3. String 클래스

#0.강의/1.자바로드맵/3.자바-중급1편

String 클래스 - 기본

자바에서 문자를 다루는 대표적인 타입은 `char`, `String` 2가지가 있다.

```
package lang.string;

public class CharArrayMain {

    public static void main(String[] args) {
        char[] charArr = new char[]{'h', 'e', 'l', 'l', 'o'};
        System.out.println(charArr);

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

실행 결과

```
hello
```

```
str = hello
```

기본형인 `char`는 문자 하나를 다룰 때 사용한다. `char`를 사용해서 여러 문자를 나열하려면 `char[]`을 사용해야 한다. 하지만 이렇게 `char[]`을 직접 다루는 방법은 매우 불편하기 때문에 자바는 문자열을 매우 편리하게 다룰 수 있는 `String` 클래스를 제공한다.

`String` 클래스를 통해 문자열을 생성하는 방법은 2가지가 있다.

```
package lang.string;

public class StringBasicMain {

    public static void main(String[] args) {
        String str1 = "hello";
        String str2 = new String("hello");

        System.out.println("str1 = " + str1);
        System.out.println("str2 = " + str2);
    }
}
```

- 쌍따옴표 사용: `"hello"`
- 객체 생성: `new String("hello");`

`String`은 클래스다. `int`, `boolean` 같은 기본형이 아니라 참조형이다. 따라서 `str1` 변수에는 `String` 인스턴스의 참조값만 들어갈 수 있다. 따라서 다음 코드는 뭔가 어색하다.

```
String str1 = "hello";
```

문자열은 매우 자주 사용된다. 그래서 편의상 쌍따옴표로 문자열을 감싸면 자바 언어에서 `new String("hello")`와 같이 변경해준다. (이 경우 실제로는 성능 최적화를 위해 문자열 풀을 사용하는데, 이 부분은 뒤에서 설명한다.)

```
String str1 = "hello"; //기존
String str1 = new String("hello"); //변경
```

String 클래스 구조

`String` 클래스는 대략 다음과 같이 생겼다.

```
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.
>
> ```
> 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.

기능(메서드)

`String` 클래스는 문자열로 처리할 수 있는 다양한 기능을 제공한다. 기능이 방대하므로 필요한 기능이 있으면 검색하거나 API 문서를 찾아보자. 주요 메서드는 다음과 같다.

- `length()` : 문자열의 길이를 반환한다.
- `charAt(int index)` : 특정 인덱스의 문자를 반환한다.
- `substring(int beginIndex, int endIndex)` : 문자열의 부분 문자열을 반환한다.
- `indexOf(String str)` : 특정 문자열이 시작되는 인덱스를 반환한다.
- `toLowerCase()` , `toUpperCase()` : 문자열을 소문자 또는 대문자로 변환한다.
- `trim()` : 문자열 양 끝의 공백을 제거한다.
- `concat(String str)` : 문자열을 더한다.

String 클래스와 참조형

`String`은 클래스이다. 따라서 기본형이 아니라 참조형이다.
참조형은 변수에 계산할 수 있는 값이 들어있는 것이 아니라 `x001`과 같이 계산할 수 없는 참조값이 들어있다. 따라서 원칙적으로 `+` 같은 연산을 사용할 수 없다.

```
package lang.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);
    }
}
```

- 자바에서 문자열을 더할 때는 `String`이 제공하는 `concat()`과 같은 메서드를 사용해야 한다.
- 하지만 문자열은 너무 자주 다루어지기 때문에 자바 언어에서 편의상 특별히 `+` 연산을 제공한다.

실행 결과

```
result1 = hello java
result2 = hello java
```

String 클래스 - 비교

`String` 클래스 비교할 때는 `==` 비교가 아니라 항상 `equals()` 비교를 해야한다.

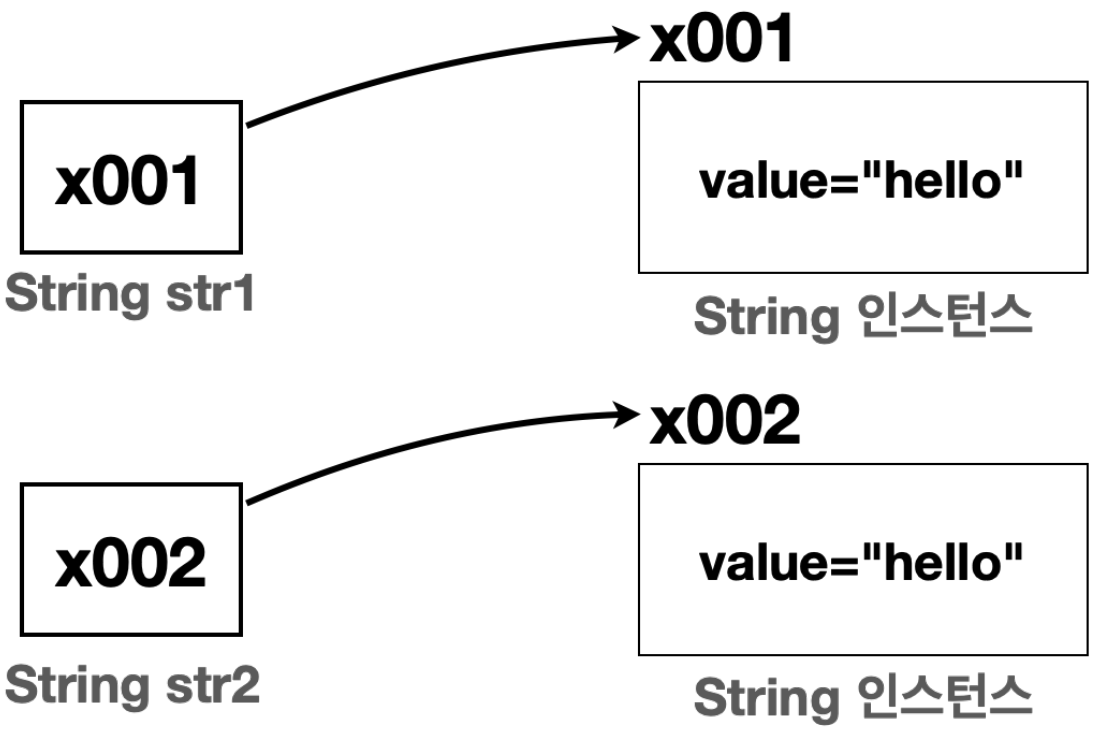
- **동일성(Identity)**: `==` 연산자를 사용해서 두 객체의 참조가 동일한 객체를 가리키고 있는지 확인
- **동등성(Equality)**: `equals()` 메서드를 사용하여 두 객체가 논리적으로 같은지 확인

```
package lang.string.equals;

public class StringEqualsMain1 {

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

        String str3 = "hello";
        String str4 = "hello";
        System.out.println("리터럴 == 비교: " + (str3 == str4));
        System.out.println("리터럴 equals 비교: " + (str3.equals(str4)));
    }
}
```

실행 결과

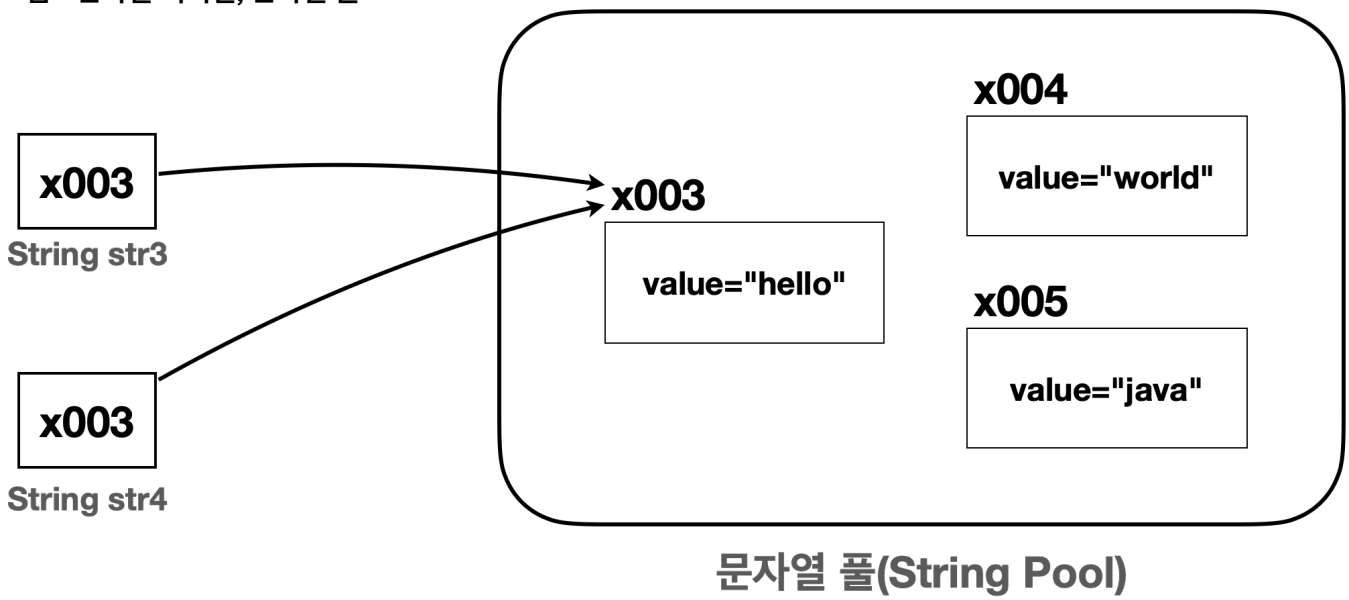
```
new String() == 비교: false
new String() equals 비교: true
리터럴 == 비교: true
리터럴 equals 비교: true
```

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

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.

> **참고**: 풀(Pool)은 자원이 모여있는 곳을 의미한다. 프로그래밍에서 풀(Pool)은 공용 자원을 모아둔 곳을 뜻한다. 여러 곳에서 함께 사용할 수 있는 객체를 필요할 때 마다 생성하고, 제거하는 것은 비효율적이다. 대신에 이렇게 문자열 풀에 필요한 `String` 인스턴스를 미리 만들어두고 여러곳에서 재사용할 수 있다면 성능과 메모리를 더 최적화 할 수 있다.
> 참고로 문자열 풀은 힙 영역을 사용한다. 그리고 문자열 풀에서 문자를 찾을 때는 해시 알고리즘을 사용하기 때문에 매우 빠른 속도로 원하는 `String` 인스턴스를 찾을 수 있다. 해시 알고리즘은 뒤에서 설명한다.

그렇다면 문자열 리터럴을 사용하면 `==` 비교를 하고, `new String()`을 직접 사용하는 경우에만 `equals()` 비교를 사용하면 되지 않을까? 다음 코드를 보자.

```
package lang.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, str2));

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

    private static boolean isSame(String x, String y) {
        return x == y;
        //return x.equals(y);
    }
```

```
}
```

실행 결과

```
메서드 호출 비교1: false
메서드 호출 비교2: true
```

`main()` 메서드를 만드는 개발자와 `isSame()` 메서드를 만드는 개발자가 서로 다르다고 가정해보자.
`isSame()`의 경우 매개변수로 넘어오는 `String` 인스턴스가 `new String()`으로 만들어진 것인지, 문자열 리터럴로 만들어 진 것인지 확인할 수 있는 방법이 없다.
따라서 문자열 비교는 항상 `equals()`를 사용해서 동등성 비교를 해야 한다.

String 클래스 - 불변 객체

`String`은 불변 객체이다. 따라서 생성 이후에 절대로 내부의 문자열 값을 변경할 수 없다.

다음 예를 보자.

```
package lang.string.immutable;

public class StringImmutable1 {

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

- `String.concat()` 메서드를 사용하면 기존 문자열에 새로운 문자열을 연결해서 합칠 수 있다.
- 이 경우 어떤 실행 결과가 나올까? 불변 객체에서 학습한 내용을 떠올려보자!

실행 결과

```
```

```
str = hello
```

실행 결과를 보면 뭔가 이상하다. 문자가 전혀 합쳐지지 않았다.

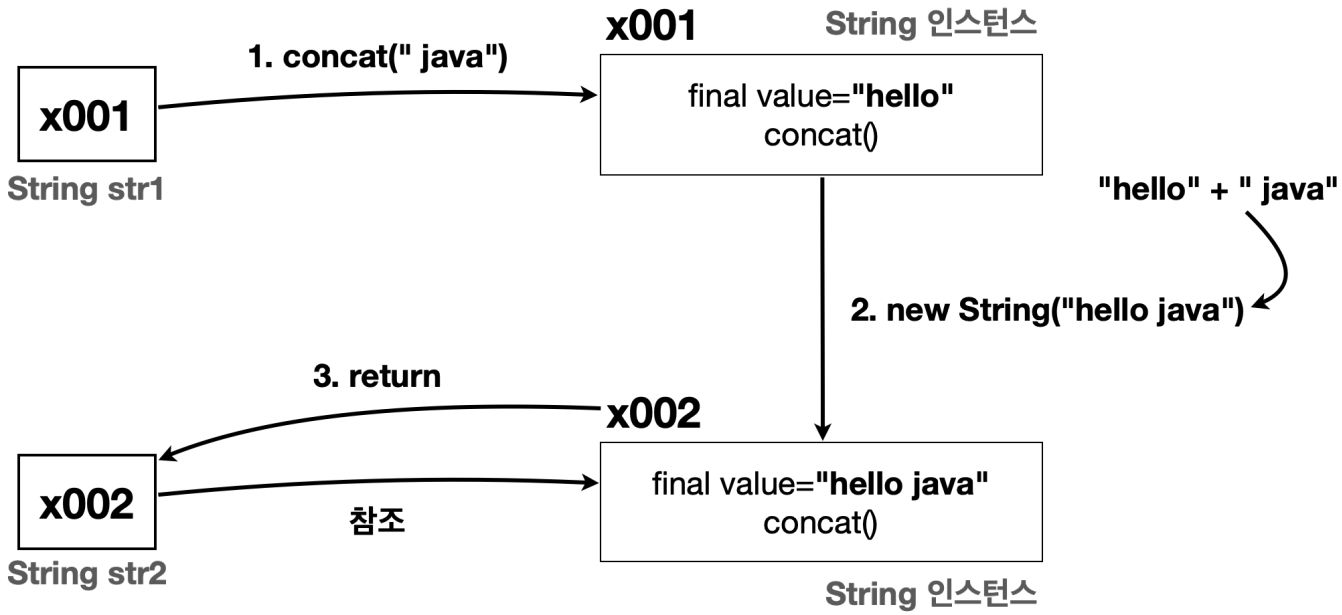
다음 코드를 보자.

```
package lang.string.immutable;

public class StringImmutable2 {

    public static void main(String[] args) {
        String str1 = "hello";
        String str2 = str1.concat(" java");
        System.out.println("str1 = " + str1);
        System.out.println("str2 = " + str2);
    }
}
```

- `String`은 불변 객체이다. 따라서 변경이 필요한 경우 기존 값을 변경하지 않고, 대신에 새로운 결과를 만들어서 반환한다.

실행 결과

```
str1 = hello
str2 = hello java
```

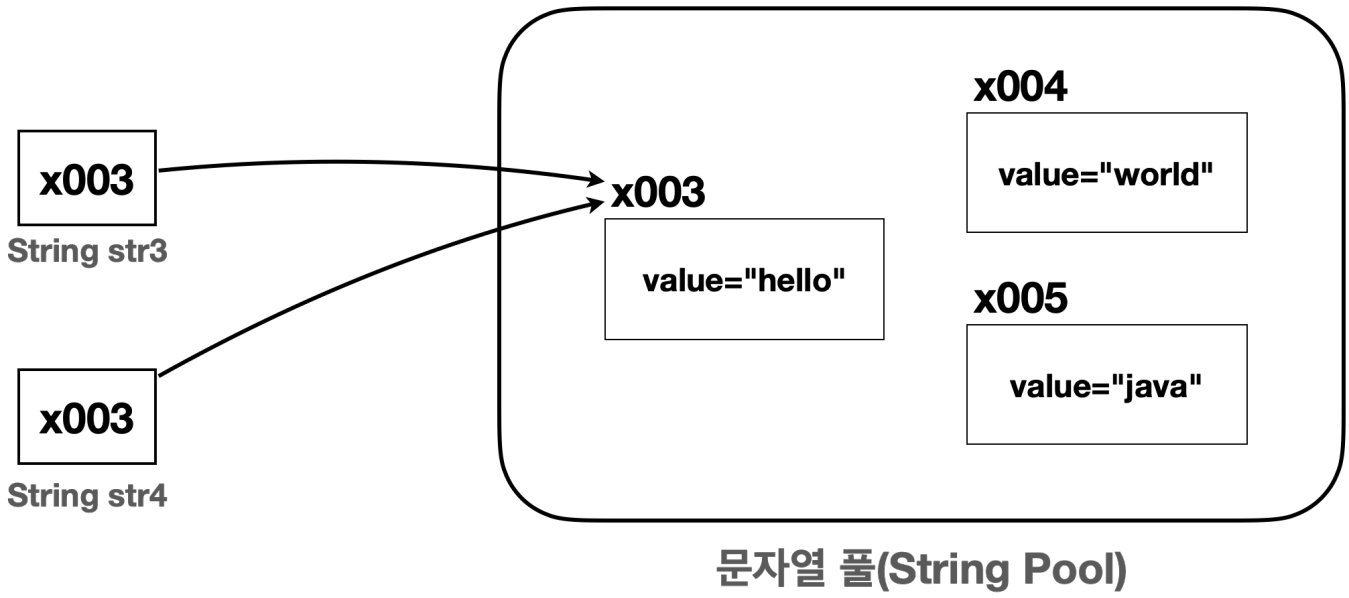


- `String.concat()`은 내부에서 새로운 `String` 객체를 만들어서 반환한다.
- 따라서 불변과 기존 객체의 값을 유지한다.

String이 불변으로 설계된 이유

`String`이 불변으로 설계된 이유는 앞서 불변 객체에서 배운 내용에 추가로 다음과 같은 이유도 있다.

문자열 풀에 있는 `String` 인스턴스의 값이 중간에 변경되면 같은 문자열을 참고하는 다른 변수의 값도 함께 변경된다.
예를 들어보자.



- `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.

String 클래스 - 주요 메서드1

주요 메서드 목록

`String` 클래스는 문자열을 편리하게 다루기 위한 다양한 메서드를 제공한다. 여기서는 자주 사용하는 기능 위주로 나열했다.
참고로 기능이 너무 많기 때문에 메서드를 외우기 보다는 주로 사용하는 메서드가 이런 것이 있구나 대략 알아두고, 필요할 때 검색하거나 API 문서를 통해서 원하는 기능을 찾는 것이 좋다.

문자열 정보 조회

- `length()` : 문자열의 길이를 반환한다.
- `isEmpty()` : 문자열이 비어 있는지 확인한다. (길이가 0)
- `isBlank()` : 문자열이 비어 있는지 확인한다. (길이가 0이거나 공백(Whitespace)만 있는 경우), 자바 11
- `charAt(int index)` : 지정된 인덱스에 있는 문자를 반환한다.

문자열 비교

- `equals(Object anObject)` : 두 문자열이 동일한지 비교한다.
- `equalsIgnoreCase(String anotherString)` : 두 문자열을 대소문자 구분 없이 비교한다.
- `compareTo(String anotherString)` : 두 문자열을 사전 순으로 비교한다.
- `compareToIgnoreCase(String str)` : 두 문자열을 대소문자 구분 없이 사전적으로 비교한다.
- `startsWith(String prefix)` : 문자열이 특정 접두사로 시작하는지 확인한다.
- `endsWith(String suffix)` : 문자열이 특정 접미사로 끝나는지 확인한다.

문자열 검색

- `contains(CharSequence s)` : 문자열이 특정 문자열을 포함하고 있는지 확인한다.
- `indexOf(String ch)` / `indexOf(String ch, int fromIndex)` : 문자열이 처음 등장하는 위치를 반환한다.
- `lastIndexOf(String ch)` : 문자열이 마지막으로 등장하는 위치를 반환한다.

문자열 조작 및 변환

- `substring(int beginIndex)` / `substring(int beginIndex, int endIndex)` : 문자열의 부분 문자열을 반환한다.
- `concat(String str)` : 문자열의 끝에 다른 문자열을 붙인다.
- `replace(CharSequence target, CharSequence replacement)` : 특정 문자열을 새 문자열로 대체한다.
- `replaceAll(String regex, String replacement)` : 문자열에서 정규 표현식과 일치하는 부분을 새 문자열로 대체한다.
- `replaceFirst(String regex, String replacement)` : 문자열에서 정규 표현식과 일치하는 첫 번째 부분을 새 문자열로 대체한다.
- `toLowerCase()` / `toUpperCase()` : 문자열을 소문자나 대문자로 변환한다.
- `trim()` : 문자열 양쪽 끝의 공백을 제거한다. 단순 `Whitespace`만 제거할 수 있다.
- `strip()` : `Whitespace`와 유니코드 공백을 포함해서 제거한다. 자바 11

문자열 분할 및 조합

- `split(String regex)` : 문자열을 정규 표현식을 기준으로 분할한다.
- `join(CharSequence delimiter, CharSequence... elements)` : 주어진 구분자로 여러 문자열을 결합한다.

기타 유틸리티

- `valueOf(Object obj)` : 다양한 타입을 문자열로 변환한다.
- `toCharArray()` : 문자열을 문자 배열로 변환한다.
- `format(String format, Object... args)` : 형식 문자열과 인자를 사용하여 새로운 문자열을 생성한다.
- `matches(String regex)` : 문자열이 주어진 정규 표현식과 일치하는지 확인한다.

이제 본격적으로 하나씩 알아보자.

> **참고**: `CharSequence`는 `String`, `StringBuilder`의 상위 타입이다. 문자열을 처리하는 다양한 객체를 받을 수 있다. `StringBuilder`는 뒤에서 설명한다.

문자열 정보 조회

- `length()` : 문자열의 길이를 반환한다.
- `isEmpty()` : 문자열이 비어 있는지 확인한다. (길이가 0)

- `isBlank()` : 문자열이 비어 있는지 확인한다. (길이가 0이거나 공백(Whitespace)만 있는 경우), 자바 11
- `charAt(int index)` : 지정된 인덱스에 있는 문자를 반환한다.

```
package lang.string.method;

public class StringInfoMain {
    public static void main(String[] args) {
        String str = "Hello, Java!";
        System.out.println("문자열의 길이: " + str.length());
        System.out.println("문자열이 비어 있는지: " + str.isEmpty());
        System.out.println("문자열이 비어 있거나 공백인지1: " + str.isBlank()); //Java
11
        System.out.println("문자열이 비어 있거나 공백인지2: " + "    ".isBlank());

        char c = str.charAt(7);
        System.out.println("7번 인덱스의 문자: " + c);
    }
}
```

실행 결과

```
문자열의 길이: 12
문자열이 비어 있는지: false
문자열이 공백인지: false
문자열이 비어 있거나 공백인지: true
7번 인덱스의 문자: J
```

문자열 비교

- `equals(Object anObject)` : 두 문자열이 동일한지 비교한다.
- `equalsIgnoreCase(String anotherString)` : 두 문자열을 대소문자 구분 없이 비교한다.
- `compareTo(String anotherString)` : 두 문자열을 사전 순으로 비교한다.
- `compareToIgnoreCase(String str)` : 두 문자열을 대소문자 구분 없이 사전적으로 비교한다.
- `startsWith(String prefix)` : 문자열이 특정 접두사로 시작하는지 확인한다.
- `endsWith(String suffix)` : 문자열이 특정 접미사로 끝나는지 확인한다.

```java
package lang.string.method;

public class StringComparisonMain {
    public static void main(String[] args) {
        String str1 = "Hello, Java!"; //대문자 일부 있음
        String str2 = "hello, java!"; //대문자 없음 모두 소문자
        String str3 = "Hello, World!";

        System.out.println("str1 equals str2: " + str1.equals(str2));
        System.out.println("str1 equalsIgnoreCase str2: " +
str1.equalsIgnoreCase(str2));

        System.out.println("'b' compareTo 'a': " + "b".compareTo("a"));
        System.out.println("str1 compareTo str3: " + str1.compareTo(str3));
        System.out.println("str1 compareToIgnoreCase str2: " +
str1.compareToIgnoreCase(str2));

        System.out.println("str1 starts with 'Hello': " +
str1.startsWith("Hello"));
        System.out.println("str1 ends with 'Java!': " +
str1.endsWith("Java!"));
    }
}
```

실행 결과

```
str1 equals str2: false
str1 equalsIgnoreCase str2: true
'b' compareTo 'a': 1
str1 compareTo str3: -13
str1 compareToIgnoreCase str2: 0
str1 starts with 'Hello': true
str1 ends with 'Java!': true
```

문자열 검색

- `contains(CharSequence s)` : 문자열이 특정 문자열을 포함하고 있는지 확인한다.
- `indexOf(String ch)` / `indexOf(String ch, int fromIndex)` : 문자열이 처음 등장하는 위치를 반환한다.

- `lastIndexOf(String ch)` : 문자열이 마지막으로 등장하는 위치를 반환한다.

```
package lang.string.method;

public class StringSearchMain {
    public static void main(String[] args) {
        String str = "Hello, Java! Welcome to Java world.";

        System.out.println("문자열에 'Java'가 포함되어 있는지: " +
str.contains("Java"));
        System.out.println("'Java'의 첫 번째 인덱스: " + str.indexOf("Java"));
        System.out.println("인덱스 10부터 'Java'의 인덱스: " + str.indexOf("Java",
10));
        System.out.println("'Java'의 마지막 인덱스: " + str.lastIndexOf("Java"));
    }
}
```

실행 결과

```
문자열에 'Java'가 포함되어 있는지: true
'Java'의 첫 번째 인덱스: 7
인덱스 10부터 'Java'의 인덱스: 24
'Java'의 마지막 인덱스: 24
```

String 클래스 - 주요 메서드2

문자열 조작 및 변환

- `substring(int beginIndex)` / `substring(int beginIndex, int endIndex)` : 문자열의 부분 문자열을 반환한다.
- `concat(String str)` : 문자열의 끝에 다른 문자열을 붙인다.
- `replace(CharSequence target, CharSequence replacement)` : 특정 문자열을 새 문자열로 대체한다.
- `replaceAll(String regex, String replacement)` : 문자열에서 정규 표현식과 일치하는 부분을 새

문자열로 대체한다.

- `replaceFirst(String regex, String replacement)` : 문자열에서 정규 표현식과 일치하는 첫 번째 부분을 새 문자열로 대체한다.
- `toLowerCase()` / `toUpperCase()` : 문자열을 소문자나 대문자로 변환한다.
- `trim()` : 문자열 양쪽 끝의 공백을 제거한다. 단순 `Whitespace` 만 제거할 수 있다.
- `strip()` : `Whitespace` 와 유니코드 공백을 포함해서 제거한다. 자바 11

```
package lang.string.method;

public class StringChangeMain1 {
    public static void main(String[] args) {
        String str = "Hello, Java! Welcome to Java";

        System.out.println("인덱스 7부터의 부분 문자열: " + str.substring(7));
        System.out.println("인덱스 7부터 12까지의 부분 문자열: " + str.substring(7,
12));

        System.out.println("문자열 결합: " + str.concat("!!!"));

        System.out.println("'Java'를 'World'로 대체: " + str.replace("Java",
"World"));
        System.out.println("첫 번째 'Java'를 'World'으로 대체: " +
str.replaceFirst("Java", "World"));
    }
}
```

실행 결과

```
인덱스 7부터의 부분 문자열: Java! Welcome to Java
인덱스 7부터 12까지의 부분 문자열: Java!
문자열 결합: Hello, Java! Welcome to Java!!!
'Java'를 'World'로 대체: Hello, World! Welcome to World
첫 번째 'Java'를 'World'으로 대체: Hello, World! Welcome to Java
```

```
package lang.string.method;
```

```
public class StringChangeMain2 {
    public static void main(String[] args) {
        String strWithSpaces = "   Java Programming ";

        System.out.println("소문자로 변환: " + strWithSpaces.toLowerCase());
        System.out.println("대문자로 변환: " + strWithSpaces.toUpperCase());

        System.out.println("공백 제거(trim): '" + strWithSpaces.trim() + "'");
        System.out.println("공백 제거(strip): '" + strWithSpaces.strip() + "'");
        System.out.println("앞 공백 제거(strip): '" +
strWithSpaces.stripLeading() + "'");
        System.out.println("뒤 공백 제거(strip): '" +
strWithSpaces.stripTrailing() + "'");
    }
}
```

실행 결과

```
소문자로 변환:    java programming
대문자로 변환:    JAVA PROGRAMMING
공백 제거(trim): 'Java Programming'
공백 제거(strip): 'Java Programming'
앞 공백 제거(strip): 'Java Programming '
뒤 공백 제거(strip): '   Java Programming'
```

문자열 분할 및 조합

- `split(String regex)` : 문자열을 정규 표현식을 기준으로 분할한다.
- `join(CharSequence delimiter, CharSequence... elements)` : 주어진 구분자로 여러 문자열을 결합한다.

```
package lang.string.method;

public class StringSplitJoinMain {
    public static void main(String[] args) {
        String str = "Apple,Banana,Orange";

        // split()
```

```
        String[] splitStr = str.split(",");
        for(String s : splitStr) {
            System.out.println(s);
        }

        // join()
        String joinedStr = String.join("-", "A", "B", "C");
        System.out.println("연결된 문자열: " + joinedStr);

        // 문자열 배열 연결
        String result = String.join("-", splitStr);
        System.out.println("result = " + result);
    }
}
```

실행 결과

```
Apple
Banana
Orange
연결된 문자열: A-B-C
result = Apple-Banana-Orange
```

기타 유틸리티

- `valueOf(Object obj)` : 다양한 타입을 문자열로 변환한다.
- `toCharArray()` : 문자열을 문자 배열로 변환한다.
- `format(String format, Object... args)` : 형식 문자열과 인자를 사용하여 새로운 문자열을 생성한다.
- `matches(String regex)` : 문자열이 주어진 정규 표현식과 일치하는지 확인한다.

```
package lang.string.method;

public class StringUtilsMain1 {
    public static void main(String[] args) {
        int num = 100;
        boolean bool = true;
```

```
        Object obj = new Object();
        String str = "Hello, Java!";

        // valueOf 메서드
        String numString = String.valueOf(num);
        System.out.println("숫자의 문자열 값: " + numString);
        String boolString = String.valueOf(bool);
        System.out.println("불리언의 문자열 값: " + boolString);
        String objString = String.valueOf(obj);
        System.out.println("객체의 문자열 값: " + objString);
        //다음과 같이 간단히 변환할 수 있음 (문자 + x -> 문자x)
        String numString2 = "" + num;
        System.out.println("빈문자열 + num:" + numString2);

        // toCharArray 메서드
        char[] strCharArray = str.toCharArray();
        System.out.println("문자열을 문자 배열로 변환: " + strCharArray);
        for (char c : strCharArray) {
            System.out.print(c);
        }
        System.out.println();
    }
}
```

실행 결과

```
숫자의 문자열 값: 100
불리언의 문자열 값: true
객체의 문자열 값: java.lang.Object@a09ee92
빈문자열 + num:100
문자열을 문자 배열로 변환: [C@30f39991
Hello, Java!
```

```
package lang.string.method;

public class StringUtilsMain2 {
    public static void main(String[] args) {
        int num = 100;
        boolean bool = true;
```

```
        String str = "Hello, Java!";

        // format 메서드
        String format1 = String.format("num: %d, bool: %b, str: %s", num,
bool, str);
        System.out.println(format1);

        String format2 = String.format("숫자: %.2f", 10.1234);
        System.out.println(format2);

        // printf
        System.out.printf("숫자: %.2f\n", 10.1234);

        // matches 메서드
        String regex = "Hello, (Java!|World!)";
        System.out.println("'str'이 패턴과 일치하는가? " + str.matches(regex));
    }
}
```

실행 결과

```
num: 100, bool: true, str: Hello, Java!
숫자: 10.12
숫자: 10.12
'str'이 패턴과 일치하는가? true
```

format 메서드에서 `%d`는 숫자, `%b`는 `boolean`, `%s`는 문자열을 뜻한다.

StringBuilder - 가변 String

불변인 `String` 클래스의 단점

불변인 `String` 클래스에도 단점이 있다. 다음 예를 보자. 참고로 실제로 작동하는 코드는 아니다.

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, 메모리 자원을 더 많이 사용하게 된다. 그리고 문자열의 크기가 클수록, 문자열을 더 자주 변경할수록 시스템의 자원을 더 많이 소모한다.

> **참고**: 실제로는 문자열을 다룰 때 자바가 내부에서 최적화를 적용하는데, 이 부분은 뒤에서 다룬다.

StringBuilder

이 문제를 해결하는 방법은 단순하다. 바로 불변이 아닌 가변 `String`이 존재하면 된다. 가변은 내부의 값을 바로 변경하면 되기 때문에 새로운 객체를 생성할 필요가 없다. 따라서 성능과 메모리 사용면에서 불변보다 더 효율적이다.

이런 문제를 해결하기 위해 자바는 `StringBuilder`라는 가변 `String`을 제공한다. 물론 가변의 경우 사이드 이펙트에 주의해서 사용해야 한다.

`StringBuilder`는 내부에 `final`이 아닌 변경할 수 있는 `byte[]`을 가지고 있다.

```
public final class StringBuilder {
    char[] value;// 자바 9 이전
    byte[] value;// 자바 9 이후

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

(실제로는 상속 관계에 있고 부모 클래스인 `AbstractStringBuilder`에 `value` 속성과 `length()` 메서드가 있다.)

StringBuilder 사용 예

실제 `StringBuilder`를 어떻게 사용하는지 확인해보자.

```
package lang.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 = " + sb);

        sb.insert(4, "Java");
        System.out.println("insert = " + sb);

        sb.delete(4, 8);
        System.out.println("delete = " + sb);

        sb.reverse();
        System.out.println("reverse = " + sb);

        //StringBuilder -> String
```

```
        String string = sb.toString();
        System.out.println("string = " + string);
    }
}
```

- `StringBuilder` 객체를 생성한다.
- `append()` 메서드를 사용해 여러 문자열을 추가한다.
- `insert()` 메서드로 특정 위치에 문자열을 삽입한다.
- `delete()` 메서드로 특정 범위의 문자열을 삭제한다.
- `reverse()` 메서드로 문자열을 뒤집는다.
- 마지막으로 `toString` 메소드를 사용해 `StringBuilder`의 결과를 기반으로 `String`을 생성해서 반환한다.

실행 결과

```
sb = ABCD
insert = ABCDJava
delete = ABCD
reverse = DCBA
string = DCBA
```

가변(Mutable) vs 불변(Immutable):

- `String`은 불변하다. 즉, 한 번 생성되면 그 내용을 변경할 수 없다. 따라서 문자열에 변화를 주려고 할 때마다 새로운 `String` 객체가 생성되고, 기존 객체는 버려진다. 이 과정에서 메모리와 처리 시간을 더 많이 소모한다.
- 반면에, `StringBuilder`는 가변적이다. 하나의 `StringBuilder` 객체 안에서 문자열을 추가, 삭제, 수정할 수 있으며, 이때마다 새로운 객체를 생성하지 않는다. 이로 인해 메모리 사용을 줄이고 성능을 향상시킬 수 있다. 단 사이드 이펙트를 주의해야 한다.

`StringBuilder`는 보통 문자열을 변경하는 동안만 사용하다가 문자열 변경이 끝나면 안전한(불변) `String`으로 변환하는 것이 좋다.

String 최적화

자바의 String 최적화

자바 컴파일러는 다음과 같이 문자열 리터럴을 더하는 부분을 자동으로 합쳐준다.

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

다음과 같이 문자열을 루프안에서 문자열을 더하는 경우에는 최적화가 이루어지지 않는다.

```
package lang.string.builder;

public class LoopStringMain {
    public static void main(String[] args) {
```

```
        long startTime = System.currentTimeMillis();

        String result = "";
        for (int i = 0; i < 100000; i++) {
            result += "Hello Java ";
        }
        long endTime = System.currentTimeMillis();

        System.out.println("result = " + result);
        System.out.println("time = " + (endTime - startTime) + "ms");
    }
}
```

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.

실행 결과

```
result = Hello Java Hello Java ....
time = 2490ms
```

- 1000ms = 1초
- M2 맥북을 기준으로 `100000`회 더했을 때 약 2.5초가 걸렸다.

이럴 때는 직접 `StringBuilder`를 사용하면 된다.

```
package lang.string.builder;
```

```java
public class LoopStringBuilderMain {
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();

        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < 100000; i++) {
            sb.append("Hello Java ");
        }
        String result = sb.toString();
        long endTime = System.currentTimeMillis();

        System.out.println("result = " + result);
        System.out.println("time = " + (endTime - startTime) + "ms");
    }
}
```

실행 결과

```
result = Hello Java Hello Java ....
time = 3ms
```

- 1000ms = 1초
- M2 맥북을 기준으로 `100000`회 더했을 때 약 0.003초가 걸렸다.

정리

문자열을 합칠 때 대부분의 경우 최적화가 되므로 `+` 연산을 사용하면 된다.

StringBuilder를 직접 사용하는 것이 더 좋은 경우

- 반복문에서 반복해서 문자를 연결할 때
- 조건문을 통해 동적으로 문자열을 조합할 때
- 복잡한 문자열의 특정 부분을 변경해야 할 때
- 매우 긴 대용량 문자열을 다룰 때

> **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 쓰레드 상황에 안전하지 않지만 동기화 오버헤드가 없으므로 속도가 빠르다.
> `StringBuffer`와 동기화에 관한 내용은 이후에 멀티 스레드를 학습해야 이해할 수 있다. 지금은 이런 것이 있

구나 정도만 알아두면 된다.

메서드 체인닝 - Method Chaining

간단한 예제 코드로 메서드 체이닝(Method Chaining)에 대해 알아보자.

```
package lang.string.chaining;

public class ValueAdder {

    private int value;

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

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

- 단순히 값을 누적해서 더하는 기능을 제공하는 클래스다.
- `add()` 메서드를 호출할 때 마다 내부의 `value`에 값을 누적한다.
- `add()` 메서드를 보면 자기 자신(`this`)의 참조값을 반환한다. 이 부분을 유의해서 보자.

```
package lang.string.chaining;

public class MethodChainingMain1 {

    public static void main(String[] args) {
        ValueAdder adder = new ValueAdder();
        adder.add(1);
        adder.add(2);
        adder.add(3);
        int result = adder.getValue();
```

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

실행 결과

```
result = 6
```

- `add()` 메서드를 여러번 호출해서 값을 누적해서 더하고 출력한다.
- 여기서는 `add()` 메서드의 반환값은 사용하지 않았다.

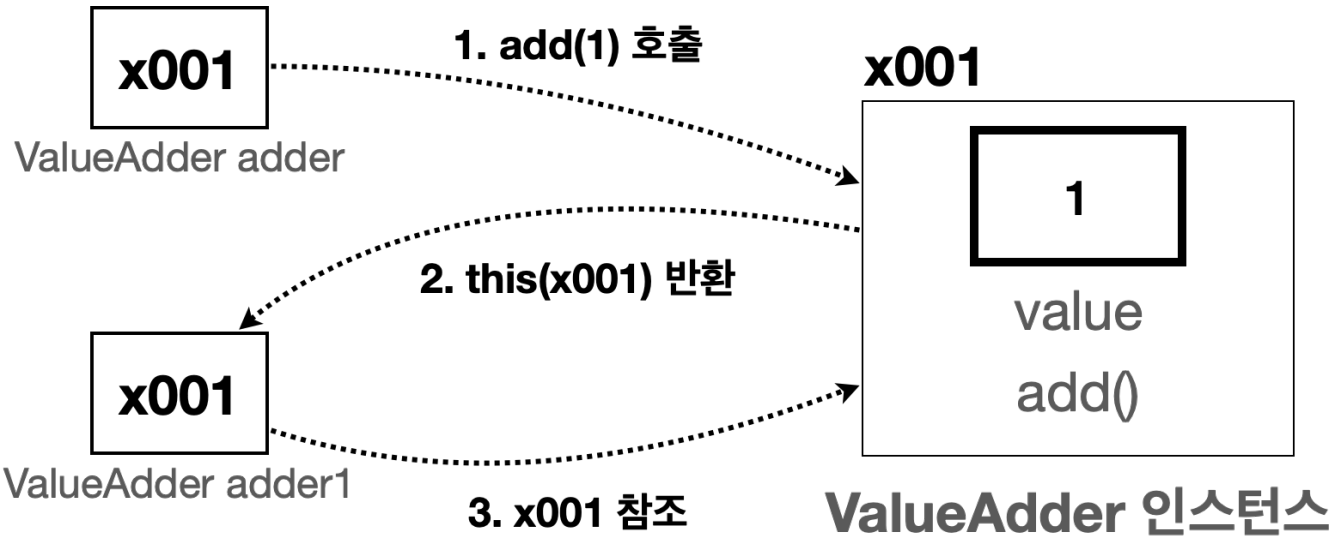
이번에는 `add()` 메서드의 반환값을 사용해보자.

```
package lang.string.chaining;

public class MethodChainingMain2 {

    public static void main(String[] args) {
        ValueAdder adder = new ValueAdder();
        ValueAdder adder1 = adder.add(1);
        ValueAdder adder2 = adder1.add(2);
        ValueAdder adder3 = adder2.add(3);
        int result = adder3.getValue();
        System.out.println("result = " + result);
    }
}
```

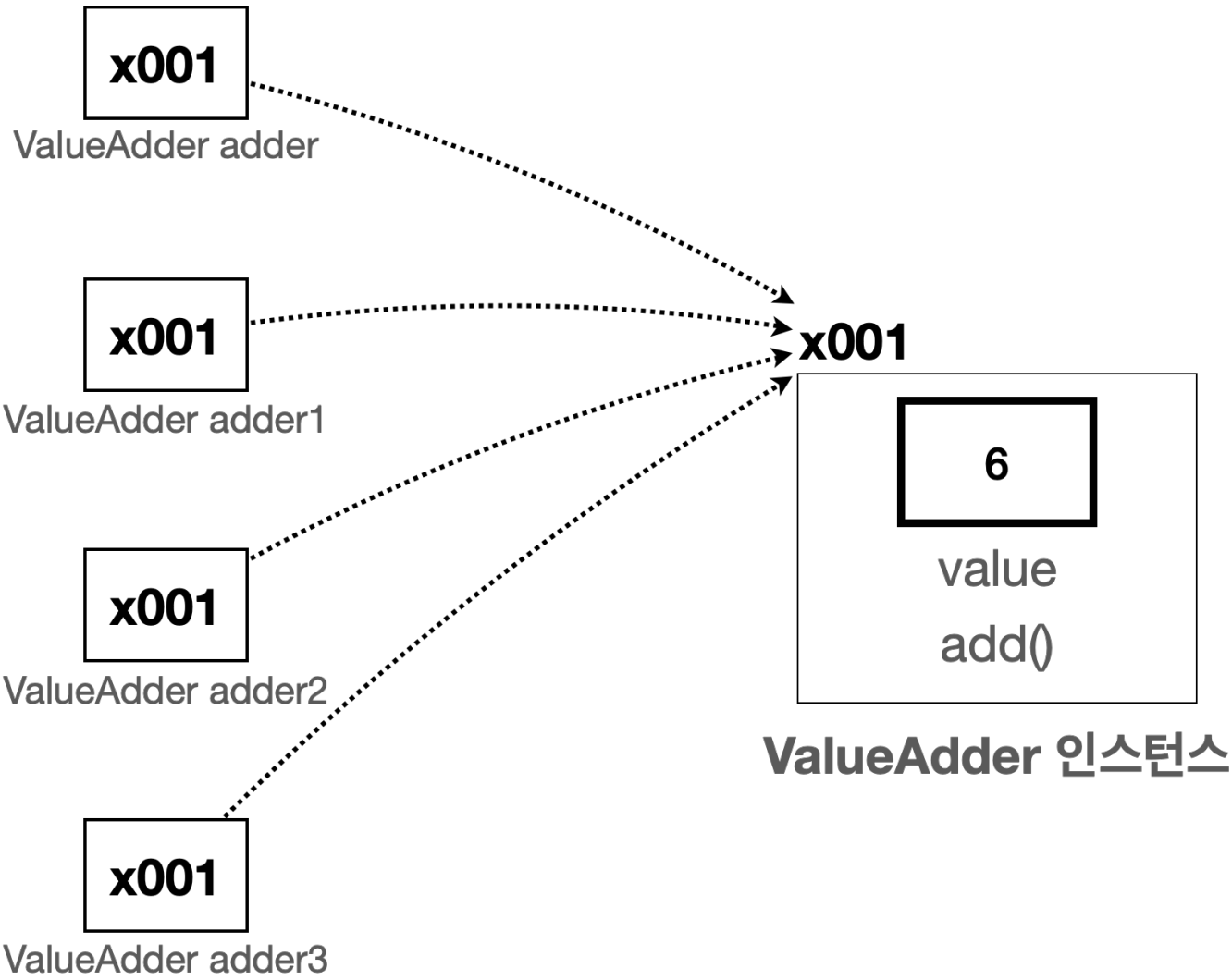
실행 결과

```
result = 6
```

실행 결과는 기존과 같다.



1. `adder.add(1)` 을 호출한다.
2. `add()` 메서드는 결과를 누적하고 자기 자신의 참조값인 `this`(`x001`)를 반환한다.
3. `adder1` 변수는 `adder`와 같은 `x001` 인스턴스를 참조한다.



- `add()` 메서드는 자기 자신(`this`)의 참조값을 반환한다. 이 반환값을 `adder1`, `adder2`, `adder3`에 보관

했다.

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

그런데 이 방식은 처음 방식보다 더 불편하고, 코드도 잘 읽히지 않는다.
이런 방식을 왜 사용하는 것일까?

이번에는 방금 사용했던 방식에서 반환된 참조값을 새로운 변수에 담아서 보관하지 않고, 대신에 바로 메서드 호출에 사용해보자.

```
package lang.string.chaining;

public class MethodChainingMain3 {

    public static void main(String[] args) {
        ValueAdder adder = new ValueAdder();
        int result = adder.add(1).add(2).add(3).getValue();
        System.out.println("result = " + result);
    }
}
```

실행 결과

```
result = 6
```

실행 순서

`add()` 메서드를 호출하면 `ValueAdder` 인스턴스 자신의 참조값(`x001`)이 반환된다. 이 반환된 참조값을 변수에 담아두지 않아도 된다. 대신에 반환된 참조값을 즉시 사용해서 바로 메서드를 호출할 수 있다.

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
```

메서드 호출의 결과로 자기 자신의 참조값을 반환하면, 반환된 참조값을 사용해서 메서드 호출을 계속 이어갈 수 있다.
코드를 보면 `.`을 찍고 메서드를 계속 연결해서 사용한다. 마치 메서드가 체인으로 연결된 것 처럼 보인다. 이러한 기법을 메서드 체이닝이라 한다.
물론 실행 결과도 기존과 동일하다.

기존에는 메서드를 호출할 때 마다 계속 변수명에 `.`을 찍어야 했다. 예) `adder.add(1)`,`adder.add(2)`
메서드 체이닝 방식은 메서드가 끝나는 시점에 바로 `.`을 찍어서 변수명을 생략할 수 있다.
메서드 체이닝이 가능한 이유는 자기 자신의 참조값을 반환하기 때문이다. 이 참조값에 `.`을 찍어서 바로 자신의 메서드를 호출할 수 있다.

메서드 체이닝 기법은 코드를 간결하고 읽기 쉽게 만들어준다.

StringBuilder와 메서드 체인(Chain)

`StringBuilder`는 메서드 체이닝 기법을 제공한다.

`StringBuilder`의 `append()` 메서드를 보면 자기 자신의 참조값을 반환한다.

```
public StringBuilder append(String str) {
    super.append(str);
    return this;
}
```

`StringBuilder`에서 문자열을 변경하는 대부분의 메서드도 메서드 체이닝 기법을 제공하기 위해 자기 자신을 반환한다.
예) `insert()`, `delete()`, `reverse()`

앞서 `StringBuilder`를 사용한 코드는 다음과 같이 개선할 수 있다.

```
package lang.string.builder;

public class StringBuilderMain1_2 {
```

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

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

실행 결과

```
string = DCBA
```

정리

"만드는 사람이 수고로우면 쓰는 사람이 편하고, 만드는 사람이 편하면 쓰는 사람이 수고롭다"는 말이 있다.

메서드 체이닝은 구현하는 입장에서는 번거롭지만 사용하는 개발자는 편리해진다.

참고로 자바의 라이브러리와 오픈 소스들은 메서드 체이닝 방식을 종종 사용한다.

문제와 풀이1

문제1 - startsWith

문제 설명

- `startsWith()`를 사용해서 `url`이 `https://`로 시작하는지 확인해라.

```
package lang.string.test;

public class TestString1 {

    public static void main(String[] args) {
        String url = "https://www.example.com";
```

```
        // 코드 작성
    }
}
```

실행 결과

```
true
```

정답

```
package lang.string.test;

public class TestString1 {

    public static void main(String[] args) {
        String url = "https://www.example.com";
        boolean result = url.startsWith("https://");
        System.out.println(result); // 출력: true
    }
}
```

문제2 - length()

문제 설명

`length()`를 사용해서 `arr` 배열에 들어있는 모든 문자열의 길이 합을 구해라.
실행 결과에 맞도록 출력하자.

```
package lang.string.test;

public class TestString2 {

    public static void main(String[] args) {
        String[] arr = {"hello", "java", "jvm", "spring", "jpa"};
        // 코드 작성
    }
}
```

실행 결과

```
hello:5
java:4
jvm:3
spring:6
jpa:3
sum = 21
```

정답

```
package lang.string.test;

public class TestString2 {

    public static void main(String[] args) {
        String[] arr = {"hello", "java", "jvm", "spring", "jpa"};
        int sum = 0;
        for (String s : arr) {
            System.out.println(s + ":" + s.length());
            sum += s.length();
        }
        System.out.println("sum = " + sum);
    }
}
```

문제3 - indexOf()

문제 설명

`str`에서 `".txt"` 문자열이 언제부터 시작하는지 위치를 찾아서 출력해라. `indexOf()`를 사용해라.

```
package lang.string.test;

public class TestString3 {

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

```
        // 코드 작성
    }
}
```

실행 결과

```
index = 5
```

정답

```
package lang.string.test;

public class TestString3 {

    public static void main(String[] args) {
        String str = "hello.txt";
        int index = str.indexOf(".txt");
        System.out.println("index = " + index);
    }
}
```

문제4 - substring()

문제 설명

`substring()`을 사용해서, `hello` 부분과 `.txt` 부분을 분리해라.
단순하게 `substring()`에 숫자를 직접 입력해서 문제를 풀면 된다.

```
package lang.string.test;

public class TestString4 {

    public static void main(String[] args) {
        String str = "hello.txt";
        // 코드 작성
    }
}
```

실행 결과

```
filename = hello
extName = .txt
```

정답

```
package lang.string.test;

public class TestString4 {

    public static void main(String[] args) {
        String str = "hello.txt";
        String filename = str.substring(0, 5);
        String extName = str.substring(5, 9);
        System.out.println("filename = " + filename);
        System.out.println("extName = " + extName);
    }
}
```

문제5 - indexOf, substring 조합

문제 설명

`str`에는 파일의 이름과 확장자가 주어진다. `ext`에는 파일의 확장자가 주어진다.
파일명과 확장자를 분리해서 출력하라.
`indexOf()`와 `substring()`을 사용해서 문제를 풀면 된다.

```
package lang.string.test;

public class TestString5 {

    public static void main(String[] args) {
        String str = "hello.txt";
        String ext = ".txt";
```

```
        // 코드 작성
    }
}
```

실행 결과

```
filename = hello
extName = .txt
```

정답

```
package lang.string.test;

public class TestString5 {

    public static void main(String[] args) {
        String str = "hello.txt";
        String ext = ".txt";

        int extIndex = str.indexOf(ext);

        String filename = str.substring(0, extIndex);
        String extName = str.substring(extIndex);
        System.out.println("filename = " + filename);
        System.out.println("extName = " + extName);
    }
}
```

문제6 - 검색 count

문제 설명

`str`에서 `key`로 주어지는 문자를 찾고, 찾은 문자의 수를 출력해라.

`indexOf()`를 반복문과 함께 풀면 된다.

```
package lang.string.test;
```

```
public class TestString6 {
    public static void main(String[] args) {
        String str = "start hello java, hello spring, hello jpa";
        String key = "hello";

        // 코드 작성
    }
}
```

실행 결과

```
count = 3
```

정답

```
package lang.string.test;

public class TestString6 {
    public static void main(String[] args) {
        String str = "start hello java, hello spring, hello jpa";
        String key = "hello";

        int count = 0;
        int index = str.indexOf(key);
        while (index >= 0) {
            index = str.indexOf(key, index + 1);
            count++;
        }
        System.out.println("count = " + count);
    }
}
```

문제와 풀이2

문제7 - 공백 제거

문제 설명

문자의 양쪽 공백을 제거해라. 예) `"    Hello Java    "` ➜ `"Hello Java"`

```
package lang.string.test;

public class TestString7 {

    public static void main(String[] args) {
        String original = "    Hello Java    ";
        // 코드 작성
    }
}
```

실행 결과

```
Hello Java
```

정답

```
package lang.string.test;

public class TestString7 {

    public static void main(String[] args) {
        String original = "    Hello Java    ";
        String trimmed = original.trim();
        System.out.println(trimmed);
    }
}
```

문제8 - replace

문제 설명

`replace()`를 사용해서 `java`라는 단어를 `jvm`으로 변경해라.

```
package lang.string.test;

public class TestString8 {

    public static void main(String[] args) {
        String input = "hello java spring jpa java";
        // 코드 작성
    }
}
```

실행 결과

```
hello jvm spring jpa jvm
```

정답

```
package lang.string.test;

public class TestString8 {

    public static void main(String[] args) {
        String input = "hello java spring jpa java";
        String result = input.replace("java", "jvm");
        System.out.println(result);
    }
}
```

문제9 - split()

문제 설명

`split()`를 사용해서 이메일의 ID 부분과 도메인 부분을 분리해라.

```
package lang.string.test;
```

```
public class TestString9 {

    public static void main(String[] args) {
        String email = "hello@example.com";
        // 코드 작성
    }
}
```

실행 결과

```
ID: hello
Domain: example.com
```

정답

```
package lang.string.test;

public class TestString9 {

    public static void main(String[] args) {
        String email = "hello@example.com";

        //@를 기준으로 email의 아이디 부분과 도메인을 분리
        String[] parts = email.split("@");
        String idPart = parts[0];
        String domainPart = parts[1];

        System.out.println("ID: " + idPart);
        System.out.println("Domain: " + domainPart);
    }
}
```

문제10 - split(), join()

문제 설명

`split()`를 사용해서 `fruits`를 분리하고, `join()`을 사용해서 분리한 문자들을 하나로 합쳐라.
실행 결과를 참고해라.

```
package lang.string.test;

public class TestString10 {

    public static void main(String[] args) {
        String fruits = "apple,banana,mango";

        // 코드 작성
    }
}
```

실행 결과

```
apple
banana
mango
joinedString = apple->banana->mango
```

정답

```
package lang.string.test;

public class TestString10 {

    public static void main(String[] args) {
        String fruits = "apple,banana,mango";

        //분리하기
        String[] splitFruits = fruits.split(",");
        for(String fruit : splitFruits) {
            System.out.println(fruit);
        }

        //합치기
        String joinedString = String.join("->", splitFruits);
        System.out.println("joinedString = " + joinedString);
    }
}
```

문제11 - reverse()

문제 설명

`str` 문자열을 반대로 뒤집어라.

`StringBuilder`에 있는 `reverse()`를 사용해라.

```
package lang.string.test;

public class TestString11 {

    public static void main(String[] args) {
        String str = "Hello Java";
        // 코드 작성
    }
}
```

실행 결과

```
avaJ olleH
```

정답

```
package lang.string.test;

public class TestString11 {

    public static void main(String[] args) {
        String str = "Hello Java";
        String reversed = new StringBuilder(str).reverse().toString();
        System.out.println(reversed); // 출력: "avaJ olleH"
    }
}
```

정리