Test Automation Lecture 7 - Strings and Collections

IT Learning & Outsourcing Center

Lector: Milen Strahinski
Skype: strahinski
E-mail: milen.strahinski@pragmatic.bg

www.pragmatic.bg

Strings

- What is String?
- How to create a String

```
String firstName;

firstName = "Ivan";

String lastName = new String("Petrov");
```

Declare variable of type String

Initialization

Another way for initialization

Concatenation of strings

+ is used for concatenation

```
String firstName = "Ivan";

String lastName = "Petrov";

String name = firstName + " " + lastName;

System.out.println(name);
```

Prints the value of name in the console

Comparing strings

- .equals() should be used because String is reference type

```
String firstName = "Ivan";
String lastName = "Petrov";

String name = firstName + " " + lastName;

System.out.println(name == "Ivan Petrov");
System.out.println(name.equals("Ivan Petrov"));
```

- The output is:
  false
  true

More about Strings

- \ should be used for escaping special characters
- .length() return the length of the string
- String has many features(methods) for manipulating the text value

```
String welcome = "Welcome to learning center
\"Pragmatic\"";

System.out.println(welcome.length());
```

The output is 38

Converting Strings to Numbers

- The Number subclasses that wrap primitive numeric types (Byte, Integer, Double, Float, Long, and Short) each provide a class method named valueOf that converts a string to an object of that type.

- **Note:** Each of the Number subclasses that wrap primitive numeric types also provides a parseXXXX() method (for example, parseFloat()) that can be used to convert strings to primitive numbers.

- StringValueOfDemo.java in the code examples

Converting Numbers to Strings

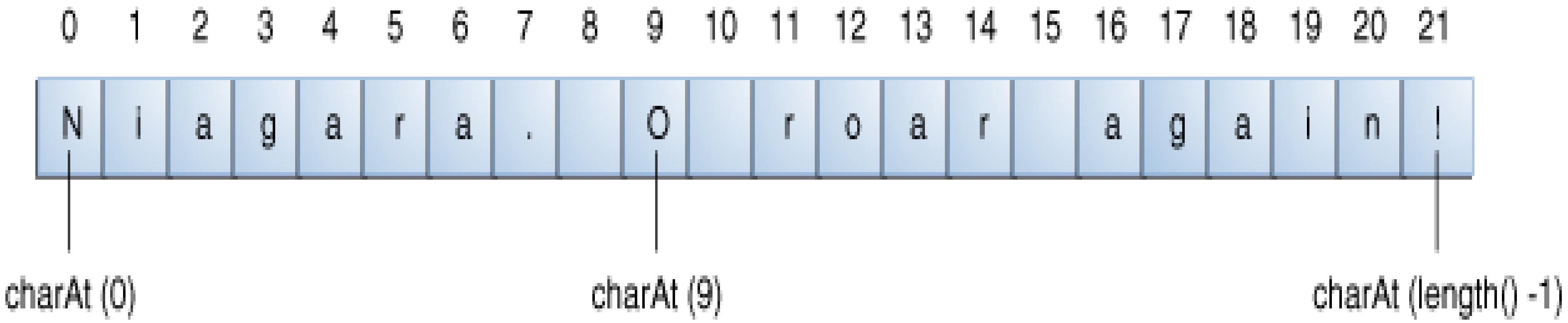
```
int i;
// Concatenate "i" with an empty string;
// conversion is handled for you.
 String s1 = "" + i;
```

```
// The valueOf class method.
String s2 = String.valueOf(i);
```

- ...or using the toString() method – check ToStringDemo.java in code examples

Getting Characters and Substrings by Index (part 1)

- You can get the character at a particular index within a string by invoking the charAt() accessor method. The index of the first character is 0, while the index of the last character is length()-1.



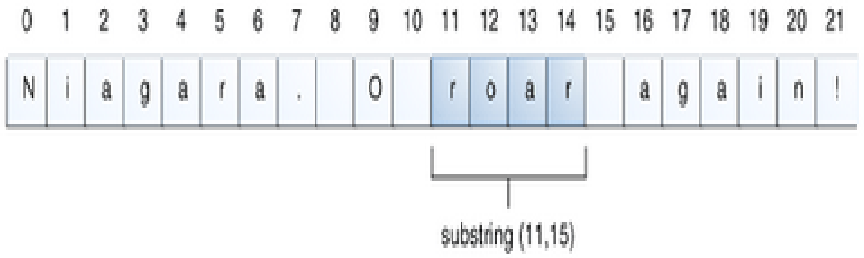
Getting Characters and Substrings by Index (part 2)

The `substring` Methods in the `String` Class

Method	Description
`String substring(int beginIndex, int endIndex)`	Returns a new string that is a substring of this string. The first integer argument specifies the index of the first character. The second integer argument is the index of the last character - 1.
`String substring(int beginIndex)`	Returns a new string that is a substring of this string. The integer argument specifies the index of the first character. Here, the returned substring extends to the end of the original string.

The following code gets from the Niagara palindrome the substring that extends from index 11 up to, but not including, index 15, which is the word "roar":

```
String anotherPalindrome = "Niagara. O roar again!";
String roar = anotherPalindrome.substring(11, 15);
```



Other Methods for Manipulating Strings

- trim() - Returns a copy of this string with leading and trailing white space removed.
- toLowerCase() or toUpperCase() - Returns a copy of this string converted to lowercase or uppercase. If no conversions are necessary, these methods return the original string.
- The indexOf() methods search forward from the beginning of the string, and the lastIndexOf() methods search backward from the end of the string. If a character or substring is not found, indexOf() and lastIndexOf() return -1.

More methods for comparing Strings

Methods for Comparing Strings

Method	Description

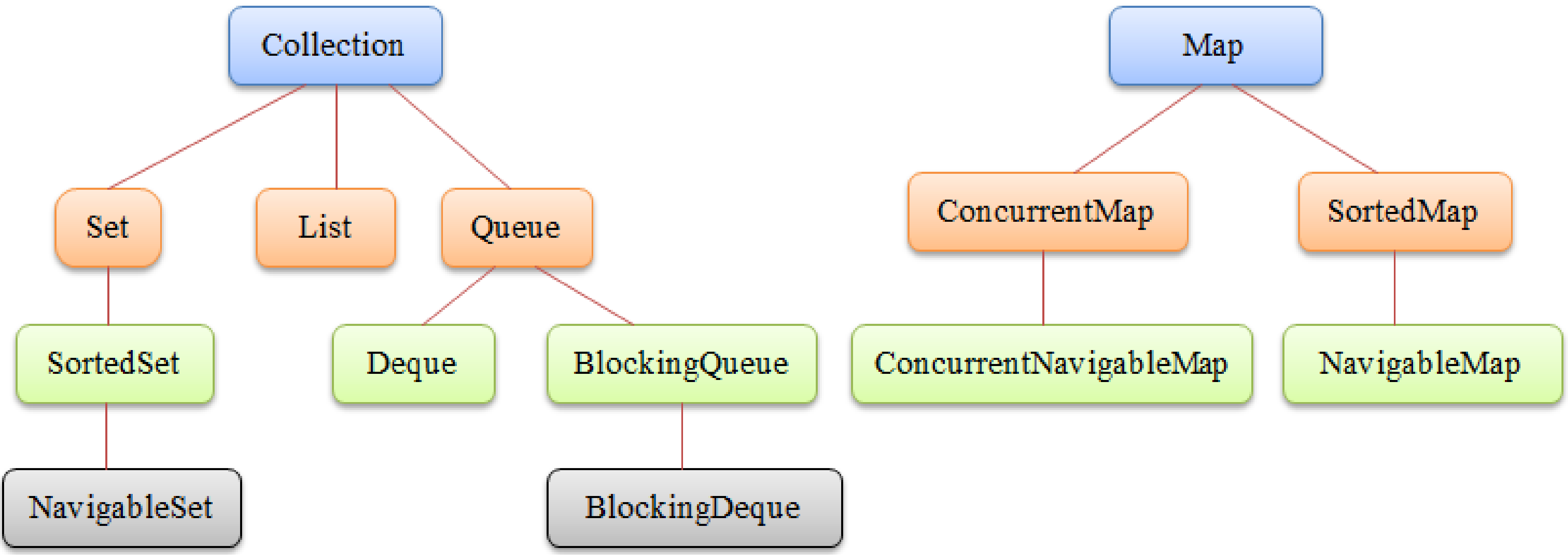
`boolean endsWith(String suffix)`
`boolean startsWith(String prefix)`
	Returns `true` if this string ends with or begins with the substring specified as an argument to the method.

`boolean startsWith(String prefix, int offset)`	Considers the string beginning at the index `offset`, and returns `true` if it begins with the substring specified as an argument.
`int compareTo(String anotherString)`	Compares two strings lexicographically. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
`int compareToIgnoreCase(String str)`	Compares two strings lexicographically, ignoring differences in case. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument.
`boolean equals(Object anObject)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object.
`boolean equalsIgnoreCase(String anotherString)`	Returns `true` if and only if the argument is a `String` object that represents the same sequence of characters as this object, ignoring differences in case.

Collections - Introduction

- A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- Typically, they represent data items that form a natural group, such as a poker hand (a collection of cards), a mail folder (a collection of letters), or a telephone directory (a mapping of names to phone numbers).

Collection Interfaces



The Collection Interface

```java
public interface Collection<E> extends Iterable<E> {
    // Basic operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    // optional
    boolean add(E element);
    // optional
    boolean remove(Object element);
    Iterator<E> iterator();

    // Bulk operations
    boolean containsAll(Collection<?> c);
    // optional
    boolean addAll(Collection<? extends E> c);
    // optional
    boolean removeAll(Collection<?> c);
    // optional
    boolean retainAll(Collection<?> c);
    // optional
    void clear();

    // Array operations
    Object[] toArray();
    <T> T[] toArray(T[] a);
}
```

Collection methods

- how many elements are in the collection (*size(), isEmpty()*)
- to check whether a given object is in the collection (*contains()*)
- to add and remove an element from the collection (*add(), remove()*)
- to provide an iterator over the collection (*iterator()*).

for-each construct

- The *for-each* construct allows you to concisely traverse a collection or array using a for loop. The following code uses the *for-each* construct to print out each element of a collection on a separate line.

```
for (Object o : collection)
    System.out.println(o);
```

Iterator interface

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

- The *hasNext* method returns true if the iteration has more elements, and the next method returns the next element in the iteration.

- The remove method removes the last element that was returned by next() from the underlying Collection. The remove method may be called only once per call to next() and throws an exception if this rule is violated.

- Lets take a look on IteratorExample.java file in code examples

for-each vs. Iterator

- Use *Iterator* instead of the *for-each* construct when you need to:
  - Remove the current element. The for-each construct hides the *iterator*, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
  - Iterate over multiple collections in parallel.

Collection interface Bulk Operations

- *containsAll* — returns true if the target Collection contains all of the elements in the specified Collection.
- *addAll* — adds all of the elements in the specified Collection to the target Collection.
- *removeAll* — removes from the target Collection all of its elements that are also contained in the specified Collection.
- *retainAll* — removes from the target Collection all its elements that are *not* also contained in the specified Collection. That is, it retains only those elements in the target Collection that are also contained in the specified Collection.
- *clear* — removes all elements from the Collection.

The *addAll*, *removeAll*, and *retainAll* methods all return true if the target Collection was modified in the process of executing the operation.

Collection Interface Array Operations

- The *toArray* methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a Collection to be translated into an array. The simple form with no arguments creates a new array of Object. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

```
Object[] a = c.toArray();
```

- Suppose that c is known to contain only strings (perhaps because c is of type Collection<String>). The following snippet dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c.

```
String[] a = c.toArray(new String[]);
```

-

Set interface

- A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains *only* methods inherited from Collection and adds the restriction that duplicate elements are prohibited.

Set Implementations

- The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, and LinkedHashSet.
- HashSet, which stores its elements in a hash table, is the best-performing implementation, however it makes no guarantees concerning the order of iteration.
- TreeSet, which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet. Objects are stored in a sorted and ascending order.
- LinkedHashSet, which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

Set Interface Bulk Operations (part 1)

- *s1.containsAll(s2)* — returns true if s2 is a **subset** of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)
- *s1.addAll(s2)* — transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)
- *s1.retainAll(s2)* — transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
- *s1.removeAll(s2)* — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

Set Interface Bulk Operations (part 2)

- Union

```
Set<Type> union = new HashSet<Type>(s1);
union.addAll(s2); //obedinenie
```

- Intersection

```
Set<Type> intersection = new HashSet<Type>(s1);
intersection.retainAll(s2); //sechenie
```

- Difference

```
Set<Type> difference = new HashSet<Type>(s1);
difference.removeAll(s2); //razlika
```

- The implementation type of the result Set in the preceding idioms is HashSet, which is the best all-around Set implementation in the Java platform. However you can use any other general Set implementation.

List Interface

- A List is an ordered Collection (sometimes called a *sequence*).
- Lists may contain duplicate elements.
- In addition to the operations inherited from *Collection*, the *List* interface includes operations for the following:
  - Positional access — manipulates elements based on their numerical position in the list
  - Search — searches for a specified object in the list and returns its numerical position
  - Iteration — extends Iterator semantics to take advantage of the list's sequential nature
  - Range-view — performs arbitrary *range operations* on the list.

List Implementations

- The Java platform contains two general-purpose List implementations:
  - ArrayList, which is usually the better-performing implementation.
  - LinkedList which offers better performance under certain circumstances.

List - Collection Operations (part 1)

- The operations inherited from *Collection* all do about what you'd expect them to do, assuming you're already familiar with them.
- The *remove* operation always removes *the first* occurrence of the specified element from the list.
- The *add* and *addAll* operations always append the new element(s) to the *end* of the list.

List - Collection Operations (part 2)

- Thus, the following idiom concatenates one list to another:

```
list1.addAll(list2);
```

- Here's a *nondestructive* form of this idiom, which produces a third List consisting of the second list appended to the first.

```
List<Type> list3 = new ArrayList<Type>(list1);
list3.addAll(list2);
```

List - Positional Access and Search Operations

- The basic positional access operations (*get, set, add* and *remove*) behave just as you expect it.
- The search operations *indexOf* and *lastIndexOf* behave just as you expect it.
- The addAll operation inserts all the elements of the specified Collection starting at the specified position. The elements are inserted in the order they are returned by the specified Collection's iterator.

Map Interface

- A Map is an object that maps keys to values:
  - A map cannot contain duplicate keys
  - Each key can map to at most one value.

Map – Iterate over it (part 1)

If you're only interested in the keys, you can iterate through the `keySet()` of the map:

```
Map<String, Object> map = ...;

for (String key : map.keySet()) {
    // ...
}
```

If you only need the values, use `values()`:

```
for (Object value : map.values()) {
    // ...
}
```

Finally, if you want both the key and value, use `entrySet()`:

```
for (Map.Entry<String, Object> entry : map.entrySet()) {
    String key = entry.getKey();
    Object value = entry.getValue();
    // ...
}
```

Map – Iterate over it (part 1)

- Lets take a look on MapExample.java in code examples

Summary

- What is String and how we can to use it?
- Collection, Iterator, For-each, List, Set, Map
- Type casting