Chapter4

Wednesday, October 8, 2025 11:47 AM

Strings

Concatenation:

1. If both operands are numeric, `+` means numeric addition.
2. If either operand is a `String`, `+` means concatenation.
3. The expression is evaluated left to right.

!!! Need to know : codePointAt()

Don't worry! You do not need to memorize the ASCII or Unicode values. You just need to know that if you see `codePointAt()` on the exam that it functions similarly to `charAt()` for ASCII characters, returning the numeric value of the character at the location.

```
System.out.println("abc".equals("ABC"));          // false
System.out.println("ABC".equals("ABC"));          // true
System.out.println("ABC".equals(6));              // false
System.out.println("abc".equalsIgnoreCase("ABC")); // true
```

```
public String strip()
public String stripLeading()
public String stripTrailing()
public String trim()
```

The following code shows how to use these methods:

```
System.out.println("abc".strip());          // abc
System.out.println("\n   a b c\n".strip()); //  a b c

String text = " abc\t ";
System.out.println(text.trim().length());       // 3
System.out.println(text.strip().length());      // 3
System.out.println(text.stripLeading().length()); // 4
System.out.println(text.stripTrailing().length()); // 4
```

Working with Indentation

Now that Java supports text blocks, it is helpful to have methods that deal with indentation. Both of these are a little tricky, so read carefully!

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

The `indent()` method adds the same number of blank spaces to the beginning of each line if you pass a positive number. If you pass a negative number, it tries to remove that number of whitespace characters from the beginning of the line. If you pass zero, the indention will not change.

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 nonblank line	Yes	Yes
`stripIndent()`	Removes all leading incidental whitespace	Yes	No

```
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`	Applies to floating point values like `float` and `double`
`%n`	Inserts a line break using the system-dependent line separator

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

You can also do the opposite and tell Java to use the string pool. The `intern()` method will use an object in the string pool if one is present.

```
public String intern()
```

If the literal is not yet in the string pool, Java will add it at this time.

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

Finally, you can type the `[]` before or after the name, and adding a space is optional. This means that all five of these statements do the exact same thing:

```
int[] numAnimals;
int [] numAnimals2;
int []numAnimals3;
int numAnimals4[];
int numAnimals5 [];
```

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

Line 7 is where this gets interesting. From the point of view of the compiler, this is just fine. A `StringBuilder` object can clearly go in an `Object[]`. The problem is that we don't actually have an `Object[]`. We have a `String[]` referred to from an `Object[]` variable. At runtime, the code throws an `ArrayStoreException`. You don't need to memorize the name of this exception, but you do need to know that this line will compile and throw an exception.

You can use 7Up, the soda, to help remember the order. Numbers (7) sort first, followed by uppercase (U), and then lowercase (p).

- A **negative** number means the first array is smaller than the second.
- A **zero** means the arrays are equal.
- A **positive** number means the first array is larger than the second.

Here's an example:

```
System.out.println(Arrays.compare(new int[] {1}, new int[] {2}));
```

Example to visualize all

```
int[] a = {1};
int[] b = {1, 2};
int[] c = {1};

System.out.println(Arrays.equals(a, b)); // false
System.out.println(Arrays.compare(a, b)); // -1 (a smaller)
System.out.println(Arrays.compare(a, c)); // -1 (1 < 2)
System.out.println(Arrays.compare(b, a)); //  1 (longer wins)
```

Compare Rules:

- If both arrays are the same length and have the same values in each spot in the same order, return zero.
- If all the elements are the same but the second array has extra elements at the end, return a negative number.
- If all the elements are the same, but the first array has extra elements at the end, return a positive number.
- If the first element that differs is smaller in the first array, return a negative number.
- If the first element that differs is larger in the first array, return a positive number.

Date and Time

You obtain date and time instances using a `static` method:

```
System.out.println(LocalDate.now());
System.out.println(LocalTime.now());
System.out.println(LocalDateTime.now());
System.out.println(ZonedDateTime.now());
```

Each of the four classes has a `static` method called `now()`, which gives the current date and time. Your output is going to depend on the date/time when you run it and where you live. The authors live in the United States, making the output look like the following when run on July 31 at 9:23 a.m.:

```
2025-07-29
09:23:07.760
2025-07-29T09:23:07.760
2025-07-29T09:23:07.760-04:00[America/New_York]
```

First, let's try to figure out how far apart the following moments are in time. Notice how India has a half hour offset, not a full hour. To approach a problem like this, you subtract the time zone from the time. This gives you the GMT equivalent of the time:

```
2025-06-20T06:59+05:30[Asia/Kolkata]     // GMT 2025-06-20 01:29
2025-06-20T07:59-04:00[US/Eastern]       // GMT 2025-06-20 11:59
```

LocalDate

```
var date1 = LocalDate.of(2025, Month.JANUARY, 20);
var date2 = LocalDate.of(2025, 1, 20);
```

LocalTime

```
var time1 = LocalTime.of(6, 15);           // hour and minute
var time2 = LocalTime.of(6, 15, 30);       // + seconds
var time3 = LocalTime.of(6, 15, 30, 200);  // + nanoseconds
```

LocalDataTime

```
var dateTime3 = LocalDateTime.of(2025, Month.JANUARY, 20, 6, 15, 30);
var dateTime2 = LocalDateTime.of(date1, time2);
```

```
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)
```

Others take a `Month` reference:

```
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)
```

ZonedDataTime

```
var zone = ZoneId.of("US/Eastern");
var zoned1 = ZonedDateTime.of(2025, 1, 20,
      6, 15, 30, 200, zone);
var zoned2 = ZonedDateTime.of(date1, time1, zone);
var zoned3 = ZonedDateTime.of(dateTime1, zone);
```

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

!!!Atention

Notice that there isn't an option to pass in the `Month` enum. Also, we did not use a constructor in any of the examples. The date and time classes have private constructors along with `static` methods that return instances. This is known as the factory pattern. The exam creators may throw something like this at you:

```
var d = new LocalDate(); // DOES NOT COMPILE
```

Don't fall for this. You are not allowed to construct a date or time object directly.

Another trick is what happens when you pass invalid numbers to `of()`, for example:

```
var d = LocalDate.of(2025, Month.JANUARY, 32); // RuntimeException
```

```
var date = LocalDate.of(2025, Month.JANUARY, 20);
date = date.plusMinutes(1);      // DOES NOT COMPILE
```

`LocalDate` and `LocalDateTime` have a method to convert themselves into `long` values, equivalent to the number of milliseconds that have passed since January 1, 1970, referred to as the *epoch*. What's special about this date? That's what Unix started using for date standards, so Java reused it.

Manipuating Date and Time

LocalDate

Manipulating Dates and Times

Adding to a date is easy. The date and time classes are immutable. Remember to assign the results of these methods to a reference variable so they are not lost.

```
12: var date = LocalDate.of(2025, Month.JANUARY, 20);
13: System.out.println(date);    // 2025-01-20
14: date = date.plusDays(2);
15: System.out.println(date);    // 2025-01-22
16: date = date.plusWeeks(1);
17: System.out.println(date);    // 2025-01-29
18: date = date.plusMonths(1);
19: System.out.println(date);    // 2025-02-28
20: date = date.plusYears(5);
21: System.out.println(date);    // 2030-02-28
```

	Can call on `LocalDate` ?	Can call on `LocalTime` ?	Can call on `LocalDateTime` or `ZonedDateTime` ?
`plusYears()` `minusYears()` `withYear()` `withDayOfYear()`	Yes	No	Yes
`plusMonths()` `minusMonths()` `withMonth()`	Yes	No	Yes
`plusWeeks()` `minusWeeks()`	Yes	No	Yes
`plusDays()` `minusDays()` `withDayOfMonth()`	Yes	No	Yes
`plusHours()` `minusHours()` `withHour()`	No	Yes	Yes
`plusMinutes()` `minusMinutes()` `withMinute()`	No	Yes	Yes
`plusSeconds()` `minusSeconds()` `withSecond()`	No	Yes	Yes
`plusNanos()` `minusNanos()` `withNano()`	No	Yes	Yes

LocalDateTime

There are also nice, easy methods to go backward in time. This time, let's work with `LocalDateTime`:

```
22: var date = LocalDate.of(2025, Month.JANUARY, 20);
23: var time = LocalTime.of(5, 15);
24: var dateTime = LocalDateTime.of(date, time);
25: System.out.println(dateTime);      // 2025-01-20T05:15
26: dateTime = dateTime.minusDays(1);
27: System.out.println(dateTime);      // 2025-01-19T05:15
28: dateTime = dateTime.minusHours(10);
29: System.out.println(dateTime);      // 2025-01-18T19:15
30: dateTime = dateTime.minusSeconds(30);
31: System.out.println(dateTime);      // 2025-01-18T19:14:30
```

TABLE 4.7 Conversion methods to `LocalDate`, `LocalTime`, and `LocalDateTime`

`LocalDate` to `LocalDateTime`	`atStartOfDay()` `atTime(int hour, int minute)` `atTime(int hour, int minute, int second)` `atTime(int hour, int minute, int second, int nanos)` `atTime(LocalTime time)`
`LocalTime` to `LocalDate`	`atDate(LocalDate date)`
`LocalDateTime` to `ZonedDateTime`	`atZone(ZoneId zoneId)`

Period & Duration

TABLE 4.8 Where to use `Duration` and `Period`

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

Instant

Working with Instants

The `Instant` class represents a specific moment in time in the GMT time zone. Suppose that you want to run a timer.

```
var now = Instant.now();
// Do something time consuming
var later = Instant.now();

var duration = Duration.between(now, later);
System.out.println(duration.toMillis()); // Returns number milliseco
```

If you have a `ZonedDateTime`, you can turn it into an `Instant`:

```
var date = LocalDate.of(2025, 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(); // 2025-05-25T15:55:00Z
System.out.println(zonedDateTime); // 2025-05-25T11:55-04:00[US/East
System.out.println(instant);       // 2025-05-25T15:55:00Z
```

- If all the elements are the same, but the first array has extra elements at the end, return a positive number.
- If the first element that differs is smaller in the first array, return a negative number.
- If the first element that differs is larger in the first array, return a positive number.

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.

TABLE 4.3 Equality vs. comparison vs. mismatch

Method	When arrays contain the same data	When arrays are different
Arrays.equals()	true	false
Arrays.compare()	0	Positive or negative number
Arrays.mismatch()	-1	Zero or positive index

Using mismatch()

Now that you are familiar with compare(), it is time to learn about mismatch(). If the arrays are equal, mismatch() returns -1. Otherwise, it returns the first index where they differ. Can you figure out what these print?

```
System.out.println(Arrays.mismatch(new int[] {1}, new int[] {1}));
System.out.println(Arrays.mismatch(new String[] {"a"},
   new String[] {"A"}));
System.out.println(Arrays.mismatch(new int[] {1, 2}, new int[] {1}));
```

In the first example, the arrays are the same, so the result is -1. In the second example, the entries at element 0 are not equal, so the result is 0. In the third example, the entries at element 0 are equal, so we keep looking. The element at index 1 is not equal. Or, more specifically, one array has an element at index 1, and the other does not. Therefore, the result is 1.

```
int[] vars4 [], space [][];  // 2D and 3D arrays
```

EXAM

1.f

2.c,e,f

2. Which of these array declarations are not legal? (Choose all that apply.)
   - A. `int[][] scores = new int[5][];`
   - B. `Object[][][] cubbies = new Object[3][0][5];`
   - C. `String beans[] = new beans[6];`
   - D. `java.util.Date[] dates[] = new java.util.Date[2][];`
   - E. `int[][] types = new int[];`
   - F. `int[][] java = new int[][];`

2. C, E, F. Option C uses the variable name as if it were a type, which is clearly illegal. Options E and F don't specify any size. Although it is legal to leave out the size for later dimensions of an array of arrays, the first one is required. Option A declares a legal 2D array. Option B declares a legal 3D array. Option D declares a legal 2D array. Remember that it is normal to see classes on the exam you might not have learned. You aren't expected to know anything about them.

3.A,C,D

4.A,C,D

5.B

5. What is the result of the following code?

```
7: var sb = new StringBuilder();
8: sb.append("aaa").insert(1, "bb").insert(4, "ccc");
9: System.out.println(sb);
```

- A. abbaaccc
- B. abbaccca
- C. bbaaaccc
- D. bbaaccca
- E. An empty line.
- F. The code does not compile.

6.C

6. C. Remember to watch return types on math operations. One of the tricks is line 24. The round() method returns an int when called with a float. However, we are calling it with a double, so it returns a long. The other trick is line 25. The random() method returns a double. Since two lines have a compiler error, option C is the answer.

7.a,e

8.a,b,f

9.a,~~g~~ C F

9. A, C, F. Arrays are zero-indexed, making option A correct and option B incorrect. They are not able to change size, which is option C. The values can be changed, making option D incorrect. An array does not override equals(), so it uses object equality. Since two different objects are not equal, option F is correct, and options E and G are incorrect.

10.a

10. How many of these lines contain a compiler error?

```
23: int one = Math.min(5, 3);
24: long two = Math.round(5.5);
25: double three = Math.floor(6.6);
26: var doubles = new double[] {one, two, three};
```

- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

11.e

12.A,~~F~~,D E

12. What is output by the following code ignoring any new lines in the output? (Choose all that apply.)

```
var numbers = "012345678".indent(1);
numbers = numbers.stripLeading();
System.out.println(numbers.substring(1, 3));
System.out.println(numbers.substring(7, 7));
System.out.println(numbers.substring(7));
```

- A. 12
- B. 123
- C. 7
- D. 78
- E. A blank line.
- F. An exception is thrown.

12. A, D, E. First, notice that the indent() call adds a blank space to the beginning of numbers, and stripLeading() immediately removes it. The substring() method has two forms. The first takes the index to start with and the index to stop immediately before. The second takes just the index to start with and goes to the end of the String. Remember that indexes are zero-based. The first call starts at index 1 and ends with index 2 since it needs to stop before index 3. This gives us option A. The second call starts at index 7 and ends in the same place, resulting in an empty String, which is option E. This prints out a blank line. The final call starts at index 7 and goes to the end of the String finishing up with option D.

13.~~D~~ B

13. What is the result of the following code?

```
public class Lion {
   public void roar(String roar1, StringBuilder roar2) {
      roar1.concat("!!!");
      roar2.append("!!!");
   }
   public static void main(String[] args) {
      var roar1 = "roar";
      var roar2 = new StringBuilder("roar");
      new Lion().roar(roar1, roar2);
      System.out.println(roar1 + " " + roar2);
} }
```

- A. roar roar
- B. roar roar!!!
- C. roar!!! roar
- D. roar!!! roar!!!
- E. An exception is thrown.
- F. The code does not compile.

14.F,~~C~~ B

14. Given the following, which can correctly fill in the blank allowing the code to compile? (Choose all that apply.)

```
var date = LocalDate.now();
var time = LocalTime.now();
var dateTime = date.______(time);
var zoneId = ZoneId.systemDefault();
var zonedDateTime = ZonedDateTime.of(dateTime, zoneId);
Instant instant = ______________________;
```

- A. atTime()
- B. atTime()
- C. withTime()
- D. dateTime.toInstant()
- E. new Instant()
- F. zonedDateTime.toInstant()

15.C,~~G~~ E

15. What is the output of the following? (Choose all that apply.)

```
var arr = new String[] { "PIG", "pig", "123"};
Arrays.sort(arr);
System.out.println(Arrays.toString(arr));
System.out.println(Arrays.binarySearch(arr, "Pippa"));
```

- A. [pig, PIG, 123]
- B. [PIG, pig, 123]
- C. [123, PIG, pig]
- D. [123, pig, PIG]
- E. -3
- F. -2
- G. The results of binarySearch() are undefined in this example.

15. C, E. Numbers sort before letters, and uppercase sorts before lowercase. This makes option C one of the answers. The binarySearch() method looks at where a value would be inserted, which is before the second element for Pippa. It then negates it and subtracts one, which is option E.

16.a,~~b,d~~,g

16. Which of these statements are true? (Choose all that apply.)

```
var letters = new StringBuilder("abcdefg");
```

- A. letters.substring(1, 2) returns a single character String.
- B. letters.substring(2, 2) returns a single character String.
- C. letters.substring(6, 5) returns a single character String.
- D. letters.substring(6, 6) returns a single character String.
- E. letters.substring(1, 2) throws an exception.
- F. letters.substring(2, 2) throws an exception.
- G. letters.substring(6, 5) throws an exception.
- H. letters.substring(6, 6) throws an exception.

16. A, G. The substring() method includes the starting index but not the ending index. When called with 1 and 2, it returns a single-character String, making option A correct and option E incorrect. Calling substring() with 2 as both parameters is legal. It returns an empty String, making options B and F incorrect. Java does not allow the indexes to be specified in reverse order. Option G is correct because it throws a StringIndexOutOfBoundsException. Finally, option H is incorrect because it returns an empty String.

17 F C

```
13: String s1 = """
14:    purr""";
15: String s2 = "";
16:
17: s1.toUpperCase();
18: s1.trim();
19: s1.substring(1, 3);
20: s1 += "two";
21:
22: s2 += 2;
23: s2 += 'c';
24: s2 += false;
25:
26: if ( s2 == "2cfalse") System.out.println("==");
27: if ( s2.equals("2cfalse")) System.out.println("equals");
28: System.out.println(s1.length());
```

- A. 2
- B. 4
- C. 7
- D. 10
- E. ==
- F. equals
- G. An exception is thrown.
- H. The code does not compile.

17. C, F. This question is tricky because it has several parts. First, you have to know that the text block on lines 13 and 14 is equivalent to a regular String. Since there is no line break at the end, this is four characters. Then, you have to know that String objects are immutable, which means the results of lines 17–19 are ignored. Finally, on line 20, something happens. We concatenate three new characters to s1 and now have a String of length 7, making option C correct.

Next, s2 += 2 expands to s2 = s2 + 2. A String concatenated with any other type gives a String. Lines 22, 23, and 24 all append to s2, giving a result of "2cfalse". The if statement on line 27 returns true because the values of the two String objects are the same using object equality. For this reason, option F is correct. The if statement on line 26 returns false because the two String objects are not the same in memory. One comes directly from the string pool, and the other comes from building using String operations.

18.A,B,D

19.

20A,C

21.E

21 What is the output of the following code?

```
var date = LocalDate.of(2025, Month.APRIL, 30);
date.plusDays(2);
date.plusYears(10);
System.out.println(date.getYear() + " " + date.getMonth()
  + " " + date.getDayOfMonth());
```

A. 2025 APRIL 30
B. 2025 MAY 2
C. 2026 APRIL 2
D. 2028 APRIL 30
E. 2028 MAY 2
F. The code does not compile.
G. A runtime exception is thrown.

33. A. The date starts out as April 30, 2025. Since dates are immutable and the plus methods' return values are ignored, the result is unchanged. Therefore, option A is correct.

22.e