정리해 봅시다

문제에 대한 답은 아래에서 직접 문제를 푸시고 확인할 수 있습니다.
https://sites.google.com/site/godofjavabook/

1 API 문서를 자동으로 생성하는 명령어를 무엇이라고 했나요?

2 API의 왼쪽 상단에는 어떤 내용이 제공되나요?

3 API의 왼쪽 하단에는 어떤 내용이 제공되나요?

4 API 우측 화면에는 보통 어떤 내용이 제공되나요?

5 Deprecated라는 것의 의미는 무엇인가요?

2장 String

이 장을 시작하기 전에

이 장의 제목은 다른 문장으로 설명할 필요가 없어 다른 장과는 다르게 아주 단순하게 제목을 정했다. 개발자분들이 언제든지 이 장을 쉽게 찾아 보고 참조할 수 있도록 하기 위한 것도 또 하나의 이유다.

만약

- char가 뭔지 모르거나
- 참조 자료형에 대해서 이해가 안 되거나,
- 메소드가 뭔지에 대해서 이해가 안 되거나,
- 앞 장에서 살펴본 API를 어떻게 사용하는지 모르는 분들은

지금까지 배운 내용들을 다시 읽고 이해한 후 이 장을 보셔야 합니다.

혹시

- String이 어떤 클래스인지,
- String의 주요 메소드들의 이름만 들어도 사용법이 머리에 팍 떠오르거나,
- StringBuffer와 StringBuilder 클래스와 String 클래스의 차이

에 대해서 아주 잘 알고 있다면 이 장을 건너 뛰어도 됩니다.

자바에서 가장 많이 사용하는 String 클래스

필자가 회사에서 주로 하는 업무는 시스템의 장애를 진단하고, 성능 튜닝 및 측정을 하는 것이다. 그런데, 그러한 진단 작업을 하면서 어떤 객체가 가장 많이 생성되는지 점검하는 경우가 있다. 그럴 때 보면 String 관련 객체는 몇백 개의 객체 중에서 상위 5개 안에는 항상 포함된다. 그만큼 여러분들이 나중에 시스템을 개발할 때 `String` 클래스를 많이 사용하기 때문에 잘 알고 있어야만 하는 중요한 클래스다.

"그냥 String은 더하면 되는 거 아니야?"라고 생각할 수도 있지만, 자바에서 String의 비중은 매우 크다. 자바 클래스들 중에서 `String`은 VVIP 정도로 특별 취급을 받는다. 클래스 중에서 더하기 연산을 제공하는 클래스는 String 밖에 없다. "그냥 객체 더하면 더해지던데 무슨 소리야?"라는 분도 있겠지만, 객체에 더하기를 하면 `toString()` 메소드가 호출되고 그 결과를 더하는 거다. 결론적으로 String을 더한다는 말이다. 그럼, 이 귀한 `String` 클래스를 집중 해부해 보자.

가장 먼저 `String` 클래스가 어떻게 선언되어 있는지 살펴보자.

```
public final class String extends Object
  implements Serializable, Comparable<String>, CharSequence
```

참고

지금까지 이 책에서는 필자가 친절하게 해당 메소드가 클래스 내의 어디에 포함되어야 하는지, 해당 코드가 어디에 포함되어야 하는지를 쉽게 찾아갈 수 있도록 클래스 선언부까지 포함하여 보여주었다. 하지만, 이제 어느 정도 자바에 친숙해졌으리라 생각되니, 실습 코드에 클래스 선언부까지 보여주지는 않을 예정이다.

클래스 선언문이 조금 길다. 하지만, 하나 하나 짚어 가보면 그리 어려운 내용은 아니다. 가장 첫 줄을 살펴보자.

```
public final class String extends Object
```


public final로 선언되었다. public이 "누구나 다 사용할 수 있는 클래스"라는 정도는 다 기억하고 있을 것이다. (이거 기억 못하고 있으면 정말 안 된다. - -;) 그 다음에 final이라고 선언되어 있다.

클래스가 final로 선언되어 있으면 뭐가 다른가?

앞(Vol.1의 12장 "인터페이스")에서 설명했지만, 클래스가 final로 선언되어 있으면 더 이상 이 클래스는 확장할 수 없다. 다시 말해서 `String` 클래스는 자식 클래스를 양산할 수 없다. 그냥 있는 대로 사용해야만 한다. 그리고, 첫 줄의 마지막을 보면 `Object` 클래스를 확장한 것을 볼 수 있다. 다시 말하면, 모든 클래스의 부모 클래스는 `Object` 클래스이므로 이 외에 따로 확장한 클래스는 없다는 말이다.

둘째 줄에 있는 implements 문장은 다음과 같이 되어 있다.

```
implements Serializable, Comparable<String>, CharSequence
```

클래스 선언문에 implements라고 하면 어떤 의미가 있는지 한번 다시 생각해보자.

implements라고 적어준 뒤 인터페이스들을 나열하면 해당 인터페이스에 선언된 메소드들을 이 클래스에서 "구현"한다는 의미다. 따라서, String은 `Serializable`, `Comparable`, `CharSequence`라는 인터페이스를 구현한 클래스다.

여기서 `Serializable` 인터페이스는 구현해야 하는 메소드가 하나도 없는 아주 특이한 인터페이스다. 이 `Serializable` 인터페이스를 구현한다고 선언해 놓으면, 해당 객체를 파일로 저장하거나 다른 서버에 전송 가능한 상태가 된다. 이 부분은 지금 설명해도 이해하기 쉽지 않을 테니, 12장의 "`Serializable`"에서 자세하게 다시 살펴보자.

그리고, `Comparable`이라는 인터페이스도 구현한다. 이 인터페이스는 `compareTo()`라는 메소드 하나만 선언되어 있다. 이 메소드는 매개 변수로 넘어가는 객체와 현재

객체가 같은지를 비교하는 데 사용된다. 간단하게 이름과 내용만으로 보기에는 그냥 `equals()` 메소드와 별 차이가 없다고 생각할 수도 있다. 하지만, 이 메소드의 리턴 타입은 int다. 같으면 0이지만, 순서 상으로 앞에 있으면 -1, 뒤에 있으면 1을 리턴한다. 다시 말해서 객체의 순서를 처리할 때 유용하게 사용될 수 있다. 이 장의 뒷부분에서 이에 대한 내용은 자세히 살펴보자. 그리고, 선언문의 꺽쇠(<>) 안에 `String`이라고 적어 주었는데, 이는 제네릭Generic이라는 것을 의미한다. 이 제네릭도 설명하자면 할 말이 많기 때문에, 4장의 "실수를 방지하기 위한 제네릭이라는 것도 있어요."를 참고하기 바란다.

가장 마지막에 있는 `CharSequence`라는 인터페이스가 있다. 이 인터페이스는 해당 클래스가 문자열을 다루기 위한 클래스라는 것을 명시적으로 나타내는 데 사용된다. 이 장의 가장 끝 부분에서 설명하는 `StringBuilder`와 `StringBuffer` 클래스도 이 `CharSequence` 인터페이스를 구현해 놓았다. 이에 대한 자세한 설명은 그 부분에서 살펴보자.

이제 본격적으로 `String` 클래스에 대한 상세한 내용을 살펴보자.

String의 생성자에는 이런 애들이 있다

먼저 생성자를 살펴보자. 대부분 문자열을 만들 때에는 다음과 같이 간단하게 만든다.

```
String name="Sangmin, Lee";
```

대부분의 경우 이렇게 선언하지만, `String`의 생성자는 매우 많다. 생성자의 목록을 보기 전에 몇가지 용어에 대해서 간단히 살펴보자. 캐릭터 셋은 문자의 집합을 의미하며, 이해가 쉽게 한마디로 이야기하면, 한글, 일본어와 같이 특정 나라의 글자를 말한다. 자세한 설명은 잠시 뒤에 알아보자. 그리고, 디코딩은 일반적으로 암호화 되어 있거나 컴퓨터가 이해할 수 있는 값들을 알아보기 쉽게 변환하는 것을 말한다. 그러면 이제 `String` 클래스의 생성자를 살펴보자.

생성자	설명
`String()`	비어있는 `String` 객체를 생성한다. 그런데 이렇게 생성하는 것은 전혀 의미가 없다. 차라리 다음과 같이 선언하는 것이 더 효율적이다. `String name=null;`
`String(byte[] bytes)`	현재 사용중인 플랫폼의 캐릭터 셋을 사용하여 제공된 byte 배열을 디코딩한 `String` 객체를 생성한다.
`String(byte[] bytes, Charset charset)`	지정된 캐릭터 셋을 사용하여 제공된 byte 배열을 디코딩한 `String` 객체를 생성한다.
`String(byte[] bytes, String charsetName)`	지정한 이름을 갖는 캐릭터 셋을 사용하여 지정한 byte 배열을 디코딩한 `String` 객체를 생성한다.
`String(byte[] bytes, int offset, int length)`	현재 사용중인 플랫폼의 기본 캐릭터 셋을 사용하여 지정한 byte 배열의 일부를 디코딩한 `String` 객체를 생성한다.
`String(byte[] bytes, int offset, int length, Charset charset)`	지정된 캐릭터 셋을 사용하여 byte 배열의 일부를 디코딩한 `String` 객체를 생성한다.
`String(byte[] bytes, int offset, int length, String charsetName)`	지정한 이름을 갖는 캐릭터 셋을 사용하여 byte 배열의 일부를 디코딩한 `String` 객체를 생성한다.
`String(char[] value)`	char 배열의 내용들을 붙여 `String` 객체를 생성한다.
`String(char[] value, int offset, int count)`	char 배열의 일부 내용들을 붙여 `String` 객체를 생성한다.
`String(int[] codePoints, int offset, int count)`	유니코드 코드 위치(Unicode code point)로 구성되어 있는 배열의 일부를 새로운 `String` 객체를 생성한다.
`String(String original)`	매개 변수로 넘어온 String과 동일한 값을 갖는 String 객체를 생성한다. 다시 말해서, 복제본을 생성한다.
`String(StringBuffer buffer)`	매개 변수로 넘어온 StringBuffer 클래스에 정의되어 있는 문자열의 값과 동일한 String 객체를 생성한다.
`String(StringBuilder builder)`	매개 변수로 넘어온 StringBuilder 클래스에 정의되어 있는 문자열의 값과 동일한 String 객체를 생성한다.

여기에 있는 여러 생성자들을 다 외울 필요는 없다. 각각의 생성자를 사용할 필요가 있을 때 적당한 생성자를 찾아 사용하면 된다. 여기에 있는 생성자 중에서 그나마 많이 사용하는 생성자를 추려보면 다음과 같다.

- String(byte[] bytes)
- String(byte[] bytes, String charsetName)

이 생성자들은 한글을 사용하는 우리나라에서는 자주 사용할 수밖에 없다. 왜냐하면, 대부분의 언어에서는 문자열을 변환할 때 기본적으로 영어로 해석하려고 하기 때문이다.

"왜 이거밖에 안쓰지?"라고 생각할 수도 있지만, String 객체는 대부분 따옴표로 묶어 생성하기 때문에 굳이 어려운 생성자를 사용할 필요가 없기 때문이다.

String 문자열을 byte로 변환하기

그런데, 생성자의 매개 변수로 받는 byte 배열은 어떻게 생성할까? 이 부분에 대해서는 고민할 필요가 없다. String 클래스에는 현재의 문자열 값을 byte 배열로 변환하는 다음과 같은 getBytes()라는 메소드들이 있기 때문이다.

리턴 타입	메소드 이름 및 매개 변수	설명
byte[]	getBytes()	기본 캐릭터 셋의 바이트 배열을 생성한다.
byte[]	getBytes(Charset charset)	지정한 캐릭터 셋 객체 타입으로 바이트 배열을 생성한다.
byte[]	getBytes(String charsetName)	지정한 이름의 캐릭터 셋을 갖는 바이트 배열을 생성한다.

보통 캐릭터 셋을 잘 알고 있거나, 같은 프로그램 내에서 문자열을 byte 배열로 만들 때에는 가장 위에 있는 getBytes() 메소드를 사용하면 된다. 하지만, 다른 시스템에서 전달 받은 문자열을 byte 배열로 변환할 때에는 두번째나 세번째에 있는 메소드를 사용하는 것이 좋다. 왜냐하면 다른 캐릭터 셋으로 되어 있을 수도 있기 때문이다.

잠시 "캐릭터 셋"에 대해서 좀 자세히 살펴보자. 자바만이 아니라, 어떤 프로그래밍 언어를 사용할 경우에도 특수문자를 표시할 일이 생긴다. 여기서 특수문자는 특별한 문자라기보다는 알파벳을 제외한 나라의 문자를 의미한다고 보는 것이 가장 쉽게 이해가 될 것이다. 그렇다면 한글은? 한글도 기본적으로는 알파벳이 아니기 때문에 고유의 캐릭터 셋을 가진다. 가끔 웹 페이지를 서핑하다 보면 한글이 깨지는 경우를 보았을 것이다. 이렇게 한글이 깨지는 이유는 브라우저에서 생각하는 캐릭터 셋과 웹 페이지에 지정된 캐릭터 셋이 다르기 때문이다. java.nio 패키지의 Charset 클래스 API에는 표준 캐릭터 셋이 정해져 있다. 목록을 한번 살펴보자.

참고로 여기서 UCS는 유니코드 캐릭터 셋Unicode Character Set의 약자다.

캐릭터 셋 이름	설명
US-ASCII	7비트 아스키
ISO-8859-1	ISO 라틴 알파벳
UTF-8	8비트 UCS 변환 포맷
UTF-16BE	16비트 UCS 변환 포맷. big-endian 바이트 순서를 가진다.
UTF-16LE	16비트 UCS 변환 포맷. little-endian 바이트 순서를 가진다.
UTF-16	16비트 UCS 변환 포맷. 바이트의 순서는 byte-order mark라는 것에 의해서 정해진다.
EUC-KR	8비트 문자 인코딩으로, EUC의 일종이며 대표적인 "한글 완성형" 인코딩
MS949	Microsoft에서 만든 "한글 확장 완성형" 인코딩

한글을 처리하기 위해서 자바에서 많이 사용하는 캐릭터 셋은 이 중에서 UTF-16이다. 예전에는 UTF-8이나 EUC-KR을 많이 사용했지만, 요즘에는 대부분 UTF-16을 많이 사용한다.

이제 예제를 통해서 String 클래스의 생성자를 살펴보자. StringSample이라는 클래스를 d.string이라는 패키지에 만들고, main() 메소드도 만들자. 생성자들을 확인해 볼 constructors()라는 메소드를 다음과 같이 만들어 놓자.

```
package d.string;

public class StringSample {
  public static void main(String[] args) {
    StringSample sample=new StringSample();
    sample.constructors();
  }
  public void constructors() {
    try {
      //예제 코드가 위치할 부분
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

그러면 이제 예제를 작성할 준비는 끝났다. 이제 중간에 있는 "//예제 코드가 위치할 부분"에 지금부터 살펴보는 코드를 추가하자.

```
String str="한글"; —❶

byte[] array1=str.getBytes(); —❷
for(byte data:array1) { —❸
  System.out.print(data + " ");
}
System.out.println();
String str1=new String(array1); —❹
System.out.println(str1);
```

처음 보면 복잡해 보이지만, 하나 하나 살펴보면 그리 어렵지 않다.

❶ "한글"이라는 값을 갖는 String 객체인 str를 생성했다.

❷ 방금 알아본 getBytes()라는 메소드를 사용하여 str를 byte 배열로 만들었다.

❸ 만들어진 byte 배열에 어떤 값들이 있는지 살펴보기 위해서 for 루프를 사용하여 각각의 byte 값을 출력하도록 해 놓았다.

❹ byte 배열을 갖고 String 객체를 만들기 위해서 byte 배열(array1)을 매개 변수로 갖는 String 객체를 생성하고, 그 문자열을 출력했다.

이렇게 코드를 작성하고 `StringSample` 클래스를 컴파일 및 실행하자. 결과는 다음과 같다.

```
-57 -47 -79 -37
한글
```

원래 만들어 놓았던 "한글"이라는 값이 그대로 출력된 것을 볼 수 있다. 왜냐하면 `getBytes()` 메소드는 플랫폼의 기본 캐릭터 셋으로 변환을 하고, String(byte[]) 생성자도 플랫폼의 기본 캐릭터 셋으로 변환을 하기 때문에 전혀 문제가 발생하지는 않았다.

여기서 중간에 있는 byte 배열의 값을 출력하는 부분이 아무리 생각해도 자주 사용될 것 같으니 다음과 같이 별도의 메소드로 만들어 놓자.


```
package d.string;

public class StringSample {
  //중간 생략
  public void printByteArray(byte[] array) {
    for(byte data:array) {
      System.out.print(data + " ");
    }
    System.out.println();
  }
}
```

이렇게 메소드 내에 자주 사용되는 부분을 별도의 메소드로 빼 놓는 버릇을 들이는 것이 향후에 재사용성을 위해서 좋다.

이 `printByteArray()` 메소드를 활용하도록 `constructors()` 메소드를 변경하면 다음과 같다.

```
String a="한글";

byte[] array1=str.getBytes();
printByteArray(array1);
String str1=new String(array1);
System.out.println(str1);
```

메소드가 보다 읽기 쉬워졌다는 것을 알 수 있다. 이번에는 "EUC-KR"이라는 캐릭터 셋으로 변환해보자. `constructors()` 메소드의 가장 아랫줄에 다음의 4줄을 추가하자.

```
byte[] array2=str.getBytes();
printByteArray(array2);
String str2=new String(array2,"EUC-KR");
System.out.println(str2);
```

이렇게 해 놓고 실행하면 어떤 결과가 나올까?

결과는 다음과 같다.

```
-57 -47 -79 -37
한글
-57 -47 -79 -37
한글
```

앞서 알아본 예제와 차이가 없이 결과가 동일하게 출력된 것을 볼 수 있다. 다시 말해서 지금 필자가 사용하는 자바 플랫폼의 기본 캐릭터 셋은 "EUC-KR"이라는 것을 알 수 있다. str2 객체를 생성할 때 캐릭터 셋을 EUC-KR이 아닌 UTF-8로 사용하면 어떻게 될까? 다시 말해서 다음과 같이 변경했을 때를 말하는 것이다.

```
byte[] array2=str.getBytes();
printByteArray(array2);
String str2=new String(array2,"UTF-8");
System.out.println(str2);
```

이렇게 변경한 후 실행 결과는 다음과 같이 나온다.

```
-57 -47 -79 -37
한글
-57 -47 -79 -37
???
```

가장 아래에 있는 결과가 물음표(???)로 나오는 것을 볼 수 있다. 즉, 잘못된 캐릭터 셋으로 변환을 하면 여러분들이 알아 볼 수 없는 문자로 표시된다. 예전에 여러분들의 선배들이 자바를 처음 배울 때 이와 같이 캐릭터 셋을 변환하는 부분에서 고생을 많이 했었다. 지금은 참고할 만한 정보들이 많아서 이러한 고생을 많이 하지 않는다. 일단 방금 UTF-8로 변경한 부분을 결과가 제대로 출력되도록 다시 EUC-KR로 바꾸어 놓자.

그렇다면, byte 배열로 변환할 때 캐릭터 셋을 변경해 버릴 수 없을까? 그럴 필요가 있을 때에는 `String` 클래스에 `getBytes()`라는 이름을 갖는 메소드 중에서 매

개 변수를 캐릭터 셋 이름으로 지정하는 메소드를 사용하면 된다. 다음의 내용을 `constructors()` 메소드의 try 블록의 가장 마지막에 추가하자.

```
byte[] array3=str.getBytes("UTF-16");
printByteArray(array3);
String str3=new String(array3,"UTF-16");
System.out.println(str3);
```

이렇게 추가한 후 실행해보자. 결과가 제대로 나올까?

결과는 다음과 같다.

```
-57 -47 -79 -37
한글
-57 -47 -79 -37
한글
-2 -1 -43 92 -82 0
한글
```

정상적으로 결과가 나온 것을 볼 수 있다. 하지만, 한 가지 다른 것이 있다. EUC-KR의 경우는 한글 두 글자를 표현하기 위해서 4 바이트를 사용하지만, UTF-16은 6 바이트를 사용한다는 점이다. 여러분들이 str의 값을 변경해 보면서 확인해보면 글자 수와 상관 없이 무조건 2바이트의 차이가 발생한다는 것을 알 수 있을 것이다.

문자열 str을 "한글"이 아닌 "최고의 자바 기본서"라고 변경하여 직접 실행해 보기 바란다.

자바에서 한글이 몇 바이트를 점유하는지 알아 두는 것은 우리나라에서 개발하면 매우 중요하다.

이번 절에서는 `String` 클래스의 생성자와 `getBytes()` 메소드에 대해서 살펴보았다. 모든 생성자를 다 외우고 있을 필요는 없지만, 많이 사용되는 몇몇 생성자와 `getBytes()` 메소드는 자바 개발을 목적으로 이 책을 보는 분들은 꼭 기억하고 있어

야만 한다. 이번 절에서 만든 constructors() 메소드의 전체 소스를 보면서 배운 내용을 다시 한번 확인해 보기 바란다.

```
public void constructors() {
  try {
    String str="한글";

    byte[] array1=str.getBytes();
    printByteArray(array1);
    String str1=new String(array1);
    System.out.println(str1);

    byte[] array2=str.getBytes();
    printByteArray(array2);
    String str2=new String(array2,"EUC-KR");
    System.out.println(str2);

    byte[] array3=str.getBytes("UTF-16");
    printByteArray(array3);
    String str3=new String(array3,"UTF-16");
    System.out.println(str3);

  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

그런데, 왜 필자가 이 메소드의 내용들을 try-catch 블록으로 감싸 놓았을까? 그 이유는 캐릭터 셋을 지정하는 메소드 및 생성자들 때문이다. byte 배열과 String 타입의 캐릭터 셋을 받는 생성자, getBytes() 메소드 중에서 String 타입의 캐릭터 셋을 받는 메소드는 UnsupportedEncodingException을 발생시킬 수 있다. 존재하지 않는 캐릭터 셋의 이름을 지정할 경우에는 이 예외가 발생하게 되므로, 반드시 try-catch로 감싸주거나 constructors() 메소드 선언시 throws 구문을 추가해 주어야만 한다(이 말이 무슨 뜻인지 이해가 안되는 분들은 Vol.1의 14장 "다 배운거 같지만, 예외라는 중요한게 있어요"를 다시 한번 읽어서 예외에 대해서 꼭 이해하고 넘어가 주기 바란다).

객체의 널 체크는 반드시 필요하답니다

여러분들이 String의 메소드를 사용하기 전에 짚고 넘어가야 하는 것이 있다. 바로 널 체크다. 널은 null로 표시하며, 어떤 참조 자료형도 널이 될 수 있다. 객체가 널이라는 말은 객체가 아무런 초기화가 되어 있지 않으며, 클래스에 선언되어 있는 어떤 메소드도 사용할 수 없다는 의미다. 다시 말해서, 여러분들이 널 체크를 하지 않으면 객체에 사용할 수 있는 메소드들은 모두 예외를 발생시킨다.

어떤 객체가 널이면 아무런 일을 못한다. 누군가가 너무 배가 고픈데, 카드도 없고 현금도 없는 상태에서 식당에 들어가서 밥을 먹는다고 생각해보자. 밥을 다 먹고 나서 그냥 식당을 나온다면 그 사람은 무전취식으로 경찰서에 가게 된다. 물론 주인이 베푸는 것을 아주 좋아하는 분이라면 그냥 보내 줄 수도 있다. 하지만, 자바는 그렇게 착한 주인이 아니다. 여러분들이 널인 객체의 메소드를 호출하는 순간 예외를 발생시킨다. 다시 말해서 돈이 없으면 주문도 할 수 없는 선불 식당이라고 생각하면 된다. 일단 예제를 살펴보자.

다음과 같이 StringSample 클래스에 nullCheck()라는 메소드를 만들자.

```
public boolean nullCheck(String text) {
  int textLength=text.length();
  System.out.println(textLength);
  if(text==null) return true;
  else return false;
}
```

여기서 메소드의 처음 두 줄은 null인 객체를 참조할 때 어떤 일이 발생하는지를 확인하기 위한 용도로 작성된 것이다. 이렇게 nullCheck() 메소드를 만들어 놓고, main() 메소드에서 이 메소드만 호출하도록 다음과 같이 변경하자.

```
public static void main(String[] args) {
  StringSample sample=new StringSample();
  //sample.constructors();
  System.out.println(sample.nullCheck(null));
}
```

null도 자바 예약어이기 때문에 어떤 객체의 이름이나 메소드 이름 등으로 사용할 수 없다. 이 소스의 컴파일은 정상적으로 된다.

코드를 실행해보자. 결과는 어떻게 나올까?

객체가 널인지 아닌지는 실행시에만 확인할 수 있기 때문이다. 이 메소드가 수행된 결과는 다음과 같이 나온다.

```
Exception in thread "main" java.lang.NullPointerException
  at d.string.StringSample.nullCheck(StringSample.java:43)
  at d.string.StringSample.main(StringSample.java:7)
```

다시 말해서, null인 객체의 메소드에 접근하면 NullPointerException이 발생한다. 이제 다음과 같이 객체를 참조하는 메소드들을 주석 처리하자.

```
public boolean nullCheck(String text) {
  //int textLength=text.length();
  //System.out.println(textLength);
  if(text==null) return true;
  else return false;
}
```

이렇게 수정한 후 컴파일 및 실행을 하면 true가 출력된다. 게다가, 예상치 못한 NullPointerException도 발생하지 않는다. 객체가 널인지 아닌지 체크하는 것은 이와 같이 !=이나 == 연산자를 사용하면 쉽게 확인할 수 있다.

여러분들이 null 체크하는 것을 우습게 봐서는 안 된다. 널 체크를 하지 않아서 애플리케이션이 비정상으로 작동하여 장애로 이어질 수도 있기 때문이다. 아무리 강조해도 지나치지 않는 것이 바로 null 체크이므로, 메소드의 매개 변수로 넘어오는 객체가 널이 될 확률이 조금이라도 있다면 반드시 한 번씩 확인하는 습관을 갖고 있어야만 한다.

그럼 이제 본격적으로 String 클래스에 어떤 메소드들이 있는지 살펴보자.

String의 내용을 비교하고 검색하는 메소드들도 있어요

String 클래스는 문자열을 나타낸다. 따라서, 문자열 내에 특정 위치를 찾거나, 값을 비교하는 등의 작업은 아주 빈번히 발생된다. String 클래스에서 제공하는 메소드에는 어떤 것들이 있고, 어떤 작업을 수행하는지를 알아두면 여러분들의 코드도 그만큼 깔끔해지고, 개발도 빨라진다. 그 여러 가지 메소드 중에서 이번 절에서는 String 클래스 객체의 내용을 비교하고 검색하는 메소드들을 알아보자. 이 메소드들을 조금 더 세밀하게 분류하면 다음과 같다.

- 문자열의 길이를 확인하는 메소드
- 문자열이 비어 있는지 확인하는 메소드
- 문자열이 같은지 비교하는 메소드
- 특정 조건에 맞는 문자열이 있는지를 확인하는 메소드

여러분들이 이렇게 구분해 놓은 것만 보면 "메소드 이름이 뭘까?"하고 고민하는 습관을 들여야 한다. 왜냐하면, 이 책으로 자바 공부를 마치면 여러분들은 앞으로 자바를 통하여 수많은 메소드들을 만들 것이고, 그 메소드들의 이름도 지정해야 하기 때문이다. 메소드 이름이나 변수 이름을 애매모호하게 지정하면 나중에 논란의 소지가 될 수도 있으므로, 메소드 이름도 직관적으로 짓는 습관을 들여야만 한다.

먼저 비교하는 데 사용되는 메소드들을 수행할 compareCheck() 메소드를 StringSample 클래스에 다음과 같이 만들어 놓고, main() 메소드에서 이 메소드만 수행하도록 해놓자.

```
public void compareCheck() {

}
```

문자열의 길이를 확인하는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
int	length()	문자열의 길이를 리턴한다.

문자열의 길이를 확인할 때에는 `length()` 메소드를 사용하면 된다. 배열도 객체이긴 하지만 메소드는 없는 특수한 객체다. 그래서 배열의 크기를 확인할 때에는 괄호가 없는 length를 사용한다. 하지만, 그 이외의 모든 클래스는 메소드를 호출해야 하며, String 객체의 길이를 확인하기 위해서는 `length()`라는 메소드를 사용해야만 한다. 다음의 사용 예를 보자.

```
public void compareCheck() {
  String text="You must know String class.";
  System.out.println("text.length()="+text.length());
}
```

이와 같이 간단하게 사용하면 된다.

이 메소드가 수행된 결과는 어떻게 나올까?

결과는 text 객체의 char 개수를 세어보면 된다. 출력되는 내용은 다음과 같다.

```
text.length()=27
```

당연한 이야기지만, 공백도 길이에 포함된다. 만약 한글일 경우에는 어떻게 결과가 나올지 궁금한 독자도 있을 것이다. 한글의 길이는 직접 text 값을 변경해서 확인해 보자.

문자열이 비어 있는지 확인하는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
boolean	isEmpty()	문자열이 비어 있는지를 확인한다. 비어 있으면 true를 리턴한다.

문자열의 길이가 0인지 아닌지를 확인하는 것보다, 이 메소드를 사용하는 것이 훨씬 간단하다. 다음의 한줄을 `compareCheck()` 메소드의 가장 아랫 줄에 넣고 실행해 보자.


```
System.out.println("text.isEmpty()="+text.isEmpty());
```

이 한줄을 추가한 후 결과는 어떻게 나올지 생각해보자.

text라는 객체는 27개의 char로 구성되어 있으므로 비어 있지 않다. 따라서 결과는 false를 리턴한다.

```
text.length()=27
text.isEmpty()=false
```

만약 text가 공백 하나로 되어 있는 문자열이라도, 이 메소드는 false를 리턴한다. 우리 눈으로 보기에 공백이나 빈칸이나 출력하면 같은 것처럼 느끼지만 프로그램에서는 다르게 인식한다.

문자열이 같은지 비교하는 메소드

String 클래스에서 제공하는 문자열이 같은지 비교하는 메소드들은 매우 많다.

리턴 타입	메소드 이름 및 매개 변수
boolean	equals(Object anObject)
boolean	equalsIgnoreCase(String anotherStr)
int	compareTo(String anotherStr)
int	compareToIgnoreCase(String str)
boolean	contentEquals(CharSequence cs)
boolean	contentEquals(StringBuffer sb)

메소드들의 이름으로 분류하면 equals로 시작하는 메소드, compareTo로 시작하는 메소드, contentEquals 메소드로 세 가지 메소드로 분류할 수 있다. 이름들은 서로 상이하지만, 이 모든 메소드들은 매개 변수로 넘어온 값과 String 객체가 같은지를 비교하기 위한 메소드다. 단지, IgnoreCase가 붙은 메소드들은 대소문자 구분을

할지 안할지 여부만 다르다. Ignore는 "무시한다"는 의미이며, Case는 "대소문자"를 말한다.

이 중에서 가장 먼저 equals() 메소드에 대해서 살펴보자. equals() 메소드에 대한 설명은 이미 Object 클래스를 설명할 때 자세히 알아봤으므로 다 기억하고 있으리라 믿는다. 다음 예제처럼 equalCheck()이라는 메소드를 만들자.

```
public void equalCheck() {
  String text="Check value";
  String text2="Check value";
  if(text==text2) {
    System.out.println("text==text2 result is same.");
  } else {
    System.out.println("text==text2 result is different.");
  }
  if(text.equals("Check value")) {
    System.out.println("text.equals(text2) result is same.");
  }
}
```

이렇게 equalsCheck() 메소드를 만들어 놓고, main() 메소드에서 이 메소드만 호출하도록 한 뒤에 컴파일 후 실행해보자. 어떤 결과가 나올까?

먼저 결과를 보자.

```
text==text2 result is same.
text.equals(text2) result is same.
```

"어? 일반적으로 생각하기에는 두 번째 if 문만 통과해서 결과가 출력될 것 같은데 아니네…"라고 생각할 수도 있다. 왜냐하면, 필자가 말한 내용을 기억하는 독자는 자바에서 객체는 equals() 메소드로 비교를 해야 한다고 한 것을 기억할 것이기 때문이다.

String 클래스도 기본적으로 equals() 메소드를 사용해서 비교를 해야만 한다. 하지만 이렇게 결과가 나오는 이유는 자바에 Constant Pool이라는 것이 존재하기 때문이다. 이에 대한 자세한 설명은 이 책에서 안하겠지만, 간단하게 이야기하면, 자바에서는 객체들을 재사용하기 위해서 Constant Pool이라는 것이 만들어져 있고, String의 경우 동일한 값을 갖는 객체가 있으면, 이미 만든 객체를 재사용한다. 따라서, text와 text2 객체는 실제로는 같은 객체다.

이 첫 번째 연산 결과가 우리가 원하는 대로 나오도록 하려면, text2 객체의 생성을 다음과 같이 변경하면 된다.

```
String text2=new String("Check value");
```

이렇게 String 객체를 생성하면 값이 같은 String 객체를 생성한다고 하더라도 Constant Pool의 값을 재활용하지 않고 별도의 객체를 생성한다. 따라서 이와 같이 변경 후에 이 메소드를 수행해보면 결과가 다음과 같이 출력된 것을 확인할 수 있다.

```
text==text2 result is different.
text.equals(text2) result is same.
```

이번에는 String 클래스에 선언되어 있는 equalsIgnoreCase() 메소드의 사용법을 확인해보자. 다음과 같이 equalsCheck() 메소드의 가장 마지막 줄 하단에 4줄을 추가하자.

```
String text3="check value";
if(text.equalsIgnoreCase(text3)) {
  System.out.println("text.equalsIgnoreCase(text3) result is same.");
}
```

text와 text3는 첫 글자가 대문자로 시작하는지, 소문자로 시작하는지의 차이만 존재한다. 이렇게 대소문자만 다른 문자열을 equalsIgnoreCase() 메소드를 이용하여 비교한 결과는 다음과 같다.

```
text==text2 result is different.
text.equals(text2) result is same.
text.equalsIgnoreCase(text3) result is same.
```

예상한 대로 대소문자를 구분하지 않고 두개의 값이 같은지 다른지만 확인하는 것을 볼 수 있다.

String 클래스의 compareTo() 메소드는 Comparable 인터페이스에 선언되어 있다고 이 장의 앞부분에서 설명했다. 다음과 같이 StringSample 클래스에 compareToCheck() 메소드를 추가하자.

```
public void compareToCheck() {
  String text="a";
  String text2="b";
  String text3="c";
  System.out.println(text2.compareTo(text));
  System.out.println(text2.compareTo(text3));
  System.out.println(text.compareTo(text3));
}
```

이렇게 만들어 놓고, 각 결과가 어떻게 출력될지 생각해보자.

compareTo() 메소드는 보통 정렬sorting을 할 때 사용한다. 따라서, true, false의 결과가 아니라, 비교하려는 매개 변수로 넘겨준 String 객체가 알파벳 순으로 앞에 있으면 양수를, 뒤에 있으면 음수를 리턴한다. 그리고, 알파벳 순서만큼 그 숫자값은 커진다. 결과를 확인해 보자.

```
1
-1
-2
```


"b"가 "a"보다 뒤에 있으므로, 첫 번째 결과는 양수를 리턴한다. 나머지 "b"와 "c", "a"와 "c"를 비교한 결과는 각각 음수를 리턴한 것을 볼 수 있다. 이러한 결과값은 반드시 외울 필요는 없지만, 여러분들이 여러 가지 각종 값들을 비교해서 어떤 결과가 나오는지는 확인해 둘 필요가 있다. compareToIgnoreCase() 메소드는 equalsIgnoreCase() 메소드와 마찬가지로, 대소문자 구분을 하지 않고 compareTo() 메소드를 수행하는 것과 같다.

String 클래스의 문자열 비교하는 메소드 중 마지막으로 contentEquals() 메소드는 매개 변수로 넘어오는 CharSequence와 StringBuffer 객체가 String 객체와 같은지를 비교하는 데 사용된다. 관련된 사용은 할 말이 많으므로, 이 장의 마지막 절에서 별도로 확인해 보자.

특정 조건에 맞는 문자열이 있는지를 확인하는 메소드

리턴 타입	메소드 이름 및 매개 변수
boolean	startsWith(String prefix)
boolean	startsWith(String prefix, int toffset)
boolean	endsWith(String suffix)
boolean	contains(CharSequence s)
boolean	matches(String regex)
boolean	regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
boolean	regionMatches(int toffset, String other, int ooffset, int len)

이 중에서 가장 많이 사용하는 것이 startsWith() 메소드다. startsWith() 메소드는 이름 그대로 매개 변수로 넘겨준 값으로 시작하는지를 확인한다. 예를 들어 "서울시 구로구 신도림동", "경기도 성남시 분당구 정자동"과 같이 주소를 나타내는 문자열들이 있을 때 "서울시"의 주소를 갖는 모든 문자열을 쉽게 찾을 수 있다. 다음 절에 있는 indexOf()라는 메소드를 사용하여 확인하는 것도 가능하지만, 이 메소드의 단점은 문자열의 모든 내용을 다 확인해 봐야 한다는 단점이 있다. 마찬가지로 endsWith() 메소드는 매개 변수로 넘어온 값으로 해당 문자열이 끝나는지를 확인하는 메소드다.

startsWith()와 endsWith 메소드를 사용하여 점검 대상의 문자열에 확인하고자 하는 문자열로 시작하는 개수와 끝나는 개수를 세어 보는 예제를 살펴보자. 다음과 같이 StringSample 클래스에 addressCheck()라는 메소드를 만들자.

```
public void addressCheck() {
  String addresses[]=new String[]{
      "서울시 구로구 신도림동",
      "경기도 성남시 분당구 정자동 개발 공장",
      "서울시 구로구 개봉동",
  };
  int startCount=0,endCount=0;
  String startText="서울시";
  String endText="동";
  for(String address:addresses) {
    if(address.startsWith(startText)) {
      startCount++;
    }
    if(address.endsWith(endText)) {
      endCount++;
    }
  }
  System.out.println("Starts with "+startText+ " count is "+startCount);
  System.out.println("Ends with "+endText+" count is "+endCount);
}
```

addresses라는 String 객체에는 3개의 주소가 있다. 이 예제는 "서울시"로 시작하는 주소와 "동"으로 끝나는 주소의 개수를 확인하여 그 결과를 출력한다. main() 메소드에서 이 메소드만 수행하도록 변경한 후 컴파일 및 실행을 해보자.

과연 결과는 어떻게 나올까?

"서울시"로 시작하는 문자열은 두개가 있으며, 모든 주소는 동으로 끝난다. 따라서, 결과는 다음과 같이 출력된다.


```
Starts with 서울시 count is 2
Ends with 동 count is 3
```

이렇게 startsWith() 메소드와 endsWith() 메소드를 사용하면 여러분들이 원하는 값이 해당 문자열에 있는지 쉽게 확인할 수 있다.

그러면, 중간에 있는 값은 어떻게 확인 가능할까? 그래서 존재하는 것이 contains() 메소드다. 이 메소드는 매개 변수로 넘어온 값이 문자열에 존재하는지를 확인한다. 그 다음에 있는 matches() 메소드는 contains() 메소드와 비슷하긴 하지만, 매개 변수로 넘어오는 값이 "정규 표현식Regular Expression"으로 되어 있어야만 한다. 정규 표현식이라는 것은 여러분들이 이메일을 점검하거나, 웹 페이지의 URL을 점검하는 등의 작업을 쉽게 하기 위해서 공식에 따라 만든 식을 말한다. 자바에서는 JDK 1.4부터 정규 표현식을 제공하며, java.util.regex 패키지의 Pattern 클래스 API에 있는 내용을 잘 읽어보면 많은 정보를 확인할 수 있다. 정규 표현식에 대해서는 책 한권으로 정리해도 부족할 정도이며, 인사이트의 〈손에 잡히는 정규 표현식〉이라는 책을 참고하기 바란다.

그러면 String 클래스의 contains() 메소드가 어떻게 사용되는지, 방금 사용한 addressCheck() 메소드를 수정하여 확인해 보자.

```
public void addressCheck() {
  String addresses[]=new String[]{
      "서울시 구로구 신도림동",
      "경기도 성남시 분당구 정자동",
      "서울시 구로구 개봉동",
  };
  int startCount=0,endCount=0;
  int containCount=0;
  String startText="서울시";
  String endText="동";
  String containText="구로";
  for(String address:addresses) {
    if(address.startsWith(startText)) {
      startCount++;
    }
    if(address.endsWith(endText)) {
      endCount++;
```

```
    }
    if(address.contains(containText)) {
      containCount++;
    }
  }
  System.out.println("Starts with "+startText+" count is "+startCount);
  System.out.println("Ends with "+endText+" count is "+endCount);
  System.out.println("Contains "+containText+" count is "+containCount);
}
```

굵은 글씨로 되어 있는 부분이 추가된 내용이다. 결과가 어떻게 나올까?

여러분들이 예상하는 대로 "구로"를 포함한 문자열의 개수는 두개이다. 따라서, 결과도 다음과 같이 나온다.

```
Starts with 서울시 count is 2
Ends with 동 count is 3
Contains 구로 count is 2
```

이 예제를 보면, "아니 뭐 그냥 눈으로 살펴보면 되겠구만. 뭐 이런 걸 힘들게 메소드로 만들까?"라는 생각을 할 수도 있다. 지금의 예제에서는 달랑 3개의 문자열이지만, 몇천, 몇만 개의 데이터를 확인해야 하는 경우도 발생한다. 그럴 때 사용하기 위한 것이다.

`regionMatches()`라는 메소드는 문자열 중에서 특정 영역이 매개 변수로 넘어온 문자열과 동일한지를 확인하는 데 사용된다. 이 메소드의 종류를 다시 보자.

리턴 타입	메소드 이름 및 매개 변수
boolean	regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)
boolean	regionMatches(int toffset, String other, int ooffset, int len)

매개 변수 목록을 보면 알겠지만, 하나는 대소문자를 구분할지 여부를 지정할 수 없고, 다른 하나는 그런 지정 자체가 불가능하다. 이 `regionMatches()` 메소드의 매개 변수로 넘어오는 값이 많으니, 매개 변수가 5개인 메소드를 기준으로 각각이 어떤 값을 뜻하는지 확인해 보자.

매개 변수	의미
ignoreCase	true일 경우 대소문자 구분을 하지 않고, 값을 비교한다.
toffset	비교 대상 문자열의 확인 시작 위치를 지정한다.
other	존재하는지를 확인할 문자열을 의미한다.
ooffset	other 객체의 확인 시작 위치를 지정한다.
len	비교할 char의 개수를 지정한다.

처음 보는 분들은 무슨 말인지 예제가 없으면 이해하기 어려울 것이다. 다음의 예제를 통해서 확인해 보자.

```
public void matchCheck() {
  String text="This is a text";
  String compare1="is";
  String compare2="this";
  System.out.println(text.regionMatches(2, compare1, 0, 1));
  //매개 변수가 4개인 메소드
  System.out.println(text.regionMatches(5, compare1, 0, 2));
  //매개 변수가 4개인 메소드
  System.out.println(text.regionMatches(true,0, compare2, 0, 4));
  //매개 변수가 5개인 메소드
}
```

예제 코드를 살펴보면 text 값이 있고, 비교할 compare1과 compare2가 있다. 이 메소드의 수행 결과가 제대로 나올지를 확인하려면, 각 char의 위치가 어디에 해당하는지를 알아야만 한다.

T	h	i	s		i	s		a		t	e	x	t
0	1	2	3	4	5	6	7	8	9	10	11	12	13

자바에서 String의 char 위치는 배열과 마찬가지로 1부터 시작하는 것이 아니라, 0부터 시작한다.

text 문장에 "is"라는 문자열이 시작되는 위치를 잘 확인해 보자. 처음 is가 나오는 것은 2번째 위치와 5번째 위치다.

```
text.regionMatches(2, compare1, 0, 1);
```

첫번째 `regionMatches()` 메소드를 자세히 살펴보자. 첫번째 매개 변수인 2라는 위치 값은 제대로 맞았다. 비교하려고 하는 compare1의 값이 "is"다. 그런데, 세번째 매개 변수가 0이고, 네번째 매개 변수가 1이기 때문에 비교하려는 것은 "i"인지 아닌지만 확인하면 된다. 따라서 첫번째 출력문의 결과는 true이다.

동일한 방법으로 두번째, 세번째 출력문의 결과를 예상해 보자.

모든 조건이 다 맞는다. 따라서 `main()` 메소드에서 이 메소드만 수행하도록 해 놓은 후 실행한 결과는 다음과 같다.

```
true
true
true
```

그런데 `regionMatches()` 메소드를 잘못 사용하면, 원하는 결과를 얻지 못할 수도 있다. 이 메소드의 선언을 다시 보자.

```
regionMatches(boolean ignoreCase, int toffset, String other,
   int ooffset, int len)
```

여러 매개 변수 중에서 값이 다음과 같은 경우에는 결과가 "무조건 false"로 나온다.

- toffset이 음수일 때
- ooffset이 음수일 때
- toffset+len이 비교 대상의 길이보다 클 때
- ooffset+len이 other 객체의 길이보다 클 때
- ignoreCase가 false인 경우에는 비교 범위의 문자들 중 같은 위치(index)에 있는 char가 다를 때,
- ignoreCase가 true인 경우에는 비교 범위의 문자들을 모두 소문자로 변경한 후 같은 위치(index)에 있는 char가 달라야 한다.

이번 절에서는 String 문자열의 내용을 비교하고 검색하는 메소드들을 알아보았다. 여기에 있는 메소드들을 꼭 외울 필요는 없다. 여러분들이 개발하다 보면 자연스레 외어질 것이다. 하지만, 어떤 기능의 메소드가 있는지 정도는 기억해 두고 다음 절로 넘어가자.

String 내에서 위치를 찾아내는 방법은 여러 가지에요

다음과 같은 문장이 있다.

"Java technology is both a programming language and a platform."

이 문장에서 "both"라는 단어가 시작하는 위치를 알고 싶을 때 어떻게 해야 할까? 자바의 `String` 클래스에서는 indexOf라는 단어가 포함되어 있는 메소드를 제공한다. 이 메소드를 사용하면 해당 객체의 특정 문자열이나 char가 있는 위치를 알 수 있다. 만약 여러분들이 찾고자 하는 문자열이나 char가 없으면 이 메소드는 -1을 리턴한다.

그럼 위치를 찾는 메소드에는 어떤 것들이 있는지 살펴보자.

리턴 타입	메소드 이름 및 매개 변수
int	indexOf(int ch)
	indexOf(int ch, int fromIndex)
	indexOf(String str)
	indexOf(String str, int fromIndex)
	lastIndexOf(int ch)
	lastIndexOf(int ch, int fromIndex)
	lastIndexOf(String str)
	lastIndexOf(String str, int fromIndex)

`indexOf()` 메소드는 `String` 클래스의 가장 많이 사용되는 메소드 중 하나다. 그만큼 중요하고 많이 사용되기 때문에 꼭 사용법을 알고 있어야만 한다. 메소드들을 살펴보면, 종류는 크게 `indexOf()`와 `lastIndexOf()`의 두 가지로 나뉜다. `indexOf()`는 앞에서부터(가장 왼쪽부터) 문자열이나 char를 찾으며, `lastIndexOf()`는 뒤에서부터(가장 오른쪽부터) 찾는다. 그런데, String을 매개 변수로 갖는 메소드는 이해가 되는데, int를 매개 변수로 갖는 메소드는 어떻게 사용해야 할까? char는 정수형이다. 따라서, 여러분들이 이 메소드의 매개 변수로 char를 넘겨주면 자동으로 형 변환이 일어나기 때문에 걱정하지 않아도 된다.

먼저 `indexOf()` 메소드의 예를 통해서 알아보자. `StringSample` 클래스에 `indexOfCheck()`를 다음과 같이 추가하고, `main()` 메소드에서 이 메소드만 호출하도록 변경하자.

```
public void indexOfCheck() {
  String text=
    "Java technology is both a programming language and a platform.";
  System.out.println(text.indexOf('a')); —❶
  System.out.println(text.indexOf("a ")); —❷
  System.out.println(text.indexOf('a',20)); —❸
  System.out.println(text.indexOf("a ",20));
  System.out.println(text.indexOf('z')); —❹
}
```


첫 번째 `indexOf()`부터 살펴보자.

❶ 필자가 말한 대로, 'a'의 형태로 매개 변수를 넘겨주어도 컴파일 및 실행하는 데 전혀 문제가 발생하지 않는다.

❷ String 타입의 매개 변수를 넘겨 주었으며, a 뒤에는 공백이 하나 있다.

❸ 세번째와 네번째 출력문은 모두 text 문자열의 20번째 자리부터 값을 확인한다.

❹ 마지막에 있는 출력문은 이 문장에 없는 "z"를 찾은 결과를 출력한다.

혹시나 해서 다시 한번 말하지만, 자바에서의 index는 0부터 시작한다. 따라서 Java의 J 앞은 0, J와 a 사이는 1이다. 종이로 위치를 확인할 때에는 그 사이를 펜으로 꾹꾹 눌러서 확인해 보면 편하고, PC에서 확인할 때에는 커서를 맨 앞으로 옮겨 놓고 우측 방향 키를 한번씩 누르면서 위치를 확인 게 제일 편하다.

그러면, 이 메소드의 출력 결과는 어떻게 나올까?

결과는 다음과 같다.

```
1
3
24
24
-1
```

Java의 a 중 첫번째 a가 가장 앞에 있는 a이므로 첫 출력문의 결과는 1이다. 두번째 출력문은 a 다음에 공백이 있는 위치를 찾는 것이므로, Java 단어 뒤에 공백이 있으므로 3이다. 그리고, 세번째와 네번째의 결과는 20번째 자리부터 찾는 작업을 하고, 제일 처음 나온 관사 a의 위치가 24번째이므로 결과가 동일하다. 마지막 `indexOf()`의 경우 z가 문장에 없으므로 -1을 출력하였다.

이번에는 문자열의 가장 뒤부터(오른쪽부터) 검색을 하는 `lastIndexOf()` 메소드의 예제를 살펴보자.

```
public void lastIndexOfCheck() {
  String text=
  "Java technology is both a programming language and a platform.";
  System.out.println(text.lastIndexOf('a'));
  System.out.println(text.lastIndexOf("a "));
  System.out.println(text.lastIndexOf('a',20));
  System.out.println(text.lastIndexOf("a ",20));
  System.out.println(text.lastIndexOf('z'));
}
```

방금 살펴본 `indexOfCheck()` 메소드 예제와 다른 것은 메소드 이름과 각 출력문에 있는 indexOf가 모두 lastIndexOf로 바뀐 것밖에 없다. `main()` 메소드에서 이 메소드만 수행하도록 한 후, 이 예제를 실행하자.

결과는 어떻게 나올까? 참고로, 이 text 문자열의 길이는 62다. 따라서, 가장 우측에 있는 점(.)의 오른쪽 위치가 62다.

`indexOf()` 메소드와 `lastIndexOf()` 메소드가 다른 점은 검색 위치만 다른 것이기 때문에, 별도의 설명은 하지 않아도 이해하리라 생각된다. 결과를 보자.

```
55
51
3
3
-1
```

예상한 대로 platform의 a의 위치, 관사 a의 마지막 위치, Java의 뒤에 있는 a 위치 등을 출력한 것을 볼 수 있다. 그리고, z는 이 문장에 포함되어 있지 않으므로, -1을 출력했다. 그런데, `lastIndexOf()` 메소드의 검색 시작 위치(fromIndex)는 어떤 값을 말하는 것일까? 여기서 시작 위치는 가장 왼쪽에서부터의 위치를 말한다. 그 위치부터 왼쪽으로 값을 찾는다.

String의 값의 일부를 추출하기 위한 메소드들은 얘네들이다

문자열의 위치를 찾는 이유는 여러 가지다. 보통 그 위치부터 어떤 값을 추출해 내거나, 그 값이 존재하는지를 확인할 때 사용한다. 그 중에서 특정 값을 추출할 때 사용하는 메소드들을 이 절에서 알아보자. 값을 추출하는 메소드의 종류는 다음과 같이 구분할 수 있다.

- char 단위의 값을 추출하는 메소드
- char 배열의 값을 String으로 변환하는 메소드
- String의 값을 char 배열로 변환하는 메소드
- 문자열의 일부 값을 잘라내는 메소드
- 문자열을 여러 개의 String 배열로 나누는 메소드

먼저 특정 위치의 char 값을 추출하는 메소드를 알아보자.

char 단위의 값을 추출하는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
char	charAt(int index)	특정 위치의 char 값을 리턴한다.
void	getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)	매개 변수로 넘어온 dst라는 char 배열 내에 srcBegin에서 srcEnd에 있는 char를 저장한다. 이때, dst 배열의 시작 위치는 dstBegin이다.
int	codePointAt(int index)	특정 위치의 유니코드 값을 리턴한다. 리턴 타입은 int지만, 이 값을 char로 형 변환하면 char 값을 출력할 수 있다.
int	codePointBefore(int index)	특정 위치 앞에 있는 char의 유니코드 값을 리턴한다. 리턴 타입은 int지만, 이 값을 char로 형 변환하면 char 값을 출력할 수 있다.
int	codePointCount(int beginIndex, int endIndex)	지정한 범위에 있는 유니코드 개수를 리턴한다.
int	offsetByCodePoints(int index, int codePointOffset)	지정된 index부터 오프셋(offset)이 설정된 인덱스를 리턴한다.

마지막에 있는 `offsetByCodePoints()` 메소드는 문자열 인코딩과 관련된 문제를 해결하기 위해서 사용되며, 아래 사이트를 참고하면 많은 도움이 될 것이다.

http://www.ibm.com/developerworks/kr/library/j-unicode/index.html

그런데, 여기에 있는 메소드들은 그리 많이 사용되지는 않는다. 그나마 많이 사용하는 메소드가 charAt() 메소드다. 앞 절에서 살펴본 예제에서 charAt() 메소드를 사용하여, indexOf()로 탐색한 위치에 있는 값을 확인하는 작업을 한번 직접 해 보자.

혹시나 해서 이야기하지만, 한글의 경우는 한 글자로 인식된다.

char 배열의 값을 String으로 변환하는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
static String	copyValueOf(char[] data)	char 배열에 있는 값을 문자열로 변환한다.
static String	copyValueOf(char[] data, int offset, int count)	char 배열에 있는 값을 문자열로 변환한다. 단 offset 위치부터 count까지의 개수만큼만 문자열로 변환한다.

copyValueOf()라는 메소드도 그리 많이 사용하지는 않는다. 그래도 이러한 메소드가 있는지 알고 있는 것은 많은 도움이 된다. 단, 이 메소드는 static 메소드이기 때문에 현재 사용하는 문자열을 참조하여 생성하는 것이 아닌, static하게 호출하여 사용해야 한다. 간단하게 어떻게 사용하는지, 다음의 코드를 보면 이해가 쉬울 것이다.

```
char values[]=new char[]{'J','a','v','a'};
String javaText=String.copyValueOf(values);
```

String의 값을 char 배열로 변환하는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
char[]	toCharArray()	문자열을 char 배열로 변환하는 메소드

여러분들이 어떤 String 객체를 만들더라도, 그 객체는 내부에 char 배열을 포함한다. 방금 전에 살펴본 copyValueOf() 메소드에서 "Java"라는 값이 char 배열로 저장되어 있듯이, String 객체 내부에는 항상 이러한 char 배열이 포함되어 있다는 말이다. 이 메소드도 그리 많이 사용되지는 않고, 사용법이 매우 간단하므로, 여러분들이 직접 char 배열로 변환한 후 그 값들을 출력해보자.

문자열의 일부 값을 잘라내는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
String	substring(int beginIndex)	beginIndex부터 끝까지 대상 문자열을 잘라 String으로 리턴한다.
String	substring(int beginIndex, int endIndex)	beginIndex부터 endIndex까지 대상 문자열을 잘라 String으로 리턴한다.
CharSequence	subSequence(int beginIndex, int endIndex)	beginIndex부터 endIndex까지 대상 문자열을 잘라 CharSequence 타입으로 리턴한다.

이번에 설명하는 메소드들은 자바에서 문자열을 다룰 때 indexOf() 메소드와 더불어 가장 많이 사용하는 메소드 중 하나다. 그래서, 꼭 사용법을 알아두어야만 한다. 다음의 예제를 보자.

```
public void substringCheck1() {
  String text="Java technology";
}
```

"Java technology"라는 문자열이 있을 때, "technology"라는 단어만 추출하려고 할 때 다음과 같이 substring() 메소드를 사용하면 된다.

```
public void substringCheck1() {
  String text="Java technology";
  String technology=text.substring(5);
  System.out.println(technology);
}
```

text라는 문자열의 t가 시작하는 위치가 5번째라는 것은 이미 여러 번 설명했으니 잘 알고 있을 것이다. 이 예제에 있는 것처럼 substring() 메소드에 5라는 하나의 정수 값만 지정하면 5번째부터 text 문자열이 끝날 때까지를 모두 잘라내라는 의미다. 이 메소드를 컴파일하고, main() 메소드에서 이 메소드만 수행하도록 변경한 후 실행한 결과는 다음과 같다.

```
technology
```

그런데, 만약 여러분들이 tech라는 단어만 잘라내고 싶을 때에는 어떻게 해야 할까? 그럴 때에는 앞에 있는 표의 두번째나 세번째에 있는 메소드를 사용하면 된다. 다음과 같이 두 줄을 추가한 후 컴파일하고 실행해보자.

```
public void substringCheck1() {
  String text="Java technology";
  String technology=text.substring(5);
  System.out.println(technology);
  String tech=text.substring(5, 4);
  System.out.println(tech);
}
```

tech가 4개의 알파벳으로 되어 있는 문자이기 때문에 두번째 index를 4로 지정했다.

결과가 어떻게 나올까?

아쉽게도 결과는 다음과 같이 예외가 발생한다.

```
technology
Exception in thread "main" java.lang.StringIndexOutOfBoundsExcepti
on: String index out of range: -1
  at java.lang.String.substring(String.java:1937)
  at d.string.StringSample.substringCheck1(StringSample.java:157)
  at d.string.StringSample.main(StringSample.java:17)
```

"왜 이런 예외가 발생하지?"라고 생각하는 독자도 있을 것이다. 앞에서 설명한 표를 잘 보면 첫번째 매개 변수는 beginIndex이고, 두번째 매개 변수는 데이터의 길이가 아닌 substring이 끝나는 위치를 말한다. 따라서, 다음과 같이 끝나는 위치를 5+4인 9로 지정해야만 한다.

```
String tech=text.substring(5,9);
```


이렇게 변경한 후 컴파일 및 실행을 해보면 다음과 같이 원하는 결과가 출력된다.

```
technology
tech
```

이와 같이 substring() 메소드를 사용하면 문자열에 있는 특정 단어를 마음대로 잘라낼 수 있다. 실제로 여러분들이 이 substring() 메소드를 사용할 때에는 보통 indexOf() 메소드와 같이 사용하여 문자열을 잘라내야만 한다. 이렇게 indexOf() 메소드와 함께 substring() 메소드를 사용하는 방법은 이 장의 뒤에 있는 "직접해 봅시다"에서 살펴보자.

문자열을 여러 개의 String 배열로 나누는 split 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
String[]	split(String regex)	regex에 있는 정규 표현식에 맞추어 문자열을 잘라 String의 배열로 리턴한다.
String[]	split(String regex, int limit)	regex에 있는 정규 표현식에 맞추어 문자열을 잘라 String의 배열로 리턴한다. 이때 String 배열의 크기는 limit 보다 커서는 안 된다.

자바에서 문자열을 여러 개의 문자열의 배열로 나누는 방법은 String 클래스에 선언된 split() 메소드를 사용하는 것과 java.util 패키지에 선언되어 있는 StringTokenizer라는 클래스를 사용하는 것이다. 만약 여러분들이 정규 표현식을 사용하여 문자열을 나누려고 한다면 String 클래스의 split() 메소드를 사용하면 된다. 그렇지 않고, 그냥 특정 String으로 문자열을 나누려고 한다면 StringTokenizer 클래스를 사용하는 것이 편하다. 특정 알파벳이나 기호 하나로 문자열을 나누려고 한다면 String 클래스의 split() 메소드를 쓰든지, StringTokenizer 클래스를 쓰든지 큰 상관 없다. StringTokenizer 클래스에 대한 설명은 8장의 "java.util"에서 자세히 다루니 참고하기 바란다.

다음의 예를 통해서 split 메소드와 친해지자.

```
public void splitCheck() {
  String text=
    "Java technology is both a programming language and a platform.";
      String[] splitArray=text.split(" ");
      for(String temp:splitArray) {
        System.out.println(temp);
  }
}
```

예제는 아주 간단하다. text에 있는 문자열을 공백space으로 나누어 splitArray라는 String 배열에 담았다. 그리고 나서 배열에 있는 각각의 값들을 출력했다. main() 메소드에서 이 메소드만 호출하도록 한 후 실행해보자. 결과는 다음과 같다.

```
Java
technology
is
both
a
programming
language
and
a
platform.
```

이번 절에서는 문자열의 내용을 자르는 split() 메소드에 대해서 살펴 보았다. 하지만, String 클래스에서 제공하는 메소드는 여기서 끝나지 않는다.

String 값을 바꾸는 메소드들도 있어요

String API 설명의 마지막으로, String 값을 바꾸는 메소드를 알아보자. 문자열의 값을 바꾸고, 변환하는 메소드도 다음과 같이 구분할 수 있다.

- 문자열을 합치는 메소드와 공백을 없애는 메소드
- 내용을 교체replace하는 메소드

- 특정 형식에 맞춰 값을 치환하는 메소드
- 대소문자를 바꾸는 메소드
- 기본 자료형을 문자열로 변환하는 메소드

먼저 가장 앞에 있는 분류의 메소드를 살펴보자.

문자열을 합치는 메소드와 공백을 없애는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
String	concat(String str)	매개 변수로 받은 str을 기존 문자열의 우측에 붙인 새로운 문자열 객체를 생성하여 리턴한다.
String	trim()	문자열의 맨 앞과 맨 뒤에 있는 공백들을 제거한 문자열 객체를 리턴한다.

솔직히 말해서 필자는 자바에 concat()이라는 메소드가 있는지 이 장을 쓰면서 처음 알았다. 그만큼 자바에서는 concat() 메소드를 쓸 일이 없다. 이렇게 concat() 메소드를 사용하여 문자열을 계속 더할 일이 있다면, 이 장의 끝에 있는 StringBuffer나 StringBuilder 클래스를 사용하여 문자열을 더하기 바란다.

trim() 메소드는 공백을 제거할 때 매우 유용하게 많이 사용된다. 문자열의 앞과 뒤에 있는 공백을 일일이 찾아서 지워버릴 필요가 없이 이 메소드만 사용하면 되기 때문이다. trim() 메소드는 아주 많이 사용되므로 다음의 예제를 통해서 확실히 어떻게 동작하는지 익혀두기 바란다.

```
public void trimCheck() {
  String strings[]=new String[]{
    " a"," b ","    c","d    ","e   f","   "
  };
  for(String string:strings) {
    System.out.println("["+string+"] ");
    System.out.println("Trim["+string.trim()+"] ");
  }
}
```

여러분들이 책으로 보기에는 잘 안보일 수도 있겠지만, a 앞에 공백, b 앞뒤에 공백, c 앞에 여러 공백, d 뒤에 여러 공백, e와 f 사이에 여러 공백, 그리고 마지막에는 공백만 존재하는 문자열이 존재한다. 그리고 나서, 원래 문자열과 `trim()` 메소드를 적용한 문자열을 출력한다.

각각 어떤 값이 나올지 생각해보자.

결과는 다음과 같다.

```
[ a]
Trim[a]
[ b ]
Trim[b]
[    c]
Trim[c]
[d    ]
Trim[d]
[e   f]
Trim[e   f]
[    ]
Trim[]
```

이와 같이 대괄호로 값들을 감싸서 보면 원래 값과 `trim()`한 값의 차이를 쉽게 확인할 수 있을 것이다. `trim()` 메소드의 용도는 매우 많지만, 작업하려는 문자열이 공백만으로 이루어진 값인지, 아니면 공백을 제외한 값이 있는지 확인하기에 매우 편리하다. 다음의 if 문을 통과하여 "OK"를 출력하면, 해당 문자열은 공백을 제외한 char 값이 하나라도 존재한다는 의미다.

```
String text=" a ";
if(text!=null && text.trim().length() >0 ) {
  System.out.println("OK");
}
```


만약 여러분들이 null 체크를 하지 않으면, 값이 null인 객체의 메소드를 호출하면 NullPointerException이 발생한다. 따라서, 이와 같이 String을 조작하기 전에 null 체크하는 습관을 갖자.

내용을 교체(replace)하는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
String	replace(char oldChar, char newChar)	해당 문자열에 있는 oldChar의 값을 newChar로 대치한다.
String	replace(CharSequence target, CharSequence replacement)	해당 문자열에 있는 target과 같은 값을 replacement로 대치한다.
String	replaceAll(String regex, String replacement)	해당 문자열의 내용 중 regex에 표현된 정규 표현식에 포함되는 모든 내용을 replacement로 대치한다.
String	replaceFirst(String regex, String replacement)	해당 문자열의 내용 중 regex에 표현된 정규 표현식에 포함되는 첫번째 내용을 replacement로 대치한다.

replace로 시작하는 메소드는 문자열에 있는 내용 중 일부를 변경하는 작업을 수행한다. 참고로,이 메소드를 수행한다고 해서, 기존 문자열의 값은 바뀌지 않는다. 메소드의 설명만 봐도 어떻게 동작하는지 감이 잡히겠지만, 예제를 통해서 살펴보자.

```
public void replaceCheck() {
  String text="The String class represents character strings.";
  System.out.println(text.replace('s', 'z')); —❶
  System.out.println(text); —❷
  System.out.println(text.replace("tring", "trike")); —❸
  System.out.println(text.replaceAll(" ", "|")); —❹
  System.out.println(text.replaceFirst(" ", "|")); —❺
}
```

❶ text 객체에 있는 char 's'를 'z'로 변환하는 작업을 수행한다.

❷ replace() 메소드를 수행한 후에 기존의 값이 변경되는지를 확인하기 위해서 현재 값을 출력한다.

❸ CharSequence 타입의 매개 변수를 사용하여 값을 변경한다.

❹ 정규 표현식을 사용하는 replaceAll() 메소드의 예이며, 공백을 파이프(|)로 변환한다.

❺ replaceFirst() 메소드를 사용하여 첫번째 공백만 파이프로 변환한다.

이 replaceCheck() 메소드를 수행한 결과가 어떻게 나오는지 생각해보자.

출력되는 결과는 다음과 같다.

```
The String clazz reprezentz character ztringz.
The String class represents character strings.
The Strike class represents character strikes.
The|String|class|represents|character|strings.
The|String class represents character strings.
```

아마도 여러분들의 생각을 크게 벗어나지는 않았을 것이다. 하지만, 첫번째 결과를 보면 소문자 s만 z로 변환된 것을 볼 수 있다. 즉, replace 관련 메소드는 대소문자를 구분하니 참고하기 바란다. 그리고, 자칫 잘못하면 변환하려고 생각하지도 않은 값이 변환되어 버릴 수도 있다. 따라서, replace 관련 메소드를 사용할 때에는 생각을 잘 하고 적용하기 바란다.

특정 형식에 맞춰 값을 치환하는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
static String	format(String format, Object... args)	format에 있는 문자열의 내용 중 변환해야 하는 부분을 args의 내용으로 변경한다.
static String	format(Locale l, String format, Object... args)	format에 있는 문자열의 내용 중 변환해야 하는 부분을 args의 내용으로 변경한다. 단 첫 매개 변수인 Locale 타입의 l에 선언된 지역에 맞추어 출력한다.

format() 메소드는 정해진 기준에 맞춘 문자열이 있으면, 그 기준에 있는 내용을 변환한다. 자바에서 %s는 String을, %d는 정수형을, %f는 소수점이 있는 숫자, %%는 %를 의미한다. 그 외에도 다양한 기준이 있는데, 이 format에 관련된 내용은 9장의 "Formatter"를 참고하기 바란다. 아주 간단한 예를 통해서 format() 메소드의 사용법을 알아보자.


```
public void formatCheck() {
  String text="제 이름은 %s입니다. 지금까지 %d 권의 책을 썼고, "
    +"하루에 %f %%의 시간을 책을 쓰는데 할애하고 있습니다.";
  String realText=String.format(text, "이상민",4,10.5);
  //String realText=String.format(text, "이상민",4);
  System.out.println(realText);
}
```

아주 간단한 예다. 이렇게 format() 메소드를 사용하면 다음과 같은 결과가 출력된다.

```
제 이름은 이상민입니다. 지금까지 4권의 책을 썼고, 하루에 10.500000%의 시간을 책을 쓰는데 할애하고 있습니다.
```

그런데, 만약 여러분들이 출력만을 위해서 이 메소드를 사용하면 굳이 이렇게 사용할 필요는 없다. System.out.println() 메소드도 있지만, System.out.format() 이라는 메소드도 있기 때문이다. 그리고, 여기서 여러분들이 주의해야 할 사항이 하나 있다. 대치해야 할 문자열이 3개인데, format 뒤에 3개 이상의 매개 변수를 나열하는 것은 상관 없다. 하지만, 2개 이하로 매개 변수만 명시하면 실행시 예외가 발생한다. 다음과 같이 가장 뒤에 있는 10.5 값을 지운 예제를 실행해보자.

```
public void formatCheck() {
  String text="제 이름은 %s입니다. 지금까지 %d 권의 책을 썼고, "
    +"하루에 %f %%의 시간을 책을 쓰는데 할애하고 있습니다.";
  //String realText=String.format(text, "이상민",4,10.5);
  String realText=String.format(text, "이상민",4);
  System.out.println(realText);
}
```

그러면 다음과 같이 예외가 출력된다.

```
Exception in thread "main" java.util.MissingFormatArgumentException:
Format specifier 'f'
  at java.util.Formatter.format(Formatter.java:2432)
  at java.util.Formatter.format(Formatter.java:2367)
  at java.lang.String.format(String.java:2769)
  at d.string.StringSample.formatCheck(StringSample.java:242)
  at d.string.StringSample.main(StringSample.java:24)
```

참고

Locale은 지역적으로 다른 표현 형식을 제공하기 위한 것이다. 보통 Locale을 지정하지 않으면 기본적으로 자바 프로그램이 수행되는 OS의 지역 정보를 기본으로 따른다.

대소문자를 바꾸는 메소드

리턴 타입	메소드 이름 및 매개 변수	설명
String	toLowerCase()	모든 문자열의 내용을 소문자로 변경한다.
String	toLowerCase(Locale locale)	지정한 지역 정보에 맞추어 모든 문자열의 내용을 소문자로 변경한다.
String	toUpperCase()	모든 문자열의 내용을 대문자로 변경한다.
String	toUpperCase(Locale locale)	지정한 지역 정보에 맞추어 모든 문자열의 내용을 대문자로 변경한다.

toLower로 시작하는 메소드는 모든 대문자를 소문자로, toUpper로 시작하는 메소드는 모든 소문자를 대문자로 변경하는 메소드다. 사용법이 그리 어렵지 않으므로 별도의 예제는 다루지 않겠다. 직접 확인해 보기 바란다.

기본 자료형을 문자열로 변환하는 메소드

리턴 타입	메소드 이름 및 매개 변수
static String	valueOf(boolean b)
static String	valueOf(char c)
static String	valueOf(char[] data)
static String	valueOf(char[] data, int offset, int count)
static String	valueOf(double d)

이어짐

리턴 타입	메소드 이름 및 매개 변수
static String	valueOf(float f)
static String	valueOf(int i)
static String	valueOf(long l)
static String	valueOf(Object obj)

여기에 나열된 메소드들은 기본 자료형을 String 타입으로 변환한다. 이 valueOf() 메소드를 사용하여 기본 자료형 값들을 문자열로 변경해도 되지만, 다음과 같이 변환해도 된다.

```
byte b=1;
String byte1=String.valueOf(b);
String byte2=b+"";
```

이렇게 byte1처럼 변환하나, byte2처럼 변환하나 출력해 보면 동일한 값이 출력된다. 다시 말해서 대부분 기본 자료형을 String 타입으로 변환할 필요가 있을 때에는 String과 합치는 과정을 거친다. 그럴 경우에는 별도로 valueOf() 메소드를 사용할 필요까지는 없다. 하지만, String으로 변환만 해 놓고 별도의 문자열과 합치는 과정이 없을 경우에는 valueOf() 메소드를 사용하는 것을 권장한다.

그리고, 또 한 가지 기억해야 하는 것이 있는데, 바로 valueOf() 메소드의 매개 변수로 객체Object가 넘어왔을 경우이다. 만약 여러분들이 toString()을 구현한 객체나, 정상적인 객체를 valueOf() 메소드에 넘겨주면 toString()의 결과를 리턴해준다. 하지만, null인 객체의 경우에는 이야기가 달라진다. null인 객체는 toString() 메소드를 사용할 수 없다. 이 장의 앞부분에서 살펴본 것처럼, NullPointerException이 발생해버린다. 그러한 결과를 방지하기 위해서는 객체를 출력할 때 valueOf() 메소드를 사용하면 좋다. valueOf() 메소드는 객체가 null이면 "null"이라는 문자열을 리턴해주기 때문이다. 만약 null이 아니면, toString() 메소드를 호출한 결과가 리턴된다. System.out.print()나 System.out.println() 메소드에서 null인 객체를 출력했을 때 NullPointerException이 발생하지 않는 이유도 이 때문이다.

이번 절에서는 문자열의 내용을 변환하는 메소드들을 알아보았다. 지금까지 이 장에서 배운 메소드가 String 클래스의 대부분이라도 해도 무방하다. 메소드들의 이름

을 꼭 외울 필요는 없더라도, 어떤 작업을 수행하는 메소드가 있다는 것을 알아두는 것은 매우 중요하다.

절대로 사용하면 안 되는 메소드가 하나 있어요!!!!!

String 클래스에 있는 여러 메소드 중에서 초보인 여러분들이 절대 사용해서는 메소드가 있다. 바로 intern()이라는 메소드다. 이 메소드는 자바로 구현되지 않고 C로 구현되어 있는 native 메소드 중 하나다. native 메소드이기 때문에 쓰지 말라는 것이 아니고, 시스템의 심각한 성능 저하를 발생시킬 수도 있기 때문이다.

앞에서 다음과 같이 String 객체를 생성했을 때의 결과를 이야기했다.

```
public void internCheck() {
  String text1="Java Basic";
  String text2="Java Basic";
  String text3=new String("Java Basic");
  System.out.println(text1==text2);
  System.out.println(text1==text3);
  System.out.println(text1.equals(text3));
}
```

다시 한번 정리하는 차원에서, 이 internCheck() 메소드의 결과를 생각해보자.

결과는 다음과 같이 나온다.

```
true
false
true
```

text1과 text2와 같이 객체를 생성하면, String 클래스에서 관리하는 Constant 풀pool에 해당 값이 있으면 기존에 있는 객체를 참조하고, text3와 같이 String 객체를 생성하면 같은 문자열이 풀에 있든 말든 새로운 객체를 생성한다고 이야기했다.

그래서, 결과가 이와 같이 나타나는 것이다. 이 메소드의 네 번째 줄에 다음과 같이 한 줄을 추가하자.

```
public void internCheck() {
  String text1="Java Basic";
  String text2="Java Basic";
  String text3=new String("Java Basic");
  text3=text3.intern();
  System.out.println(text1==text2);
  System.out.println(text1==text3);
  System.out.println(text1.equals(text3));
}
```

이렇게 해 놓고 수행하면 결과가 어떻게 나올까?

결과는 예상 외로 다음과 같이 나온다.

```
true
true
true
```

왜 이러한 결과가 나올까? new String(String)으로 생성한 문자열 객체라고 할지라도, 풀에 해당 값이 있으면, 풀에 있는 값을 참조는 객체를 리턴한다. 만약 동일한 문자열이 존재하지 않으면 풀에 해당 값을 추가한다. 따라서, intern() 메소드를 수행한 뒤에 문자열은 equals() 메소드가 아닌, == 으로 동일한지 비교할 수가 있다.

equals() 메소드로 비교하는 것과 ==으로 비교하는 것의 성능 차이는 많다. ==으로 비교하는 것이 훨씬 빠르다. 그런데, 왜 필자가 이 메소드를 쓰지 말라고 하는 것일까? 만약 새로운 문자열을 쉴새 없이 만드는 프로그램에서 intern() 메소드를 사용하여 억지로 Constant 풀에 값을 할당하도록 만들면, 저장되는 영역은 한계가 있기 때문에 그 영역에 대해서 별도로 메모리를 청소하는 단계를 거치게 된다. 따라서, 작은 연산 하나를 빠르게 하기 위해서 전체 자바 시스템의 성능에 악영향을 주게 된다.

여러분들이 만드는 애플리케이션에서 생성하는 문자열이 정해져 있고, 그 문자열에 대해서만 intern() 메소드를 호출하여 사용할 경우에는 문제가 되지 않을 수도 있다. 하지만, 생성되는 문자열이 완전히 정해져 있는 시스템은 거의 없다. 따라서, 절대로 intern() 메소드는 사용하지 마라. 이 절의 모든 설명을 이해하지 못해도 되지만, intern() 메소드를 사용해서는 안 된다는 것만은 꼭 기억하기 바란다.

immutable한 String의 단점을 보완하는 클래스에는 StringBuffer와 StringBuilder가 있다

String은 immutable한 객체다. immutable이라는 말은 사전적인 의미로 "불변의"라는 의미다. 다시 말해서 한번 만들어지면 더 이상 그 값을 바꿀 수 없다. "무슨 말이야? 더하기 하면 잘만 더해지는데…". 아니다. 그 생각은 틀렸다. String 객체는 변하지 않는다. 만약 여러분들이 String 문자열을 더하면 새로운 String 객체가 생성되고, 기존 객체는 버려진다. 그러므로, 여러분들이 계속 하나의 String을 만들어 계속 더하는 작업을 한다면, 계속 쓰레기를 만들게 된다. 방금 이야기한 내용은 다음과 같은 경우를 말한 것이다.

```
String text="Hello";
text=text+" world";
```

이 경우, "Hello"라는 단어를 갖고 있는 객체는 더 이상 사용할 수 없다. 즉, 쓰레기가 되며, 나중에 GCGarbage collection(가비지 컬렉션)의 대상이 된다.

이러한 String 클래스의 단점을 보완하기 위해서 나온 클래스가 StringBuffer와 StringBuilder다. 두 클래스에서 제공하는 메소드는 동일하다. 하지만, StringBuffer는 Thread safe하다고 하며, StringBuilder는 Thread safe하지 않다고 한다. 아직 여러분들이 쓰레드라는 것에 대해서 배우지 않았기 때문에 무슨 말인지 이해가 되지는 않겠지만, 여하튼 기능은 같지만 StringBuffer가 StringBuilder 보다 더 안전하다고만 기억해 두기 바란다. 속도는 Thread safe하지 않는 StringBuilder 클래스가 더 빠르다.

StringBuffer와 StringBuilder 클래스는 문자열을 더하더라도 새로운 객체를 생성하지 않는다. 그렇다고 더하기(+) 기호를 사용하여 더할 수 있다는 말이 아니다. 이 두개의 클래스에서 가장 많이 사용하는 메소드는 append()라는 메소드다. append() 메소드는 매개 변수로 모든 기본 자료형과 참조 자료형을 모두 포함한다. 따라서, 어떤 값이라도 이 메소드의 매개 변수로 들어갈 수 있다.

보통 다음과 같이 사용한다.

```
StringBuilder sb=new StringBuilder();
sb.append("Hello");
sb.append(" world");
```

지금 언뜻 보기에는 별 차이가 없어 보일 것이다. 하지만, append() 메소드에 넘어가는 매개 변수가 이처럼 정해져 있는 문자열이라면 사용하나 마나지만, 매개 변수가 변수로 받은(항상 변하는) 값이라면 이야기는 달라진다. 그리고, 이 메소드는 다음과 같이 append() 메소드를 여러 개 붙여서 사용해도 무방하다.

```
StringBuilder sb=new StringBuilder();
sb.append("Hello").append(" world");
```

세미콜론이 나오기 전에 계속 append() 메소드를 붙여도 상관 없다. 왜냐하면, append() 메소드를 수행한 후에는 해당 StringBuilder 객체가 리턴되므로, 그 객체에 계속 붙이는 작업을 해도 무방한 것이다.

추가로, JDK 5 이상에서는 여러분들이 String의 더하기 연산을 할 경우, 컴파일할 때 자동으로 해당 연산을 StringBuilder로 변환해 준다. 따라서, 일일이 더하는 작업을 변환해 줄 필요는 없으나, for 루프와 같이 반복 연산을 할 때에는 자동으로 변환을 해주지 않으므로, 꼭 필요하다.

마지막으로 String과 StringBuilder, StringBuffer 클래스의 공통점에 대해서 알아보자. 공통점이라면, 모두 문자열을 다룬다는 점이다. 그런데, 또 다른 공통점은 CharSequence 인터페이스를 구현했다는 점이다. 따라서, 여러분들이 이 세 가지 중 하나의 클래스를 사용하여 매개 변수로 받는 작업을 할 때 String이나 StringBuilder 타입으로 받는 것보다는 CharSequence 타입으로 받는 것이 좋다.

그러면 언제 StringBuilder를 사용하고, 언제 `StringBuffer` 클래스를 사용해야 할까? 일반적으로 하나의 메소드 내에서 문자열을 생성하여 더할 경우에는 StringBuilder를 사용해도 전혀 상관 없다. 그런데, 어떤 클래스에 문자열을 생성하여 더하기 위한 문자열을 처리하기 위한 인스턴스 변수가 선언되었고, 여러 쓰레드에서 이 변수를 동시에 접근하는 일이 있을 경우에는 반드시 StringBuffer를 사용해야만 한다.

정리하며

이번 장에서는 자바에서 가장 많이 활용되는 `String` 클래스와 이 클래스에서 제공하는 메소드들에 대해서 아주 자세히 살펴보았다. `String` 클래스가 그만큼 자주 사용되고 중요하기 때문에 필자가 일부러 자세히 설명한 것이다. 앞으로 설명하는 클래스들은 이렇게까지 자세하게 설명하지 않고, 주요 클래스의 주요 메소드만 알아볼 예정이다.

`String` 클래스를 잘 사용해야만 메모리를 효율적으로 사용할 수 있고, 여러 String을 더하는 연산이 존재할 경우에는 `StringBuilder`나 `StringBuffer` 클래스를 적절하게 선택하여 활용해야만 한다.

직접해 봅시다

어떤 언어로 개발하든간에 문자열에서 필요한 데이터를 추출해 내는 작업은 매우 많이 사용되고, 중요하다. 따라서, 여러분들은 String 클래스에서 제공하는 메소드들을 이용하여 문자열을 떡 주무르듯이 다룰수 있어야 한다. 다음의 문장이 있다.

```
The String class represents character strings.
```

이 문장은 String 클래스의 API 문서에 있는 가장 첫 문장이다. 이 문장에서 여러분들이 필요한 데이터를 추출하는 것을 연습해보자.

1 API 문서에서 `String` 클래스를 찾아 필요할 때마다 참조할 수 있도록 하자.

2 d.string.practice 패키지에 `UseStringMethods`라는 클래스를 만들고, main() 메소드도 만들자.

3 `public void printWords(String str)`로 선언된 메소드를 만들자. 이 메소드는 str 문장의 단어들을 출력한다. 예제 문장을 str 값으로 전달하여, `main()` 메소드에서 이 메소드를 호출하여 결과를 확인해 보자.

힌트 | split() 메소드를 사용

결과

```
The
String
class
represents
character
strings.
```

직접해 봅시다

4 `public void findString(String str,String findStr)` 메소드를 만들자. 이 메소드는 str 중에서 findStr로 넘겨준 값과 동일한 단어의 첫번째 위치를 출력한다. 예제 문장을 str 값으로 전달하고, findStr에는 "string"을 넘겨주자. `main()` 메소드에서 이 메소드를 호출하여 결과를 확인해 보자.

힌트 | indexOf() 메소드를 사용

결과

```
string is appeared at 38
```

5 `public void findAnyCaseString(String str,String findStr)` 메소드를 만들자. 이 메소드는 str 중에서 findStr로 넘겨준 값과 "대소문자 구분 없이" 동일한 단어의 첫번째 위치를 출력한다. 예제 문장을 str 값으로 전달하고, findStr에는 "string"을 넘겨주자. `main()` 메소드에서 이 메소드를 호출하여 결과를 확인해 보자.

힌트 | toLowerCase(), indexOf() 메소드를 사용

결과

```
string is appeared at 4
```

6 `public void countChar(String str,char c)` 메소드를 만들자. 이 메소드는 str에서 c와 동일한 char의 개수를 출력한다. 예제 문장을 str 값으로 전달하고, c는 's'를 넘겨주자. `main()` 메소드에서 이 메소드를 호출하여 결과를 확인해 보자.

힌트 | toCharArray() 메소드 사용

결과

```
char 's' count is 6
```


직접해 봅시다

7 `public void printContainWords(String str,String findStr)` 메소드를 만들자. 이 메소드는 str 문자열에서 findStr이 포함된 단어를 출력한다. 예제 문장을 str 값으로 전달하고, findStr는 "ss"를 넘겨주자. `main()` 메소드에서 이 메소드를 호출하여 결과를 확인해 보자.

힌트 | split(), contains() 메소드 사용

결과

```
class contains ss
```

정리해 봅시다

문제에 대한 답은 아래에서 직접 문제를 푸시고 확인할 수 있습니다.
https://sites.google.com/site/godofjavabook/

1 String 클래스는 final 클래스인가요? 만약 그렇다면, 그 이유는 무엇인가요?

2 String 클래스가 구현한 인터페이스에는 어떤 것들이 있나요?

3 String 클래스의 생성자 중에서 가장 의미없는 (사용할 필요가 없는) 생성자는 무엇인가요?

4 String 문자열을 byte 배열로 만드는 메소드의 이름은 무엇인가요?

5 String 문자열의 메소드를 호출하기 전에 반드시 점검해야 하는 사항은 무엇인가요?

6 String 문자열의 길이를 알아내는 메소드는 무엇인가요?

7 String 클래스의 equals() 메소드와 compareTo() 메소드의 공통점과 차이점은 무엇인가요?

8 문자열이 "서울시"로 시작하는지를 확인하려면 String의 어떤 메소드를 사용해야 하나요?

정리해 봅시다

9 문자열에 "한국"이라는 단어의 위치를 찾아내려고 할 때에는 String의 어떤 메소드를 사용해야 하나요?

10 9번 문제의 답에서 "한국"이 문자열에 없을 때 결과값은 무엇인가요?

11 문자열의 1번째부터 10번째 위치까지의 내용을 String으로 추출하려고 합니다. 어떤 메소드를 사용해야 하나요?

12 문자열의 모든 공백을 * 표시로 변환하려고 합니다. 어떤 메소드를 사용하는 것이 좋을까요?

13 String의 단점을 보완하기 위한 두개의 클래스는 무엇인가요?

14 13번의 답에서 문자열을 더하기 위한 메소드의 이름은 무엇인가요?