**Chapter 4 Mathematical Functions, Characters, and Strings**

Contents

[Intro to the chapter 2](#_Toc43111374)

[Math Methods 2](#_Toc43111375)

[Incompatible types 3](#_Toc43111376)

[Common Math Methods 3](#_Toc43111377)

[Character Data Type 3](#_Toc43111378)

[Unicode Format 4](#_Toc43111379)

[ASCII 4](#_Toc43111380)

[Casting between char and Numeric Types 5](#_Toc43111381)

[Escape Characters 5](#_Toc43111382)

[Comparing and Testing Characters 6](#_Toc43111383)

[Examples: 6](#_Toc43111384)

[The String Type 6](#_Toc43111385)

[Memory 7](#_Toc43111386)

[How/Why Counting Starts at Zero: 7](#_Toc43111387)

[Index 7](#_Toc43111388)

[How to use the methods: 8](#_Toc43111389)

[Common Methods used with Strings 8](#_Toc43111390)

[Getting Characters from a String 8](#_Toc43111391)

[String Concatenation 8](#_Toc43111392)

[Converting Strings 8](#_Toc43111393)

[Converting Strings to Integers 9](#_Toc43111394)

[Converting Strings to Doubles 9](#_Toc43111395)

[Reading a String from the Console 9](#_Toc43111396)

[Reading a Character from the Console 9](#_Toc43111397)

[Strings cannot use the relational comparison operators! 9](#_Toc43111398)

[Methods for Comparing Strings 11](#_Toc43111399)

[Obtaining Substrings 12](#_Toc43111400)

[Finding a Character or a Substring in a String 12](#_Toc43111401)

[Formatting Console Output 13](#_Toc43111402)

[Syntax 13](#_Toc43111403)

[Frequently-Used Specifiers 13](#_Toc43111404)

[Specify the exact number 13](#_Toc43111405)

[Alignment 14](#_Toc43111406)

[Aligning to the right 14](#_Toc43111407)

[Aligning to the left: 15](#_Toc43111408)

[Why do we want to learn to format the output? 15](#_Toc43111409)

# Intro to the chapter

This chapter is all about learning how to use methods that are defined in the Java Math class, the Character class, and the String class.

What is a method? A method is a group of statements that perform a specific task. A method is predefined code for manipulation of the data, modifying the data, basically working with the data in some fashion. A good method does **one** thing, and does that one thing well! We will learn how to write our own methods in chapter 6, for now, we just need to learn how to use them.

First, why do we want to learn about these methods? Methods enable a programmer to do something repeated times, while only having to write the code once. Benefit right there! So nice to be able to write something once and get to use again. Methods have other benefits that we will look at in chapter 6.

Second, why do we want to learn about these methods in these classes? They are useful, in fact, so useful-and commonly used-, that the Java developers wrote those methods so we don’t have to! Now, while we are in the intro stages, some of these methods we won’t be seeing the benefit of yet, but you will as you advanced onward.

So, how do we use a method? We invoke (or call) a method. What does that mean? It means that you, the programmer, will write that method name and if a method returns a value to you, then you have to deal with that value. When a value is returned, you have to “catch” that result/return value (just like you have being doing when you used the Math.pow method). You can handle this two ways 1) by printing the result, or 2) by storing the result in a data variable.

In reality, you have already been using methods – since day 1 in fact! Remember the Welcome program? The line: System.out.println(“Welcome to Java!”); println is a method. Surprise! println takes the contents inside the ( ) and prints its. The Scanner class – every time you read in values from the screen, you are using methods. So, now we are just really learning more methods to use.

# Math Methods

You learned about the Math.pow method in chapter 2, and the Math.random in chapter 3 (chapter 4 is where the formula that we used is from).

All of the Math class methods return a value. Most of the Math methods will return a double value.

Examples:

double result = Math.pow(3, 4);

System.out.println(result); // 81.0 is displayed

**NOTE**: that when you use the Math class methods, you have to state Math.methodName(parameters)

Remember that just using the method does not allow you to see the result of that statement! To see the result of the method that is being invoked, you have to display that to the screen.

System.out.println(Math.round(54.36));

If you want to be able to use that value again, assign it to a variable.

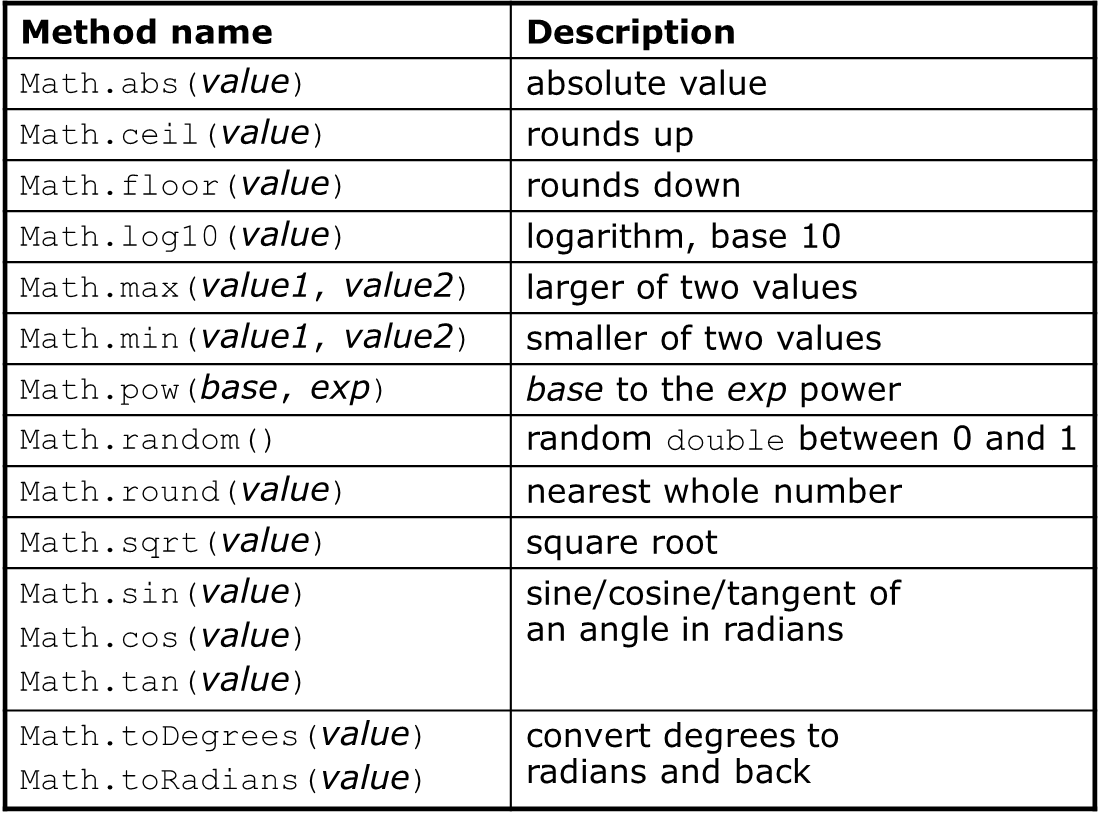
## Incompatible types

Some Math methods return double or other non-int types.

int x = Math.pow(10, 3); **// ERROR: incompatible types**

You can cast - int x = **(int)**Math.pow(10, 3); **// 1000**

## Common Math Methods



Some Math methods are overloaded. Overloading means that we keep the same name and provide different parameters (or arguments). We will go into detail about overloading in chapter 6. The min, max, pow are overloaded methods. We can use Math.min for ints or for doubles.

# Character Data Type

* A character data type represents a single character – it can be any character that is represented in the Unicode
* A character literal is enclosed in signal quotations:

char letter = ‘A’; // assigns A to the variable letter

char numChar = ‘4’; // assigns digit character 4 to the variable numChar

How it looks in main memory:

|  |  |
| --- | --- |
| letter | A |
| numChar | 4 |

* Clarification:
  + A string literal must be enclosed in *double* quotation marks (“”).
  + A character literal must be enclosed in a *single* quotation mark (‘’).
* The increment and decrement operators can also be used on char variables to get the next or preceding Unicode character. For example, the following statements display character b.

char ch = 'a';

System.out.println(++ch);

## Unicode Format

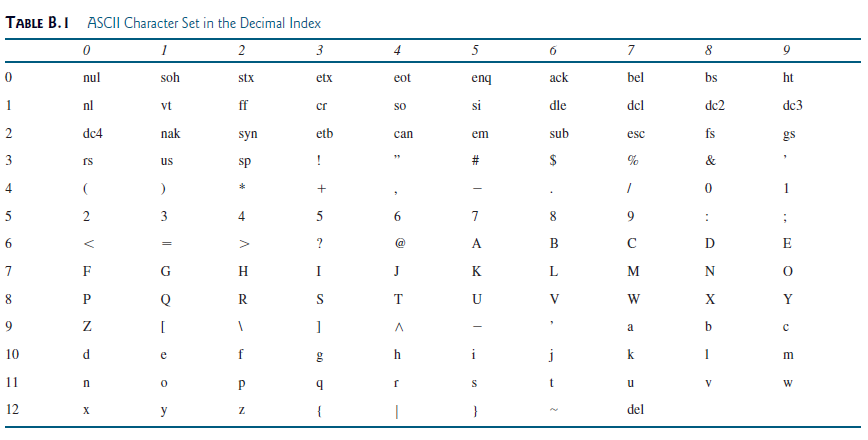
* Encoding: mapping a character to its binary representation
* Encoding scheme: how characters are encoded
* Java characters use *Unicode*, a 16-bit encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world’s diverse languages. Unicode takes two bytes, preceded by \u, expressed in four hexadecimal numbers that run from '\u0000' to '\uFFFF'. So, Unicode can represent 65535 + 1 characters.
* The Unicode standard has been extended to allow up to 1,112,064 characters, called supplementary characters that go beyond the original 16-bit limit.
  + Unicode is built upon ASCII
* Example:

char letter = ‘\u0041’; // the variable letter is assigned A in Unicode

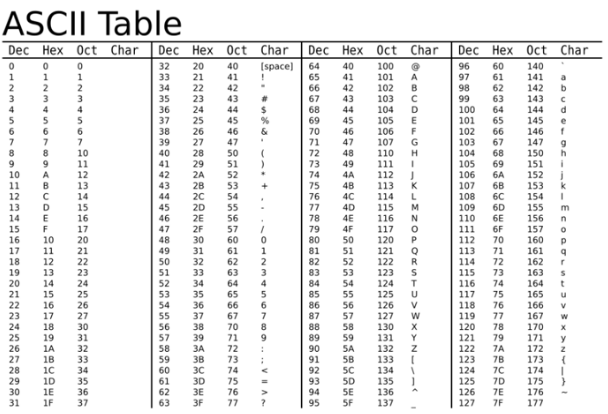
## ASCII

* Most computers use ASCII, American Standard Code for Information Interchange
* 7-bit encoding scheme for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters.
* Example:

char letter = ‘A’; // the variable letter is assigned A in ASCII



Another view:



## Casting between char and Numeric Types

* A char can be cast into any numeric type (like an int or double) and vice versa
* When an integer is cast into a char, only its lower 16 bits of data are used (the other part is ignored)
* This is possible because the char is mapped to the Unicode/ASCII table, which is a numeric value
* when a char is cast into a numeric type, the character’s Unicode is cast into the specified numeric type
* when a floating-point value is cast into a char, the floating-point is first cast into an int, then cast into a char (think of why! – no decimals in the Unicode/ASCII table)
* Examples:
  + int i = 'a'; // Same as int i = (int)'a'; and is assigned the value of 97
  + char c = 97; // Same as char c = (char)97; and is assigned the value of a

## Escape Characters

* Escape characters overcome the compile errors when using characters that are used in the language.
* Consists of a backslash followed by a character or a character sequence
* Table of the Escape Characters:

*Description Escape Sequence*

Backspace \b

Tab \t

Linefeed \n

Carriage return \r

Backslash \\

Single Quote \'

Double Quote \"

* Example: \t for the Tab character to space out headings on a table:

System.out.println(“Description \t Escape Sequence”);

This prints: Description Escape Sequence

## Comparing and Testing Characters

The char data type is a primitive data type and relational operators (<, <=, >, >=, ==, !=) can be used to compare between char variables.

if (ch >= 'A' && ch <= 'Z') // same as if(isUpperCase(ch))

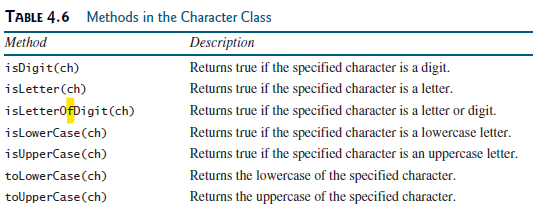
System.out.println(ch + " is an uppercase letter");

else if (ch >= 'a' && ch <= 'z') // same as else if(isLowerCase(ch))

System.out.println(ch + " is a lowercase letter");

else if (ch >= '0' && ch <= '9') // same as else if(isDigit(ch))

System.out.println(ch + " is a numeric character");



\*\*\* that is supposed to be: isLetterOrDigit(ch), not an f (typo in the text book)

**NOTE**: When you are using the Character class methods (just like when you are using the Math class methods), you have to state Character.methodName(parameters)

Also note, that when you are using these methods that you have a value returned. You will have to either assign that value to a variable or display that value.

The first 5 methods return a boolean value, the last two return the ch (modified) back.

## Examples:

char aLetter = ‘a’;

aLetter = Character.toUpperCase(aLetter);

boolean isItAnUpperCaseLetter = Character.isUpperCase(aLetter);

boolean isItALetter = Character.isLetter(aLetter);

boolean isItADigit = Character.isDigit(aLetter);

# The String Type

* a String is a sequence of characters
* String class is actually a predefined class in the Java library just like the System class, Math class, Character class, and File class.
* The String type is not a primitive type. It is known as a *reference type*. Any Java class can be used as a reference type for a variable.

There are a few things I like to point out about Strings: 1) memory storage, 2) how/why counting starts at zero, and 3) methods are used with the identifier (like you do with the Scanner object).

Memory:

* Strings have two different ways of being stored in memory. (<https://www.baeldung.com/java-string-pool>)
* String literals (String st = “word”; , which is what we are learning here in this course) have a special memory storage that is just for them, called the String Pool.
  + Why use the String Pool? Because it saves on memory space.
* String objects (String st = new String(“word”); , which you will learn about in CSCI 2302) are stored in memory as a normal reference data type is.

## How/Why Counting Starts at Zero:

Let’s use a picture to work this out.

String st = “Learning”;

st stored in the String Pool:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| L | e | a | r | n | i | n | g |

If we use a “pointer” (and yes, pointer is in quotes because you do not have access to the pointer in Java for security purposes) to access one of those letters, we can then see why we (in computer science) start counting at zero.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| L | e | a | r | n | i | n | g |

There is the “pointer” in purple. I want to access the first letter, L. Do I have to move the “pointer” to have that value returned? The answer: no. Hence, we start counting at zero.

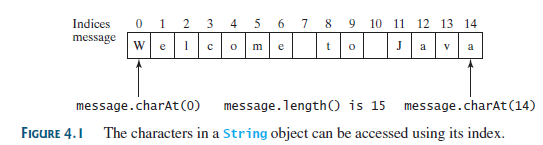
Let’s put this together with the idea of index values now.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Character value: | L | e | a | r | n | i | n | g |
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

## Index

What is an index? (Since I brought it up with no definition) an index is a way of accessing the individual letters / elements of an object, in this case String.

So, looking at the memory space, I can see that there are index values 0 to 7. I can also count that Learning has 8 characters in it, which we call length. The relationship between the index values and length is always as: index values are 0 – (length-1).



If you try to access an element at index value length, then you will get an index out of bounds error during runtime.

## How to use the methods:

We use the methods in the String class, the way we have been using methods in the Scanner class, by the identifier.

Example:

String st = “Learning”;

char firstLetter = st.charAt(0); // the String method charAt returns the letter L

int howManyCharacters = st.length(); // returns the value of 8

## Common Methods used with Strings



## Getting Characters from a String

String s = “example”;

char ch = s.charAt(2); // ch value is a

ch = s.charAt(s,length()); // is a StringIndexOutOfBoundsException

ch = s.charAt(s,length()-1); //is the last letter of the String

## String Concatenation

// Three strings are concatenated

String message = "Welcome " + "to " + "Java";

This would print: Welcome to Java

// String Chapter is concatenated with number 2

String s = "Chapter" + 2; // s becomes Chapter2

// using the concat() method

String s1 = s.concat(" adding at the end"); //s1 is Chapter2 adding at the end

## Converting Strings

toLowerCase() 🡪 a new String with all lower case letters

toUpperCase() 🡪 a new String with all upper case letters

## Converting Strings to Integers

* The input returned from the input dialog box is a string. If you enter a numeric value such as 123, it returns “123”. To obtain the input as a number, you have to convert a string into a number.
* To convert a string into an int value, you can use the static parseInt method in the Integer class as follows:

  int intValue = Integer.parseInt(intString);

  where intString is a numeric string such as “123”

## Converting Strings to Doubles

* To convert a string into a double value, you can use the static parseDouble method in the Double class as follows:

 double doubleValue =Double.parseDouble(doubleString);

where doubleString is a numeric string such as “123.45”.

# Reading a String from the Console

next() 🡪 a string that ends with a whitespace character, i.e. the space (by hitting the space bar)

nextLine() 🡪 a string that that is the entire line of the text; ends when the *Enter* key is pressed

using nextLine() will yield an input error when used after nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), nextDouble(), or next()

If you only have one word – then use the next( ) – it will save you headaches.

Please see the DifferenceBetweenNextAndNextLine.java (in Example Programs) for a more detailed explanation and how to deal with it.

# Reading a Character from the Console

Use next() or nextLine(), then invoke charAt(0) to get the character.

Scanner input = new Scanner(System.in);

System.out.print("Enter a character: ");

String s = input.next();

char ch = s.charAt(0);

System.out.println("The character entered is " + ch);

OR

char letter = input.next().charAt();

Strings cannot use the relational comparison operators! Those are for primitive data types only! Strings have to use the methods!

Relational operators work at the memory content level, and Strings have that funny way on being stored in memory.

String s = "something";

String t = "something else";

if (s == t) // Legal, but usually WRONG, because of memory management.

if (s.equals(t)) // CORRECT

if (s > t) // ILLEGAL

if (s.compareTo(t) > 0) // CORRECT

<http://docs.oracle.com/javase/tutorial/java/data/comparestrings.html>

|  |  |
| --- | --- |
| Methods for Comparing Strings | |
| **Method** | **Description** |
| boolean endsWith(String suffix) boolean startsWith(String prefix) | Returns true if this string ends with or begins with the substring specified as an argument to the method. |
| boolean startsWith(String prefix, int offset) | Considers the string beginning at the index offset, and returns true if it begins with the substring specified as an argument. |
| int compareTo(String anotherString) | Compares two strings lexicographically. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument. |
| int compareToIgnoreCase(String str) | Compares two strings lexicographically, ignoring differences in case. Returns an integer indicating whether this string is greater than (result is > 0), equal to (result is = 0), or less than (result is < 0) the argument. |
| boolean equals(Object anObject) | Returns true if and only if the argument is a String object that represents the same sequence of characters as this object. |
| boolean equalsIgnoreCase(String anotherString) | Returns true if and only if the argument is a String object that represents the same sequence of characters as this object, ignoring differences in case. |
| boolean regionMatches(int toffset, String other, int ooffset, int len) | Tests whether the specified region of this string matches the specified region of the String argument.  Region is of length len and begins at the index toffset for this string andooffset for the other string. |
| boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) | Tests whether the specified region of this string matches the specified region of the String argument.  Region is of length len and begins at the index toffset for this string andooffset for the other string.  The boolean argument indicates whether case should be ignored; if true, case is ignored when comparing characters. |
| boolean matches(String regex) | Tests whether this string matches the specified regular expression. Regular expressions are discussed in the lesson titled "Regular Expressions." |

## Obtaining Substrings

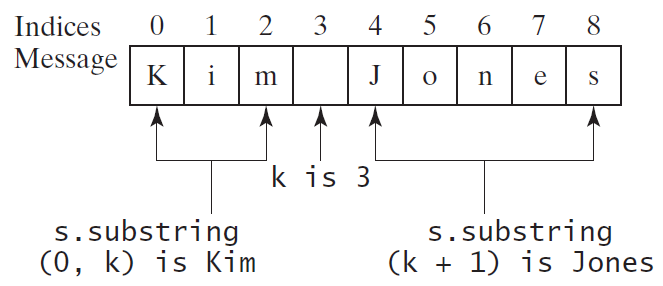


Example:

int k = s.indexOf(' ');

String firstName = s.substring(0, k);

String lastName = s.substring(k + 1);

****

**\*\*\*\*\* NOTICE\*\*\*\*\*\*** when using substring(beginIndex, endIndex), as in the above example, k = endIndex, it is upto to **NOT** including k.

## Finding a Character or a Substring in a String



# Formatting Console Output

Use the printf statement.

Syntax: System.out.printf(format, items);

Where format is a String that may consist of substrings and format specifiers.

And items are the variables.

A format specifier specifies how an item should be displayed. An item may be a numeric value, character, boolean value, or a String. Each specifier begins with a percent sign.

## Frequently-Used Specifiers

**Specifier Output Example**

%b a boolean value true or false

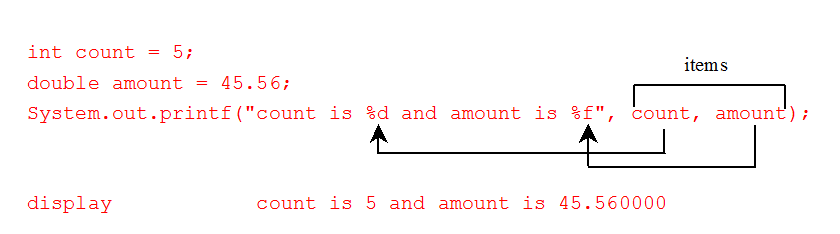
%c a character 'a'

%d a decimal integer 200

%f a floating-point number 45.460000

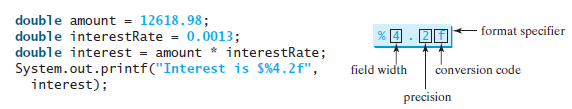
%e a number in standard  
 scientific notation 4.556000e+01

%s a string "Java is cool"



## Specify the exact number

When we use the format specifiers, we can specify the exact number of spaces that we want to have displayed on the screen.



Produces the output: 

For this example, we want the money to only print with the two decimal places (as money should be displayed in). Let’s break this down.

Field width: how many total spaces for the variable to be displayed in.

Precision: how many decimal places will be printed (the stuff on the right side of the decimal)

Format specifier: the data type

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

Putting it together: field width of 4:

That is how many spaces this variable’s output will take up on the line.

If that variable will be displaying a decimal (the dot/period), then that decimal is *included* in the field width.

BUT – let me make something clear here. If that variable takes up more than 4 spaces (not including the stuff on the right side of the decimal), then it will expand past the 4 spaces.

Precision: this value will never change – it will not be expanded.

Another example:

double number = 123.456789;

We want number to be displayed with 3 decimal places: System.out.printf(“%7.3f”, number);

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | . | 4 | 5 | 6 |

Output:

Another example:

String example = “example”;

We incorrectly specify the field width: System.out.printf(“%5s”, example);

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a | m | p | l | e |

Output: e x

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| e | x | a | m | p | l | e |

“Actually output”:

The e and x are still printed out – it takes up more than the 5 spaces allocated.

## Alignment

Definition: the proper positioning or state of adjustment of parts in relation to each other.

In the printf, the is the right alignment & the left alignment.

Aligning to the right:

Meaning we start placing the last characters on the right side and add the remaining characters towards the left – starting at the end (right side) working the way to the beginning of the variable (going right to left).

Left Side Right Side

First characters of the variable & extra spaces last characters of the variable

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |

Example:

String rtAlignment = “aligned along the right side.”;

System.out.printf(“%35s” , rtAlignment);

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  | a | l | i | g | n | e | d |  | a | l | o | n | g |  | t | h | e |  | r | i | g | h | t |  | s | i | d | e | . |

Right alignment uses positive numbers, and the extra spaces are at the beginning / prior to the variable’s characters.

This is useful when aligning decimal numbers, the decimal will always be the 3rd from the right and in a nice neat column – like we normally see numbers lined up.

## Aligning to the left:

Meaning we start placing the first characters on the left and work to the right till the last character is placed (going left to right). The extra spaces (if there are any) will be on the left side.

Example:

String ltAlignment = “aligned along the left side”;

System.out.printf(“%-35s”, ltAlignment);

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | l | i | g | n | e | d |  | a | l | o | n | g |  | t | h | e |  | l | e | f | t |  | s | i | d | e | . |  |  |  |  |  |  |  |

Left alignment uses negative numbers, and the extra spaces are at the end - after the variable’s characters.

## Why do we want to learn to format the output?

It looks nicer!

If we have to display a money value, then the display should look like we represent money, with only 2 decimal places. This provides a way of doing that without losing the integrity of variable. You never want to do something that will alter the value of a variable, you will always want to keep the integrity of a variable.

You can also have your results displayed in a table format, in nice neat columns that are easy to read. Overall, it is a helpful feature for clean output.

Whenever we need a nice output, or outputting $, we will be using printf.