08-02 [Java]

소유자	종수 김
태그	

java.lang 패키지

자바가 기본으로 제공하는 라이브러리 중 가장 기본이 되는 패키지

[대표적인 클래스]

Object : 모든 자바 객체의 부모 클래스

String : 문자열

Integer, Long, Double : 래퍼 타입, 기본형 데이터 타입을 객체로 만든 것

Class : 클래스 메타 정보

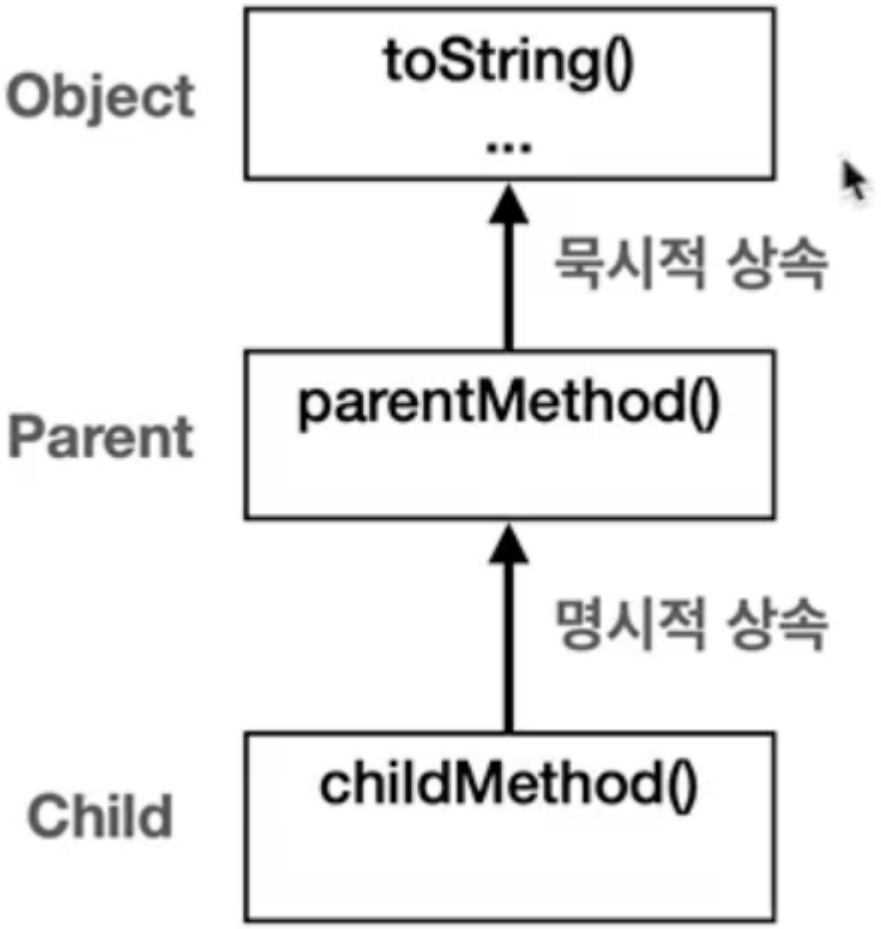
System : 시스템과 관련된 기본 기능 제공

- java.lang 패키지는 Import 생략이 가능하다.

Object

모든 클래스의 최상위 부모 클래스

= extends Object

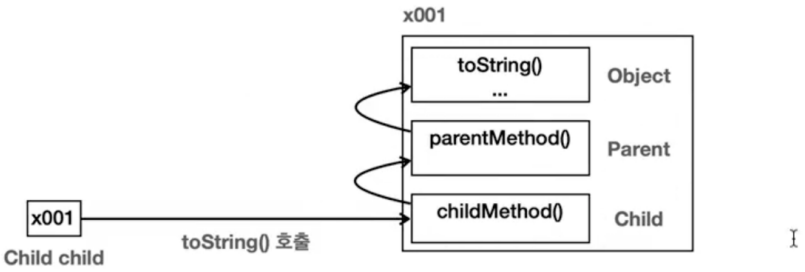


- 묵시적 : 개발자가 코드에 직접 기술하지 않아도 컴파일러가 자동으로 동작
- 명시적 : 개발자가 직접 코드에 명시해주어야함

toString(), equals() 메서드가 Object 클래스의 메서드

실행 결과 그림

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



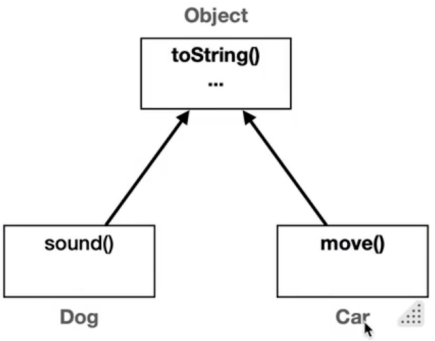
1. `child.toString()`을 호출한다.
2. 먼저 본인의 타입인 `Child`에서 `toString()`을 찾는다. 없으므로 부모 타입으로 올라가서 찾는다.
3. 부모 타입인 `Parent`에서 찾는다. 없으므로 부모 타입으로 올라가서 찾는다.
4. 부모 타입인 `Object`에서 찾는다. `Object`에 `toString()`이 있으므로 이 메서드를 호출한다.

[자바에서 Object 클래스가 최상위 부모 클래스인 이유]

1. 공통기능 제공
    a. 객체의 정보를 제공하고, 이 객체가 다른 객체와 같은지 비교하고, 객체가 어떤 클래스로 만들어졌는지 확인하는 기능은 모든 객체에 필요한 '기본 기능'
    b. 비교, 정보제공 등을 각각 만든다면 일관성(네이밍, 구현방법)이 떨어짐.
2. 다형성의 기본 구현
    a. 부모는 자식을 담을 수 있음.
    따라서 Object는 모든 객체를 참조할 수 있음.
    b. 모든 자바 객체는 Object 타입으로 처리될 수 있으므로 다양한 타입의 객체를 통합적으로 처리할 수 있음.

Object의 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;

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();
//        obj.move();

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

Object Class는 Dog의 Sound Method, Car의 move()가 없으므로 컴파일 에러가 발생
Object는 가장 최상의 부모 클래스이므로, 다운캐스팅이 다 가능.

[한계]

Object는 모든 객체를 대상으로 다형적 참조를 할 수 있음.
Object를 통해 전달 받은 객체를 호출하려면 각 객체에 맞는 다운캐스팅 과정이 필요.

- Object는 모든 메서드를 알고있지 않기 때문.

다형성을 제대로 활용하려면 다형적 참조 + 메서드 오버라이딩을 함께 사용해야함.

Object는 모든 객체의 부모이므로 모든 객체를 대상으로 다형적 참조를 할 수 있음.
하지만, Object에는 Dog.sound, Car.move가 없으므로 오버라이딩이 불가능.

다형적 참조만 가능하고, 메서드 오버라이딩이 안되는것.

Object 배열

Object는 모든 객체를 담을 수 있으므로, Object 배열은 모든것을 담을 수 있음.

```
    private static void size(Object[] objects) {
        System.out.println("objects.size = " + objects.length
    }
```

배열의 개수를 리턴해주는 간단한 메서드로, Object를 인자로 받기 때문에 모든 곳에서 사용이 가능.

Object가 없다면, 위처럼 모든 객체를 받을 수 있는 메서드를 만들 수 없음.

- 물론, MyObject를 통해 가능은 하나, 전세계 모든 사람이 직접 만든다면 호환성(일관성)이 떨어질 것.

toString()

Object 클래스가 제공하는 메서드
기본적으로 패키지를 포함한 객체의 이름과 객체의 참조값(해시코드)를 16진수로 제공

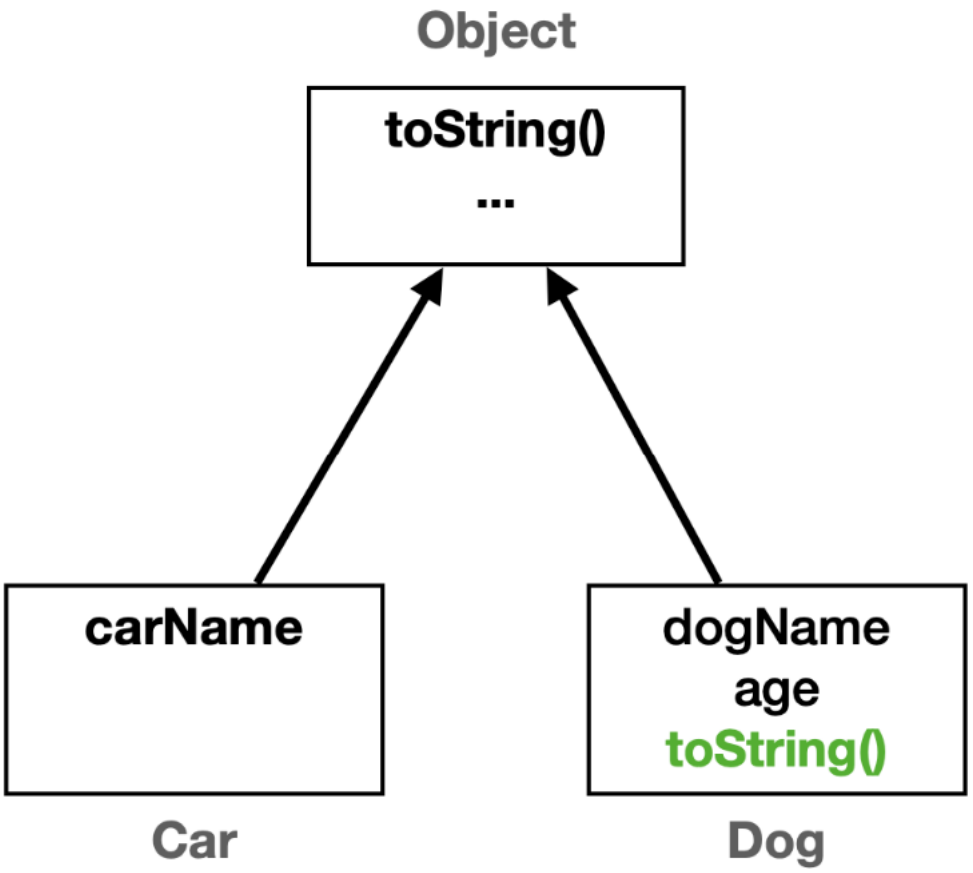
```
package lang.object.tostring;

public class ToStringMain1 {
    public static void main(String[] args) {
        Object object = new Object();
        String string = object.toString();
```

```
        //toString() 반환 값 출력
        System.out.println(string);
        System.out.println(object);

    }
}


// java.lang.Object@372f7a8d
// java.lang.Object@372f7a8d
//Process finished with exit code 0
```

해당 코드는 결과가 완전히 동일한데, 이는 println이 해당 메서드의 toString()을 호출하기 때문.

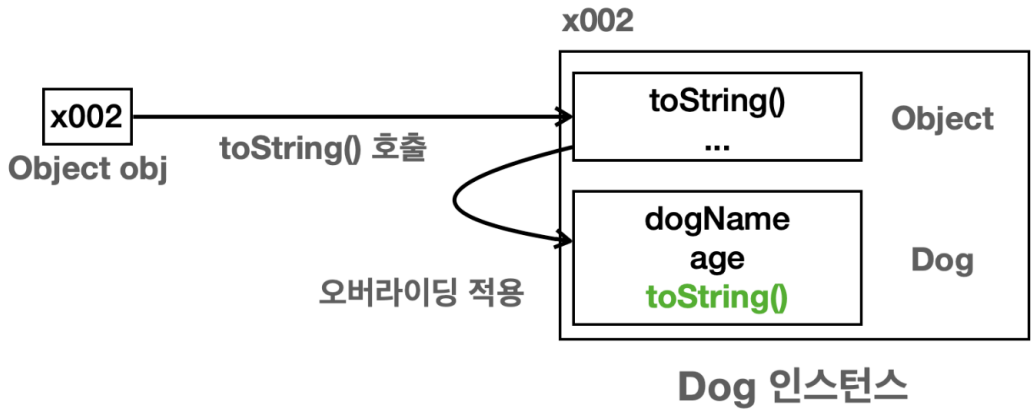
toString() Override

오버라이딩 되지 않은 toString()은 클래스 정보와 참조값을 제공하지만, 이 정보만으로는 객체의 상태를 적절히 나타낼 수 없기에 toString()을 오버라이딩하여 해당 클래스의 정보를 제공하는 것이 일반적



Override를 하는 경우 코드 진행

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.

- 객체의 참조 값을 출력하는 방법

`toString()` 은 기본으로 객체의 참조값을 출력한다. 그런데 `toString()` 이나 `hashCode()` 를 재정의하면 객체

의 참조값을 출력할 수 없다. 이때는 다음 코드를 사용하면 객체의 참조값을 출력할 수 있다.

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

- ***실행 결과***

```
 refValue = 72ea2f77
```

Object와 OCP

Object가 없고, Object가 제공하는 toString()이 없다면, 공통의 부모가 없기에 클래스마다 별도의 메서드를 작성해야함.

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

BadObjectPrinter는 구체적인 타입인 Car, Dog를 사용하며, 구체적인 클래스에 의존하고 있다.

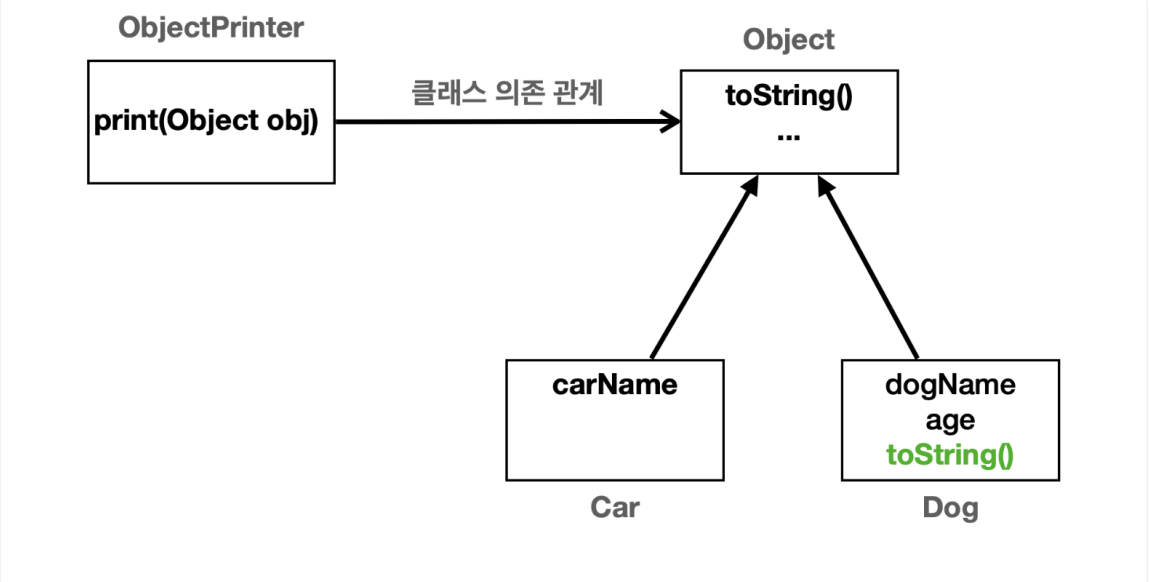
추상적인 것에 의존

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

ObjectPrinter는 구체적인 클래스인 Car, Dog를 사용하는 것이 아닌 추상적인 Object 클래스에 의존하고있다.

- 추상적

  Animal / Dog Cat 과 같은 관계가 있다면 Animal 같은 부모 타입으로 올라갈수록 개념은 더 추상적이게 되고, Dog, Cat과 같은 자식 타입으로 내려갈수록 개념은 더 구체적이게 된다.



- 다형성을 잘 활용한다는 것은 다형적 참조와 메서드 오버라이딩을 적절하게 사용한다는 것.

`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)` 메서
드는 `Dog` , `Car` 와 같은 구체적인 타입에 의존(사용)하지 않고, 추상적인 `Object` 타입에 의존하면서 런타임에 각 인스턴스의 `toString()` 을 호출할 수 있다.

OCP 원칙

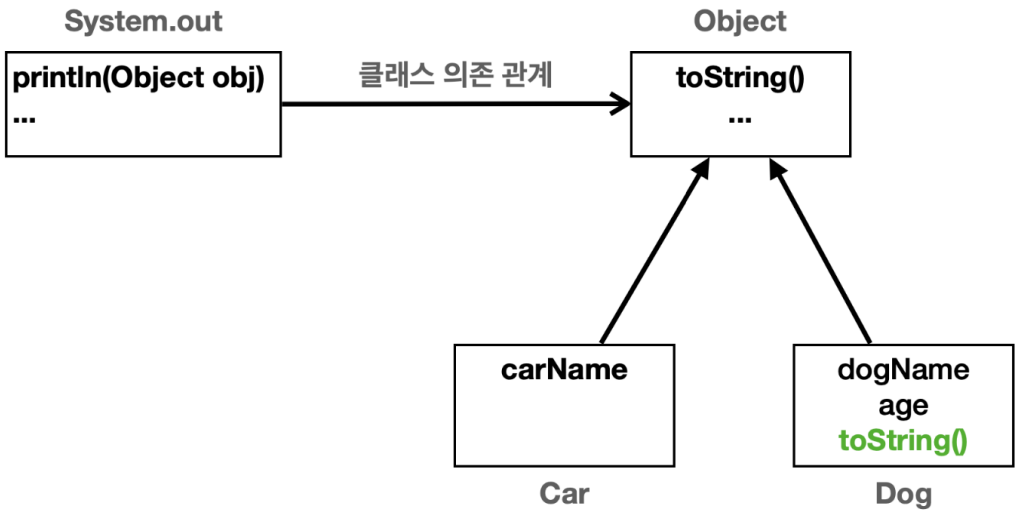
Open : 새로운 클래스를 추가하고, toString()을 오버라이딩해서 기능을 확장할 수 있음.

Close : 새로운 클래스를 추가해도 Object와 toString()을 사용하는 Client 코드인 ObjectPrinter는 변경할 필요 없음.

System.out.println()

ObjectPrinter는 System.out.println과 매우 유사.

내부적으로 Object 매개변수를 사용하며, 내부에서 toString()을 호출.



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.

정적 의존관계와 동적 의존관계

- 정적 의존관계
  - 컴파일 시점에 결정.
  - 주로 클래스 간의 관계를 의미.
  - 프로그램을 실행하지 않고, 클래스 내에서 사용하는 타입들만 보면 쉽게 의존 관계 파악 가능
- 동적 의존관계
  - 프로그램을 실행하는 런타임 시점에 결정
  - 추상화 된 메서드의 매개변수로 넘어온 타입이 Car 인스턴스인지, Dog 인스턴스인지 프로그램을 실행시키고 해당 로직을 돌려봐야 알 수 있는 의존 관계
- 보통 어디에 의존한다, 라는 말은 정적 의존관계를 의미.

equals() - 1. 동일성과 동등성

Object는 동등성 비교를 위한 equals() 메서드를 제공.

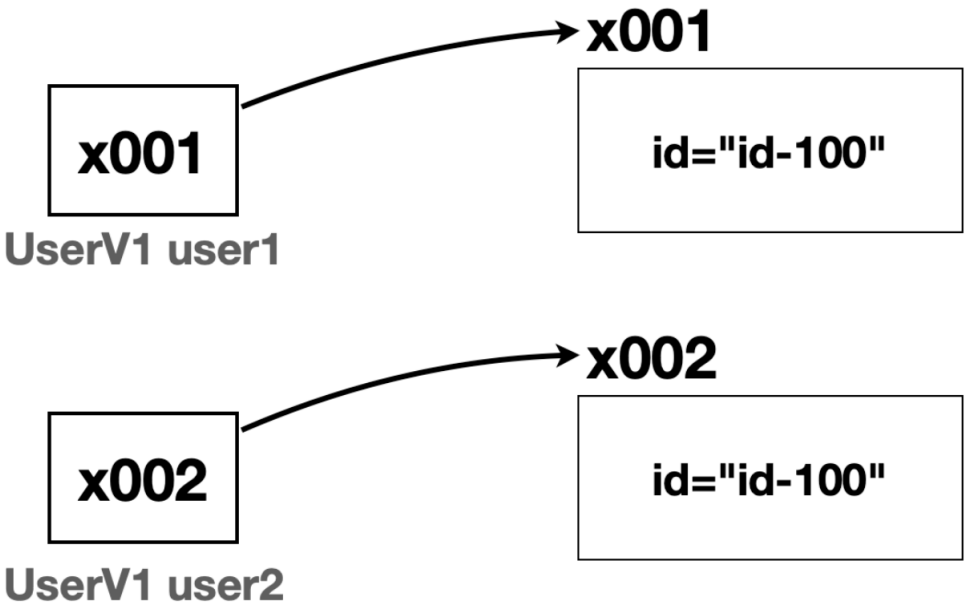
자바는 두 객체가 같다 라는 표현을 2가지로 분리해서 제공.

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

동일은 완전히 같음을 의미.
동등은 같은 가치나 수준을 의미하지만 그 형태나 외관 등이 완전히 같진 않을 수 있음.

- 동일성은 자바 머신 기준이고, 메모리의 참조가 기준이므로 물리적.
- 동등성은 보통 사람이 생각하는 논리적인 기준에 맞추어 비교.

```
User user1 = new User("id-100");
User user2 = new User("id-100");
```



이 경우, 다른 메모리에 있는 다른 객체이나 회원 번호를 기준으로 생각해보면 논리적으론 같은 회원.
동일성은 다르지만 동등성은 같음.

```
    public static void main(String[] args) {
        UserV1 v1 = new UserV1("id-100");
        UserV1 v2 = new UserV1("id-100");
```

```
        System.out.println("identity = " + (v1 == v2));
        System.out.println("equality = " + v1.equals(v2));

    }

    //false
    //false
```

- equals가 false인 이유.
    - 오버라이딩하기 전까지 equals 메서드는 동일성과 마찬가지로 == 연산자를 통해 해당 객체를 비교하기 때문.
    - 동등성이라는건 객체마다 다르기 때문에(ex. 주민번호, 회원번호 등) 해당 객체에 맞는 동등성을 재 정의해줄 필요가 있음.

equals() - 2. 예제

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.

```
public static void main(String[] args) {
     UserV2 v1 = new UserV2("id-100");
     UserV2 v2 = new UserV2("id-100");


     System.out.println("identity = " + (v1 == v2));
     System.out.println("equality = " + v1.equals(v2));

 }

 //false
 //true
```

- 아까와 같이 인스턴스는 따로 생성되기에 동일성은 다름.
- equals 재정의를 통해 동일해야 같은 객체가 아닌, 유저의 id가 같다면 두 객체가 동등하다 라고 정의해줬으므로 true.

정확한 equals의 구현.

```
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return f
        UserV2 userV2 = (UserV2) o;
        return Objects.equals(id, userV2.id);
    }
```

규칙

1. 반사성(Reflexivity)
    a. 객체는 자기 자신과 동등해야 한다. (x.equals(x)는 항상 true)
2. 대칭성(Symmetry)
    a. 두 객체가 서로에 대해 동일하다고 판단하면, 이는 양방향으로 동일해야한다. (x.equals(y)가 true 면 y.equals(x)도 true)
3. 추이성(Transitivity)
    a. 만약 한 객체가 두 번째 객체와 동일하고 두 번째 객체가 세 번째 객체와 동일하다면, 첫 번째 객체와 세 번째 객체는 동일하다.
4. 일관성(Consistency)
    a. 두 객체의 상태가 변경되지 않는 한, equals() 메서드는 항상 동일한 값을 반환해야 한다.
5. null에대한 비교
    a. 모든 객체는 null과 비교했을 때 false를 반환해야 한다.