Object-Oriented Programming Using Java

Java String Handling

Introduction

- Java implements strings as objects of type String
- String objects can be constructed a number of ways
- String objects are immutable:
  - When we create a String object, we are creating a string that cannot be changed!
  - Any alteration leads to creation of a new String object!!
  - Somewhat Unexpected!!!
- **Reason:** Fixed, Immutable strings can be implemented more efficiently than changeable ones
- For cases in which a modifiable string is desired, Java provides two options:
  - StringBuffer Class
  - StringBuilder Class
- **String**, **StringBuffer**, and **StringBuilder**: All are defined in java.lang package
- All are declared final, meaning that none of these may be subclassed
- All three implement the CharSequence interface

String Constructors

- String class supports several constructors
- To create an empty string:

  ```
  String s = new String();
  ```

- To create a String object initialized by an array of characters:

  ```
  String(char chars[])          and
  String(byte chrs[])    /* ASCII string */
  ```

- To create a String object with a specified subrange of a character array as an initializer:

  ```
  String(char chars[], int startIndex, int numChars)          and
  String(byte chrs[], int startIndex, int numChars)           and
  String(int chrs[], int startIndex, int numChars)    /* Unicode string*/
  ```

More String Constructors

- To create a String object that contains the same character sequence as another String object:

  String(String strObj)

- To create a String object using a StringBuffer object as initializer:

  String(StringBuffer strBufObj)

- To create a String object using a StringBuilder object as initializer:

  String(StringBuilder strBuildObj)

String Literals

- Java automatically constructs a String object for each string literal in the program
- Essentially, following statements are equivalent:
  - String s1 = “Example”;
  - String s2 = new String(“Example”);
  - String s3 = “Example”;
- The difference between the three statements is that, s1 and s3 are pointing to the same memory location i.e. the string pool; s2 is pointing to a memory location on the heap
- Using a new operator creates a memory location on the heap
- Concatenating s1 and s3 leads to creation of a new string in the pool

Length Of A String

- The length() method:

  int length()

- Example:

```
char chars = {‘a’, ‘b’, ‘c’};
String s = new String(chars);
System.out.println( s.length() );
```

- Since a string literal is a String object, the following is valid:

```
System.out.println( “abcd”.length() );
```

String Concatenation

- The only operator to be applied to String objects is ‘+’
  - The string concatenation
- Essentially useful when dealing with very long strings
- Java allows to concatenate strings with other types of data
- String concatenation may yield weird result if not used carefully:

```
String s = “four: “ + 2 + 2;          /* output: four: 22 */
```

- Correct use is as follows:

```
String s = “four: “ + (2 + 2);        /* output: four: 4 */
```

String Conversion: toString() Method

- The Object class defines toString() method
  - So, every class implements toString() method **implicitly**
  - However, default implementation of toString( ) may not be very useful
  - A class may override toString() method explicitly
- General syntax:

  String toString()

- To override toString() method, we may simply return a String object
  - The object should contains a **human-readable** string that appropriately describes the object of the class

Character Extraction: charAt() Method

- As Java strings are not character arrays, they cannot be indexed directly
- However, Java provides way-out to it through a number of methods
- To extract a single character from a String, we may refer directly to an individual character via the charAt() method:

```
char charAt(int where)
```

- In the above, where is an index and must be nonnegative

```
char ch;
ch = "abc".charAt(1);
```

- The above code would assign the value *b* to variable *ch*

Character Extraction: getChars() Method

- To extract more than one characters at a time, we use the getChars() method
- The general form:

```
void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)
```

❖**sourceStart** specifies the index of the beginning of the substring

❖**sourceEnd** specifies an index that is one past the end of the desired substring

❖**target** is the array that would receive the characters

❖**targetStart** is the index within target where substring would be copied from

- **Caution:** The target array must be large enough to hold the number of characters in the specified substring
- An alternative to getChars() is getBytes()
  - Particularly useful when we export a String value into an environment that does not support 16-bit Unicode characters

Character Extraction: toCharArray() Method

- To convert all the characters in a String object into a character array
- The general form:

  char[] toCharArray()

- Provided only for convenience!
- We may get the same effect using getChars() method

String Comparison: equals() and equalsIgnoreCase()

- Compares two strings for equality
- General form:

  boolean equals(Object str)

- The comparison in this method is case-sensitive

- To Perform string comparison ignoring case differences, use equalsIgnoreCase() method
- General form:

  boolean equalsIgnoreCase(String str)

String Comparison: regionMatches() Method

- Compares a specific region inside a string with another specific region in another string
- General form:

  boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)

- The same has an overloaded form as follows:

  boolean regionMatches(Boolean ignoreCase, int startIndex, String str2,
  int str2StartIndex, int numChars)

String Comparison: startsWith(), endsWith()

- More or less, specialized forms of regionMatches() method
- startsWith()/endsWith() method determines whether a given String begins/ends with a specified string
- General forms:

  boolean startsWith(String str)

  boolean endsWith(String str)

- A second form for startsWith():

  boolean startsWith(String str, int startIndex)

String Comparison: compareTo() Method

- There are situations when it is not enough to merely know whether two strings are identical or not
  - Example: Sorting-like applications
- The method compareTo() serves the purpose
- Specified by Comparable<T> interface, that String class implements
- General form:

  int compareTo(String str)

- The method returns negative, zero or positive integral values if the invoking string is lower than, equal to or higher than, respectively in dictionary order, compared to the argument string
- The comparison is case-sensitive
- If we wish to ignore cases when comparing strings, we have the following:

  int compareToIgnoreCase(String str)

Searching Strings: indexOf() and lastIndexOf()

- Two methods that allow us to search in a string for a specified character/substring
- Both the methods are overloaded in several different ways
- Return the index at which the character/substring was found and -1 on failure
- Various available formats:

int indexOf(int ch)	and	int lastIndexOf(int ch)
int indexOf(String str)	and	int lastIndexOf(String str)
int indexOf(int ch, int startIndex)	and	int lastIndexOf(int ch, int startIndex)
int indexOf(String str, int startIndex)	and	int lastIndexOf(String str, int startIndex)

Modify String: substring()

- The method has two forms:

  String substring(int startIndex)

  String substring(int startIndex, int endIndex)
- startIndex specifies the beginning index
- endIndex specifies the stopping point (excluding the point)

- **Note:** Recall that Java String objects are **immutable**. So, whenever we want to modify a String, either we use a String method that constructs a **new copy** of the string with your modifications complete, or we must either copy it into a **StringBuffer** or **StringBuilder**

Modify String: concat()

- Concatenate two strings
- Format:

  String concat(String str)

- **Note:** Output is same as ‘+’ operator

Modify String: replace()

- Two general forms:

  String replace(char original, char replacement)

  String replace(CharSequence original, CharSequence replacement)

- The former replaces all occurrences of *original* character in the invoking string with *replacement* character
- The latter replaces all occurrences of *original* character sequence in the invoking string with *replacement* character sequence
- Example:

  String s = "Hello".replace('l', 'w’); /* **Output: Hewwo** */

Modify String: trim() and strip()

- The **trim()** method returns a copy of the invoking string from which any leading and trailing spaces have been removed
- General form:

  String trim()
- Example:

  System.out.println(“ Hello ”.trim()); **/* Output: Hello */**
- Particularly useful when we have to process user commands
- The **strip()** method removes all whitespace characters (as defined by Java) from the beginning and end of the invoking string and returns the result
- Available since the beginning with JDK 11
- JDK 11 also provides methods **stripLeading()** and **stripTrailing()**
- The general forms: String strip() String stripLeading() String stripTrailing()

Data Conversion: valueOf() method

- The method converts data from its internal format into a human-readable (string) form
- Defined as a static method
- Overloaded within String class for all of Java's built-in types
- Also overloaded for type Object, so an object of any class type (built-in or user-defined) can also be used as an argument!
- Few forms:

  static String valueOf(double num)

  static String valueOf(long num)

  static String valueOf(char chars[])

  static String valueOf(Object ob)

- For an object, valueOf() method simply calls the toString() method

Changing Case: toLowerCase(), toUpperCase()

- Simplest form:

  String toLowerCase()

  String toUpperCase()

- The name explains the purpose

Joining Strings: join()

- Used concatenate two or more strings, separating each string with a delimiter
- Simplest form:

  static String join(CharSequence delim, CharSequence . . . strs)

- ***delim*** specifies the delimiter used to separate the character sequences
- Method added since JDK 8

StringBuffer Class

- StringBuffer class supports a modifiable string
  - StringBuffer automatically grows to make room for addition of characters/substrings that are inserted in the middle or appended to the end
  - To allow room for growth, StringBuffer often has more characters pre-allocated than are actually needed
- By reserving room for additional characters, the StringBuffer reduces the number of reallocations that may take place otherwise
  - Essentially, reallocation is a costly process in terms of time
  - Frequent reallocations can fragment memory

StringBuffer Constructors

- StringBuffer defines the following four constructors:

  StringBuffer()

  StringBuffer(int size)

  StringBuffer(String str)

  StringBuffer(CharSequence chars)

- The default constructor (the one with no parameters) reserves room for sixteen (16) characters without reallocation
- In second version of StringBuffer Constructor, the integer argument *size* explicitly sets the size of the buffer
- The third and the fourth versions of StringBuffer Constructor sets the initial contents of the StringBuffer object as *str* or *chars* and reserves room for sixteen (16) more characters without reallocation

StringBuffer: length() and capacity()

- Method length() finds the current length of a StringBuffer

  int length()

- Method capacity() finds the total allocated capacity

  int capacity()

- Example:

```
StringBuffer sb = new StringBuffer("Hello");
System.out.println("Buffer = " + sb);
 System.out.println("length = " + sb.length());
 System.out.println("capacity = " + sb.capacity());
```

- Output:

buffer = Hello

length = 5

capacity = 21

StringBuffer: ensureCapacity()

- After a StringBuffer has been constructed, we may use ensureCapacity() to set the size of the buffer
- Particularly useful if we know in advance that we will be appending a large number of small strings to a StringBuffer
- General form:

  void ensureCapacity(int minCapacity)

- A buffer larger than **minCapacity** may be set due to efficiency reason

StringBuffer: setLength()

- Method setLength() is used to set the length of the string within a StringBuffer object
  - If we try increasing the size of the string, null characters are added to the end
  - If we try to shorten the string, the characters stored beyond the new length will be lost
- General form:

  void setLength(int len)

StringBuffer: charAt(), setCharAt(), getChars()

- charAt() method return the value of a single character from a StringBuffer
- We can set the value of a character within a StringBuffer using setCharAt() method
- getChars() method copies a substring of a StringBuffer into an array
- General forms:

```
char charAt(int where)

void setCharAt(int where, char ch)

void getChars(int sourceStart, int sourceEnd, char target[], int targetStart)
```

StringBuffer: append()

- Concatenates the string representation of any other type of data to the end of the invoking StringBuffer object
- Several overloaded versions:

  StringBuffer append(String str)

  StringBuffer append(int num)

  StringBuffer append(Object obj)

StringBuffer: insert()

- Inserts one string into another string
- overloaded to accept values of all the primitive types, plus Strings, Objects, and CharSequences
- Few General forms:

    StringBuffer insert(int index, String str)

    StringBuffer insert(int index, char ch)

    StringBuffer insert(int index, Object obj)

StringBuffer: reverse(), delete(), deleteCharAt()

- We can reverse the characters within a StringBuffer object using reverse() method
- General form:

  StringBuffer reverse()

- We can delete characters within a StringBuffer by using the methods delete() and deleteCharAt()
- General form:

  StringBuffer delete(int startIndex, int endIndex)

  StringBuffer deleteCharAt(int loc)

StringBuffer: replace() and substring()

- We can replace one set of characters with another set inside a StringBuffer object
- Genral form:

  StringBuffer replace(int startIndex, int endIndex, String str)

- We can obtain a portion of a StringBuffer by calling substring()
- General forms:

  String substring(int startIndex)

  String substring(int startIndex, int endIndex)

StringBuilder Class

- Traditionally, the Java platform has always provided two classes: String and StringBuffer
- The StringBuilder class, introduced in JDK 5.0, is a faster, drop-in replacement for string buffers
- Similar to StringBuffer class, except for one important difference:
  - Not synchronized, so not thread-safe
  - However, performance is faster than StringBuffer
- We use a StringBuilder in the same way as a StringBuffer, but only if it is going to be accessed by a single thread!

Summary

- If no or limited string modification expected
  - Suggested → String
- Severe string modification expected & Multithreaded implementation
  - Suggested → StringBuffer
- Severe string modification expected & Single-threaded implementation
  - Suggested → StringBuilder