Chapter 3: Core Java APIs.

API stands for Application Programming Interface. In java, an interface is something special. In the context of an API, it can be a group of class of interface definitions that gives you access to a service or functionality.

Creating and Manipulating Strings.

String class is fundamental in Java. It doesn't need a new keyword statement to create an instance:

```
String name = "Fluffy";
String name = new String("Fluffy");
```

Both creates a reference type variable that points to Fluffy.

Concatenation: placing one String before the other and combining them. + operand is used for this purpose. If both operands are numeric, it's an addition. If at least one operand is a String, it's concatenation. The expression is always evaluated from left to right.

Immutability: once a String is created is not allowed to change. It can't be made larger or smaller, and you cannot change one of the characters inside it.

The String pool: also known as the intern pool, is a location in the JVM that collects all strings, for reusing them and taking less memory while the program is executing. For the first instance of "Fluffy" above, Java will add it to the pool. For the second one with the new keyword, we are requesting Java to not use the pool and create an object, though is less efficient. This is not a good practice, but it's allowed.

Important String Methods.

Remember that String is a sequence of characters and Java counts from 0 when indexed. Here are most commonly used methods:

- length()
  Returns the number of characters in the String.
  ```
  variable.length()
  ```

- charAt()
  Lets you query the string to find out what character is at a specific index. Throws an exception if match not found.
  ```
  variable.charAt(2)
  ```

- indexOf()
  Search characters in the string and finds the first index that matches the desired value. It can work with individual characters or a whole String as input. It doesn't throw an exception if it cannot find a match.
  ```
  int indexOf(int ch)
  int indexOf(int ch, int fromIndex)
  int indexOf(String str)
  ```

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

- substring()
  Also looks for characters in a string and returns parts of it. Frist parameter is the index t start with for the returned string. The second parameter is optional (end of string). It throws an exception if indexes are backwards or greater than the length of the string.
  ```
  String substring(int beginIndex)
  String substring(int beginIndex, int endIndex)
  ```

- toLowerCase() and toUpperCase()
  Converts your data to lower and upper case.
  ```
  string.toLowerCase();
  string.toUpperCase();
  ```

- equals() and equalsIgnoreCase()
  Checks whether two String objects contain exactly the same characters in the same order. The second one does the same with the exception that it will convert the characters' case if needed.
  ```
  boolean equals(Object obj)
  boolean equalsIgnoreCase(String str)
  ```

- startsWith() and endsWith()
  They look at whether the provided value matches part of the String.
  ```
  boolean startsWith(String prefix)
  boolean endsWith(String suffix)
  ```

- contains()
  Also looks for matches in the String. The searched value can be anywhere within the String.
  ```
  boolean contains(String str)
  ```

- replace()
  It does a simple search and replaces the value in the String.
  ```
  boolean replace(char oldChar, char newChar)
  boolean replace(CharSequence oldChar, CharSequence newChar)
  ```

- trim()
  It removes whitespaces from the begging and end of the String but leaves the ones in the middle of it. Whitespaces consists of spaces along with the \t (tab) and \n (newline) characters; also including \r (carriage return).
  ```
  public String trim()
  ```

Method chaining.
Method chaining refers to calling methods / functionality chained one to another, which helps to save on lines of code and to avoid instantiating new variables. This is an example:

```
String result = "AniMaL     ".trim().toLowerCase().replace('a', 'A');
System.out.println(result);
```

Using the StringBuilder Class.

A small program can create a lot of String objects very quickly. Due to immutability, new objects will be created when trying to "modify" a string. With StringBuilder, the same space is reused when appending characters in a string. StringBuilder is not immutable, and when applying method chaining, the object changes its own state and returns a reference to itself.

```
StringBuilder a = new StringBuilder("abc");
StringBuilder b = a.append("de");
B = b.append("f").append("g");
System.out.println("a=" + a);
System.out.println("b=" + b);
// Both print "abcdefg" as they reference to the same object.
```

Creating a StringBuilder: We can make an instance in three ways using the new keyword, in which we may pass nothing, a string or a size. In the last scenario, we are suggesting Java what the size of the string will be, but the size can be increased if needed.

```
StringBuilder sb1 = new StringBuilder();
StringBuilder sb2 = new StringBuilder("animal");
StringBuilder sb3 = new StringBuilder(10);
```

Important StringBuilder Methods.

- charAt(), indexOf(), length() and substring()
  These work the same as in the String class.

- append()
  It adds the parameter to the StringBuilder and returns a reference to the current StringBuilder.
  ```
  StringBuilder append(String str)
  ```

- insert()
  Same as append, adds a parameter and returns a reference to the current StringBuilder, with the exception that we need to specify the offset / index where to insert it.
  ```
  StringBuilder insert(int offset, String str)
  ```

- delete() and deleteCharAt()
  Delete() removes characters from the sequencer and returns the current StringBuilder. DeleteCharAt() is convenient when you want to delete only one character in the sequence.
  ```
  StringBuilder delete(int start, int end)
  StringBuilder deleteCharAt(int index)
  ```

- reverse()

Reverses the characters in the sequence and returns a reference to the current StringBuilder.
`StringBuilder reverse()`

- toString()
  Converts a StringBuilder to String. `String s = sb.toString();`

StringBuilder vs. StringBuffer.
StringBuilder was introduced in Java 5. Before that, StringBuffer was used, which does the same thing but it's slower as it is thread safe.

Understanding Equality.
In Chapter two we saw equality for primitive types. Now, we see for non-primitives and to point out to never use == in real life scenarios, but rather for the certification exam.

```
StringBuilder one = new StringBuilder();
StringBuilder two = new StringBuilder();
StringBuilder three = one.append("a");
System.out.println(one == two);           // false
System.out.println(one == three);         // true
```

Both above are checking on reference equality. Next String examples check for reference equality as well. Remember how JVM reuses String literals by adding a single one to the pool / memory.

```
String x = "Hello World";
String y = "Hello World";
System.out.println(x == y);         // true
String z = " Hello World".trim();
System.out.println(x == z);         // false
```

Above, z variable is computed at runtime. Since x and z are not the same at compile time, a new String object is created. Below, we check for logical object equality. StringBuilder does not implements equals(), therefore if this method is called, it will check reference equality only.

```
System.out.println(x.equals(z));          // true
```

Understanding Java Arrays.
An array is an area of memory on the heap with space for a designated number of elements. A String is implemented as an array with some methods that you might want to use when dealing with characters. A StringBuilder is implemented as an array where the array object is replaced with a new bigger one when it runs out of space to store all the characters. A big difference is that an array can be of any other Java type.

```
char[] letters;
```

The variable letters is a reference, char is a primitive but it's the data type that goes into the array itself. It's an ordered list and can contain duplicates.

Creating an Array of primitives.

```
int[] numbers1 = new int[3];
```

Where int is the type of array, the brackets is required right after the type and the brackets at the end of the right-hand side indicates the size. All elements are set to the default value of the type. In this case, 0.

```
int[] numbers2 = new int[] {42, 55, 99};
```

In this case, we specify the initial values of the elements instead of using the default. The right-hand size is redundant for which Java lets you write the following:

```
int[] numbers2 = {42, 55, 99};
```

These declarations are the same:

```
int[] variable; int [] variable; int variable[]; int variable [];
```

Multiple declarations in a single line:

```
int[] ids, types; int ids[], types[]; int ids[], types;
```

Creating an Array of reference types.

```
String [] bugs = {"cricket", "bettle", "ladybug");
String [] alias = bugs;
System.out.println(bugs.equals(alias));   // true
System.out.println(bugs.toString());      // [Ljava.lang.String;@160bc7c0
```

In the sample above, we use String (non-primitive type) to build the object. Equals method checks for reference equality, it doesn't look for the values contained in the object. This array doesn't allocate space for the String objects. The array allocates space for a reference (pointer) to where the String objects are really stored. If not specified, the default values is null.

Using an Array.

Sorting. Java makes it easy by providing a sort method that you can pass almost anything to it: Arrays.sort(). Package java.util.Arrays or java.util.* is needed for this operation.

```
Arrays.sort(arrayVariable);
```

Searching. Java provides binary search but the array needs to be sorted. If not, the result is not predictable. When value is found, the index of the match is returned, otherwise a negative value.

```
Arrays.binarySearch(arrayVariable, searchedValue);
```

Multidimensional Arrays. It's basically putting an array inside of an array. These are the ways that can be declared:

```
int[][] vars1;
int vars2 [][];
int[] vars3[];
int[] vars4 [], space [][]; // a 2D and a 3D array
```

Size can be specified in the declaration. The following is a symmetric array:

```
String[][] rectangle = new String[3][2];
```

This is an asymmetric array:

```
int[][] differentSize = {{1,4}, {3}, {9,8,7}};
```

Another way to declare an asymmetric array:

```
int[][] args = new int[4][];
args[0] = new int[5];
args[1] = new int[3];
```

In order to iterate through a dimensional array, nested loops are used. These can handle both symmetric and asymmetric arrays. These are enhanced loops but regular ones are used as well:

```
for(int[] inner : twoD) {
     for(int num : inner)
          System.out.print(num + " ");
     System.out.println();
}
```

Understanding an ArrayList.

It has one shortcoming: you need to know how many elements will be in the array when you create it and then you are stuck with that choice. Like a StringBuilder, the size can change at runtime as needed. Like an Array, it is an ordered sequence that allows duplicates. It needs one of the following imports: java.util.* or java.util.ArrayList.

How to instance it:

```
ArrayList list1 = new ArrayList();
ArrayList list2 = new ArrayList(10);
ArrayList list3 = new ArrayList(list2);
```

Using generics, introduced in Java 5:

```
ArrayList<String> list4 = new ArrayList<String>();
ArrayList<String> list5 = new ArrayList<>();
```

Since ArrayList implements the interface List, the following can be done:

```
List<String> list6= new ArrayList<>();
ArrayList<String> list7 = new List<>();        // Does not compile
```

Important ArrayList Methods.

A new class or element is used in the method signatures: E.

E is used by convention in generics to mean any class that this array can hold. E means anything you put inside < >, and when nothing is specified, E means Object. ArrayLists implement toString().

- add()
  Insert a new value in the ArrayList. Method signature:
  ```
  boolean add(E element)       // Always returns true
  void add(int index, E element)
  ```

- remove()

Removed the first matching value in the ArrayList or remove the element at the specified index. Method signature:

```
boolean remove(Object object) // Returns true or false
E remove(int index)           // Returns element removed
```

- set()
  Changes one of the element of the ArrayList without changing its size. Method signature:
  ```
  E set(int index, E newElement)
  ```

- isEmpty() and size()
  Look at how many slots are in use. Methods signatures:
  ```
  boolean isEmpty()
  int size()
  ```

- clear()
  It discards all elements of the ArrayList at once. Method signature:
  ```
  void clear()
  ```

- contains()
  Checks whether a certain value is in the ArrayList. Method signature:
  ```
  boolean contains(Object object)
  ```

- equals()
  This is a custom implementation of equals) and it's able to compare two lists to see if they contain the same elements in the same order. Method signature:
  ```
  boolean equals(Object object)
  ```

Wrapper Classes.

What happens if we want to put a primitive type in an ArrayList? We can put a wrapper class, which is an object type that corresponds to each primitive type.

Primitive	Wrapper Class	
boolean	Boolean	new Boolean(true)
byte	Byte	new Byte((byte) 1)
short	Short	new Short((short) 1)
int	Integer	new Integer(1)
long	Long	new Long(1)
float	Float	new Float(1.0)
double	Double	new Double(1.0)
char	Character	new Character('c')

Converting from a String:

Wrapper Class	Converting String to Prim.	Converting String to Wrapper
Boolean	Boolean.parseBoolean("true");	Boolean.valueOf("TRUE");
Byte	Byte.parseByte("1");	Byte.valueOf("2");

Short	Short.parseShort("1");	Short.valueOf("2");
Integer	Integer.parseInt("1");	Integer.valueOf("2");
Long	Long.parseLong("1");	Long.valueOf("2");
Float	Float.parseFloat("1");	Float.valueOf("2.2");
Double	Double.parseDouble("1");	Double.valueOf("2.2");
Character	None	None

Autoboxing. When converting from String to primitive or wrapper class, we don't need to worry which one is returned. Since Java 5, you can type any primitive value and Java will convert it to the relevant wrapper class for you. Examples in WrapperClasses.java file.

Converting between array and List.
From List to Array:

```
List<String> list = new ArrayList<>();
Object[] objectArray = list.toArray();
String[] stringArray = list.toArray(new String[0]);
```

The last one is the right way to convert from ArrayList to Array. By specifying the size as 0, Java will create a new array with the proper size for the return value.
From Array to List:

```
String[] array = {"hawk", "robin"};
List<String> list = Arrays.asList(array);
```

Now both array and list variables reference to the same data, and the data objects can be modified through any of the two references. Deleting and adding not possible since the object is a fixed size, even when we convert array into list.

Sorting.
In order to sort, you need to use a different helper class:

```
Collections.sort(numbers2);
```

Working with Dates and Times.
In order to work with Dates and Times, the following import is necessary: java.time.*
Three ways to work with dates in java and the OCA exam: LocalDate (contains just a date – no time, no time zone), LocalTime (contains just time – no date, no time zone) and LocalDateTime (contains both date and time – no time zone). Oracle recommends to avoid time zones unless they are really needed. For this, ZonedDateTime handles it.

```
LocalDate.now();
LocalTime.now();
LocalDateTime.now();
```

Method signatures for Date:

```
public static localDate of(int year, int month, int dayOfMonth)
public static localDate of(int year, Month month, int dayOfMonth)
```

OCA Chapter 3 Notes

Month is an enum (enmus are java classes). For months, Java counts starting from 1, making Months an exception from everything else, which Java starts counting from 0.

Method signatures for Time:

```
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)
```

Method signatures for DateTime:

```
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour,
int minute)
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour,
int minute, int second)
public static LocalDateTime of(int year, int month, int dayOfMonth, int hour,
int minute, int second, int nanos)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int
hour, int minute)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int
hour, int minute, int second)
public static LocalDateTime of(int year, Month month, int dayOfMonth, int
hour, int minute, int second, int nanos)
public static LocalDateTime of(LocalDate date, LocalTime time)
```

The date and time classes have private constructors to for you to use the static method. Therefore, the new keyword / constructor instance can't be applied:

```
LocalDate d = new LocalDate();     // Does not compile
```

Dates and Times are immutable, just like String was. This means we need to remember to assign the results to a reference variable, so they are not lost.

Working with Periods.

We use a Java class called Periods to work with date times that change in our code, so the code can be reusable by just adjusting the period. There are 5 ways to create a Period class:

```
Period annually = Period.ofYears(1);
Period quarterly = Period.ofMonths(3);
Period everyThreeWeeks = Period.ofWeeks(3);
Period everyOtherDay = Period.ofDays(2);
Period everyYearAndWeek = Period.of(1, 0, 7);
```

Method chaining for Periods is not allowed, since only the last method in the chain will be applied to the period. E.g.: `Period.ofYears(1).ofWeeks(1);`

There's another class called *Duration*, which is applied to hours, minutes, seconds and nanos, but it's out of the scope for the OCA.

Formatting Dates and Times.

We can specify the format we want to handle the data of the objects. ISO is the standard for dates and would like this:

```
dateTime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
```

Where dateTime is an object and DateTimeFormatter is the implementation of the class. Same can be applied to either Date or Time.

Also, a format can be defined in a object before applying to the Date object as follow:

```
DateTimeFormatter shortDateTime =
DateTimeFormatter._____(FormatStyle.SHORT);
```

Where blank can be filled with: ofLocalizedDate [for Date and DateTime], ofLocalizedDateTime [for DateTime only] or ofLocalizedTime [for Time and DateTime].

There are two predefined formats: SHORT and MEDIUM.
A SHORT DateTime format looks like this: *01/20/20 11:12 AM*
A MEDIUM DateTime format looks like this: *Jan 20, 2020 11:12:34 AM*

You can also create your own format by using the method ofPattern(String str). E.g.:

```
DateTimeFormatter f = DateTimeFormatter.ofPattern("MMMM dd, yyyy,
hh:mm");
```

The more Ms, the more verbose Java will display the month.

You can also use the parse() method to pass a String to be converted to Date / Time / DateTime. E.g.:

```
LocalDate date = LocalDate.parse(“01 02 2015”, f);
```

Where f is a DateTimeFormatter object.

```
LocalTime time = LocalTime.parse(“11:22”);
```