

Chapter 4

Core APIs

Creating and Manipulating Strings

Concatenating

3 rules of +

If both operands are numeric → addition

If either operand is a String → concatenation

The expression is evaluated left to right

1 + 2; //3

"a" + "b"; //ab

"a" + "b" + 3; //ab3

1 + 2 + "c"; //3c

"c" + 1 + 2; //c12

"c" + null; //cnull

Important String Methods (String is immutable)

1/ Determining the length

public int `length()`

2/ Getting a single character

public char `charAt(int index)`

→ throws exception if index is out of range

3/ Finding an index

public int `indexOf(int ch)`

(int ch, int fromIndex)

(String str)

(String str, int fromIndex)

→ returns the 1st index matching the desired value

→ returns -1 if it can't find the match

4/ Getting a substring

public String `substring(int beginIndex)`

(int beginIndex, int endIndex)

→ begin = end : empty string

→ begin > end or end is out of range : exception

5/Adjusting case

public String toLowerCase()

public String toUpperCase()

6/Checking for equality

public boolean equals(Object obj)

public boolean equalsIgnoreCase(String str)

7/Search for substrings

public boolean startsWith(String prefix)

public boolean endsWith(String suffix)

public boolean contains(CharSequence charSeq)

8/Replacing values

public String replace(char oldChar, char newChar)

(CharSequence target,

CharSequence replacement)

9/ Removing whitespace

whitespace = space + \t(tab) + \n(newline)

public String strip() → supports Unicode
public String stripLeading()
public String stripTrailing()
public String trim()

10/ Working with indentation

public String indent(int numberSpaces)
public String stripIndent()

Method	Indent change	Normalizes existing line breaks	Adds line break at end if missing
indent(n) where n > 0	Adds n spaces to beginning of each line	Yes	Yes
indent(n) where n == 0	No change	Yes	Yes
indent(n) where n < 0	Removes up to n spaces from each line where the same number of characters is removed from each non-blank line	Yes	Yes
stripIndent()	Removes all leading incidental whitespace	Yes	No

\r\n becomes \n

```
10: var block = """;  
11:     a\n  
12:     b\n  
13:     c""";  
14: var concat = " a\n" + " b\n"  
15:           + " c";  
16:  
17: System.out.println(block.length()); // 6  
18: System.out.println(concat.length()); // 9  
19: System.out.println(block.indent(1).length()); // 10  
20: System.out.println(concat.indent(-1).length()); // 7  
21: System.out.println(concat.indent(-4).length()); // 6  
22: System.out.println(concat.stripIndent().length()); // 6
```

11/ Translating escapes

```
public String translateEscapes()
```

```
var str = "1\\t2";
```

```
System.out.println(str); // 1\t2
```

```
System.out.println(str.translateEscapes()); // 1 2
```

12/ Checking for empty or blank Strings

```
public boolean isEmpty()
```

```
public boolean isBlank()
```

13/ Formatting values

public static String format(String format, Object args...)
(Locale loc, String format,
Object args...)

public String formatted(Object args...)

```
var name = "Kate";  
var orderId = 5;  
// All print: Hello Kate, order 5 is ready  
System.out.println("Hello "+name+", order "+orderId+" is ready");  
System.out.println(String.format("Hello %s, order %d is ready",  
    name, orderId));  
System.out.println("Hello %s, order %d is ready"  
.formatted(name, orderId));
```

Symbol	Description
%s	Applies to any type, commonly String values
%d	Applies to integer values like int and long
%f <i>.1f</i>	Applies to floating-point values like float and double (default: 6 digits past the .)
%n	Inserts a line break using the system-dependent line separator

The following example uses all four symbols from Table 4.2:

```
var name = "James";  
var score = 90.25;  
var total = 100;  
System.out.println("%s:%n Score: %f out of %d"  
.formatted(name, score, total));
```

This prints the following:

James:
Score: 90.250000 out of 100

```
var pi = 3.14159265359;  
System.out.format("[%f]",pi); // [3.141593]  
System.out.format("[%12.8f]",pi); // [ 3.14159265]  
System.out.format("[%012f]",pi); // [00003.141593]  
System.out.format("%12.2f",pi); // [ 3.14]  
System.out.format("%.3f",pi); // [3.142]
```

Using the String Builder

```
String alpha = "";  
for (char current = 'a'; current <= 'z'; current++)  
    alpha += current;
```

Loop 1: alpha → 

Loop 2: alpha →  → eligible for garbage collection

→ new object, since String is immutable

Mutability & Chaining

```
StringBuilder a = new StringBuilder("abc");  
StringBuilder b = a.append("de");  
b = b.append("f").append("g");
```

Both **a** & **b** are "abcdefg"

Creating a StringBuilder

```
new StringBuilder(); // empty  
new StringBuilder("abc"); // "abc"  
new StringBuilder(10); // 10 slots for char.
```

Important StringBuilder Methods

1/ Common methods: `substring()`, `indexOf()`, `length()`, `charAt()`

returns a String

rather than a StringBuilder

2/ Appending values

```
public StringBuilder append(String str)  
    (int val)  
    (char val)...
```

```
new StringBuilder().append(1)  
    .append('c')  
    .append("-")  
    .append(true);
```

// 1c-true

3/ Inserting data (lots of method signatures for different types)

```
public StringBuilder insert(int offset, String str)
```

insert the param to this index

4/ Deleting contents

```
public StringBuilder delete(int startIndex, int endIndex)
```

```
public StringBuilder deleteCharAt(int index)
```

past the end → OK, delete to the end

5/Replacing portions

```
public StringBuilder replace(int startIndex,  
                           int endIndex,  
                           String newString)
```

deletes characters,
then inserts

6/Reversing

```
public StringBuilder reverse()
```

Understanding Equality

```
var x = "Hello World";  
var y = "Hello World".trim();  
System.out.println(x.equals(y)); //true
```

StringBuilder doesn't implement `equals()`

→ calling `equals()` will check reference equality (same as `==`)

The String Pool

A location in the JVM, containing literal values & constants that appear in our program

```
var x = "Hello World";  
var y = "Hello World";  
System.out.println(x == y); // true
```

only 1 literal
in memory

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

computed at
runtime → new obj

```
var singleString = "hello world";  
var concat = "hello ";  
concat +=" world";  
System.out.println(singleString == concat); // false
```

new str.

`intern()` method will use an object in the string pool if one is present. Otherwise, Java will add this literal to the string pool.

```
var name = "Hello World";  
var name2 = new String("Hello World").intern();  
System.out.println(name == name2); //true
```

```
var first = "rat" + 1;  
var second = "r" + "a" + "t" + "1";  
var third = "r" + "a" + "t" + new String("1");
```

Both first & second values are compile-time constants
→ share the same string pool reference
third does not point to a reference in the string pool

Understanding Arrays

Creating an Array of Primitives

Type Array symbol

↓

int [] numbers = new int[3];

Size ↑

All elements are set to the default value for that type

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

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

```
int [] arr1;
```

```
int [] arr2;
```

```
int arr3[];
```

```
int arr4 [];
```

Creating an Array with Reference Variables

```
String[] bugs = {"cricket", "beetle", "ladybug"};
```

```
3: String[] strings = { "stringValue" }; > String[]
4: Object[] objects = strings;
5: String[] againStrings = (String[]) objects;
6: againStrings[0] = new StringBuilder(); // DOES NOT COMPILE
7: objects[0] = new StringBuilder(); // Careful!
```

throws an `ArrayStoreException`

at runtime (since "objects" still refers to `String[]`)

Sorting

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

```
or import java.util.Arrays;
```

→ `Arrays.sort(NSArray);` // Sort ascending

String sorts in alphabetic order: number < upper < lower

Ex: `{"10", "g", "100"}` → 10 100 g

Searching

Scenario	Result
Target element found in sorted array	Index of match
Target element not found in sorted array	Negative value showing one smaller than the negative of the index, where a match needs to be inserted to preserve sorted order
Unsorted array	A surprise; this result is undefined <i>unpredictable not error</i>

Let's try these rules with an example:

```
3: int[] numbers = {2,4,6,8};  
4: System.out.println(Arrays.binarySearch(numbers, 2)); // 0  
5: System.out.println(Arrays.binarySearch(numbers, 4)); // 1  
6: System.out.println(Arrays.binarySearch(numbers, 1)); // -1  
7: System.out.println(Arrays.binarySearch(numbers, 3)); // -2  
8: System.out.println(Arrays.binarySearch(numbers, 9)); // -5
```

Comparing compare()

Arrays.compare(arr1, arr2);

< 0 : arr1 < arr2

= 0 : arr1 = arr2

> 0 : arr1 > arr2

* arr1 & arr2 must be **the same array type**

- **null** is smaller than any other value.
 - For numbers, normal numeric order applies.
 - For strings, one is smaller if it is a prefix of another.
 - For strings/characters, numbers are smaller than letters.
 - For strings/characters, uppercase is smaller than lowercase.
- Table 4.4 shows examples of these rules in action.

TABLE 4.4 Arrays.compare() examples

First array	Second array	Result	Reason
new int[] {1, 2}	new int[] {1}	Positive number	The first element is the same, but the first array is longer.
new int[] {1, 2}	new int[] {1, 2}	Zero	Exact match
new String[] {"a"}	new String[] {"aa"}	Negative number	The first element is a substring of the second.
new String[] {"a"}	new String[] {"A"}	Positive number	Uppercase is smaller than lowercase.
new String[] {"a"}	new String[] {null}	Positive number	null is smaller than a letter.

mismatch()

Arrays.mismatch(arr1, arr2);

arr1 = arr2 : -1

arr1 ≠ arr2 : the 1st index where they differ

Using Methods with Varargs

```
public static void main(String... args)
```

Working with Multidimensional Arrays

```
int[][] vars1;  
int vars2 [][];  
int [] vars3[];  
int [] vars4 [], space [][] // 2D & 3D
```

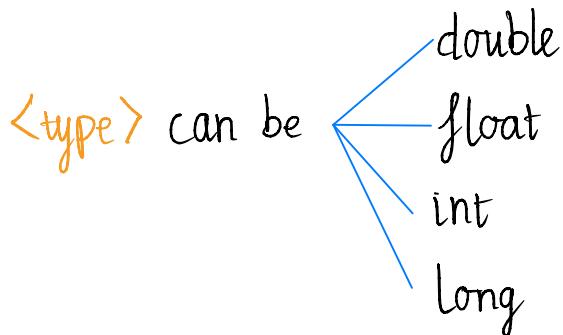
```
String[][] rectangle = new String[3][2];  
int[][] differentSizes = {{1,4},{3},{9,8,7}};
```

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

Calculating with Math APIs

Minimum & Maximum

```
public static <type> min (<type> a, <type> b)
```



The same for `max()` method

Rounding Numbers

```
public static long round (double num)  
public static int round (float num)
```

$\gg .5 \rightarrow$ round up ↑

$\ll .5 \rightarrow$ round down ↓

Ceiling & Floor

```
public static double ceil(double num)
```

- * whole number → returns the same value
- * has any fractional value → rounds up to the next whole number

Ex: double c = Math.ceil(3.14); //4.0

```
public static double floor(double num)
```

discards any fractional value

Ex: double f = Math.floor(3.14); //3.0

Exponents

```
public static double pow(double number,  
                      double exponent)
```

Ex: double squared = Math.pow(5, 2); //25.0

Random Numbers

```
public static double random()
```

$0.0 \leq \text{result} \leq 1.0$

Working with Dates & Times

```
import java.time.*;
```

Creating Dates & Times

`LocalDate.now();` // just date (no time & no timezone)

`LocalTime.now();` // just time (no date & no timezone)

`LocalDateTime.now();` // date & time but no timezone

`ZonedDateTime.now();` // date, time, & timezone

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

```
var date1 = LocalDate.of(2022,  
                         Month.JANUARY, 20);  
var date2 = LocalDate.of(2022, 1, 20);
```

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

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

public static ZonedDateTime of(int year, int month, int day,
int hour, int minute, int second, int nanos,
ZoneId zone)

of(LocalDate date,
LocalTime time,
ZoneId zone)

of(LocalDateTime dateTime,
ZoneId zone)

no
Month enum

```
var zone = ZoneId.of("US/Eastern");  
var zoned1 = ZonedDateTime.of(2022, 1, 20,  
6, 15, 30, 200, zone);
```



The date & time classes have private constructors along with static methods that return instances

```

var d = new LocalDate(); //Error
var d = LocalDate.of(2022, 1, 32);
//DateTimeException

```

Manipulating Dates & Times

The date & time classes are **immutable**

	Can call on LocalDate?	Can call on LocalTime?	Can call on LocalDateTime or ZonedDateTime?
plusYears()	Yes	No	Yes
minusYears()	Yes	No	Yes
plusMonths()	Yes	No	Yes
minusMonths()	Yes	No	Yes
plusWeeks()	Yes	No	Yes
minusWeeks()	Yes	No	Yes
plusDays()	Yes	No	Yes
minusDays()	Yes	No	Yes
plusHours()	No	Yes	Yes
minusHours()	No	Yes	Yes
plusMinutes()	No	Yes	Yes
minusMinutes()	No	Yes	Yes
plusSeconds()	No	Yes	Yes
minusSeconds()	No	Yes	Yes
plusNanos()	No	Yes	Yes
minusNanos()	No	Yes	Yes

Working with Periods

Period.ofYears(1); // every 1 year

Period.ofMonths(3); // every 3 months

Period.ofWeeks(3); // every 3 weeks

Period.ofDays(2); // every 2 days

Period.of(1, 0, 7); // every year & 7 days

var wrong = Period.ofYears(1).ofWeeks(1);



only the last method is used

⇒ { var wrong = Period.ofYears(1);
 wrong = Period.ofWeeks(1); }

System.out.println(Period.of(1, 2, 3)); // P1Y2M3D

System.out.println(Period.ofMonths(3)); // P3M

```
3: var date = LocalDate.of(2022, 1, 20);
4: var time = LocalTime.of(6, 15);
5: var dateTime = LocalDateTime.of(date, time);
6: var period = Period.ofMonths(1);
7: System.out.println(date.plus(period)); // 2022-02-20
8: System.out.println(dateTime.plus(period)); // 2022-02-20T06:15
9: System.out.println(time.plus(period)); // Exception
```

Working with Durations

Duration.ofDays(1); // PT24H

Duration.ofHours(1); // PT1H

Duration.ofMinutes(1); // PT1M

Duration.ofSeconds(10); // PT10S

Duration.ofMillis(1); // PT0.001S

Duration.ofNanos(1); // PT0.000000001S

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

Duration.of(1, ChronoUnit.DAYS);

implementation of
TemporalUnit



```
var one = LocalTime.of(5, 15);  
var two = LocalTime.of(6, 30);  
var date = LocalDate.of(2016, 1, 20);  
ChronoUnit.HOURS.between(one, two); // 1  
ChronoUnit.MINUTES.between(one, two); // 75  
ChronoUnit.MINUTES.between(one, date); // DateTimeExc.
```

```
LocalTime time = LocalTime.of(3, 12, 45);  
System.out.println(time); // 03:12:45  
LocalTime truncated = time.truncatedTo(  
    ChronoUnit.MINUTES);  
System.out.println(truncated); // 03:12
```

	Can use with Period?	Can use with Duration?
LocalDate	Yes	No
LocalDateTime	Yes	Yes
LocalTime	No	Yes
ZonedDateTime	Yes	Yes

Working with Instants

The Instant class represents a specific moment in time in the GMT time zone

```
var now = Instant.now();
```

```
// do something time consuming
```

```
var later = Instant.now();
```

```
var duration = Duration.between(now, later);
```

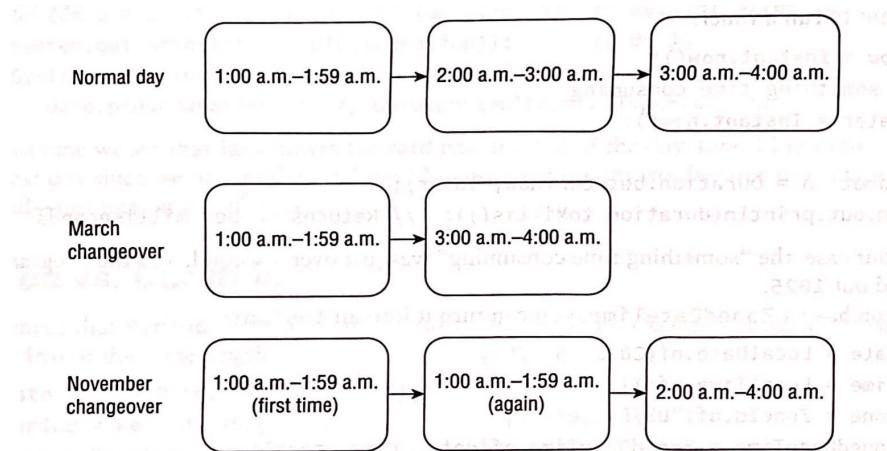
```
System.out.println(duration.toMillis());
```

```
var date = LocalDate.of(2022, 5, 25);
var time = LocalTime.of(11, 55, 00);
var zone = ZoneId.of("US/Eastern");
var zonedDateTime = ZonedDateTime.of(date, time, zone);
var instant = zonedDateTime.toInstant(); // 2022-05-25T15:55:00Z
System.out.println(zonedDateTime); // 2022-05-25T11:55-04:00[US/Eastern]
System.out.println(instant); // 2022-05-25T15:55:00Z
```

gets rid of the timezone & turns it into
an Instant of time in GMT

CANNOT convert a `LocalDateTime` to an Instant
(not contain a timezone)

Accounting for Daylight Saving Time



```
var date = LocalDate.of(2022, Month.MARCH, 13);
var time = LocalTime.of(1, 30);
var zone = ZoneId.of("US/Eastern");
var dateTime = ZonedDateTime.of(date, time, zone);
System.out.println(dateTime); // 2022-03-13T01:30-05:00[US/Eastern]
System.out.println(dateTime.getHour()); // 1
System.out.println(dateTime.getOffset()); // -05:00

dateTime = dateTime.plusHours(1);
System.out.println(dateTime); // 2022-03-13T03:30-04:00[US/Eastern]
System.out.println(dateTime.getHour()); // 3
System.out.println(dateTime.getOffset()); // -04:00
```

```
var date = LocalDate.of(2022, Month.NOVEMBER, 6);  
var time = LocalTime.of(1, 30);  
var zone = ZoneId.of("US/Eastern");  
var dateTime = ZonedDateTime.of(date, time, zone);  
System.out.println(dateTime); // 2022-11-06T01:30-04:00[US/Eastern]  
  
dateTime = dateTime.plusHours(1);  
System.out.println(dateTime); // 2022-11-06T01:30-05:00[US/Eastern]  
  
dateTime = dateTime.plusHours(1);  
System.out.println(dateTime); // 2022-11-06T02:30-05:00[US/Eastern]  
  
  
var date = LocalDate.of(2022, Month.MARCH, 13);  
var time = LocalTime.of(2, 30);  
var zone = ZoneId.of("US/Eastern");  
var dateTime = ZonedDateTime.of(date, time, zone);  
System.out.println(dateTime); // 2022-03-13T03:30-04:00[US/Eastern]
```