10-13 [Java - String 클래스]

소유자	종수 김
태그	

String 클래스

기본

자바에서 문자를 다루는 대표적인 타입 char, String

```
package string;

import java.util.Arrays;

public class CharArrayMain {
    public static void main(String[] args) {
        char a = '가';
        System.out.println("a = " + a);
        char[] charArr = new char[]{'h', 'e', 'l', 'l', 'o'};
        System.out.println(charArr);
    }
}
```

기본형인 char는 문자 하나를 다루며, 문자열을 표현하기 위해서는 배열이 필요하다. 이처럼 char로 문자열을 표현하기에는 매우 불편하여 자바에서는 String 클래스를 따로 제공한다.

String을 생성하는 두 가지 방법

```
package string;

public class StringBasicMain {
    public static void main(String[] args) {
        String str = "Hello";
        String str2 = new String("Hello");
```

```
        System.out.println("str = " + str);
        System.out.println("str2 = " + str2);
    }
}
```

String은 클래스로, int boolean과 같은 기본형이 아닌 참조형.

따라서 str변수에는 인스턴스의 참조 값 만이 들어갈 수 있는데 String은 기본형처럼 문자열 자체를 대입하고 있음.

- 문자열은 매우 자주 사용되므로 편의상 쌍따옴표로 문자열을 감싸면 new String(”Hello”)와 같이 변경
  성능 최적화를 위해 문자열 풀을 사용.

String 클래스 구조

```
public final class String {

    //문자열 보관
    private final char[] value;// 자바 9 이전
    private final byte[] value;// 자바 9 이후
```

```
    //여러 메서드
    public String concat(String str) {...}
    public int length() {...}
    ...
}
```

클래스이므로 속성과 기능을 가진다.

속성(필드)

```
private final char[] value;
```

여기에는 `String`의 실제 문자열 값이 보관된다. 문자 데이터 자체는 `char[]`에 보관된다.
`String` 클래스는 개발자가 직접 다루기 불편한 `char[]`을 내부에 감추고 `String` 클래스를 사용하는 개발자가 편리하게 문자열을 다룰 수 있도록 다양한 기능을 제공한다. 그리고 메서드 제공을 넘어서 자바 언어 차원에서도 여러 편의 문법을 제공한다.

> **참고: 자바 9 이후 String 클래스 변경 사항**
> 자바 9부터 `String` 클래스에서 `char[]` 대신에 `byte[]`을 사용한다.
>
> ```
> java
> private final byte[] value;
> ```
>
> 자바에서 문자 하나를 표현하는 `char`는 `2byte`를 차지한다. 그런데 영어, 숫자는 보통 `1byte`로 표현이 가능하다. 그래서 단순 영어, 숫자로만 표현된 경우 `1byte`를 사용하고(정확히는 Latin-1 인코딩의 경우 `1byte` 사용) , 그렇지 않은 나머지의 경우 `2byte`인 UTF-16 인코딩을 사용한다. 따라서 메모리를 더 효율적으로 사용할 수 있게 변경되었다.

String 클래스와 참조형

String은 클래스이다. 기본형이 아닌 참조형.

참조형은 변수에 계산할 수 있는 값이 들어있는 것이 아니라 x001과 같이 계산할 수 없는 참조 값이 들어가 있기 때문에 원칙적으로 + 연산자가 사용 불가능해야 함.

```
package string;

public class StringConcatMain {
    public static void main(String[] args) {
        String a = "hello";
        String b = "java";

        String result1 = a.concat(b);
        String result2 = a + b;

        System.out.println("result1 = " + result1);
        System.out.println("result2 = " + result2);
    }
}

// hellojava
// hellojava
```

concat은 합치는 함수이기에 결과가 예상되지만, 참조형이 가지는 참조값 변수는 + 연산이 불가능함에도 같은 결과가 출력됨.

이는 자바에서 문자열이 매우 자주 다루어지기에 편의상 + 연산자를 제공해준 것.

String 클래스 - 비교

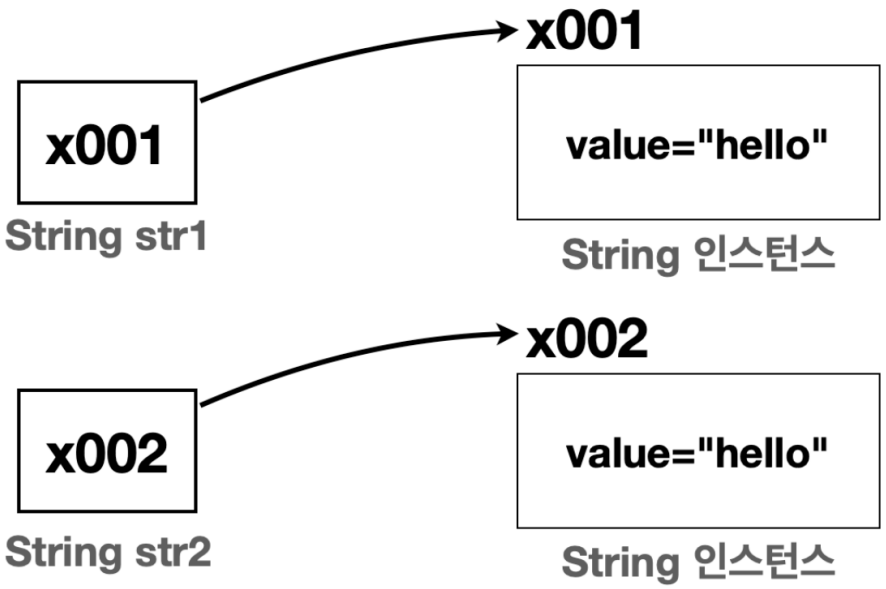
String 클래스를 비교할 때는 == 이 아닌, equals를 통하여 비교해야 함.

```
package string.equals;

public class StringEqualsMain1 {
    public static void main(String[] args) {
        String str1 = new String("hello");
        String str2 = new String("hello");
        System.out.println(str1 == str2);
        System.out.println(str1.equals(str2));

        String str3 = "hello";
        String str4 = "hello";
        System.out.println(str3 == str4);
```

```
        System.out.println(str3.equals(str4));
    }
}

//false
//true
//true
//true
```

str1과 str2는 다른 인스턴스이므로 당연히 false가 나옴.
하지만, String 클래스는 equals 오버라이드를 통해 해당 문자열을 비교하도록 해놓았음.

그림 - new String() 비교

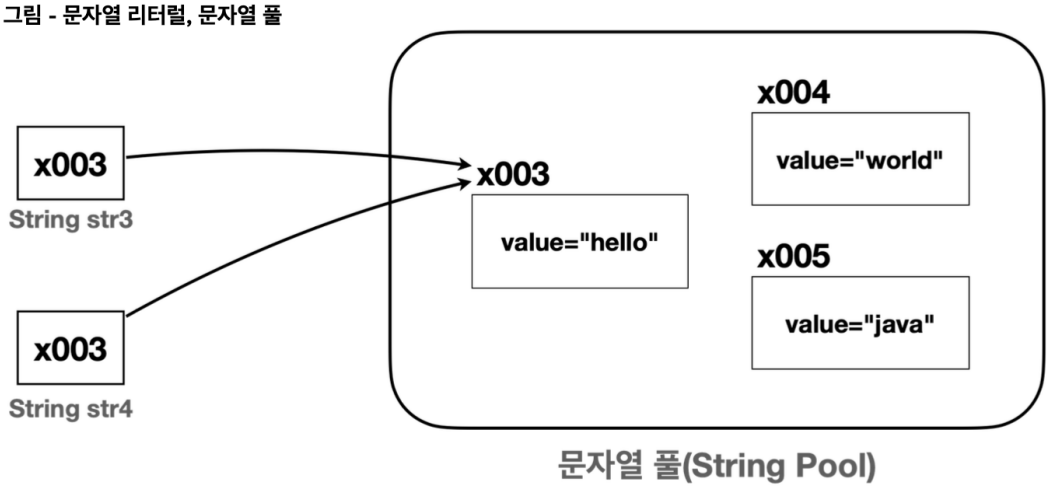


- `str1`과 `str2`는 `new String()`을 사용해서 각각 인스턴스를 생성했다. 서로 다른 인스턴스이므로 동일성(`==`) 비교에 실패한다.
- 둘은 내부에 같은 `"hello"` 값을 가지고 있기 때문에 논리적으로 같다. 따라서 동등성(`equals()`) 비교에 성

공한다. 참고로 `String` 클래스는 내부 문자열 값을 비교하도록 `equals()` 메서드를 재정의 해두었다.

str3과 str4는 위에서 설명한 것이라면 new String이 생략된 것 이기에 서로 다른 인스턴스여야 하지만 true가 출력됨.

그림 - 문자열 리터럴, 문자열 풀



- `String str3 = "hello"`와 같이 문자열 리터럴을 사용하는 경우 자바는 메모리 효율성과 성능 최적화를 위해 문자열 풀을 사용한다.
- 자바가 실행되는 시점에 클래스에 문자열 리터럴이 있으면 문자열 풀에 `String` 인스턴스를 미리 만들어둔다. 이때 같은 문자열이 있으면 만들지 않는다.
- `String str3 = "hello"`와 같이 문자열 리터럴을 사용하면 문자열 풀에서 `"hello"`라는 문자를 가진 `String` 인스턴스를 찾는다. 그리고 찾은 인스턴스의 참조(`x003`)를 반환한다.
- `String str4 = "hello"`의 경우 `"hello"` 문자열 리터럴을 사용하므로 문자열 풀에서 `str3`과 같은 `x003` 참조를 사용한다.
- 문자열 풀 덕분에 같은 문자를 사용하는 경우 메모리 사용을 줄이고 문자를 만드는 시간도 줄어들기 때문에 성능도 최적화 할 수 있다.

따라서 문자열 리터럴을 사용하는 경우 같은 참조값을 가지므로 `==` 비교에 성공한다.

- String Constant Pool

  여러 곳에서 함께 사용할 수 있는 객체를 필요할 때 마다 생성하고, 제거하는 것은 비 효율적.
  이렇게 문자열 풀에 필요한 String 인스턴스를 만들어 둔 뒤 여러 곳에서 재사용할 수 있다면, 성능과 메모리 최적화가 가능.

  Heap 영역을 사용하며 해시 알고리즘을 사용하기 때문에 매우 빠른 속도로 String 인스턴스를 찾을 수 있음.

항상 equals를 사용해야하는 이유

```
package string.equals;

public class StringEqualsMain2 {
    public static void main(String[] args) {
        String str1 = new String("hello");
        String str2 = new String("hello");
        System.out.println("메서드 호출 비교1 : " + isSame(str1,

        String str3 = "hello";
        String str4 = "hello";
        System.out.println("메서드 호출 비교2 : " + isSame(str3,
    }

    private static boolean isSame(String x, String y) {
        return x == y;
    }
}

// false
// true
```

결과는 위와 동일하나, isSame 메서드는 new String으로 만들어진 String 변수가 넘어올지, 리터럴을 통해 만들어진 String 변수가 넘어올 지 알 수 없기 때문에 메서드의 안정성 및 사용성을 위해 equals로 비교하는 것이 무조건 옳다.

String 클래스 - 불변 객체

String은 불변 객체로 생성된 후 내부 문자열 값을 변경할 수 없음.

```
@Stable
private final byte[] value;
```

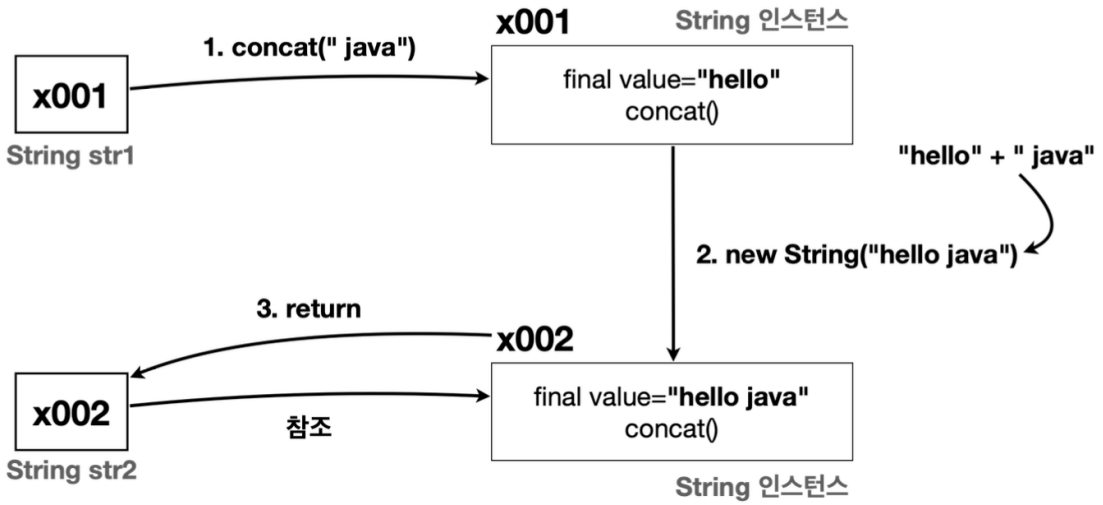
문자열이 합쳐지지 않음.

```
package string.immutable;

public class StringImmutable1 {
    public static void main(String[] args) {
        String str = "hello";
```

```
        str.concat("java");
        System.out.println("str = " + str);
    }
}
// hello
```

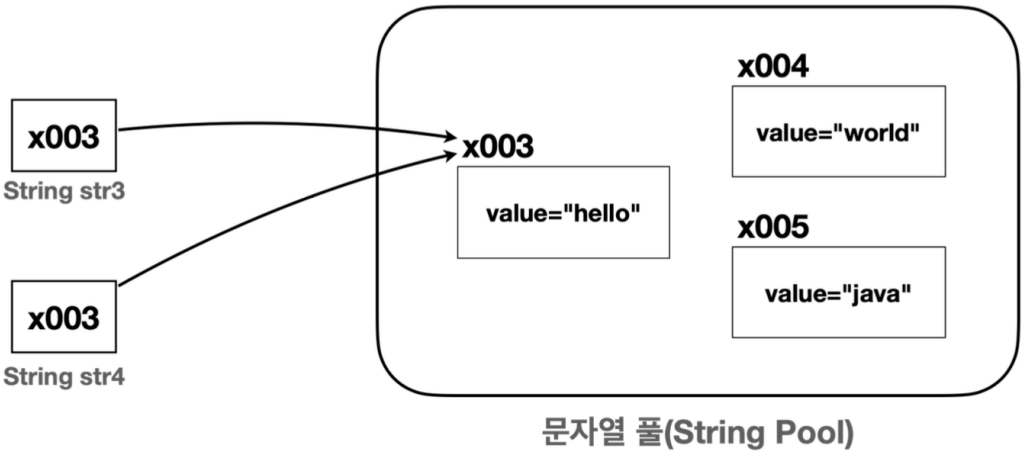
String은 불변 객체이므로 변경이 필요한 경우 기존 값을 변경하지 않고, 새로운 결과를 만들어 반환.

```
package string.immutable;

public class StringImmutable2 {
    public static void main(String[] args) {
        String str = "hello";
        str = str.concat("java");
        System.out.println("str = " + str);
    }
}
hellojava
```



String이 불변으로 설계된 이유

문자열 풀에 있는 String 인스턴스의 값이 중간에 변경되면 같은 문자열을 참고하는 다른 변수의 값도 함께 변경.



- `String`은 자바 내부에서 문자열 풀을 통해 최적화를 한다.
- 만약 `String` 내부의 값을 변경할 수 있다면, 기존에 문자열 풀에서 같은 문자를 참조하는 변수의 모든 문자가 함께 변경되어 버리는 문제가 발생한다. 다음의 경우 `str3`이 참조하는 문자를 변경하면 `str4`의 문자도 함께 변경되는 사이드 이펙트 문제가 발생한다.
  - `String str3 = "hello"`
  - `String str4 = "hello"`

`String` 클래스는 불변으로 설계되어서 이런 사이드 이펙트 문제가 발생하지 않는다.

StringBuilder - 가변 String

불변인 String 클래스의 단점

두 문자를 더하는 경우 다음과 같이 작동한다.

```
"A" + "B"
String("A") + String("B") //문자는 String 타입이다.
String("A").concat(String("B"))//문자의 더하기는 concat을 사용한다.
new String("AB") //String은 불변이다. 따라서 새로운 객체가 생성된다.
```

불변인 `String`의 내부 값은 변경할 수 없다. 따라서 변경된 값을 기반으로 새로운 `String` 객체를 생성한다.

더 많은 문자를 더하는 경우를 살펴보자.

```
String str = "A" + "B" + "C" + "D";
String str = String("A") + String("B") + String("C") + String("D");
String str = new String("AB") + String("C") + String("D");
String str = new String("ABC") + String("D");
String str = new String("ABCD");
```

- 이 경우 총 3개의 `String` 클래스가 추가로 생성된다.
- 그런데 문제는 중간에 만들어진 `new String("AB")`, `new String("ABC")`는 사용되지 않는다. 최종적으로 만들어진 `new String("ABCD")`만 사용된다.
- 결과적으로 중간에 만들어진 `new String("AB")`, `new String("ABC")`는 제대로 사용되지도 않고, 이후 GC의 대상이 된다.

불변인 `String` 클래스의 단점은 문자를 더하거나 변경할 때 마다 계속해서 새로운 객체를 생성해야 한다는 점이다. 문자를 자주 더하거나 변경해야 하는 상황이라면 더 많은 `String` 객체를 만들고, GC해야 한다. 결과적으로 컴퓨터의 CPU, 메모리를 자원을 더 많이 사용하게 된다. 그리고 문자열의 크기가 클수록, 문자열을 더 자주 변경할수록 시스템의 자원을 더 많이 소모한다.

StringBuilder

위의 문제를 해결하는 방법은 가변 String을 사용하는 것.
가변은 내부의 값을 바로 변경하면 되기 때문에 새로운 객체를 생성할 필요가 없어 메모리 측면에서 더 효율적

StringBuilder내부는 final이 아닌 byte[]가 존재.

```
package string.builder;

public class StringBuilderMain1_1 {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        sb.append("A");
        sb.append("B");
        sb.append("C");
        sb.append("D");
        System.out.println("sb.toString() = " + sb.toString()
```

```
        // 문자열 추가
        sb.insert(4, "Java");
        System.out.println("sb.toString() = " + sb.toString()

        // 문자열 삭제
        sb.delete(4, 8);
        System.out.println("sb.toString() = " + sb.toString()

        // 문자열 뒤집기
        sb.reverse();
        System.out.println("sb.toString() = " + sb.toString()


    }
}

// ABCD
// ABCDJAVA
// ABCD
// DCBA
```

- StringBuilder는 가변이므로 문자열을 조작하는 메서드에 리턴 값이 존재하지 않고 본인 객체를 바로 수정.
- 보통 문자열을 변경하는 동안만 사용하다가 문자열 변경이 끝나면 String(불변)으로 변환하는 것이 좋음.

가변 vs 불변

String은 불변하다. 즉 한번 생성되면 그 내용을 변경할 수 없기에 새로운 String 객체가 생성되고 기존 객체는 버려지는 등의 메모리 낭비가 일어난다.

반면에, StringBuilder는 가변적으로 객체 안에서 문자열을 추가, 삭제, 수정할 수 있으며 이때 마다 새로운 객체를 생성하지 않기에 메모리 사용을 줄이고 성능을 향상시킬 수 있으나 사이드 이펙트에 주의하여야 함.

String 최적화

문자열 리터럴 최적화

컴파일 전

```
String helloWorld = "Hello, " + "World!";
```

컴파일 후

```
String helloWorld = "Hello, World!";
```

따라서 런타임에 별도의 문자열 결합 연산을 수행하지 않기 때문에 성능이 향상된다.

String 변수 최적화

문자열 변수의 경우 그 안에 어떤 값이 들어있는지 컴파일 시점에는 알 수 없기 때문에 단순하게 합칠 수 없다.

```
String result = str1 + str2;
```

이런 경우 예를 들면 다음과 같이 최적화를 수행한다. (최적화 방식은 자바 버전에 따라 달라진다.)

```
String result = new StringBuilder().append(str1).append(str2).toString();
```

> 참고: 자바 9부터는 `StringConcatFactory`를 사용해서 최적화를 수행한다.

이렇듯 자바가 최적화를 처리해주기 때문에 지금처럼 간단한 경우에는 `StringBuilder`를 사용하지 않아도 된다. 대신에 문자열 더하기 연산(`+`)을 사용하면 충분하다.

String 최적화가 어려운 경우

문자열을 루프 안에서 문자열을 더하는 경우.

```
package string.builder;

public class LoopStringMain {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        String result = "";
        for (int i = 0; i < 100000; i++) {
            result += "Hello Java";
        }

        long endTime = System.currentTimeMillis();
        System.out.println(result);
        System.out.println(endTime - start);
    }
}
```

```
// 1257
```

왜냐하면 대략 다음과 같이 최적화 되기 때문이다. (최적화 방식은 자바 버전에 따라 다르다)

```
String result = "";
for (int i = 0; i < 100000; i++) {
    result = new StringBuilder().append(result).append("Hello Java
").toString();
}
```

반복문의 루프 내부에서는 최적화가 되는 것 처럼 보이지만, 반복 횟수만큼 객체를 생성해야 한다.

반복문 내에서의 문자열 연결은, 런타임에 연결할 문자열의 개수와 내용이 결정된다. 이런 경우, 컴파일러는 얼마나 많은 반복이 일어날지, 각 반복에서 문자열이 어떻게 변할지 예측할 수 없다. 따라서, 이런 상황에서는 최적화가 어렵다.

`StringBuilder`는 물론이고, 아마도 대략 반복 횟수인 100,000번의 `String` 객체를 생성했을 것이다.

개발자가 직접 StringBuilder를 사용하면 됨.

```
package string.builder;

public class LoopStringBuilderMain {
    public static void main(String[] args) {
        long start = System.currentTimeMillis();
        StringBuilder result = new StringBuilder();
        for (int i = 0; i < 100000; i++) {
            result.append("Hello Java");
        }

        long endTime = System.currentTimeMillis();
        System.out.println(result);
        System.out.println(endTime - start);
    }
}

// 4
```

StringBuilder 직접 사용하는 것이 더 좋은 경우

1. 반복문에서 반복해서 문자를 연결할 때

2. 조건문을 통해 동적으로 문자열을 조합할 때
3. 복잡한 문자열의 특정 부분을 변경해야 할 때
4. 매우 긴 대용량의 문자열을 다룰 때

참고

> **참고: StringBuilder vs StringBuffer**
> `StringBuilder`와 똑같은 기능을 수행하는 `StringBuffer` 클래스도 있다.
> `StringBuffer`는 내부에 동기화가 되어 있어서, 멀티 스레드 상황에 안전하지만 동기화 오버헤드로 인해 성능이 느리다.
> `StringBuilder`는 멀티 쓰레드에 상황에 안전하지 않지만 동기화 오버헤드가 없으므로 속도가 빠르다.
> `StringBuffer`와 동기화에 관한 내용은 이후에 멀티 스레드를 학습해야 이해할 수 있다. 지금은 이런 것이 있구나 정도만 알아두면 된다.

메서드 체이닝 - Method Chaining

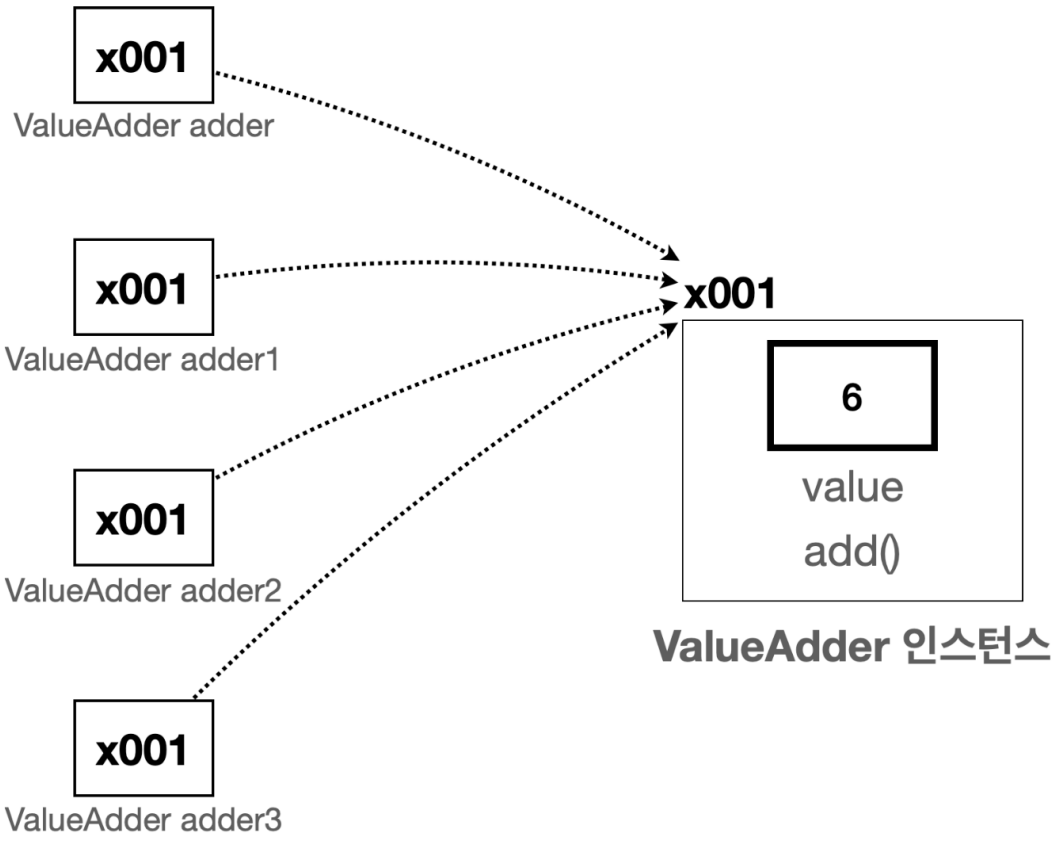
인스턴스 자신의 참조 값을 반환하고 이를 변수에 담아두지 않고 참조값을 즉시 사용하여 메서드를 이어 나가는 것이 가능.

메서드 체이닝이 가능한 이유는 자기 자신의 참조값을 반환하기 때문.

코드를 간결하고 읽기 쉽게 만들어줌.

```
package string.chaining;

public class ValueAdder {
    private int value;

    public ValueAdder add(int addValue) {
        value += addValue;
        return this;
    }

    public int getValue() {
        return value;
    }
```

```
}


package string.chaining;

public class MethodChainingMain3 {
    public static void main(String[] args) {
        ValueAdder adder = new ValueAdder();
        adder.add(1).add(2).add(3);

        int result = adder.getValue();
        System.out.println("result = " + result);
    }
}

// 6
```



- `add()` 메서드는 자기 자신(`this`)의 참조값을 반환한다. 이 반환값을 `adder1`, `adder2`, `adder3`에 보관했다.
- 따라서 `adder`, `adder1`, `adder2`, `adder3`은 모두 같은 참조값을 사용한다. 왜냐하면 `add()` 메서드가 자기 자신(`this`)의 참조값을 반환했기 때문이다.

다음과 같은 순서로 실행된다.

```
adder.add(1).add(2).add(3).getValue() //value=0
x001.add(1).add(2).add(3).getValue() //value=0, x001.add(1)을 호출하면 그 결과로 x001을 반환한다.
x001.add(2).add(3).getValue() //value=1, x001.add(2)을 호출하면 그 결과로 x001을 반환한다.
x001.add(3).getValue() //value=3, x001.add(3)을 호출하면 그 결과로 x001을 반환한다.
x001.getValue() //value=6
6
```

StringBuilder는 메서드 체이닝 기법을 사용. - append(), delete(), reverse() 등 자기 자신을 반환.

```
package string.builder;
```

```
public class StringBuilderMain1_2 {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        String result = sb.append("A").append("B").append("C"
                .insert(4, "Java")
                .delete(4, 8)
                .reverse()
                .toString();

        System.out.println("result = " + result);

    }
}
```