일해야 한다.
- **일관성(Consistency)**: 두 객체의 상태가 변경되지 않는 한, `equals()` 메소드는 항상 동일한 값을 반환해야 한다.
- **null에 대한 비교**: 모든 객체는 `null`과 비교했을 때 `false`를 반환해야 한다.

실무에서는 대부분 IDE가 만들어주는 `equals()`를 사용하므로, 이 규칙을 외우기 보다는 대략 이렇구나 정도로 한번 읽어보고 넘어가면 충분하다.

정리

- 참고로 동등성 비교가 항상 필요한 것은 아니다. 동등성 비교가 필요한 경우에만 `equals()`를 재정의하면 된다.
- `equals()`와 `hashCode()`는 보통 함께 사용된다. 이 부분은 뒤에 컬렉션 프레임워크에서 자세히 설명한다.

문제와 풀이

문제: toString(), equals() 구현하기

문제 설명

- 다음 코드와 실행 결과를 참고해서 `Rectangle` 클래스를 만들어라.
- `Rectangle` 클래스에 IDE의 기능을 사용해서 `toString()`, `equals()` 메서드를 실행 결과에 맞도록 재정의해라.
- `rect1`과 `rect2`는 너비(`width`)와 높이(`height`)를 가진다. 너비와 높이가 모두 같다면 동등성 비교에 성공해야 한다.

```
package lang.object.test;

public class RectangleMain {
    public static void main(String[] args) {
        Rectangle rect1 = new Rectangle(100, 20);
        Rectangle rect2 = new Rectangle(100, 20);
        System.out.println(rect1);
        System.out.println(rect2);
        System.out.println(rect1 == rect2);
        System.out.println(rect1.equals(rect2));
    }
}
```

실행 결과

```
Rectangle{width=100, height=20}
Rectangle{width=100, height=20}
false
true
```

정답 - Rectangle 클래스

```
package lang.object.test;

public class Rectangle {

    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
```

```
        if (o == null || getClass() != o.getClass()) return false;
        Rectangle rectangle = (Rectangle) o;
        return width == rectangle.width && height == rectangle.height;
    }

    @Override
    public String toString() {
        return "Rectangle{" +
                "width=" + width +
                ", height=" + height +
                '}';
    }
}
```

> **참고**: 최신 IntelliJ에서는 equals 구현에서 다음 첫 줄 코드가 제거된다.

```
if (this == o) return true;
```

- 이유: 내부 분석 결과, 대부분의 실무 코드에서는 자기 자신을 비교하는 경우가 거의 없다. 따라서 분기 예측 미스가 더 큰 비용을 유발한다는 리포트(IDEA-357686)가 반영되었다.

정리

Object의 나머지 메서드

- `clone()` ➜ 객체를 복사할 때 사용한다. 잘 사용하지 않으므로 다루지 않는다.
- `hashCode()` ➜ `equals()`와 `hashCode()`는 종종 함께 사용된다. `hashCode()`는 뒤에 컬렉션 프레임워크에서 자세히 설명한다.
- `getClass()` ➜ 뒤에 `Class`에서 설명한다.
- `notify()`, `notifyAll()`, `wait()` ➜ 멀티쓰레드용 메서드이다. 멀티쓰레드에서 다룬다.