Reading and Working with the Java API Documentation

(References used: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html and https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html)

Purpose

The purpose of this document is to explain how a Java API works and how to use it to determine appropriate syntax while working with Java code. By the end of the document, the information provided in the Java API documentation should be fully understandable.

I. Examining the Header

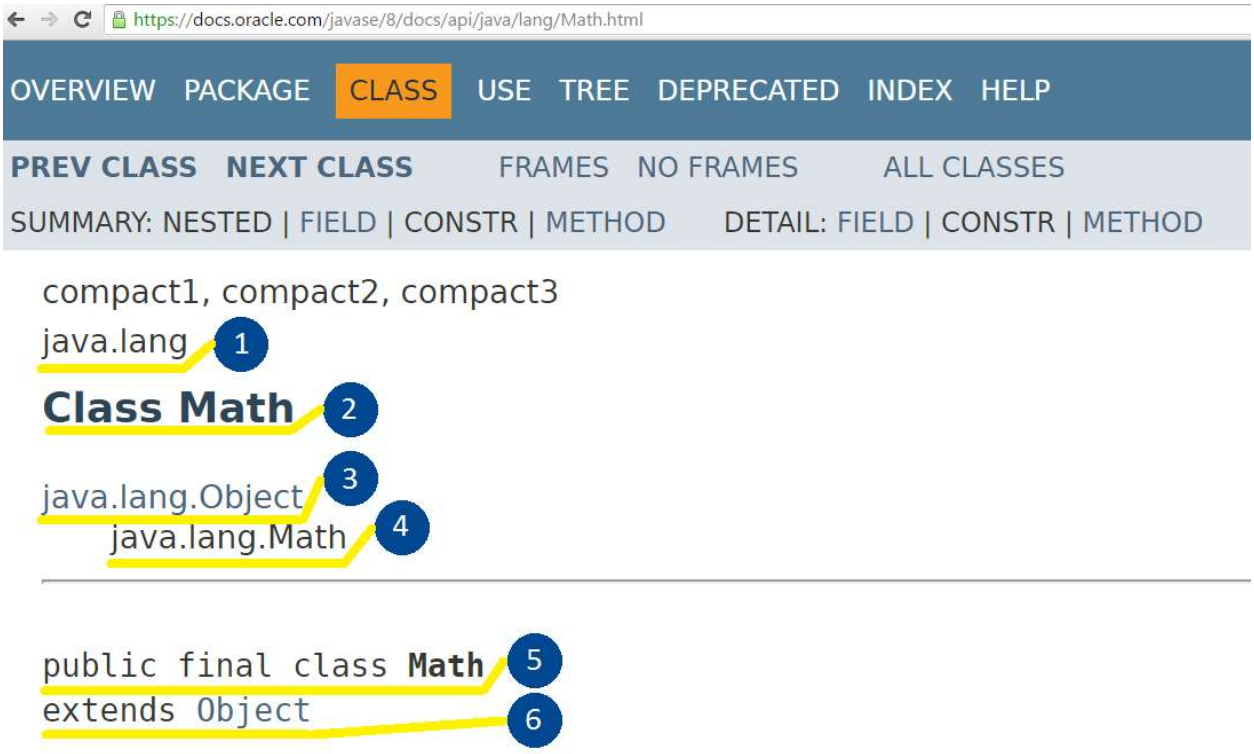


Figure 1 - The Header contains the class name and hierarchy information, including package and inheritance references.

The first part of the documentation for an API contains critical naming and inheritance information regarding the class library. Each of the numbered sections above helps to determine the class hierarchy and package imports that are needed to use the code.

1. The information on the first line is the package where this class exists. For java.lang.Math, this first line is telling us the "Math" class is in the "java.lang" package. (java.lang is imported into every project by default, so "Math" is always accessible).
2. The second line contains the declaration of the object type (here it is a Class). Choices can be Package, Class, Abstract Class, Interface, or Enum. The second part of the line contains the name of the object (also called the "identifier"). In this example we know that this object is a

"Class" and its identifier (name) is "Math." From line 1, we know this Class Math exists in the package java.lang.

3. Lines 3 and 4 here show the object-inheritance hierarchy. In this case, we see that the line java.lang.Object. In this instance, there is only one "superclass" to "Math". In other APIs, the tree may show multiple lines here denoting the hierarchy in depth. For example, take a look at the API for the "Vector" class:

Using what we know already, Vector is the name of the class, and it lives in the package "java.util." Note the object inherits directly from "java.util.AbstractList" which inherits from "java.util.AbstractCollection" which inherits from "java.lang.Object."

- Now we know three things:
    - How to read the hierarchy tree of an object
    - All Classes (Object Types) we create in Java will ultimately inherit from "java.lang.object", in some direct or indirect way
    - Classes in one package may inherit from classes in another package

4. Line 4 is once again the identifier for this class, fully referenced with its direct package implementation (i.e. "java.lang.Math" or "java.util.Vector").
5. Lines 5 and 6 also go together. In the last part of the header shown in figure one, we get the class declaration as it would appear in code. In this case, "public final class Math extends Object". Line 5 will be the public declaration of the object.
6. Line 6 may be multiple lines depending on the implementation. In the "Math" class there is only one item to extend, "Object". Remember that the "extends Object" is implicit on any class declaration so it is never directly typed in code.

There is one last part of the header to consider. Going back to the Vector class is a great way to show this missing part, because there is a class that extends Vector, and Vector also implements interfaces. When classes implement interfaces or are extended by other classes, they are listed in the header. This not only reiterates the hierarchy, but allows the user to click on each item to see those specific implementations:

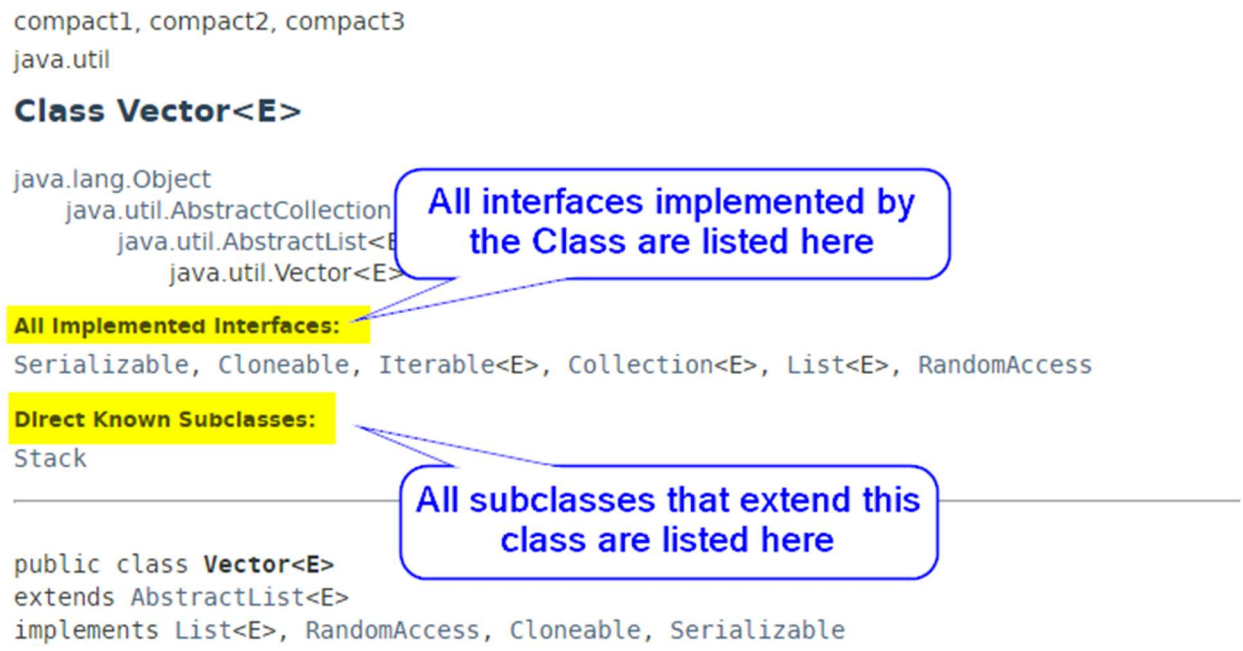


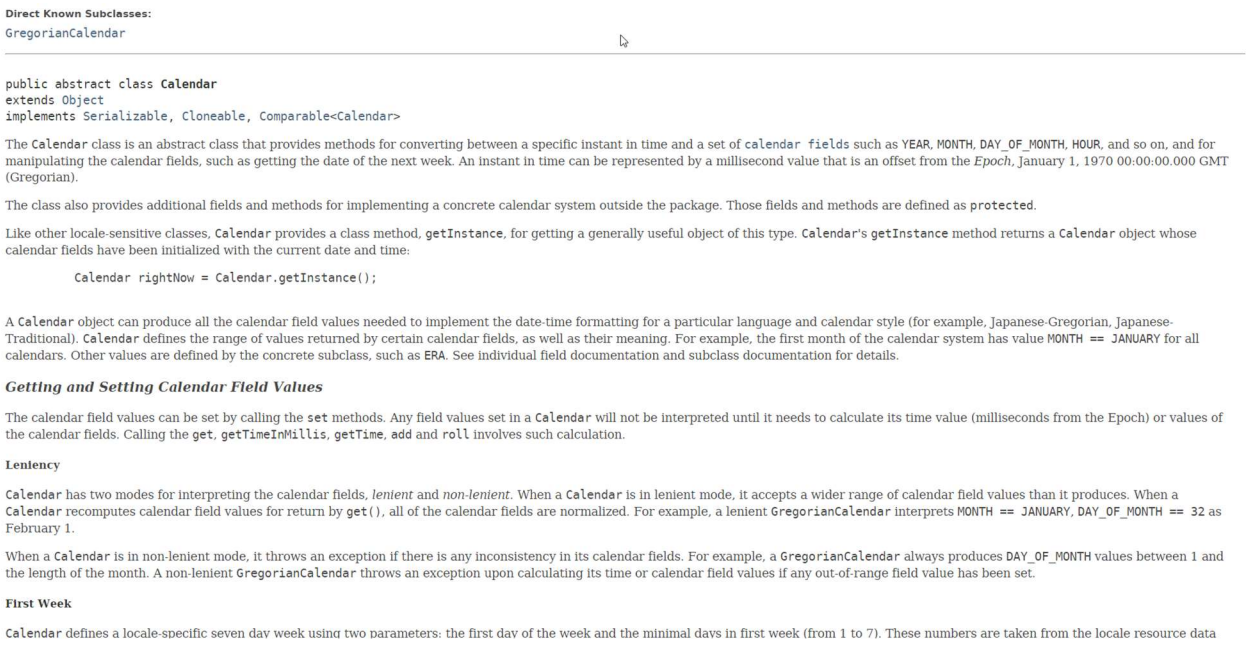
Figure 2- The Vector class implements interfaces and has a subclass that extends the Vector class

II. The Description

The next part of the documentation contains a description of the object that is created. This often gives in-depth information that is critical to understanding the operation of the object in our code. Therefore it is important to read through the information when we are not familiar with the obect. For example, the Calendar Object has an intense description because of the complexity of how the calendar works:

https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html

(Oracle)



Direct Known Subclasses:

GregorianCalendar

```
public abstract class Calendar
extends Object
implements Serializable, Cloneable, Comparable<Calendar>
```

The `Calendar` class is an abstract class that provides methods for converting between a specific instant in time and a set of `calendar fields` such as `YEAR`, `MONTH`, `DAY_OF_MONTH`, `HOUR`, and so on, and for manipulating the calendar fields, such as getting the date of the next week. An instant in time can be represented by a millisecond value that is an offset from the *Epoch*, January 1, 1970 00:00:00.000 GMT (Gregorian).

The class also provides additional fields and methods for implementing a concrete calendar system outside the package. Those fields and methods are defined as `protected`.

Like other locale-sensitive classes, `Calendar` provides a class method, `getInstance`, for getting a generally useful object of this type. `Calendar`'s `getInstance` method returns a `Calendar` object whose calendar fields have been initialized with the current date and time:

```
Calendar rightNow = Calendar.getInstance();
```

A `Calendar` object can produce all the calendar field values needed to implement the date-time formatting for a particular language and calendar style (for example, Japanese-Gregorian, Japanese-Traditional). `Calendar` defines the range of values returned by certain calendar fields, as well as their meaning. For example, the first month of the calendar system has value `MONTH == JANUARY` for all calendars. Other values are defined by the concrete subclass, such as `ERA`. See individual field documentation and subclass documentation for details.

Getting and Setting Calendar Field Values

The calendar field values can be set by calling the `set` methods. Any field values set in a `Calendar` will not be interpreted until it needs to calculate its time value (milliseconds from the Epoch) or values of the calendar fields. Calling the `get`, `getTimeInMillis`, `getTime`, `add` and `roll` involves such calculation.

Leniency

`Calendar` has two modes for interpreting the calendar fields, *lenient* and *non-lenient*. When a `Calendar` is in lenient mode, it accepts a wider range of calendar field values than it produces. When a `Calendar` recomputes calendar field values for return by `get()`, all of the calendar fields are normalized. For example, a lenient `GregorianCalendar` interprets `MONTH == JANUARY`, `DAY_OF_MONTH == 32` as February 1.

When a `Calendar` is in non-lenient mode, it throws an exception if there is any inconsistency in its calendar fields. For example, a `GregorianCalendar` always produces `DAY_OF_MONTH` values between 1 and the length of the month. A non-lenient `GregorianCalendar` throws an exception upon calculating its time or calendar field values if any out-of-range field value has been set.

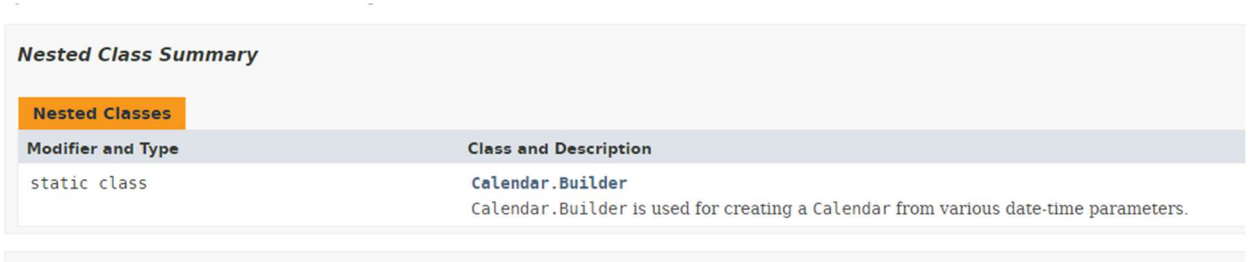
First Week

`Calendar` defines a locale-specific seven day week using two parameters: the first day of the week and the minimal days in first week (from 1 to 7). These numbers are taken from the locale resource data

Figure 3- The Calendar class has a very lengthy description as only part of it is shown here. This is based on the complexity of the Calendar object.

III. Nested Classes

If the object we are viewing has any nested classes within the object, they will be the next item listed in the API. For example, the Calendar object has a nested class “Builder,” which can be used to create a Calendar object with specific settings.



Nested Class Summary

Nested Classes

Modifier and Type	Class and Description
static class	**Calendar.Builder** Calendar.Builder is used for creating a Calendar from various date-time parameters.

Figure 4 - The nested Calendar.Builder class is listed in the API for Calendar

IV. Fields and Constants

Some classes will expose fields to the public for use. This is again very common in the Calendar object, because we have a lot of properties we want direct access to, such as “DAY_OF_MONTH”, “FRIDAY”, “JANUARY”, “HOUR”, etc. Because we want to be able to quickly get access to these values, they are exposed to be easily retrieved. We can see many fields that are exposed in the “Calendar” object, and we will use these, often when we want to do any type of loop or condition check against standard values (i.e. current month < October, current day == Friday, etc).

Field Summary

Fields

Modifier and Type	Field and Description
`static int`	`ALL_STYLES` A style specifier for `getDisplayNames` indicating names in all styles, such as "January" and "Jan".
`static int`	`AM` Value of the `AM_PM` field indicating the period of the day from midnight to just before noon.
`static int`	`AM_PM` Field number for `get` and `set` indicating whether the `HOUR` is before or after noon.
`static int`	`APRIL` Value of the `MONTH` field indicating the fourth month of the year in the Gregorian and Julian calendars.
`protected boolean`	`areFieldsSet` True if `fields[]` are in sync with the currently set time.
`static int`	`AUGUST` Value of the `MONTH` field indicating the eighth month of the year in the Gregorian and Julian calendars.
`static int`	`DATE` Field number for `get` and `set` indicating the day of the month.
`static int`	`DAY_OF_MONTH` Field number for `get` and `set` indicating the day of the month.
`static int`	`DAY_OF_WEEK` Field number for `get` and `set` indicating the day of the week.
`static int`	`DAY_OF_WEEK_IN_MONTH` Field number for `get` and `set` indicating the ordinal number of the day of the week within the current month.
`static int`	`DAY_OF_YEAR` Field number for `get` and `set` indicating the day number within the current year.
`static int`	`DECEMBER` Value of the `MONTH` field indicating the twelfth month of the year in the Gregorian and Julian calendars.
`static int`	`DST_OFFSET` Field number for `get` and `set` indicating the daylight saving offset in milliseconds.
`static int`	`ERA` Field number for `get` and `set` indicating the era, e.g., AD or BC in the Julian calendar.
`static int`	`FEBRUARY` Value of the `MONTH` field indicating the second month of the year in the Gregorian and Julian calendars.
`static int`	`FIELD_COUNT`

Figure 5 - Some of the fields that are listed for the Calendar object are shown here.

The Math API also has fields for constants such as E and PI:

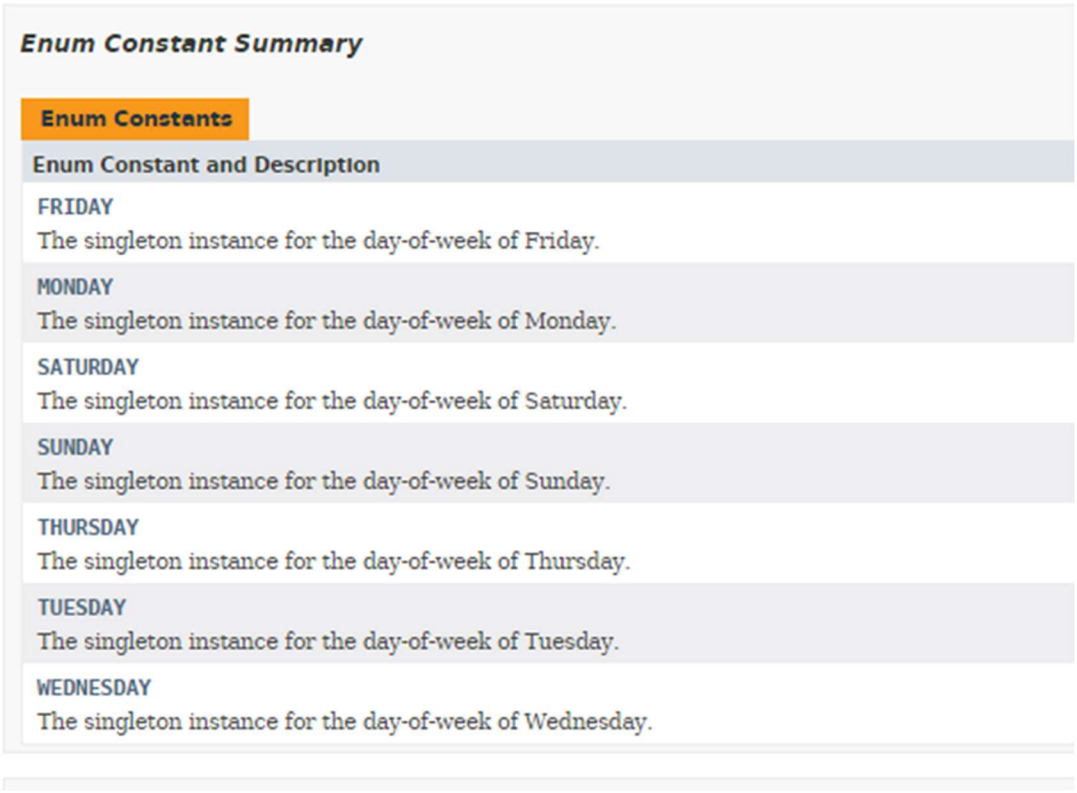
Field Summary

Fields

Modifier and Type	Field and Description
`static double`	`E` The `double` value that is closer than any other to *e*, the base of the natural logarithms.
`static double`	`PI` The `double` value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Figure 6 - Math fields include values for the constants E and PI

In the absence of “Fields,” objects like an Enum (Enumeration) would list their “Constants” instead. For example, the Enum for “DayOfWeek” shows the following Constants:



Enum Constant Summary

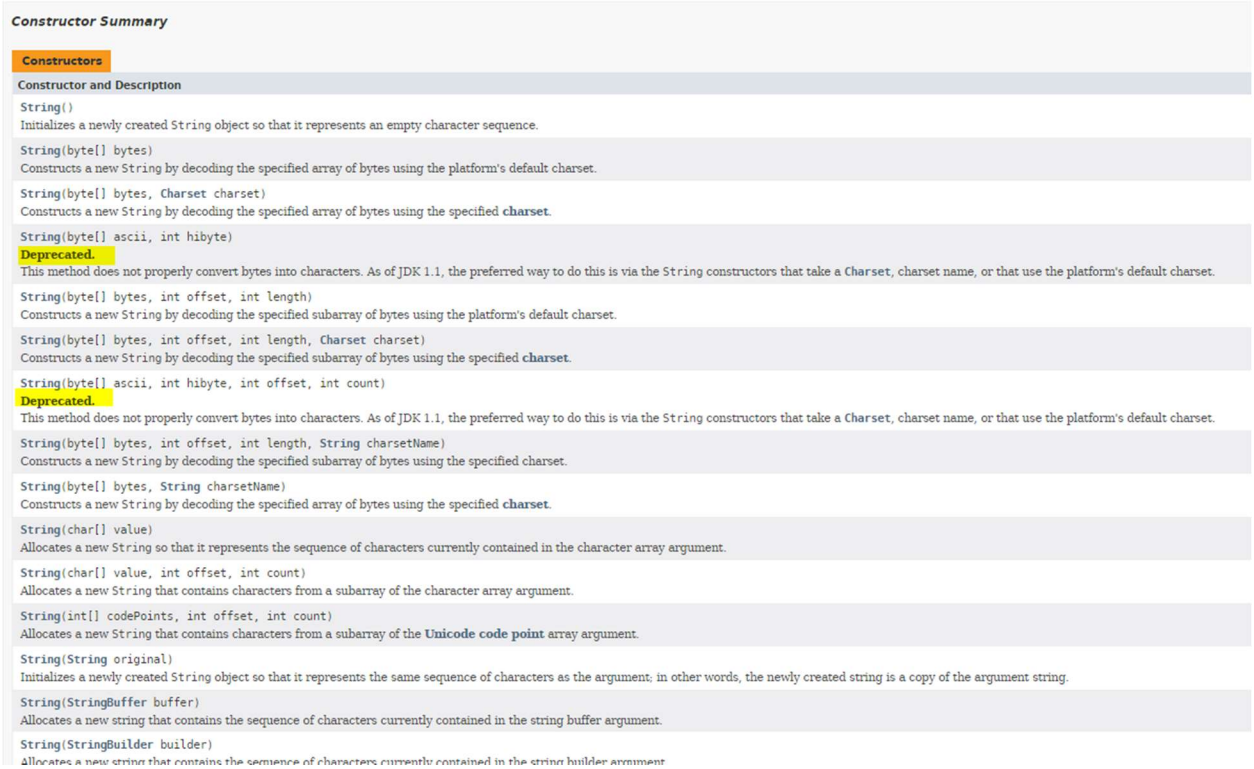
Enum Constants

Enum Constant and Description
FRIDAY The singleton instance for the day-of-week of Friday.
MONDAY The singleton instance for the day-of-week of Monday.
SATURDAY The singleton instance for the day-of-week of Saturday.
SUNDAY The singleton instance for the day-of-week of Sunday.
THURSDAY The singleton instance for the day-of-week of Thursday.
TUESDAY The singleton instance for the day-of-week of Tuesday.
WEDNESDAY The singleton instance for the day-of-week of Wednesday.

Figure 7- The constants from the Enum "DayOfWeek" https://docs.oracle.com/javase/8/docs/api/java/time/DayOfWeek.html

V. Constructors

Constructors are the defining methods in a Class that allow creation of the object in code. All Java objects have a default constructor, and some will also contain explicit constructors. Whenever an explicit constructor (one that takes parameters) exists, then the default is not implicit and needs to be specifically defined in order to be used. The Math class does not have a constructor defined, because all of the methods are static in the class. This means the object can be used in code directly, without instantiation. Most classes will contain one or more constructors, however. The Java class "String" has a multitude of constructors, which allows creating Strings in quite a few different ways:



Constructor Summary

Constructors

Constructor and Description
`String()` Initializes a newly created `String` object so that it represents an empty character sequence.
`String(byte[] bytes)` Constructs a new `String` by decoding the specified array of bytes using the platform's default charset.
`String(byte[] bytes, Charset charset)` Constructs a new `String` by decoding the specified array of bytes using the specified **charset**.
`String(byte[] ascii, int hibyte)` **Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.
`String(byte[] bytes, int offset, int length)` Constructs a new `String` by decoding the specified subarray of bytes using the platform's default charset.
`String(byte[] bytes, int offset, int length, Charset charset)` Constructs a new `String` by decoding the specified subarray of bytes using the specified **charset**.
`String(byte[] ascii, int hibyte, int offset, int count)` **Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the preferred way to do this is via the `String` constructors that take a `Charset`, charset name, or that use the platform's default charset.
`String(byte[] bytes, int offset, int length, String charsetName)` Constructs a new `String` by decoding the specified subarray of bytes using the specified charset.
`String(byte[] bytes, String charsetName)` Constructs a new `String` by decoding the specified array of bytes using the specified **charset**.
`String(char[] value)` Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument.
`String(char[] value, int offset, int count)` Allocates a new `String` that contains characters from a subarray of the character array argument.
`String(int[] codePoints, int offset, int count)` Allocates a new `String` that contains characters from a subarray of the **Unicode code point** array argument.
`String(String original)` Initializes a newly created `String` object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
`String(StringBuffer buffer)` Allocates a new string that contains the sequence of characters currently contained in the string buffer argument.
`String(StringBuilder builder)` Allocates a new string that contains the sequence of characters currently contained in the string builder argument.

Figure 8- The list of constructors for the String class in Java is extensive

Notice that even Deprecated methods are shown. When the constructor or method (or even object) is listed as "Deprecated," it is wise to find another alternative to using that method, constructor, or object.

VI. Methods

Methods are how the code is able to be used to perform some functionality. Often methods will require one or more parameters. The API helps us determine how to correctly call methods, as well as gives us the ability to get a brief description of the method functionality. Clicking on the method will give more specific detail. Additionally, the method will have a return type.

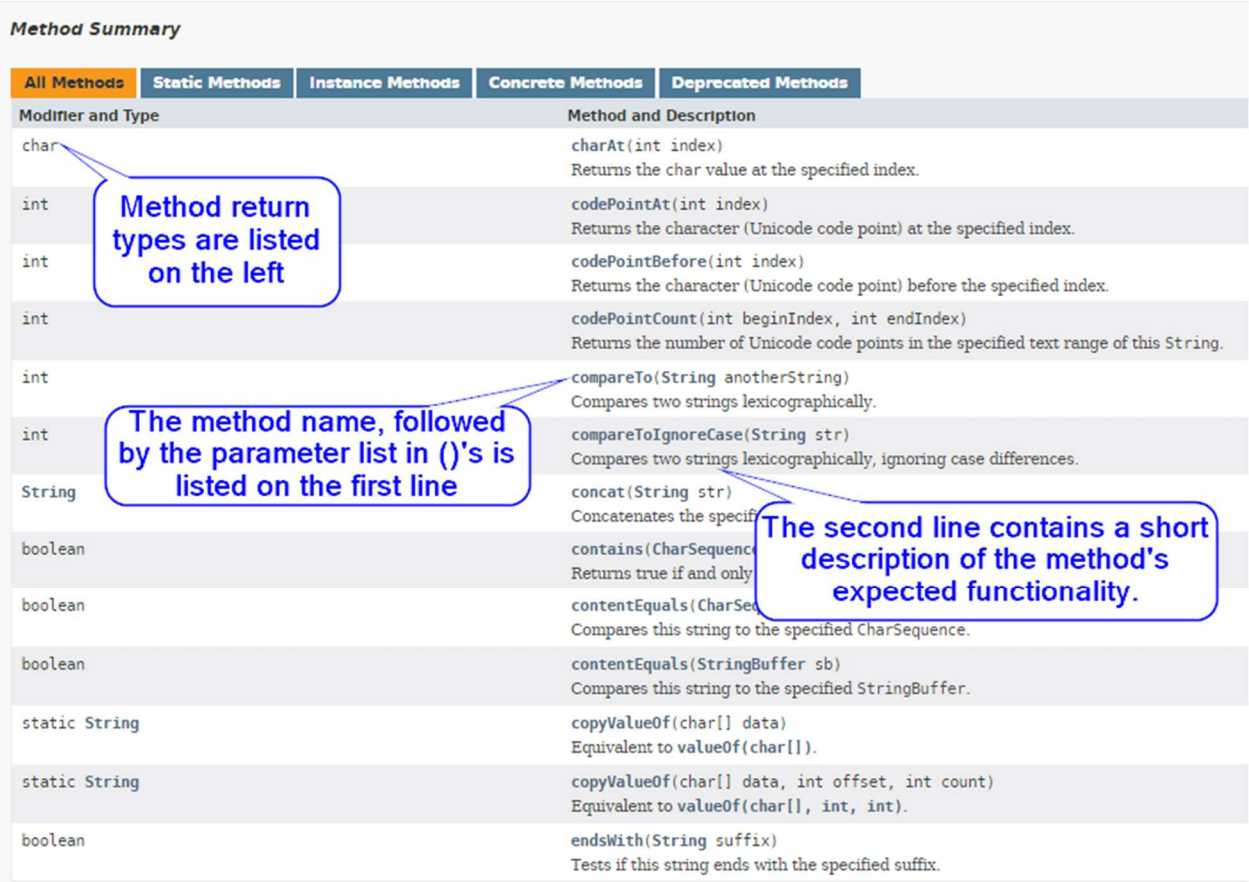
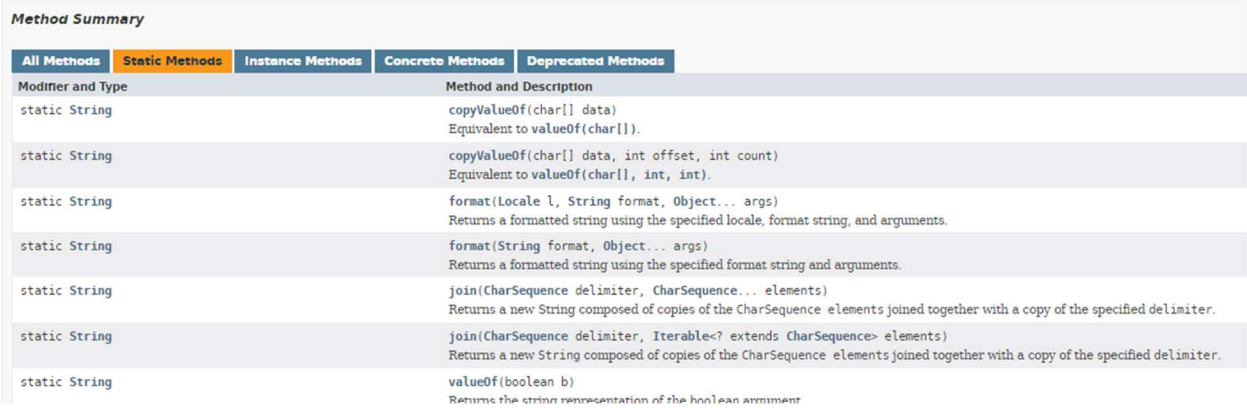


Figure 9- Some of the methods from "String" are listed here, with callouts to show how the section is organized

The method summary also allows filtering by clicking on the columns at the top. For example, if only the static methods are of interest, a user can click on “Static Methods” to see just the list of those methods that are static on the object:



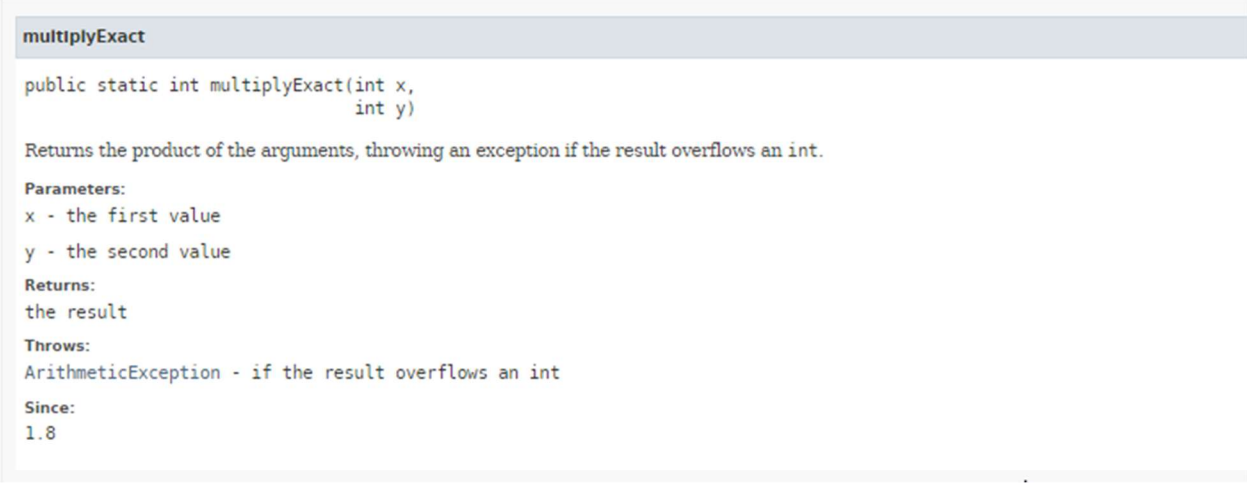
Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method and Description
static String	copyValueOf(char[] data) Equivalent to valueOf(char[]).
static String	copyValueOf(char[] data, int offset, int count) Equivalent to valueOf(char[], int, int).
static String	format(Locale l, String format, Object... args) Returns a formatted string using the specified locale, format string, and arguments.
static String	format(String format, Object... args) Returns a formatted string using the specified format string and arguments.
static String	join(CharSequence delimiter, CharSequence... elements) Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.
static String	join(CharSequence delimiter, Iterable<? extends CharSequence> elements) Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter.
static String	valueOf(boolean b) Returns the string representation of the boolean argument.

Figure 10 - Only static methods are shown when the list is filtered using the buttons on the top of the Method Summary

Clicking on any method gives more information about the specific method in detail:



multiplyExact

```
public static int multiplyExact(int x,
                                int y)
```

Returns the product of the arguments, throwing an exception if the result overflows an int.

Parameters:

x - the first value

y - the second value

Returns:

the result

Throws:

ArithmeticException - if the result overflows an int

Since:

1.8

Figure 11 - Clicking on the "MultiplyExact" method in the Math API brings further detail about the multipleExact method.

VII. Conclusion

The Java API documentation available online is a powerful tool that should be referenced frequently when working with the pre-defined Java API objects. By looking over the API documentation, it is possible to understand where the object stands in the Object hierarchy, how to create the object, and how to work with the object's methods and properties to accomplish the task at hand in code. Whenever in doubt, information about these objects is never more than a quick Google search away!