1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Lecture 31: Strings and basic string operations

References and learning objects

- Today's sessions are largely based on ***Java: How to Program***, Chapter 14, and ***Java in a Nutshell***
- After today's session, you should be able to:
  - Use strings, string builders, and string operations in Java programs
  - Use basic regular expression (RegExp) operations
  - Use list and iterators to implement collections in Java programs

Introduction to strings

- In many Java programs, strings and string operations are essential
  - An instance of class **`String`** represents a string, i.e., a sequence of characters
  - Class **`String`** provides several methods to create and manipulate strings: we have already used some basic string operations in the earlier sessions
- `String` objects are immutable
  - The contents of `String` objects cannot be changed after creation; this is why many String methods actually create a copy of the original `String` that is manipulated

String constructors (1)

- Class **`String`** provides constructors for initialising **`String`** objects in a variety of different ways:
  - Without an argument, an empty string is created; however, since strings are immutable, empty strings are usually worthless:

```
String s0 = new String();
```

s0 = “”

  - You can use a constant string as an argument:

```
String s1 = new String("hello");
```

s1 = “hello”

  - You can use a `String` object as an argument to create a copy:

```
String s2 = new String(s1);
```

s2 = “hello”

String constructors (2)

- Class `String` provides also constructors accepting character or byte arrays as arguments:

```
char[] charArray = {'b','i','r','t','h',' ','d','a','y'};

String s3 = new String(charArray);

String s4 = new String(charArray, 6, 3);
```

s3 = “birth day”

s4 = “day”

The starting position (offset) where the characters in the array are accessed

The number of characters (count) to access

Initialising strings as literals

- In Java, `String` objects can be also created by assigning a *string literal* without keyword `new`:

```
String s = "hello";
```

- It should be noted that keyword `new` creates *always* a new `String` object, whereas strings created by literal will refer to an existing object, if similar string exists in the pool of string literals already
  - Since `String` objects are immutable, the difference is practically insignificant

Example of string literals (1)

```
1  public class StringLiteralExample {
2    public static void main(String[] args){
3      String s1 = "hello";
4      String s2 = new String("hello");
5      String s3 = "hello";
6      System.out.println("Are s1 and s2 same? " + (s1 == s2));
7      System.out.println("Are s1 and s3 same? " + (s1 == s3));
8      System.out.println("Are s2 and s3 same? " + (s2 == s3));
9    }
10 }
```

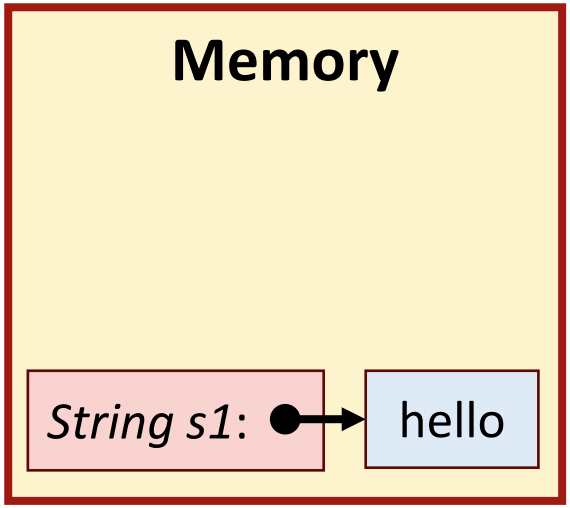
```
$ java StringLiteralExample
```

Memory

Example of string literals (2)

```
1  public class StringLiteralExample {
2    public static void main(String[] args){
3      String s1 = "hello";
4      String s2 = new String("hello");
5      String s3 = "hello";
6      System.out.println("Are s1 and s2 same? " + (s1 == s2));
7      System.out.println("Are s1 and s3 same? " + (s1 == s3));
8      System.out.println("Are s2 and s3 same? " + (s2 == s3));
9    }
10 }
```

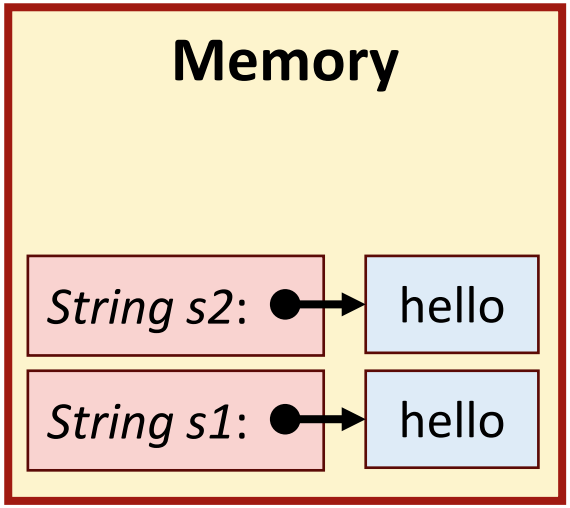
```
$ java StringLiteralExample
```



Example of string literals (3)

```
1  public class StringLiteralExample {
2    public static void main(String[] args){
3      String s1 = "hello";
4      String s2 = new String("hello");
5      String s3 = "hello";
6      System.out.println("Are s1 and s2 same? " + (s1 == s2));
7      System.out.println("Are s1 and s3 same? " + (s1 == s3));
8      System.out.println("Are s2 and s3 same? " + (s2 == s3));
9    }
10 }
```

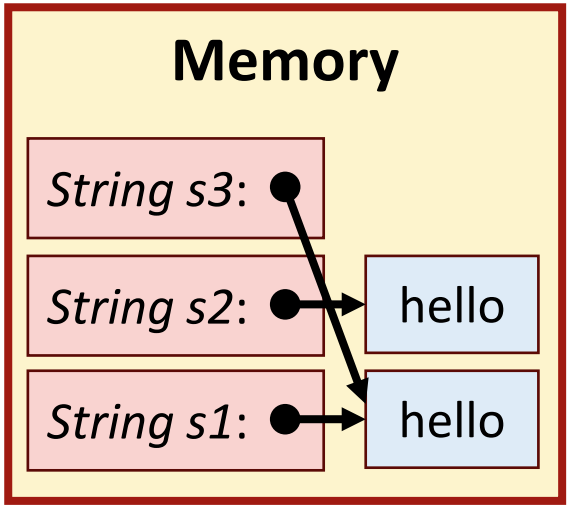
```
$ java StringLiteralExample
```



Example of string literals (3)

```
1  public class StringLiteralExample {
2    public static void main(String[] args){
3      String s1 = "hello";
4      String s2 = new String("hello");
5      String s3 = "hello";
6      System.out.println("Are s1 and s2 same? " + (s1 == s2));
7      System.out.println("Are s1 and s3 same? " + (s1 == s3));
8      System.out.println("Are s2 and s3 same? " + (s2 == s3));
9    }
10 }
```

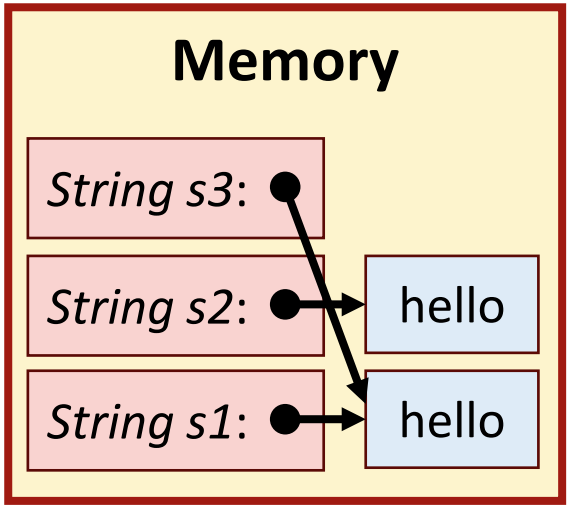
```
$ java StringLiteralExample
```



Example of string literals (4)

```
1  public class StringLiteralExample {
2    public static void main(String[] args){
3      String s1 = "hello";
4      String s2 = new String("hello");
5      String s3 = "hello";
6      System.out.println("Are s1 and s2 same? " + (s1 == s2));
7      System.out.println("Are s1 and s3 same? " + (s1 == s3));
8      System.out.println("Are s2 and s3 same? " + (s2 == s3));
9    }
10 }
```

```
$ java StringLiteralExample
Are s1 and s2 same? False
Are s1 and s3 same? True
Are s2 and s3 same? False
$
```



Note that comparison (==) applies to the pointers, not the data they point to!

Basic String methods

- The basic `String` methods include the following:
  - `int` **`length`**(): returns the length (number of characters in the string)
  - `char` **`charAt(pos):`** returns the character at the position given in argument `pos` of type `int` (note that the first character is in position 0)
  - `void` **`getChars(`**`beg,end,dest,destBeg`**`):` copies characters from the string to the character array
    - `int beg:` the index (position) where copying starts
    - `int end:` the index (position) *next to the last character* to be copied
    - char[] dest: the character array where the characters are copied
    - `int destBeg:` the index (position) in `dest` where copying in starts

Basic String methods example (1)

```
1  public class BasicStringExample {
2    public static void main(String[] args){
3      String str = "hello world!";
4      for(int i=str.length()-1; i >= 0; i--) {
5          System.out.printf("%c", str.charAt(i));
6      }
7      System.out.println();
8      char[] charArr = new char[5];
9      str.getChars(6,11,charArr,0);
10     System.out.println(charArr);
11   }
12 }
```

Basic String methods example (2)

```
1  public class BasicStringExample {
2    public static void main(String[] args){
3      String str = "hello world!";
4      for(int i=str.length()-1; i >= 0; i--) {
5          System.out.printf("%c", str.charAt(i));
6      }
7      System.out.println();
8      char[] charArr = new char[5];
9      str.getChars(6,11,charArr,0);
10     System.out.println(charArr);
11   }
12 }
```

```
$ java BasicStringExample
!dlrow olleh
```

Loops through the characters backwards and prints them one by one. Note that the last character is at index position `length()-1`.

Basic String methods example (3)

```
1   public class BasicStringExample {
2     public static void main(String[] args){
3       String str = "hello world!";
4       for(int i=str.length()-1; i >= 0; i--) {
5           System.out.printf("%c", str.charAt(i));
6       }
7       System.out.println();
8       char[] charArr = new char[5];
9       str.getChars(6,11,charArr,0);
10      System.out.println(charArr);
11    }
12  }
```

```
$ java BasicStringExample
!dlrow olleh
World
$
```

Copies characters from position index 6 to position index 10 of `str` to the character array `charArr`.

Comparing strings

- Note that the Java comparison operator == compares the references, not the contents of the strings
- For comparing the contents of two strings, methods **`equals()`** and **`compareTo()`** can be used
  - Method **`equals()`** returns true if the argument string contains the same sequence of characters as this object.
  - Method **`equalsIgnoreCase()`** is like **`equals()`**, but ignores case
  - Method **`compareTo()`** returns a negative integer if this string lexicographically (alphabetically) precedes the argument string, zero if the strings are equal, and positive integer if this string lexicographically follows the argument string.

String comparison example

```
1  public class BasicStringComparisonExample {
2    public static void main(String[] args){
3      String s1 = "albert";
4      String s2 = "Albert";
5      String s3 = "Bertha";
6      System.out.printf("%s equals %s: %b%n", s1, s2, s1.equals(s2));
7      System.out.printf("%s equalsIgnoreCase %s: %b%n", s1, s2,
8                        s1.equalsIgnoreCase(s2));
9      System.out.printf("%s compareTo %s: %d%n", s2, s3, s2.compareTo(s3));
10     System.out.printf("%s compareTo %s: %d%n", s3, s2, s3.compareTo(s2));
11   }
12 }
```

```
$ java BasicStringComparisonExample
albert equals Albert: false
albert equalsIgnoreCase Albert: true
Albert compareTo Bertha: -1
Bertha compareTo Albert: 1
$
```

Comparing string regions (substrings)

- For comparing *regions* of strings rather than full strings, method **`regionMatches()`** can be used
  - Returns true if the substrings in specified regions are equal
- Two versions with either four or five arguments:
  - Method `regionMatches(off1,str2,off2,len)` returns true if the substrings of length `len` starting at position `off1` in this string and at position `off2` in argument string `str2` are equal
  - In method `regionMatches(ignoreCase,off1,str2,off2,len)` there is an additional first argument of type `boolean` to determine whether the case should be ignored

Extracting substrings from a string

- For extracting substrings, method **`substring()`** can be used
  - Returns a new `String` object created by copying part of an existing `String` object
- Two versions with either one or two arguments:
  - Method `substring(start)` returns the substring starting from position index `start` and ending in the last character of the string
  - Method `substring(start,end)` returns the substring starting from position index `start` up to, *but not including*, the position index end

Using regionMatches() and substring()

```
1  public class RegionMatchesExample {
2    public static void main(String[] args){
3      String s1 = "Hello World!";
4      String s2 = "good morning world!";
5      System.out.println("Regions matching: " +
6                         s1.regionMatches(6,s2,13,6));
7      System.out.println("Regions matching with case ignored: " +
8                         s1.regionMatches(true,6,s2,13,6));
9      System.out.println("Substring of s1 from 6 to end: " + s1.substring(6));
10     System.out.println("Substring of s2 from 13 to 17: " + s2.substring(13,18));
11   }
12 }
```

```
$ java RegionMatchesExample
Regions matching: false
Regions matching with case ignored: true
Substring of s1 from 6 to end: World!
Substring of s2 from 13 to 17: world
$
```

Questions, comments?