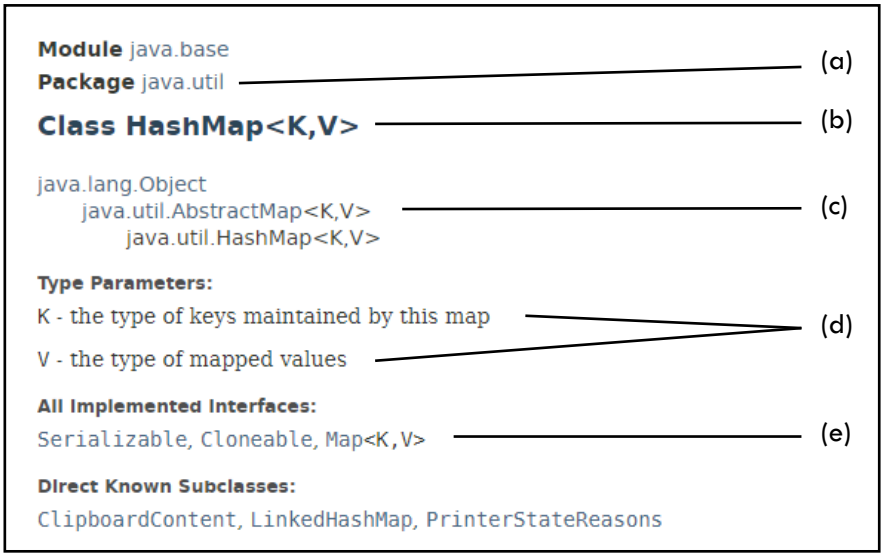
GUIDE TO JAVADOC

Throughout this guide, I'll be using the Documentation of the following classes:

- java.util.HashMap
- java.lang.String

OVERVIEW

When you first open the Java Documentation for any Java class, the first thing you see is usually something like this:



Overview of the HashMap class

(a) The package tells you the full path of the class, as well as what you need to import to use the class.

In this case, to use a HashMap, you should have the following line at the top of your code:

```
import java.util.HashMap;
```

Alternatively, you can import the entire package (which is what the skeleton file does).

```
import java.util.*;
```

Usually, you don't have to include any additional import statements. Most of the things you need can be found in the `java.util` package.

The `java.lang` package is imported by default. (i.e. you don't have to import anything to use the `java.lang.String` class or the `java.lang.Math` class, etc.)

(b) If you see those angular braces next to the class name, remember that these are type parameters for Java Generics. When using the class, substitute these with the types you need.

```
HashMap<String, Integer> studentScoreMap;
```

Remember that you cannot use primitive types for type arguments. (e.g. No `int` or `char`, instead use `Integer` or `Character`)

(c) The inheritance chain of the class, where you can see all the base classes (i.e. superclasses).

In this case, a `HashMap<K,V>` is a subclass of `Map<K,V>` and a subclass of `Object`. This means you can use `HashMap<K,V>` for any method that takes in a `Map<K,V>` or `Object` as an argument.

(d) The meaning of each type parameter.

(e) The list of implemented `Interface`s of this class.

For example, if you look at the documentation for the `String` class, you'll notice that it implements the `Comparable<String>` interface, meaning you can call `Collections.sort()` on a List of `String`s by default (i.e. without creating your own custom comparator).

CONSTRUCTORS

The constructor of a class is **that thing you use with the `new` keyword.**

```
char[] catParts = new char[]{'c', 'a', 't'};
String cat = new String(catParts); // String constructor
```

In the documentation, you can see the list of available constructors.

Constructor Summary

Constructors

Constructor	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[] 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[] ascii, int hibyte, int offset,`	**Deprecated.** This method does not properly convert bytes into characters. As of JDK 1.1, the

Constructor Summary of the String class (truncated)

You can click on any of the constructors for more information.

String

`public String(char[] value)`

Allocates a new `String` so that it represents the sequence of characters currently contained in the character array argument. The contents of the character array are copied; subsequent modification of the character array does not affect the newly created string.

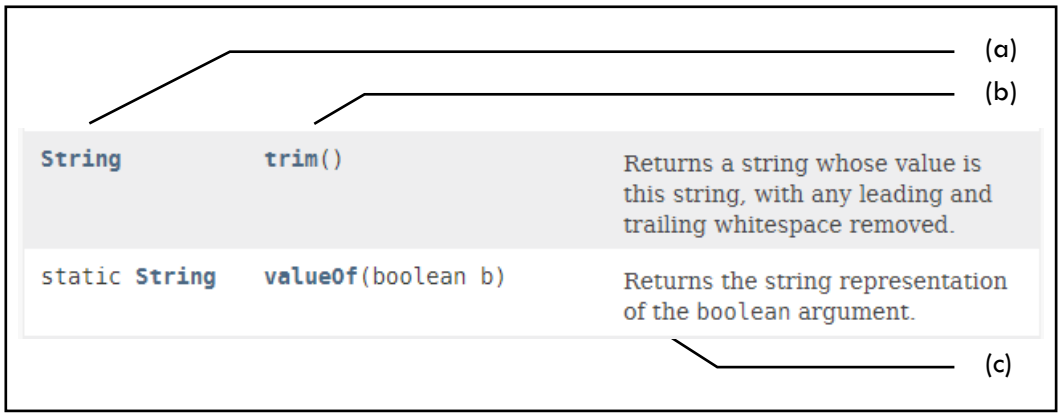
Parameters:

`value` - The initial value of the string

Detailed information about `String(char[])` constructor

METHODS

In the method summary, you'll see the list of methods that the class supports.



Method summary of the `String` class (truncated)

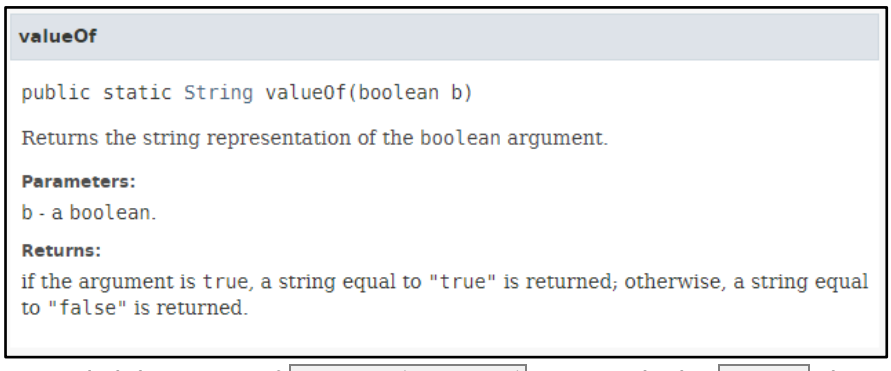
(a) The return type of the method. If the keyword `static` is there, then it is a **static method**. Otherwise, it is an **instance method**.

```
// Instance method
String strInstance = "  spacious  ";
System.out.println(strInstance.trim()); // "spacious"

// Static method
System.out.println(String.valueOf(true)); // "true"
```

If you're unsure what instance methods or static methods are, go to the section labelled **"Instance vs Static"**.

(b) The method signature, consisting of its name and argument type(s). Clicking the method name will give you a more detailed description.



valueOf

`public static String valueOf(boolean b)`

Returns the string representation of the `boolean` argument.

Parameters:

`b` - a `boolean`.

Returns:

if the argument is `true`, a string equal to `"true"` is returned; otherwise, a string equal to `"false"` is returned.

Detailed description of `valueOf(boolean)` static method in `String` class

(c) Summary of the method.

INSTANCE VS STATIC

If you're unclear about the differences between "instance methods" and "static methods", here's a short explanation.

Look at this Ball class:

```
class Ball {
    int radius;

    public Ball(int radius) {...}
    public double getVolume() {...}
}
```

The class itself can be seen as a "blueprint" or a "factory" for a `Ball` (as opposed to saying that "this class represents a `Ball`"). This blueprint says that:

- A Ball should have a radius.
- A Ball should be able to calculate its volume.

It wouldn't make sense to ask the class itself for information about its "radius" or "volume", since the blueprint/factory doesn't have that information.

```
// Hey factory, what's your radius?
System.out.println(Ball.radius); // wat?
// Hey factory, what's your volume?
System.out.println(Ball.getVolume()); // wat?
```

It only makes sense to ask this from actual `Ball`s. We refer to these "actual `Ball`s" as `Ball` **instances.**

```
Ball ballInstance = new Ball(7);

// Hey ball, what's your radius?
System.out.println(ballInstance.radius); // 7
// Hey ball, what's your volume?
System.out.println(ballInstance.getVolume()); // 0. I'm deflated.
```

As such, we refer to these as **instance variables** and **instance methods.**

But it is possible to have some methods that can be called from the blueprint/factory (i.e. the class itself). Let's have the class keep track of the number of balls created.

```
class Ball {
    static int numConstructedBalls = 0;
    int radius;

    public Ball(int radius) {
        ++numConstructedBalls;
        // ...
    }
    public double getVolume() {...}
    public static int getNumConstructedBalls() {
        return numConstructedBalls;
    }
}
```

These are different from the instance methods in the sense that it makes sense to ask these questions from the factory (i.e. the class).

```
// Hey factory, how many balls have you constructed?
System.out.println(Ball.getNumConstructedBalls()); // 1
```

As such, we refer to these as **class methods** (or **static methods**).