String

CharSequence – this is an interface implemented by StringBuffer, String, StringBuilder, Segment, CharBuffer

length()

charAt(int) – returns a character at a specific position. This method uses zero indexes. Can throw StringIndexOutOfBoundsException.

IndexOf() - returns -1 when a match isn’t found. The method doesn’t start counting from 0 if we specify a starting position. It always returns the concrete position in the string. It is zero indexed.

int indexOf(char ch)

int indexOf(char ch, index fromIndex)

int indexOf(String str)

int indexOf(String str, index fromIndex)

String string = "animals";

System.out.println(string.indexOf('a')); //0

System.out.println(string.indexOf("al")); //4

System.out.println(string.indexOf('a', 4)); //4

System.out.println(string.indexOf("al", 5)); //-1

substring – returns parts of the string. It is zero-based. The endIndex position is not included. Thus, substring(3, 3) returns an empty string.

String substring(int beginIndex) – returns a substring from the position to the end.

String substring(int beginIndex, int endIndex)

toLowerCase()

toUpperCase()

equals()

equalsIgnoreCase()

System.out.println("abc".equals("ABC")); // false

System.out.println("ABC".equals("ABC")); // true

System.out.println("abc".equalsIgnoreCase("ABC")); // true

intern() - returns a “string pooled” version of a string

startsWith(), endsWith()

boolean startsWith(String prefix)

boolean endsWith(String suffix)

The following code shows how to use these methods:

System.out.println("abc".startsWith("a")); // true

System.out.println("abc".startsWith("A")); // false

System.out.println("abc".endsWith("c")); // true

System.out.println("abc".endsWith("a")); // false

contains() - It isn’t as particular as startsWith() and endsWith() —the match can be anywhere in the String . The method

signature is as follows: boolean contains(String str)

The following code shows how to use these methods:

System.out.println("abc".contains("b")); // true

System.out.println("abc".contains("B")); // false

replace() - replace all the found symbols, not only the first one

String replace(char oldChar, char newChar)

String replace(CharSequence oldChar, CharSequence newChar)

System.out.println("abcabc".replace('a', 'A')); // AbcAbc

System.out.println("abcabc".replace("a", "A")); // AbcAbc

trim() - removes whitespace from the beginning and end of a String . In terms of the exam, whitespace consists of spaces along with the \t (tab) and \n (newline) characters. Other characters, such as \r (carriage return), are also included in

what gets trimmed.

Math.floor – takes a floating point value. Returns the largest double that is less than or equal to the argument.

Math.ceil – returns the smaller double value that is greater than or equal to the argument and is equal to a mathematical integer. If the argument value is already equal to a mathematical integer, then the result is the same as the argument.

public static double ceil(double a)

Math.round – returns the closest long or int depending on the methods return type

long round(double d)

int round(float f)

Math.round(100.675) //101

Math.round(100.500) //100

Math.round(100) //100

Math.round(90f) //90

All numeric primitive wrapper classes extend the Number class. The Number class implements the Comparable interface.

StringBuilder

charAt()

indexOf()

length()

substring()

append()

Appends a char sequence to the end of the string builder

insert()

StringBuilder insert(int offset, String str)

ensureCapacity(int maxCapacity)

Collections:

1. When you declare an ArrayList variable you can omit specifying a class. If you use the non-generic version the compile time type checking is lost.

2. An ArrayList, just like an array is not thread safe. If you have multiple threads trying to add and remove elements from an ArrayList, you have to write additional code to ensure thread safety.

3. In Java 8 ArrayList declares a removeIf method which is used with lambdas. It processes the ArrayList members and removes the ones for which the lambda returns true.

4. Each time we add an element to an ArrayList that is not at the next position but supersedes it a

java.lang.IndexOutOfBoundsException is thrown. E.g.:

List list = new ArrayList();

list.add("val1"); //1

list.add(2, "val3"); //3 – this will throw an IndexOutOfBounds exception

list.add(3, "val2"); //2

Collections written without generics are also known as raw collections.

Java 8 provides a replaceAll() method that uses an UnaryOperator functional interface. It is used with lambdas to replace all the members within a List.

ArrayList has a method size() but doesn’t have a property length.

Converting between array and array list:

List<String> list = new ArrayList<>();

list.add(“hawk”);

list.add(“robin”);

Object[] objectArray = list.toArray();

System.out.println(objectArray.length);

String[] stringArray = list.toArray(new String[0]);

System.out.println(stringArray.length);

Collection interfaces:

List – an ordered collection that can contain duplicate entries. They have a dynamic size.

Map – a map is a collection that maps keys to values with no duplicate keys allowed

Set – a collection that does not allow repeating entries

Queue – a collection that orders its elements in a specific order for processing. A typical queue processes its elements in a first-in, first-out order but other orderings are possible.

List, Set and Queue inherit from the java.util.Collection interface. Map doesn’t implement it but it is considered part of the Java Collections framework.

**Common collection methods – methods for all sets, lists and queues.**

There are slight differences in implementations.

add() - for some collections add always returns true (e.g. ArrayList)

remove()

isEmpty()

size()

clear()

contains()

boolean contains(Object object)

ArrayList - The main benefit of an ArrayList is that you can look up any element in constant time. Adding or removing an element is slower than accessing an element. This makes an ArrayList a good choice when you are reading more often than (or the same amount as) writing to the ArrayList.

add()

boolean add(E element) – this boolean version always returns true.

void add(int index, E element)

remove()

boolean remove(Object object) – this boolean version returns true if the element was removed successfully and false otherwise. It removes only the first matching corresponding object within the ArrayList.

E remove(int index) – this version returns a reference to the removed object

set()

E set(int index, E newElement) – the E return type is the element that got replaced.

IsEmpty()

boolean isEmpty()

size()

int size()

clear() - void clear() - removes all the elements in the list

contains()

boolean contains(Object object)

equals() - ArrayList has a custom implementation of Object.equals() to compare two arrays

removeIf(Predicate<Object>) - mutates the array and removes all element for which the lambda returned true

int indexOf(E elem) – returns the first index where such element is found or -1 if there isn’t such an elements

int lastIndexOf(E elem) – returns the last index where such element is found or -1 if there isn’t such an elements

java.util.Arrays.sort(ary);

java.util.Collections.sort(col);

LinkedList – this collection implements both the Queue and List interface. It has all the methods that an ArrayList has and additional methods to manipulate elements at the beginning and the end of the LinkedList. **The main benefits of a LinkedList are that you can access, add, and remove from the**

**beginning and end of the list in constant time. The tradeoff is that dealing with an arbitrary index takes linear time.**

In previous versions of Java Vector was the only implementation of a List. ArrayList gradually had replaced it. Vector does the same thing but is slower because it is thread-safe.

Stack is another old API. It extends Vector.

ArrayDeque is used instead of a Stack nowadays.

Set interface – you use a set when you don’t want to allow duplicate entries and you aren’t concerned with the order of the entries.

A **HashSet** stores its elements in a hash table. This means that it uses the hashCode() method to retrieve them more efficiently. The main benefit is that adding elements and checking if an element is in the set both have constant time. The tradeoff is that you lose the order in which you inserted the elements. Most of the time we aren’t concerned with this when using a set, so this makes HashSet the most commonly used Set implementation. A HashSet may have empty rows.

A **TreeSet** stores its elements in a sorted tree structure. The main benefit is that the set is always in a sorted order. The tradeoff is that adding and checking if an element is present are both O(log n). TreeSet implements a special interface called NavigableSet, which lets you slice up the collection.

Here are some of the methods implemented in TreeSet from NavigableSet:

E lower(E e) Returns greatest element that is < e, or null if no such element

E floor(E e) Returns greatest element that is <= e, or null if no such element

E ceiling(E e) Returns smallest element that is >= e, or null if no such element

E higher(E e) Returns smallest element that is > e, or null if no such element

The hashCode() method is used to know which bucket to look in so that Java doesn’t have to look through the whole set to find out if an object is there.

boolean add() - returns true if an element is added to the set. It returns false if an element is already in the set.

Queue – we use a queue when elements need to be added in a specific order. Queues are typically used for sorting elements prior to accessing them. Unless stated otherwise, a queue is assumed to be FIFO (first-in, first-out). Some queue implementations change this to use a different order. The other common format is LIFO (last-in, first-out).

A LinkedList is a double-ended queue, besides being a List. A double-ended queue is different from the regular queue in that you can insert and remove elements from the front and from the end of the queue.

The main benefit of a **LinkedList** is that it implements both the List and Queue interfaces. The tradeoff is that it isn’t as efficient as a “pure” double-ended queue.

The ArrayDeque is a “pure” double-ended queue. Its stores its elements as a resizable array. It is designed especially to work as a double-ended queue.

Array Deque methods

Method → Description → For queue → For stack

boolean add(E e) Adds an element to the back of the queue and Yes No

returns true or throws an exception

E element() Returns next element or throws an exception if Yes No

empty queue

boolean offer(E e) Adds an element to the back of the queue and Yes No

returns whether successful

E remove() Removes and returns next element or throws an Yes No

exception if empty queue

E poll() Removes and returns next element or returns Yes No

null if empty queue

void push(E e) Adds an element to the front of the queue Yes Yes

void peek(E e) Returns next element or returns null if empty Yes Yes

void pop(E e) Removes and returns next elements or throws Yes Yes

an exception of empty

Map – it is a separate interface, it doesn’t belong to the Collection interface.

HashMap – stores the keys in a hash table. This means that it uses the hashCode() method of the keys to retrieve their values more efficiently. The main benefit is that adding and retrieving the elements by key both have constant time. The tradeoff is that you lose the order in which you inserted the elements.

Most of the time we use a Map we aren’t concerned with the order.

LinkedHasMap – a Map implementation where elements are stored in a specific order according to the way they are inserted.

TreeMap – stores the keys in a sorted three structure. The main benefit is that keys are always in sorted order. The tradeoff is that adding and checking if a key is present are both O(log n).

Hashtable – an old Map implementation, which like Vector is thread-safe which makes it not that optimal. HashMap is prefered over Hashtable nowadays.

Map methods:

void clear() Removes all keys and values from the map.

boolean isEmpty() Returns whether the map is empty.

int size() Returns the number of entries (key/value pairs) in the map.

V get(Object key) Returns the value mapped by key or null if none is mapped.

V put(K key, V value) Adds or replaces key/value pair. Returns previous value or null.

V remove(Object key) Removes and returns value mapped to key. Returns null if none.

boolean containsKey(Object key) Returns whether key is in map.

boolean containsValue(Object) Returns value is in map.

Set<K> keySet() Returns set of all keys.

Collection<V> values() Returns Collection of all values.

putIfAbsent(K key, V value) – sets the element only if there isn’t an element with such a key

merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction) - if the value is not present or null, associate it with the given value. If the value is present change its value according to the result of the BiFunction lambda.

The data structures that involve sorting don’t allow null. Those include ArrayDeque, Hashtabe, TreeMap, TreeSet.

Date and Time API

LocalDate - Contains just a date—no time and no time zone. Implements java.time.chrono.ChronoLocalDate

public static LocalDate of(int year, int month, int dayOfMonth)

public static LocalDate of(int year, Month month, int dayOfMonth)

isBefore()

isEqual()

isLeapYear()

isAfter()

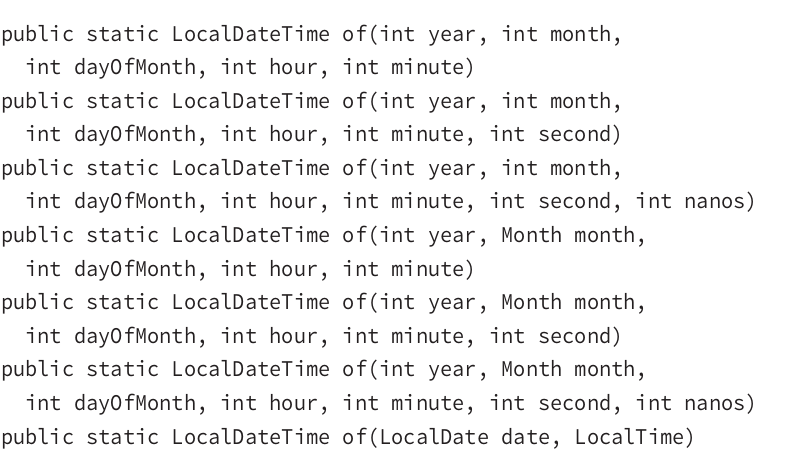
LocalTime - Contains just a time—no date and no time zone

public static LocalTime of(int hour, int minute)

public static LocalTime of(int hour, int minute, int second)

public static LocalTime of(int hour, int minute, int second, int nanos)

LocalDateTime - Contains both a date and time but no time zone



Each of the three classes has a static method called now() The key is to notice the type of information in the output. Here are corresponding sample results:

2015-01-20

12:45:18.401

2015-01-20T12:45:18.401

Java uses T to separate the date and time when converting LocalDateTime to a String.

Java tends to use a 24-hour clock.

The date and time classes have private constructors to force you to use the static methods. Those classes are immutable, just like the String class. This means that we need to remember to assign the results of the methods which manipulate the objects to a reference variable so they are not lost.

Another trick is to see what happens when you pass invalid numbers to of() . For example:

LocalDate.of(2015, Month.JANUARY, 32) // throws DateTimeException

A nanosecond is a billionth of a second. Java works with nanoseconds, unlike JS which work with milliseconds. (Miliseconds are a longer period).

ZonedDateTime

TemporalAmount – interface implemented by Duration and Period

Period – this class is immutable. There’s one catch. You cannot chain methods when creating a Period .Only the last method is used because the Period. ofXXX methods are static methods.

Period p = Period.ofMonths(1).ofDays(5); //only the ofDays call will take effect

We cannot use the Period class with LocalTime objects (objects that represent only time). We can use it with LocalDate and LocalDateTime.

With LocalTime.plus() and LocalTime.minus() we can use Duration objects, which represents a time duration.

Duration – this class is immutable. It cannot be instantiated directly. It works similarly to Period but it represents s time period of hours, minutes, seconds etc. It can be used with LocalTime methods.

The date and time classes belong to java.time package since Java 8.

java.time.format.DateTimeFormatter - can be used to format any type of date and/or time object.

LocalDate date = LocalDate.of(2020, Month.JANUARY, 20);

LocalTime time = LocalTime.of(11, 12, 34);

LocalDateTime dateTime = LocalDateTime.of(date, time);

System.out.println(date .format(DateTimeFormatter.ISO\_LOCAL\_DATE));

System.out.println(time.format(DateTimeFormatter.ISO\_LOCAL\_TIME));

System.out.println(dateTime.format(DateTimeFormatter.ISO\_LOCAL\_DATE\_TIME));

ISO is a standard for dates. The output of the previous code looks like this:

2020-01-20 //DateTimeFormatter.ISO\_LOCAL\_DATE

11:12:34 //DateTimeFormatter.ISO\_LOCAL\_TIME

2020-01-20T11:12:34 //DateTimeFormatter.ISO\_LOCAL\_DATE\_TIME

DateTimeFormatter shortDateTime = DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT);

System.out.println(shortDateTime.format(dateTime)); // 1/20/20

System.out.println(shortDateTime.format(date)); // 1/20/20

System.out.println(shortDateTime.format(time)); // UnsupportedTemporalTypeException

Available

FormatStyle.SHORT, FormatStyle.LONG, FormatStyle.MEDIUM, FormatStyle.FULL

java.time.format.FormatStyle is an enum type.

FormatStyle.LONG and FormatStyle.FULL support time zones.

Primitive wrapper classes comparison

Equals method of a primitive wrapper class ( e.g. java.lang.Integer, Double, Float etc) are

1. symmetric => a.equals(b) returns same as b.equals(a)

2. transitive => if a.equals(b) and b.equals(c) return true, then a.equals(c) returns true.

3. reflexive => a.equals(a) return true.

NumberFormat – class for formatting numbers and currency to a given locale

NumberFormat.getInstance() -> general purpose format

NumberFormat.getInstance(locale)

NumberFormat.getNumberInstance() -> same as getInstance

NumberFormat.getNumberInstance(locale)

NumberFormat.getCurrencyInstance() -> for formatting currency

NumberFormat.getCurrencyInstance(locale)

NumberFormat.getPercentageInstance() -> for formatting percentage

NumberFormat.getPercentageInstance(locale)

NumberFormat.getIntegerInstance() -> rounds decimal values

NumberFormat.getIntegerInstance(locale)

Once we have a NumberFormat instance we can call format() to turn a number into a String and parse()

to turn a string into a number. The format classes are not thread-safe. They shouldn’t be stored in instance variables or static variables.

NumberFormat nf = NumberFormat.getInstance();

String one = "456abc";

String two = "-2.5165x10";

String three = "x85.3";

System.out.println(nf.parse(one)); // 456

System.out.println(nf.parse(two)); // -2.5165

System.out.println(nf.parse(three));// throws ParseException because there are no numbers at the beginning of a String

The parse method parses only the beginning of a string. After it reaches a character that cannot be parsed, the parsing stops and the value is returned. The parse method can also be used for parsing currency:

String amt = “$92,087.99”;

NumberFormat cf = NumberFormat.getCurrencyInstance();

double value = (Double) cf.parse(amt);

System.out.println(value);

The currency string "$92,807.99" contains a dollar sign and a comma. The parse

method strips out the characters and converts the value to a number. The return value of parse is a Number object.

Applying the Fork/Join framework requires us to perform three steps:

1. Create a ForkJoinTask
2. Create the ForkJoinPool
3. Start the ForkJoinTask