**문자열 클래스**

2017-09-08

이승진

**학습목표**

String 클래스, StringBuilder 클래스, StringBuffer 클래스

**목차**

[1. Java 문자열 2](#_Toc492662671)

[1) Java 문자열의 종류 2](#_Toc492662672)

[2) CharSequence 인터페이스의 메소드 3](#_Toc492662673)

[3) StringBuilder 클래스 메소드 4](#_Toc492662674)

[4) equals 메소드 5](#_Toc492662675)

[2. 예제 코드 6](#_Toc492662676)

[1) StringBuilder1.java 6](#_Toc492662677)

[2) StringBuilder2.java 7](#_Toc492662678)

[3) StringBuilder3.java 8](#_Toc492662679)

[4) 과제 9](#_Toc492662680)

# Java 문자열

## Java 문자열의 종류



### CharSequence interface

Java 언어에서 문자열에 해당하는 클래스는 여러 개다. String, StringBuffer, StringBuilder 등

이 문자열 클래스들의 공통 부모 클래스 역할을 하는 것이 CharSequence inteface 이다.

### String 클래스

대표적인 Java 문자열은 String 클래스이다.

String 객체는 생성된 뒤 문자열 값이 수정될 수 없다.

String 클래스의 문자열 수정 메소드는, this 문자열을 수정하는 것이 아니고, 수정된 새 문자열 객체를 리턴한다. this 문자열은 수정되지 않는다.

### StringBuilder 클래스

수정 가능한 문자열 객체는 StringBuilder 클래스이다.

StringBuilder 클래스의 문자열 수정 메소드는 내부 문자열을 직접 수정한다.

### StringBuffer 클래스

StringBuffer 클래스는 StringBuilder 클래스와 기능이 같고 메소드도 같고 사용법도 같다.

차이점은 StringBuffer는 thread safe 하지만 StringBuilder는 thread safe 하지 않다는 점이다.

thread safe 하다는 말은 멀티 스레드(multi-thread)로 실행되어도 안전하다는 말이다.

StringBuffer 클래스는 thread safe 하기 때문에, StringBuilder 보다 느리다.

따라서 thread safe 해야 할 때에만 StringBuffer를 사용해야 한다.

### 공통 부모 인터페이스의 목적

공통 부모 인터페이스를 구현(implements)한 클래스들은, 인터페이스에 선언된 메소드들을 전부 구현해야 한다.

그 메소드들의 사용법도 서로 같아야 한다.

사용법이 같아서 호환되는 클래스들을 만들기 위해서, 인테페이스 문법을 활용한다.

안드로이드에는, 폰트 정보까지 저장할 수 있는 문자열 클래스가 있다.

이 클래스도 CharSequence 인터페이스를 구현하고 있다.

## CharSequence 인터페이스의 메소드

|  |
| --- |
| char charAt(int index)  this 문자열에서 index 위치의 문자(char)를 리턴한다. |
| int length()  문자열의 길이를 리턴한다. |
| CharSequence subSequence(int start, int end)  this 문자열에서 beginIndex 위치에서 endIndex 직전까지의 문자열을 리턴한다. |
| String toString()  내부 문자열을 String 클래스 객체로 리턴한다. |

CharSequence는 클래스가 아니고 인터페이스(interface) 이다.

인터페이스에는 메소드 선언만 들어있고 메소드 구현은 들어있지 않다.

CharSequence 인터페이스 소스 코드는 다음과 같은 형태일 것이다.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | package java.lang;  public interface CharSequence {  char charAt(int index);  int length();  CharSequence subSequence(int start, int end);  String toString();  } |

인터페이스의 메소드는, public abstract 키워드를 붙여주지 않아도,

기본적으로 전부 public abstract 메소드이다.

인터페이스를 implements한 클래스는 인터페이스의 메소드를 재정의(override) 해야 한다.

인터페이스를 implements한 클래스에서, 재정의되지 않은 인터페이스의 메소드가 하나라도 있다면,

그 클래스는 abstract 클래스이어야 한다. 즉 class 앞에 abstract 키워드가 붙어있어야 한다.

그렇지 않으면 컴파일 에러가 발생한다.

인터페이스를 implements한 클래스가 abstract 클래스가 아니라면, 인터페이스의 메소드를 전부 재정의(override) 해야 한다. 그렇지 않으면 컴파일 에러가 발생한다.

CharSequence 인터페이스를 implements한 String 클래스와 StringBuilder 클래스는 abstract 클래스가 아니다.

따라서 String 클래스와 StringBuilder 클래스는 CharSequence 인터페이스의 모든 메소드를 재정의 하였다.

Java에서 인터페이스의 역할은, 그 인터페이스를 implements한 자식 클래스들에게, 그 인터페이스의 메소드를 재정의하라고 강요하는 것이다.

인터페이스의 메소드들이 자식 클래스들에서 내부 구현만 다르고 사용법은 같게 재정의 되어야 한다.

이것을 강요하는 것이 인터페이스의 역할이다.

내부 구현만 다르고 사용법은 서로 같은 클래스들, 즉 서로 호환되는 클래스들은 사용하기 편하다.

서로 호환되는 클래스들을 만드는 것이 인터페이스를 활용하는 목적이다.

자식 클래스들의 모든 메소드의 사용법이 같은 것은 아니고, 인터페이스에 정의된 메소드의 사용법만 서로 같다.

CharSequence 인터페이스의 메소드들이 String 클래스와 StringBuilder 클래스에도 있고,

이 메소드들의 사용 방법이 서로 같다.

## StringBuilder 클래스 메소드

|  |
| --- |
| StringBuilder append(CharSequence s)  문자열 s를 this 문자열 뒤에 추가한다.  이 메소드의 리턴 값은 this 객체이다. |
| StringBuilder append(int i)  i 값을 문자열로 변환하여 this 문자열 뒤에 추가한다.  이 메소드의 리턴 값은 this 객체이다. |
| StringBuilder append(Double d)  d 값을 문자열로 변환하여 this 문자열 뒤에 추가한다.  이 메소드의 리턴 값은 this 객체이다. |
| char charAt(int index)  this 문자열에서 index 위치의 문자(char)를 리턴한다. |
| StringBuilder delete(int beginIndex, int endIndex)  this 문자열에서 beginIndex 위치에서 endIndex 직전까지의 문자열을 삭제한다.  이 메소드의 리턴 값은 this 객체이다. |
| StringBuilder deleteCharAt(int index)  this 문자열에서 index 위치의 문자를 삭제한다.  이 메소드의 리턴 값은 this 객체이다. |
| int indexOf(String s)  this 문자열에서 문자열 s를 찾아서 그 위치(index)를 리턴한다.  this 문자열에 문자열 s가 들어있지 않다면 -1을 리턴한다. |
| StringBuilder insert(int index, CharSequence s)  this 문자열의 index 위치에 문자열 s를 끼워 넣는다.  이 메소드의 리턴 값은 this 객체이다. |
| StringBuilder insert(int index, Object obj)  obj 객체를 문자열로 변환하여 this 문자열의 index 위치에 끼워 넣는다.  이 메소드의 리턴 값은 this 객체이다. |
| int lastIndexOf(String s)  this 문자열에서 문자열 s를 찾아서 그 위치(index)를 리턴한다.  문자열 s가 여러 번 들어있다면 마지막 위치를 리턴한다.  this 문자열에 문자열 s가 들어있지 않다면 -1을 리턴한다. |
| int length()  문자열의 길이를 리턴한다. |
| StringBuilder replace(int beginIndex, int endIndex, String s)  this 문자열에서 beginIndex 위치에서 endIndex 직전까지의 문자열을  문자열 s로 치환한다.  이 메소드의 리턴 값은 this 객체이다. |
| StringBuilder reverse()  문자열을 거꾸로 뒤집는다.  이 메소드의 리턴 값은 this 객체이다. |
| void setCharAt(int index, char c)  this 문자열에서 index 위치의 문자를 c로 변경한다. |
| CharSequence subSequence(int start, int end)  this 문자열에서 beginIndex 위치에서 endIndex 직전까지의 문자열을 리턴한다.  this 문자열은 수정되지 않는다. |
| String substring(int beginIndex)  this 문자열에서 beginIndex 위치에서 끝까지의 문자열을 리턴한다.  this 문자열은 수정되지 않는다. |
| String substring(int beginIndex, int endIndex)  this 문자열에서 beginIndex 위치에서 endIndex 직전까지의 문자열을 리턴한다.  this 문자열은 수정되지 않는다. |
| String toString()  this 문자열을 String 객체로 변환하여 리턴한다. |

## equals 메소드

동일한 객체인지 비교할 때 사용하는 equals 메소드는 Object 클래스에 정의되어 있다.

많은 클래스들이 이 equals 메소드를 재정의(override) 한다.

String 클래스는 equals 메소드를 재정의(override) 하였다.

String 클래스의 equals 메소드는 문자열 값이 일치하면 true를 리턴한다.

그래서 String 객체 두 개를 equals 메소드로 비교하면, 문자열이 동일한 경우에 true가 리턴된다.

그런데 StringBuilder 클래스는 재정의 하지 않고 Object 클래스의 equals 메소드를 상속 받는다.

Object 클래스의 equals 메소드는 객체 내부 값은 비교하지 않고, 객체가 동일한 객체이어야 true를 리턴한다.

그래서 StringBuilder 객체 두 개를 equals로 비교하면, 문자열이 동일해도 false가 리턴된다.

### 내용이 동일한 두 문자열 객체를 비교할 때

String 클래스의 equals 메소드로 비교하면 true를 리턴한다.

StringBuilder 클래스의 equals 메소드로 비교하면 false를 리턴

== 연산자로 비교하면 false를 리턴한다.

### StringBuilder 클래스에 equals 메소드가 재정의되지 않은 이유는?

equals 메소드를 재정의 하지 않은 클래스는, 값이 동일한지 비교도 할 수 없기 때문에,

여러가지 용도로 사용하기 어렵다.

그렇기 때문에 StringBuilder 클래스를 문자열 보관, 비교, 조회 용으로 사용하기 어렵다.

문자열을 조립해서 생성할 때만 잠깐 StringBuilder 객체를 사용하고,

문자열 조립이 완성되자 마자, toString 메소드를 호출하여 String 객체로 변환해서 사용해야 한다.

문자열을 조립해서 생성할 때만 잠깐 StringBuilder 객체를 사용하라는 의미에서

equals 메소드가 재정의 되지 않았다.

# 예제 코드

## StringBuilder1.java

src/net/skhu/lecture03/StringBuilder1.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | package net.skhu.lecture03;  public class StringBuilder1 {    public static void main(String[] args) {  String[] list = new String[] { "one", "two", "three", "four", "five", "six", "seven" };    String temp = "";  for (String s : list) {  if (temp.length() > 0) temp = temp + ", ";  temp = temp + "\"" + s + "\"";  }  System.out.println(temp);    StringBuilder builder = new StringBuilder();  for (String s : list) {  if (builder.length() > 0) builder.append(", ");  builder.append('"');  builder.append(s);  builder.append('"');  }  System.out.println(builder.toString());  }  } |

위 코드의 실행 결과 출력은 다음과 같다.

"one", "two", "three", "four", "five", "six", "seven"

"one", "two", "three", "four", "five", "six", "seven"

위 코드에서, 줄8~13의 코드와, 줄15~22의 코드는 같은 일을 한다.

위 코드들은 String[] 배열에 들어있는 문자열들을 하나로 합쳐서 출력하는 일을 한다.

줄8~13은 String 클래스를 사용하여 구현되었다.

String 객체는 일단 생성된 후에는 내부 문자열이 변경될 수 없으므로,

줄11의 문자열 더하기 코드는, 문자열들이 결합된 새 문자열 객체를 생성한다.

for 루프를 돌 때마다 매번 줄11에서 새 문자열 객체가 생성되어 temp 변수에 대입된다.

줄15~22는 StringBuilder 클래스를 사용하여 구현되었다.

줄15~22의 코드는 StringBuilder 객체 하나만 사용하여 구현되었다.

줄15에서 StringBuilder 객체 하나를 생성하고,

이 객체의 append 메소드를 호출하여 문자열을 조금씩 덧붙여 나가는 형태로 구현되었다.

마지막으로 줄22에서 StringBuilder 객체의 toString() 메소드를 호출하여 String 객체를 생성한다.

즉 줄8~13의 코드는 for 루프를 돌 때마다 메번 줄11에서 새 문자열 객체가 생성되지만,

줄15~22의 코드는 줄15에서 StringBuilder 객체 하나만 생성하고, 줄22에서 String 객체를 하나만 생성한다.

위와 같이 문자열을 조금씩 만들어 나갈 때는 StringBuilder 객체를 사용해서 구현하는 쪽이 효율적이다.

## StringBuilder2.java

src/net/skhu/lecture03/StringBuilder2.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | package net.skhu.lecture03;  import net.skhu.lecture03.bloodType1.BloodType;  import net.skhu.lecture03.bloodType1.Person;  public class StringBuilder2 {    public static void main(String[] args) {  Person[] list = new Person[] {  new Person("홍길동", 18, BloodType.AB),  new Person("전우치", 24, BloodType.O),  new Person("임꺽정", 22, BloodType.B)  };    StringBuilder builder = new StringBuilder();  builder.append("<table>\n");  builder.append(" <tr><td>이름</td></td><td>나이</td><td>혈액형</td></tr>\n");  for (Person p : list) {  builder.append(" <tr>");  builder.append("<td>");  builder.append(p.getName());  builder.append("</td>");  builder.append("<td>");  builder.append(p.getAge());  builder.append("</td>");  builder.append("<td>");  builder.append(p.getBloodType());  builder.append("</td>");  builder.append("</tr>\n");  }  builder.append("</table>\n");  System.out.println(builder.toString());  }  } |

위 코드는 다음과 같이 출력한다.

|  |
| --- |
| <table>  <tr><td>이름</td></td><td>나이</td><td>혈액형</td></tr>  <tr><td>홍길동</td><td>18</td><td>AB</td></tr>  <tr><td>전우치</td><td>24</td><td>O</td></tr>  <tr><td>임꺽정</td><td>22</td><td>B</td></tr>  </table> |

## StringBuilder3.java

src/net/skhu/lecture03/StringBuilder3.java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | package net.skhu.lecture03;  import net.skhu.lecture03.bloodType1.BloodType;  import net.skhu.lecture03.bloodType1.Person;  public class StringBuilder3 {    public static void main(String[] args) {  Person[] list = new Person[] {  new Person("홍길동", 18, BloodType.AB),  new Person("전우치", 24, BloodType.O),  new Person("임꺽정", 22, BloodType.B)  };    StringBuilder builder = new StringBuilder();  builder.append("<table>\n");  builder.append(" <tr><td>이름</td></td><td>나이</td><td>혈액형</td></tr>\n");  for (Person p : list) {  builder.append(" <tr>");  builder.append("<td>").append(p.getName()).append("</td>");  builder.append("<td>").append(p.getAge()).append("</td>");  builder.append("<td>").append(p.getBloodType()).append("</td>");  builder.append("</tr>\n");  }  builder.append("</table>\n");  System.out.println(builder.toString());  }  } |

StringBuilder3.java의 코드와 StringBuilder2.java의 코드의 출력은 동일하다.

StringBuilder 클래스의 append 메소드의 리턴 타입은 StringBuilder 이다.

이 메소드는 this 객체를 리턴한다.

(가)

|  |
| --- |
| builder.append("<td>");  builder.append(p.getName()); |

위 코드는 builder 객체의 append 메소드를 2번 호출한다.

(나)

|  |
| --- |
| builder.append("<td>").append(p.getName()); |

위 코드도 builder 객체의 append("<td>") 메소드를 호출한다.

이 메소드는 this 객체를 리턴한다. 즉 builder 객체를 리턴한다.

그 다음 리턴된 객체의 append(p.getName()) 메소드를 호출한다.

결국 (가)와 (나)는 완전히 같은 일을 하는 코드이다.

(나)와 같이 메소드 호출을 연속해서 구현하는 코딩 기법을 method chaining 이라고 부른다.

코드를 간결하게 해주기 때문에 프로그래머들이 즐겨 사용하는 코딩 기법이다.

StringBuilder 클래스의 메소드들 중에서 리턴 타입이 StringBuilder인 메소드를 찾아보자.

이 메소들의 리턴 타입이 StringBuilder인 이유가 바로 method chaining 코딩 기법을 활용하기 위함이다.

## 과제

StringBuilder 클래스의 메소드를 적어도 10개 이상 사용하는 예제 코드를 구현하라.

즉 StringBuilder 클래스의 메소드를 사용해 보라는 얘기이다.