Lecture - 09

String and *HashMap<K,V>* Classes

Arpit Rana

22nd Feb 2022

Character Sequences

In Java, character sequences are handled through three *`final`* classes: `String`, `StringBuilder`, and `StringBuffer`.

- `String` class implements ***immutable*** character strings.
- The `StringBuilder` class implements ***dynamic*** character strings.
- The `StringBuffer` class is a ***thread-safe*** version of the `StringBuilder` class.

Java platform uses the variable-length UTF-16 encoding to store characters in char arrays and in the string handling classes.

Immutability

`String` class implements ***immutable*** character strings.

- Once the string has been created and initialized, it is *read-only*.
- Any operation on a `String` object that modify the characters returns a new `String` object.

Creating and Initializing Strings

Using a `String` literal -

```
String str1 = "You cannot change me!";
```

- A string literal is a reference to a `String` object (value is the enclosed character sequence).
- A string literal can be used to invoke methods on its `String` object:

```
int strLength = "You cannot change me!".length(); // 21
```

String Literal Pool

The compiler optimizes handling of string literals and compile-time constant expressions that evaluate to strings.

- Only one `String` object is shared by all string-valued constant expressions with the same character sequence.
- Such strings are said to be ***interned***, meaning that they share a unique `String` object if they have the same content.

```
String str2 = "You cannot change me!";      // Already interned.

String str3 = "You cannot" + " change me!"; // Compile-time constant
expression

String can1 = 7 + "Up";  // Value of compile-time constant expression: "7Up"
String can2 = "7Up";     // "7Up"

String word = "Up";
String can4 = 7 + word;  // Not a compile-time constant expression.
```

String Literal Pool

The compiler optimizes handling of string literals and compile-time constant expressions that evaluate to strings.

- Only one `String` object is shared by all string-valued constant expressions with the same character sequence.
- Such strings are said to be ***interned***, meaning that they share a unique `String` object if they have the same content.

```
String str2 = "You cannot change me!";      // Already interned.

String str3 = "You cannot" + " change me!"; // Compile-time constant
expression

String can1 = 7 + "Up";  // Value of compile-time constant expression: "7Up"
String can2 = "7Up";     // "7Up"

String word = "Up";
String can4 = 7 + word;  // Not a compile-time constant expression.
```

What if one reference changes the string literal?

Creating and Initializing Strings

Using a `String` constructor -

```
String()

String(String str)
```

- The first constructor will create an empty string (“”) and the second will create a new string with object passed as argument.

```
String(char[] value)
String(char[] value, int offset, int count)
```

- These constructors create a new `String` object, whose contents are copied from a `char` array.
- In this, the second constructor allows extraction of a certain number of characters (`count`) from a given `offset` in the array.
- A constructor creates a brand-new `String` object, it does not *intern* the string.

Creating and Initializing Strings

```
// File: StringConstruction.java
class Auxiliary {
  static String str1 = "You cannot change me!";          // Interned
}
//_____________________________________________________________________
public class StringConstruction {

  static String str1 = "You cannot change me!";          // Interned

  public static void main(String[] args) {
    String emptyStr = new String();                      // ""
    System.out.println("emptyStr: "" + emptyStr + """);

    String str2 = "You cannot change me!";               // Interned
    String str3 = "You cannot" + " change me!";          // Interned
    String str4 = new String("You cannot change me!");   // New String object

    String words = " change me!";
    String str5 = "You cannot" + words;                  // New String object
```

Creating and Initializing Strings

```
System.out.println("str1.equals(str5): " + str1.equals(str5));

System.out.println("str1 == str5:      " + (str1 == str5));

System.out.println("str1.equals(str4): " + str1.equals(str4));

System.out.println("str1 == str4:      " + (str1 == str4));

System.out.println("str1.equals(str3): " + str1.equals(str3));

System.out.println("str1 == str3:      " + (str1 == str3));

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

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

System.out.println("str1 == Auxiliary.str1:      " +
                   (str1 == Auxiliary.str1));
System.out.println("str1.equals(Auxiliary.str1): " +
                    str1.equals(Auxiliary.str1));
```

The `CharSequence` Interface

- This interface defines a readable sequence of char values.
- It is implemented by all three classes: `String`, `StringBuilder`, and `StringBuffer`.
- This interface facilitates interoperability between these classes.
- It defines the following methods:

```
char charAt(int index)

int length()

CharSequence subSequence(int start, int end)

String toString()
```

Reading Characters from a String

The following methods can be used for character-related operations on a string:

```
char charAt(int index)

int length()

boolean isEmpty()

void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

char[] toCharArray()
```

Reading Characters from a String

```
public class ReadingCharsFromString {
  public static void main(String[] args) {
    int[] frequencyData = new int [Character.MAX_VALUE];     // (1)
    String str = "You cannot change me!";                    // (2)

    // Count the frequency of each character in the string.
    for (int i = 0; i < str.length(); i++) {                 // (3)
      try {
        frequencyData[str.charAt(i)]++;                      // (4)
      } catch(StringIndexOutOfBoundsException e) {
        System.out.println("Index error detected: "+ i +" not in range.");
      }
    }

    // Print the character frequency.
    System.out.println("Character frequency for string: "" + str + """);
    for (int i = 0; i < frequencyData.length; i++) {
      if (frequencyData[i] != 0)
        System.out.println((char)i + " (code "+ i +"): " + frequencyData[i]);
    }

    System.out.println("Copying into a char array:");
    char[] destination = new char [str.length() - 3];        // 3 characters
less.
    str.getChars( 0,            7, destination, 0);          // (5) "You can"
    str.getChars(10, str.length(), destination, 7);          // (6) " change
me!"
                                                             // "not" not
copied.
    // Print the character array.
    for (int i = 0; i < destination.length; i++) {
      System.out.print(destination[i]);
    }
    System.out.println();
  }
}
```

Comparing Strings

Characters are compared based on their Unicode values:

```
boolean test = 'a' < 'b';     // true since 0x61 < 0x62
```

Strings are compared ***lexicographically*** i.e. by successively comparing their corresponding characters at each position in the two strings, starting with the characters in the first position. e.g.

- The string "`abba`" is less than "`aha`", since the second character `'b'` in the string "`abba`" is less than the second character `'h'` in the string "`aha`".

Comparing Strings

The following public methods can be used for comparing strings:

```
boolean equals(Object obj)
boolean equalsIgnoreCase(String str2)
```

The `String` class overrides the `equals()` method from the `Object` class.

Comparing Strings

```
int compareTo(String str2)
```

The `String` class implements the `Comparable<String>` interface.

The `compareTo()` method compares the two strings, and returns a value based on the outcome of the comparison:

- The value 0, if this string is equal to the string argument
- A value less than 0, if this string is lexicographically less than the string argument
- A value greater than 0, if this string is lexicographically greater than the string argument

Character Case in a String

```
String toLowerCase()

String toUpperCase()
```

Note that the original string is returned if none of the characters needs its case changed, but a new `String` object is returned if any of the characters needs its case changed.

Concatenation of Strings

Concatenation of two strings results in a new string that consists of the characters of the first string followed by the characters of the second string.

- The overloaded operator **+** is used for string concatenation.
- The following method can also be used to concatenate two strings:

```
String concat(String str)
```

The `concat()` method does not modify the `String` object on which it is invoked but, returns a reference to a brand-new `String` object:

```
String billboard = "Just";
billboard.concat(" lost in space."); // (1) Returned reference value not
stored.
System.out.println(billboard);       // (2) "Just"
billboard = billboard.concat(" advertise").concat(" here.");  // (3)
Chaining.
System.out.println(billboard);       // (4) "Just advertise here."
```

Searching for Characters and Substrings

The following overloaded methods can be used to find the index of a character or the start index of a substring in a string.

- These methods either search forward toward the end of the string or backward toward the start of the string.
- If the search is unsuccessful, the value –1 is returned.

```
int indexOf(int ch)
int indexOf(int ch, int fromIndex)
int indexOf(String str)
int indexOf(String str, int fromIndex)

int lastIndexOf(int ch)
int lastIndexOf(int ch, int fromIndex)
int lastIndexOf(String str)
int lastIndexOf(String str, int fromIndex)
```

Searching for Characters and Substrings

```
String funStr = "Java Jives";
//              0123456789

int jInd1a = funStr.indexOf('J');             // 0
int jInd1b = funStr.indexOf('J', 1);          // 5
int jInd2a = funStr.lastIndexOf('J');         // 5
int jInd2b = funStr.lastIndexOf('J', 4);      // 0

String banner = "One man, One vote";
//              01234567890123456

int subInd1a = banner.indexOf("One");         // 0
int subInd1b = banner.indexOf("One", 3);      // 9
int subInd2a = banner.lastIndexOf("One");     // 9
int subInd2b = banner.lastIndexOf("One", 10); // 9
int subInd2c = banner.lastIndexOf("One", 8);  // 0
int subInd2d = banner.lastIndexOf("One", 2);  // 0

String newStr = funStr.replace('J', 'W');          // "Wava Wives"
String newBanner = banner.replace("One", "No");    // "No man, No vote"
boolean found1 = banner.contains("One");           // true
boolean found2 = newBanner.contains("One");        // false

String song = "Start me up!";
//            012345677890
boolean found3    = song.startsWith("Start");      // true
boolean notFound1 = song.startsWith("start");      // false
boolean found4    = song.startsWith("me", 6);      // true
boolean found5    = song.endsWith("up!");          // true

boolean notFound2 = song.endsWith("up");           // false
```

Extracting Substrings

`trim()` method can be used to create a string where whitespace has been removed from the front (leading) and the end (trailing) of a string.

```
String trim()
```

The `String` class provides the following overloaded methods to extract substrings from a string.

```
String substring(int startIndex)
String substring(int startIndex, int endIndex)
```

A new `String` object containing the substring is created and returned.

```
String utopia = "\t\n  Java Nation \n\t  ";
utopia = utopia.trim();                          // "Java Nation"
utopia = utopia.substring(5);                    // "Nation"
String radioactive = utopia.substring(3,6);      // "ion"
```

HashMap<K, V> Class

The `HashMap` class is roughly equivalent to HashTable, except that it is *unsynchronized* and permits `null` as keys and as values.

- This class does not guarantee that the order will remain constant over time.
- It has two type parameters:
  - K: the type of keys maintained by this map
  - V: the type of mapped values
- It is available in `java.util` package.

HashMap<K, V> Class

- Create a HashMap

```
HashMap<String, Integer> numbers = new HashMap<>();
```

- Add elements to HashMap

```
numbers.put("One", 1);
numbers.put("Two", 2);
numbers.put("Three", 3);
```

- Access elements of HashMap

```
Integer val = numbers.get(“One”);
System.out.println("Keys:" + numbers.keySet());
System.out.println("Values:" + numbers.values());
System.out.println("Key/Value:" + numbers.entrySet());
```

HashMap<K, V> Class

- Change a HashMap Value

```
numbers.replace("Two", 22);
```

- Remove a HashMap Element

```
String value = numbers.remove(“Two”);
System.out.println("Updated HashMap: " + numbers);
// {“one”=1, “Three”=3}
```

- Iterate through HashMap

```
for (String key : numbers.keySet()) { }
for (Integer value : numbers.values()) { }
for (Entry<String, Integer> entry : numbers.entrySet()) { }
```

Next lecture -

Exception Types and Handling