Collections & Regex

MCIT 591 Online

Agenda

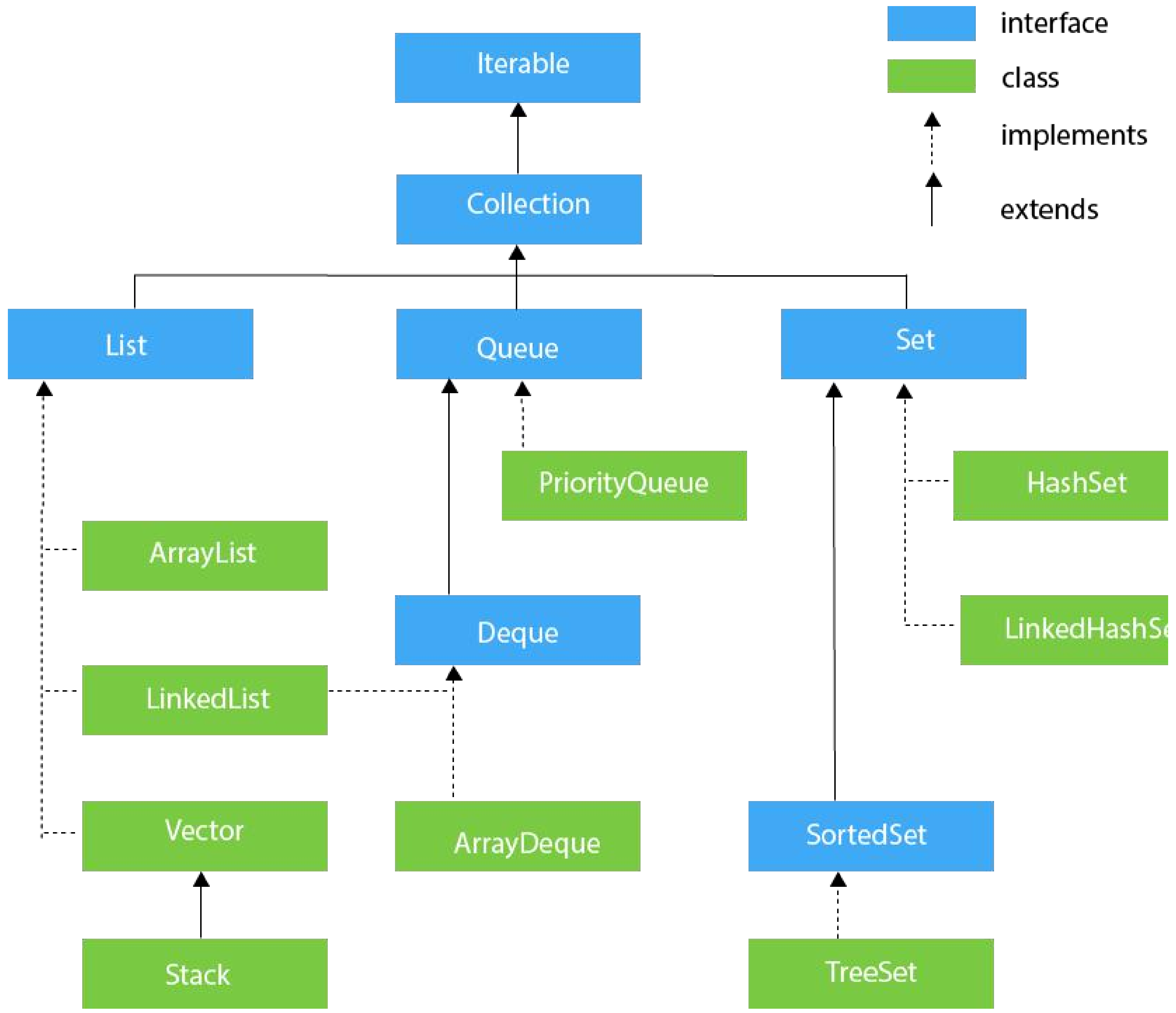
- Course Materials
- Practice Assignment Review
- Homework 9: Student Management System

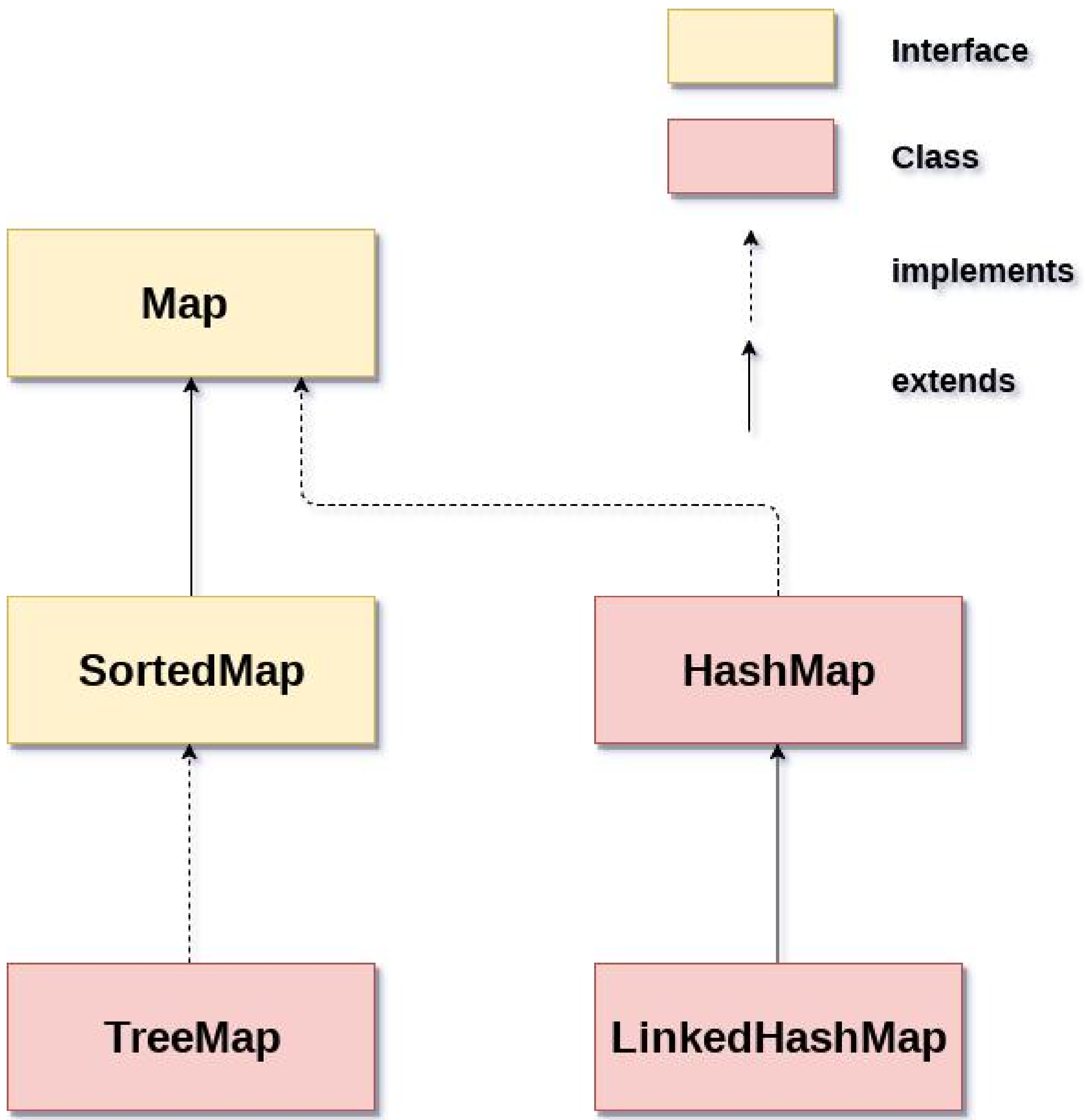
Collection: a structured group of objects

- The Collection interface is the root interface of the Collection hierarchy
- Types of Collections (subinterface):
    - List(ArrayList): May contain duplicate elements, order is important
        - ArrayList is an implementation of List, which is a subinterface of Collection
        - Array is not a collection
    - Set: Cannot contain duplicate elements, order is not important
    - SortedSet: Like a Set, but order is important
- Types of Collection-like (subinterface):
    - Map: A “dictionary” that associates keys with values, where order is not important
    - SortedMap: A map where order is important

Collection: a structured group of objects

- Each interface has at least one implementation.
- List – ArrayList, LinkedList
- Deque (Double-ended queue, front & rear) – ArrayDeque, ConcurrentLinkedDeque, LinkedList
- Set – HashSet, LinkedHashSet,
  - SortedSet - TreeSet
- Map – HashMap,
  - SortedMap - TreeMap


interface
class
implements
extends
Iterable
Collection
List
Queue
Set
PriorityQueue
HashSet
ArrayList
Deque
LinkedHashSe
LinkedList
Vector
ArrayDeque
SortedSet
Stack
TreeSet


Interface
Class
implements
extends
Map
SortedMap
HashMap
TreeMap
LinkedHashMap

Methods in the Collection Interface

- All subinterfaces include the methods in the Collection interface
- For example
  - boolean add(E o)
  - boolean contains(Object o)
  - boolean remove(Object o)
  - boolean isEmpty()
  - int size()
  - Object[] toArray()
  - Iterator<E> iterator()

Methods in the Subinterfaces of Collection

- List interface
    - void add(int index, E element)
    - void set(int index, E element)
- Deque Interface
    - Element access only at the front or the rear of the Collection
    - Can be used as both a stack (LIFO) and a queue (FIFO)
        - LIFO: last in, first out
        - FIFO: first in, first out
    - void addFirst(E o), void addLast(E o)
    - E getFirst(), E getLast()
    - E removeFirst(), E removeLast()
- Set Interface

Methods in the Collection-like Interfaces

- Map Interface
  - Cannot contain duplicate keys
  - Each key can map to at most one value
  - Important methods:
    - V put(K key, V value)
    - V get(Object key)
    - It does not inherit Collection methods

General Rules for Selecting an Implementation

- List
  - If you need fast access to random elements in the list, choose the ArrayList.
  - If you will frequently remove or insert elements to the list, choose a LinkedList.
- Deque – If you only need access at the ends (beginning or end) of the sequence.
- Set
  - Use a TreeSet if you need to traverse the set in sorted order.
  - Otherwise, use a HashSet, it's more efficient.
- Map
  - Chose TreeMap if you want to access the collection in key order.
  - Otherwise, choose HashMap.

Iterator<E> iterator()

- Recall: we use for loop to iterate a list,
    - either with index: int i=0, i<n, i++
    - or in an element way: for (String s: ListOfString)
- Use an iterator instead of a for loop to <u>modify the collection while traversing</u>
    - A ConcurrentModificationException is thrown when a collection is modified while traversing, by any means other than through its iterator

Iterator<E> iterator()

```
//get iterator object
Iterator<String> it = treeSet.iterator();


//modify (remove) the values using the iterator while traversing the
treeset
while(it.hasNext()) {                      //if exist
    if(it.next().equals("red")) {          //access to the element
        it.remove();                       //modify the values
    }
}
```

Sorted order

- To store an object in a sorted Collection, the Object must have a natural order (numeric values)
- OR it must implement the Comparable interface compareTo (for example: alphabetically)
- Recall: Compare the two instance variables of an author's name

Example: CompareTo

- Here's an example Author class that implements Comparable and *compareTo*

```
public class Author implements Comparable<Author> {

    String firstName;
    String lastName;

    @Override
    public int compareTo(Author other) {
        //compare the names of authors
        //returns negative number if this is supposed to be less than the other
        //returns positive number if this is supposed to be greater than the other
        //otherwise returns 0 if they are supposed to be equal
        int last = this.lastName.compareTo(other.lastName);
        return last == 0 ? this.firstName.compareTo(other.firstName) : last;
    }
}
```

Static method

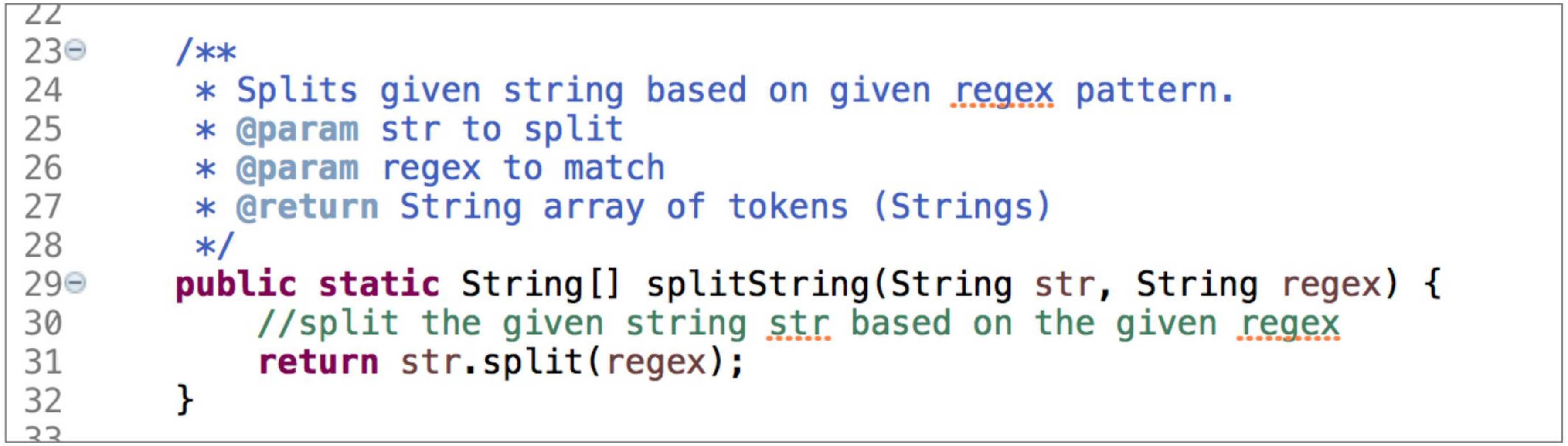
- The Collections class has some convenient static methods for working with collections
- For example:
    - Collections.sort()
        - sorts list
    - Collections.binarySearch(arrayList, String)
        - returns position in list where object is found

- Static methods of Array: Arrays.asList(), Arrays.sort()

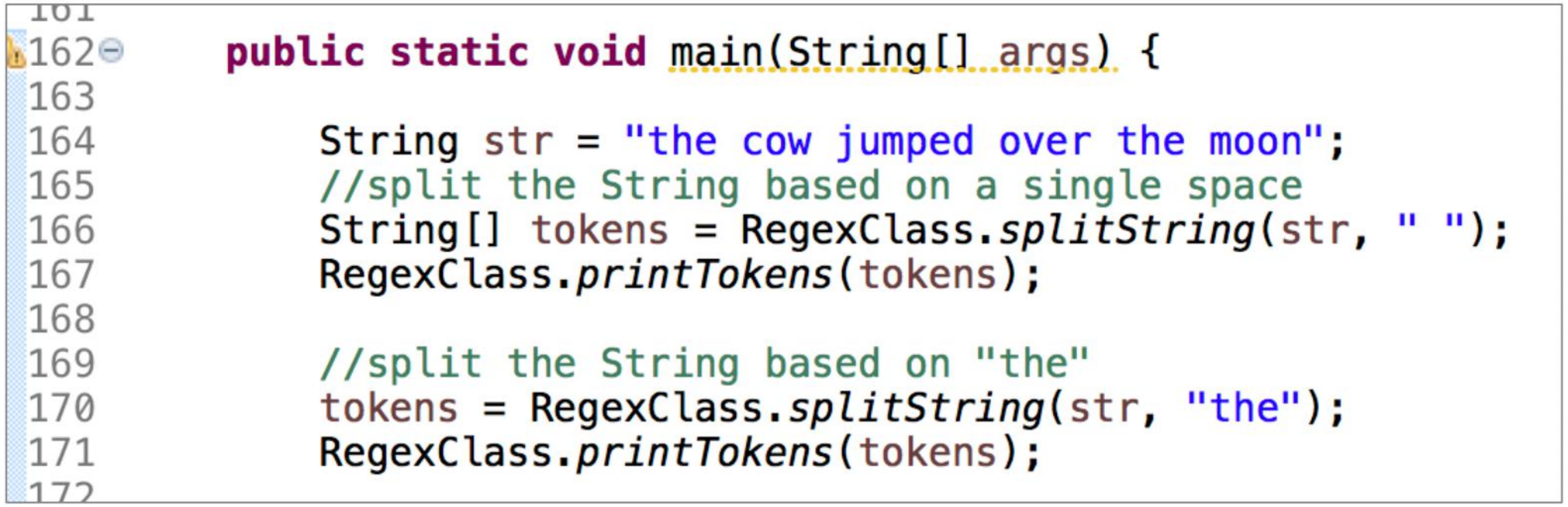
Regular Expressions

- A regular expression (or regex) is a special sequence of characters that describes a pattern used for searching, editing, and manipulating text and data
- For example, regular expressions are widely used to define the constraint on Strings in password and email validation

Split a String

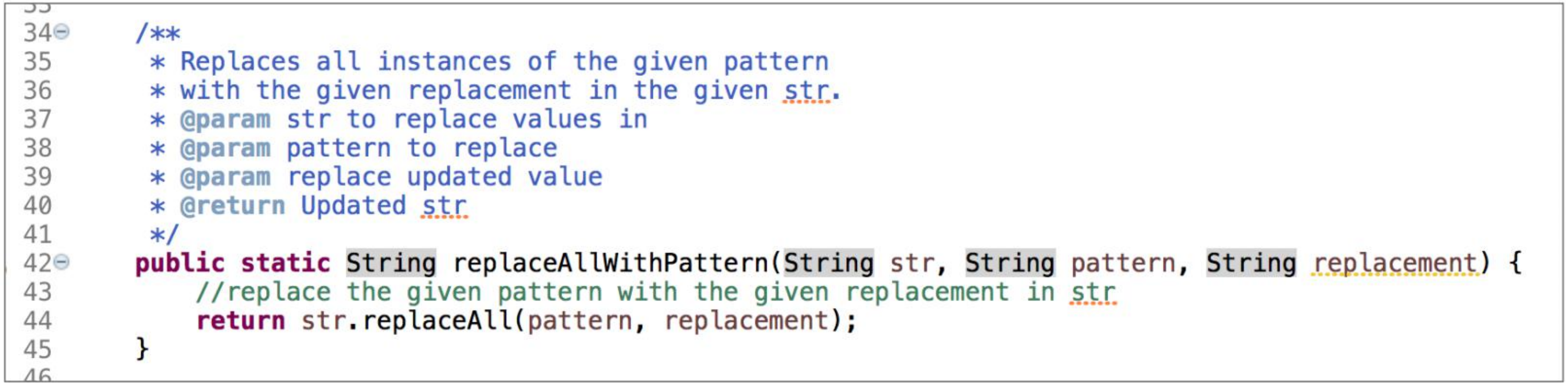


```
    /**
     * Splits given string based on given regex pattern.
     * @param str to split
     * @param regex to match
     * @return String array of tokens (Strings)
     */
    public static String[] splitString(String str, String regex) {
        //split the given string str based on the given regex
        return str.split(regex);
    }
```

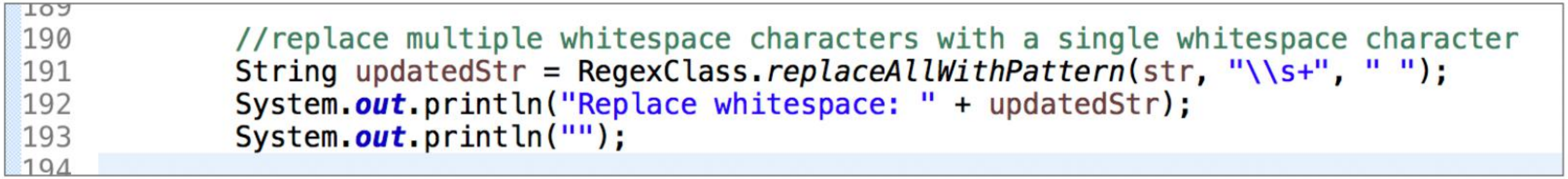


```
    public static void main(String[] args) {

        String str = "the cow jumped over the moon";
        //split the String based on a single space
        String[] tokens = RegexClass.splitString(str, " ");
        RegexClass.printTokens(tokens);

        //split the String based on "the"
        tokens = RegexClass.splitString(str, "the");
        RegexClass.printTokens(tokens);
```

Replace all with a pattern

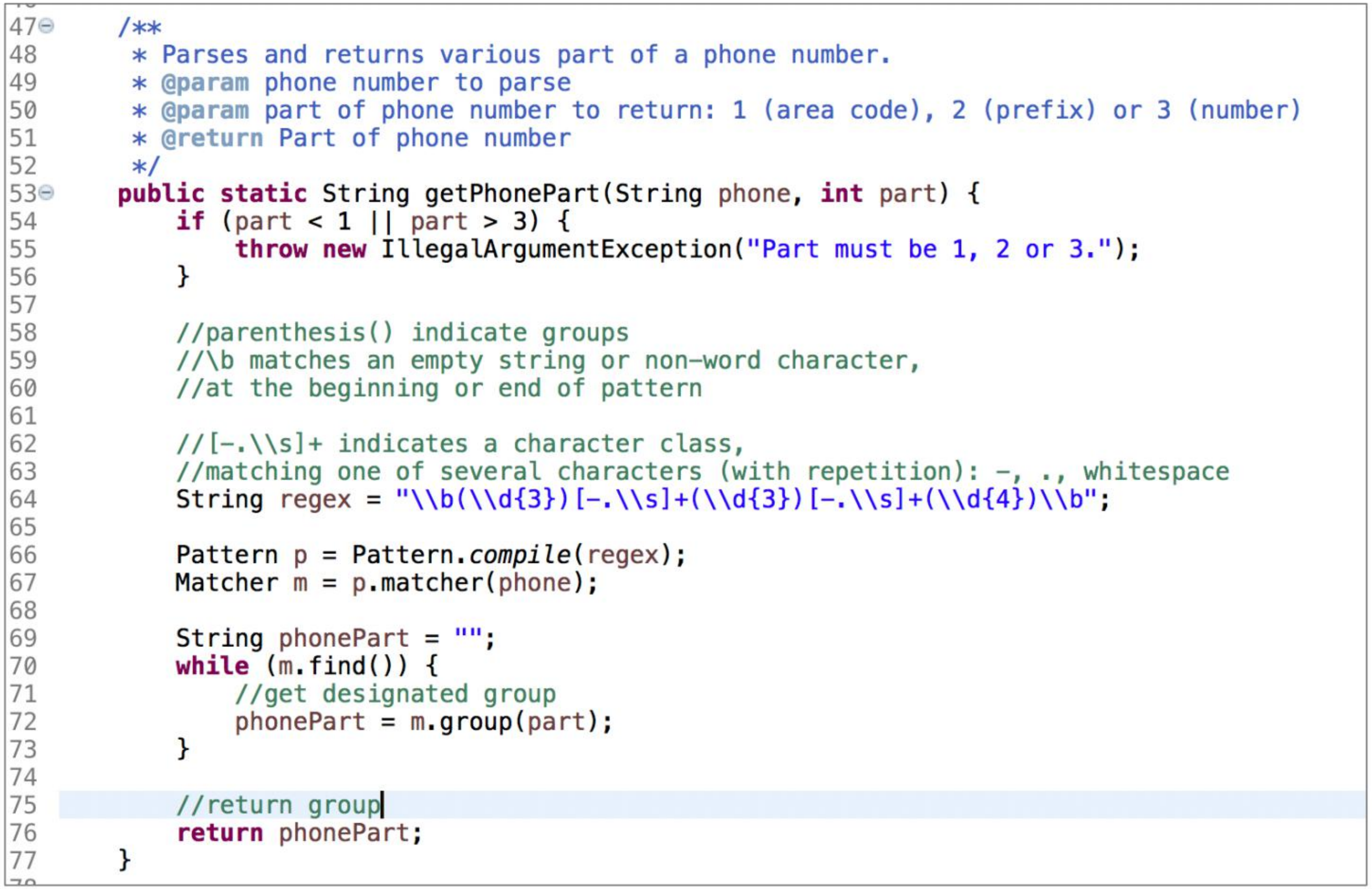


```java
    /**
     * Replaces all instances of the given pattern
     * with the given replacement in the given str.
     * @param str to replace values in
     * @param pattern to replace
     * @param replace updated value
     * @return Updated str
     */
    public static String replaceAllWithPattern(String str, String pattern, String replacement) {
        //replace the given pattern with the given replacement in str
        return str.replaceAll(pattern, replacement);
    }
```



```java
        //replace multiple whitespace characters with a single whitespace character
        String updatedStr = RegexClass.replaceAllWithPattern(str, "\\s+", " ");
        System.out.println("Replace whitespace: " + updatedStr);
        System.out.println("");
```

Get Parts of a Phone Number



```java
    /**
     * Parses and returns various part of a phone number.
     * @param phone number to parse
     * @param part of phone number to return: 1 (area code), 2 (prefix) or 3 (number)
     * @return Part of phone number
     */
    public static String getPhonePart(String phone, int part) {
        if (part < 1 || part > 3) {
            throw new IllegalArgumentException("Part must be 1, 2 or 3.");
        }

        //parenthesis() indicate groups
        //\b matches an empty string or non-word character,
        //at the beginning or end of pattern

        //[-.\\s]+ indicates a character class,
        //matching one of several characters (with repetition): -, ., whitespace
        String regex = "\\b(\\d{3})[-.\\s]+(\\d{3})[-.\\s]+(\\d{4})\\b";

        Pattern p = Pattern.compile(regex);
        Matcher m = p.matcher(phone);

        String phonePart = "";
        while (m.find()) {
            //get designated group
            phonePart = m.group(part);
        }

        //return group
        return phonePart;
    }
```