1 Data Types

1.1 Date and Time

1.1.1 LocalDate

LocalDate is an immutable date-time object that represents a date, often viewed as year-month-day. Other date fields, such as day-of-year, day-of-week and week-of-year, can also be accessed.

```
// to obtain, e.g.
static LocalDate of(int year, int month, int dayOfMonth)
static LocalDate of(int year, Month month, int dayOfMonth)
static LocalDate ofInstant(Instant instant, ZoneId zone)
static LocalDate parse(CharSequence text, DateTimeFormatter
    formatter)

// instance methods, e.g.
// return a copy!
LocalDateTime atTime(int hour, int minute, int second, int
    nanoOfSecond)
LocalDateTime atTime(LocalTime time)

int getDayOfMonth()
DayOfWeek getDayOfWeek()
int getDayOfYear()
Month getMonth()
int getMonthValue()
int getYear()

// same for plus
// return a copy!
LocalDate minus(long amountToSubtract, TemporalUnit unit)
LocalDate minusDays(long daysToSubtract)
LocalDate minusMonths(long monthsToSubtract) //etc
```

Beware immutability:

```
var date = LocalDate.of(2022, Month.APRIL, 30);
date.plusDays(2); // does not change date
```

1.1.2 LocalTime

LocalTime is an immutable date-time object that represents a time, often viewed as hour-minute-second. Time is represented to nanosecond precision. For example, the value ”13:45.30.123456789” can be stored in a LocalTime.

```java
// to obtain, e.g.
static LocalTime of(int hour, int minute, int second, int
    nanoOfSecond)
static LocalTime ofInstant(Instant instant, ZoneId zone)

// instance methods, e.g.
// return a copy!
LocalDateTime atDate(LocalDate date)

int getHour()
int getMinute() //etc.

// same for minus
// return a copy!
LocalTime plus(long amountToAdd, TemporalUnit unit)
LocalTime plusNanos(long nanosToAdd) // etc.

// return a copy!
LocalTime withHour(int hour)
LocalTime withMinute(int minute) //etc.
```

1.1.3 LocalDateTime

```java
// to obtain, e.g.
static LocalDateTime of(int year, Month month, int
    dayOfMonth, int hour, int minute, int second, int
    nanoOfSecond)
static LocalDateTime of(LocalDate date, LocalTime time)
// instance methods analogous to above
```

1.1.4 Month

In addition to the textual enum name, each month-of-year has an int value (1-12). Do not use ordinal() to obtain the numeric representation of Month. Use getValue() instead.

```java
// to obtain, e.g.
static Month of(int month)
Month e = Month.of(10); // DECEMBER
static Month valueOf(String name)
Month m = Month.valueOf("DECEMBER"); // DECEMBER

// instance methods, e.g.
int getValue()
int length(boolean leapYear)
minus(long months)
```

```
plus(long months)
```

1.1.5 ChronoUnit

```
// to obtain, e.g.
static ChronoUnit valueOf(String name)

// instance methods, e.g.
<R extends Temporal> R addTo(R temporal, long amount) //
    returns a copy!
long between(Temporal temporal1Inclusive, Temporal
    temporal2Exclusive)

// Usage example
//
    https://stackoverflow.com/questions/77587361/chronounit-months-inconsisten
var d1 = OffsetDateTime.parse("2023-01-31T10:00:00Z");
var d2 = ChronoUnit.MONTHS.addTo(d1, 1);
d2 = ChronoUnit.HOURS.addTo(d2, 23);
System.out.println(ChronoUnit.MONTHS.between(d1, d2)); // 0
}
```

1.1.6 Instant

An Instant represents a specific moment in time using GMT. Consequently, there is no time zone information.

```
// to obtain, e.g.
static Instant from(TemporalAccessor temporal)
static Instant now()
static Instant ofEpochMilli(long epochMilli)

// instance methods, e.g.
OffsetDateTime atOffset(ZoneOffset offset)
ZonedDateTime atZone(ZoneId zone)

Instant minus(long amountToSubtract, TemporalUnit unit)
    //returns copy! others too
Instant minus(TemporalAmount amountToSubtract)

Instant minusMillis(long millisToSubtract)
Instant minusNanos(long nanosToSubtract)
```

```
var instant = trainDay.toInstant(); // will not compile if
    this is a LocalDateTime!
```

1.1.7 Period

This class models a quantity or amount of time in terms of years, months and days. See Duration for the time-based equivalent to this class.

Durations and periods differ in their treatment of daylight savings time when added to ZonedDateTime. A Duration will add an exact number of seconds, thus a duration of one day is always exactly 24 hours. By contrast, a Period will add a conceptual day, trying to maintain the local time.

For example, consider adding a period of one day and a duration of one day to 18:00 on the evening before a daylight savings gap. The Period will add the conceptual day and result in a ZonedDateTime at 18:00 the following day. By contrast, the Duration will add exactly 24 hours, resulting in a ZonedDateTime at 19:00 the following day (assuming a one hour DST gap).

The supported units of a period are YEARS, MONTHS and DAYS. All three fields are always present, but may be set to zero.

The period is modeled as a directed amount of time, meaning that individual parts of the period may be negative.

```
// to obtain, e.g.
static Period between(LocalDate startDateInclusive,
    LocalDate endDateExclusive)
static Period of(int years, int months, int days)
static Period ofDays(int days) // other fields will be 0
static Period ofMonths(int months)

// instance methods, e.g.
Temporal addTo(Temporal temporal)
Period minusDays(long daysToSubtract) // all return copies!
minusMonths(long monthsToSubtract)

Period withMonths(int months) // copies, too!
Period withYears(int years)

int getDays()
```

1.1.8 Duration

This class models a quantity or amount of time in terms of seconds and nanoseconds. It can be accessed using other duration-based units, such as minutes and hours. In addition, the DAYS unit can be used and is treated as exactly equal to 24 hours, thus ignoring daylight savings effects.

See Period for the date-based equivalent to this class.

```
// to obtain, e.g.
static Duration of(long amount, TemporalUnit unit)
// convenience method
static Duration ofDays(long days)

// instance methods, e.g.
Duration dividedBy(long divisor) // all these copy
long dividedBy(Duration divisor)

long get(TemporalUnit unit)
int getNano()
long getSeconds()

// static methods, e.g.
// throws UnsupportedTemporalTypeException if passed a LocalDate
Duration between(LocalDateTime now, LocalDateTime other);
```

1.2 String and StringBuilder

1.2.1 String

```
 // Instance methods e.g.
char charAt(itn index)
boolean contains(CharSequence s);
String concat(String str);
int indexOf(int ch);
String replace(char oldChar, char newChar);
String[] split(String regex);

// strip()-related methods (these are the only ones)
strip(), stripLeading(), stripTrailing(), stripIndent()

// indent():
// indent(n) splits into lines and then indents each; also adds
    newline after the last line if there's none.
// indent(0) does not change the indentation, but still adds a
    newline after the last line
// indent(-n) removes indentation iff there is any, and also
    adds a newline after the last line

// indent() example
var phrase = "prickly \nporcupine";
System.out.println(phrase);
System.out.println("........................................");
System.out.println(phrase.indent(1));
```

```
    System.out.println("......................................");
    System.out.println(phrase.indent(0));
    System.out.println("......................................");
    System.out.println(phrase.indent(-1));
    System.out.println("......................................");

    // Output:
    prickly
    porcupine
    ......................................
     prickly
     porcupine

    ......................................
    prickly
    porcupine

    ......................................
    prickly
    porcupine

    ......................................


    // translateEscapes()
    // these print 2 lines:
    System.out.println("cheetah\ncub");
    System.out.println("cheetah\ncub".translateEscapes());
    System.out.println("cheetah\\ncub".translateEscapes());
   - this prints 1:
    System.out.println("cheetah\\ncub");

    // there is no reverse()
```

1.2.2 StringBuilder

```
        // instance methods, e.g.
        char charAt(int index)
        IntStream chars()

        int indexOf(String str)

        int length()

        StringBuilder
        delete(int start, int end)
```

```
// beware: returns a new String!
String substring(start, end)
```

1.3 Numbers

1.3.1 Number types: automatic promotion

Integer literals are considered int by default (size notwithstanding). But they can be automatically promoted to long, float, or double.

```
final ___ song = 6; // can be int, long, float, double
    (automatic promotion from int)
```

When used as an argument to overloaded methods, ints will preferentially be promoted to long (or double) before autoboxing to Integer is considered.

1.3.2 Autoboxing

Beware: cannot autobox and promote at the same time!

1.3.3 Math methods

```
Math.round() // double -> double
Math.max() // overloaded, returns passed-in type
Math.pow() // double -> double
```

1.3.4 Parsing Strings

```
var numPigeons = Long.parseLong("100"); // returns long
var numPigeons2 = Long.valueOf("100"); // returns Long
```

Examples:

```
// compiles - Boolean's being lenient
Boolean.valueOf("8").booleanValue() // false

// Character has no byteValue()
Character.valueOf('x').byteValue(); // does not compile
```

```
// cannot have underscore next to decimal point
Double.valueOf("9_.3").byteValue(); // NumberFormatException

Long.valueOf(128).byteValue(); // - 128
```

1.4 Arrays

These are all legal array declarations:

```
// no var allowed
String[][] gamma;
String[] delta[];
String epsilon[][];
```

1.5 java.util.Optional<T>

```
// to obtain:
static <T> Optional<T>   empty()
static <T> Optional<T>   of(T value)empty Optional
// if value null return
static <T> Optional<T>   ofNullable(T value)

// instance methods, e.g.

// If a value is present and predicate matched, return Optional
    with value, otherwise return an empty Optional.
Optional<T>  filter(Predicate<? super T> predicate)

// may throw NoSuchElementException
T get()

void ifPresent(Consumer<? super T> consumer)

// If value present, apply mapping function to it, and if the
    result is non-null, return an Optional describing the result.
<U> Optional<U>  map(Function<? super T,? extends U> mapper)

// Return the value if present, otherwise return other.
T orElse(T other)

//  Return value if present, otherwise invoke other
T orElseGet(Supplier<? extends T> other)
```

```java
<X extends Throwable> T orElseThrow(Supplier<? extends X>
    exceptionSupplier)
```

2 Operators

2.1 Kinds

2.1.1 Logical Operators

```java
&&
||
// no ~!
```

2.1.2 Bitwise Operators: Logical

```java
&
int result = 5 & 6; // 4     101 & 110 = 100
|
int result = 5 | 6; // 7     101 | 110 = 111
^
int result = 5 ^ 6; // 3     101 ^ 110 = 011
~
int result = ~6; // -7       0000 0110 -> 1111 1001 -> 0000
    0110 + 1 -> 0000 0111
```

2.1.3 Bitwise NOT, complements, etc.

Ex. 1: Bitwise NOT computation steps:

1. to compute 6, first write 6 in binary: 0000 0110
2. negate each bit: 1111 1001 // this is the 1's complement
3. to get the 2's complement (since numbers are stored as 2's complement), add 1: 1111 1010

Ex. 2: To find the binary representation of -17, take the 2's complement of 17:

1. 17 = 0001 0001
2. Take the bitwise complement: 1110 1110

3. Add 1: 1110 1110 + 1 = 1110 1111

Ex. 3: Take the 2's complement of negative number:

1. Start from binary -17: 1110 1111
2. Take the the bitwise complement: 0001 0000
3. Add 1: 0001 0001
4. This gets back the 17!

Ex. 4: To find the decimal representation of a number given in binary, reverse steps

1. Subtract 1: 1110 1111 - 1 = 1110 1110
2. Take the complement of the complement: 0001 0001
3. Change from base 2 back to base 10 16 + 1 = 17
4. Rewrite this as a negative integer: -17

2.1.4 Bitwise operators: arithmetic

```
// shift left (signed)
12  << 2: 48 // *2^n

// shift right (signed)
12 >> 2: 3 // 1100 -> 0011 (pos.: fill with 0)
-12 >> 2: -3 // (neg..: fill with 1)

// shift right (unsigned)
12 >>> 2: 3 // 1100 -> 0011
-12 >>> 2: 1073741821 // fill with 0 too
```

2.2 Precedence

Default evaluation order is left-to-right.

Post-unary operator	x++, x–	
Pre-unary operator	++x, ++x	
Other unary operators	-, !,∼, +	Right-to-left
Cast(type)reference		Right-to-left
Multiplication/division/modulus	*, /, %	
Addition/subtraction	+, -	
Shift operators	≪, ≫, ⋙	
Relational operators	<, >, <=, >=, instanceof	
Equal to/not equal to	==, !=	
Logical AND	&	
Logical exclusive OR	ˆ	
Logical inclusive OR	|	
Conditional OR	||	
Conditional AND	&&	
Ternary operator	e1 ? e2 : e3	Right-to-left
Assignment operators	=, +=, -=, *=, /=, %=, &=, ˆ, |, <<= , >>=, >>>=	Right-to-left
Arrow operator	− >	

In a nutshell:

- shift ops after +, -
- relational before equality before logical
- & | before && ||
- ternary thereafter but before assignment
- assignment last

3 Syntax

3.1 Allowed variable names

Names may contain: underscore, currency symbol, numbers, letters (first may not be a number)

3.2 Reserved Words

See 1.

3.3 Variable declarations and initialization topics

For instance variables, on a single line only one type should be specified.

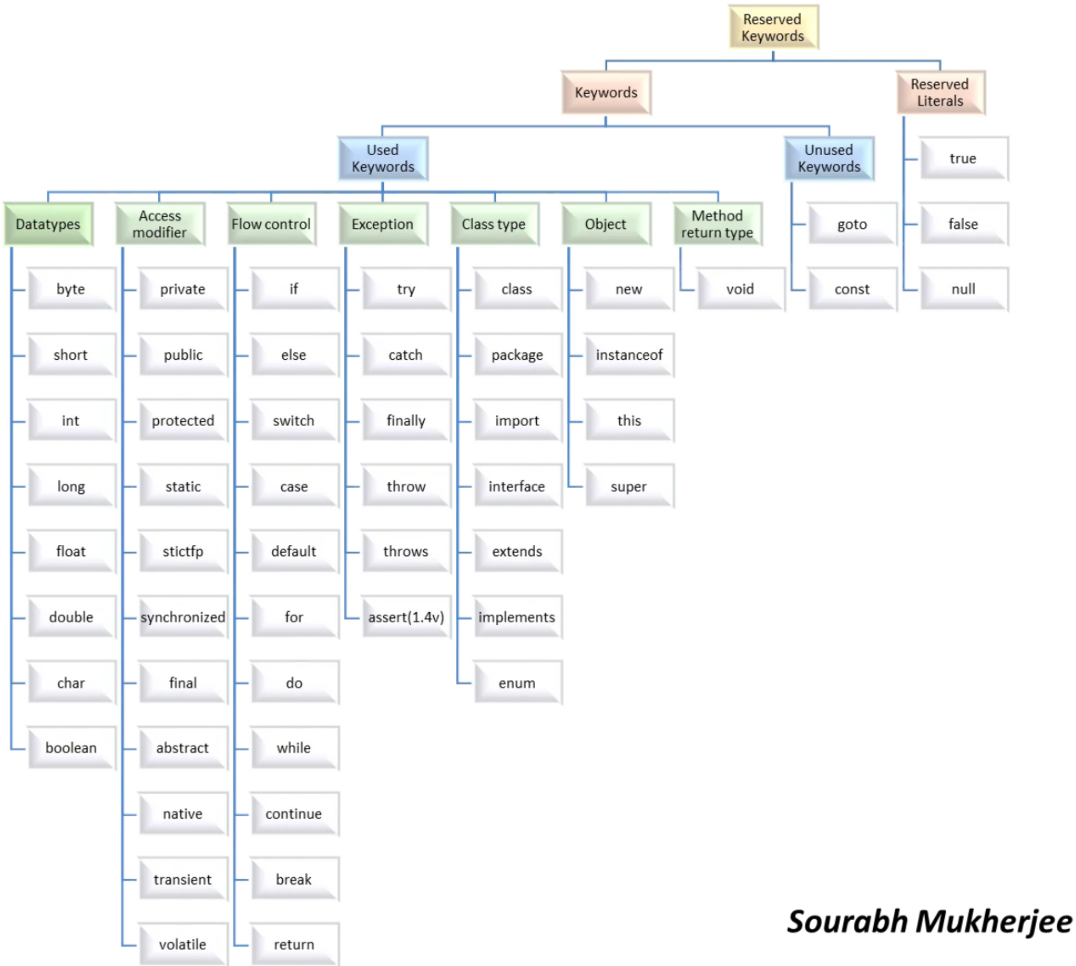

Figure 1: Reserved Words

3.4 Text blocks

Imagine a vertical line drawn on the leftmost non-whitespace character in the text block. Everything to the left of it is incidental whitespace, and everything to the right is essential whitespace.

Note:

- \at end of line means the line gets continued
- Space at end of line is ignored
- \s yields two spaces
- \n yields additional line break

Example:

```
// starting quotes on sep. line
String block = """
doe \
 // ending quotes on same line
doof""";

 var quotes = """
\"The Quotes that Could\" // could remove both backslashes
   here
\"\"\"                    // could remove 2 backslashes
   (otherwise end of format string)
""";
```

3.5 var

A var cannot be initialized with a null value without giving a type.

var cannot be used in a multiple-variable assignment.

3.6 underscore

An underscore can be placed in any numeric literal, as long as it is not at the beginning, at the end, or next to a decimal point. Underscores can even be placed next to each other.

3.7 switch

3.7.1 General

- Supports the following data types as arguments to switch(): enum, byte, Byte, short, Short, char, Character, int, Integer, String, var (if resolves to one of those types)
- Does not support: boolean, Boolean, double, Double, float, Float, long, Long

3.7.2 Statement

- The value passed to a *case* (not switch!) statement must be a constant, a literal value, or a final variable (not, e.g., **case** red:)
- May use comma to separate case constants in statements: e.g.,

```
public int getAverageTemperate(Season s) {
    switch (s) {
        default:
        case WINTER, SUMMER: return 30;
}}
```

3.7.3 Expression

- A switch expression requires all possible case values to be handled, or a default to be added.
- switch expressions execute exactly one branch and do not use break statements.
- switch expressions need a semicolon.
- Can omit default clause in when either all the values of an enum are covered or no value is returned.
- Case labels must be compile-time constants.
- Every path must return a value.

Example:

```
case 10 -> {"Jane";} // yield implied
case 20 -> {yield "Lisa";}
// expression may also have multiple values passed to case
case 30, 40 -> "Kelly";
default -> "Unassigned";
```

3.8 for (x : y)

A for-each loop accepts arrays and classes that implement java.lang.Iterable, such as List. Not: String, StringBuilder

3.9 Flow Scoping

Example:

```
static void printIfString(Object o) {
    if (!(o instanceof String s)) {
        // 's' is NOT in scope
        return;
    } else {
        // 's' is in scope
        // s is a string
        System.out.println(s);
    }
}

// this is valid:
if (x instanceof Foo(var v) && v != null) {
    A;
}

//this is NOT:
if (x instanceof Foo(var v) || v != null) {
    A;
}
```

```
if (number instanceof Integer i && Math.abs(i) == 0) // ok
if (number instanceof Integer i || Math.abs(i) == 0) // i
    not defined
if (number instanceof int i && Math.abs(i) == 0) // can't
    use primitives
```

3.10 Loops

for and while (but not do-while) don't need braces if just one statement follows.

```
while (i<6) System.out.println("");
for (;;) System.out.println();
```

4 Methods and functions

4.1 General notes

Java does not support setting default method parameter values.

4.2 Pass-by-value and references

Java creates a copy of references and pass it to the invoked method, but they still point to same memory reference. Thus, changes made to the object are reflected in the caller.

However, changes are not reflected back if we *change the reference itself* to refer some other location or object!

4.3 Method references

4.3.1 static methods

```
interface Converter {
    long round(double num);
}

Converter methodRef = Math::round;

// same as lambda
Converter lambda = x -> Math.round(x);
```

Like with lambdas, Java uses type inference to infer that x here is a double.

4.3.2 Instance methods on an instance object

Here are two ways of calling an *instance method that takes no parameters*.

With the first, the interface method, too, takes no parameters:

```
interface StringChecker {
    boolean check();
}

var str = "";
// uses concrete instance, not class method
StringChecker methodRef = str::isEmpty;

// same as lambda
StringChecker lambda = () -> str.isEmpty();
```

With the second, the interface method takes one parameter:

```
interface StringParameterChecker {
    boolean check(String text);
}

// now uses class!
StringParameterChecker methodRef = String::isEmpty;
```

```
        // same as lambda
        StringParameterChecker lambda = s -> s.isEmpty();
```

How about *instance methods that take one or more parameters*? Here, the *interface method takes two parameters*:

```
            // the lambda
            StringTwoParameterChecker lambda = (s, p) ->
                s.startsWith(p);

            interface StringTwoParameterChecker {
                boolean check(String text, String prefix);
            }

            // uses class method!
            StringTwoParameterChecker methodRef = String::startsWith;

            // first parameter is instance, all others are method
                parameters
            methodRef.check("Zoo", "A"));
```

4.3.3 Constructors

This example mimics a *lambda calling a constructor that takes no parameters.*

```
        interface EmptyStringCreator {
            String create();
        }

        EmptyStringCreator methodRef = String::new;
        var myString = methodRef.create();
        System.out.println(myString.equals("Snake"))

        // lambda
        EmptyStringCreator lambda = () -> new String();
```

Here, Java sees that we are passing a String parameter and *calls the constructor of String that takes such a parameter*:

```
        // lambda
        StringCopier lambda = x - new String(x);

        interface StringCopier {
            String copy(String value);
        }
```

```
// same method reference!
StringCopier methodRef = String::new;

var myString = methodRef.copy("Zebra");
```

4.4 Mixed notes on functions

4.4.1 compose()

The a.compose(b) method calls the Function parameter b before the reference Function variable a.

To memorize:

```
after.compose(before)

// is like
after o before
```

5 Classes, Interfaces, Records and Enums

5.1 General Notes

5.1.1 Member (Class and Instance) Variables

All member fields (static and non-static) are initialized to their default values.

Objects are initialized to null, numeric types to 0 (or 0.0), and boolean to false.

If a static variable is final, it must be initialized, either in the declaration or in a static initializer.

5.1.2 Inheritance

When a parent defines a with-argument constructor only, the child must call the parent constructor explicitly. It is not enough for both to take the same arguments. See:

```
class Cinema {
    private String name = "Sequel";
    public Cinema(String name) {
        this.name = name;
    }
}
public class Movie extends Cinema {
    private String name = "adaptation";
    public Movie(String movie) {
        this.name = "Remake";
```

```
        }
    }
```

An instance variable with same name in the child hides the parent variable.

When a parent method is static, the child cannot "override" (*nor complement*) this by a same-arg instance method.

5.2 Classes

5.2.1 Top-level classes

Top-level classes may only have public or package access.

5.2.2 Nested Classes

There are four types of nested classes: inner, static, local, and anonymous. Nested classes can be public.

An *inner* class requires an instance of the outer class to use. Three ways that are legal:

```
public class Dinosaur {
    class Pterodactyl extends Dinosaur {}

    // it all happens in an instance method
    public void roar() {
        var dino = new Dinosaur();

        // uses instance to create inner
        dino.new Pterodactyl();

        // relies on the fact that roar() is an instance method
        //, which means there's an implicit instance of the
            outer class Dinosaur available
        new Dinosaur.Pterodactyl();

        // The Dinosaur. prefix is optional, though
        new Pterodactyl();
    } }
```

While a *static* nested class does not:

```
    new Lion.Den()
```

A *local* class is commonly defined within a method or block. Local classes can only access local variables that are final or effectively final.

Anonymous classes are a special type of local class that does not have a name. Anonymous classes are required to extend exactly one class or implement one interface.

Note:

Inner, local, and anonymous classes can *access private members* of the class in which they are defined, provided the latter two are used inside an instance method.

Static nested classes cannot access the instance variables or methods of the outer class.

All four types of nested classes can now define static variables and methods.

5.2.3 Sealed Classes

A sealed class is a class that restricts which other classes may directly extend it:

```
sealed class Friendly extends Mandrill permits Silly {}
```

Every class that directly extends a sealed class must specify exactly one of the following three modifiers: final, sealed, or non-sealed.

Rules regarding *permit*:

- A sealed class/interface must have a permits clause. An exception is that if all the direct subclasses/subinterfaces are defined in the same compilation unit, then the permits clause is not needed.
- Beware though: the extends clause still is! See:

```
// MyClass.java:25: error: cannot find symbol
//sealed class Friendly extends Mandrill permits Silly {}
//                    ^
// symbol: class Mandrill
// MyClass.java:25: error: invalid permits clause
//
sealed class Friendly extends Mandrill permits Silly {}
//                   ^
// (subclass Silly must extend sealed class)
// 2 errors
```

- If a sealed class C is associated with a *named module*, then every class specified in the permits clause must be associated with the same *module* as C.
- If a sealed class C is associated with an *unnamed module* , then every class specified in the permits clause must belong to the same *package* as C.

Note:

- Sealed classes cannot be final (otherwise the permits clause would make no sense).

- We can have sealed interfaces (permitting both extensions and implementations). There are no sealed enums or records, though. Records are implicitly final, so it doesn't make sense to mark them as sealed. Enums are either implicitly final (if its declaration contains no enum constants that have a class body) or implicitly sealed (if its declaration contains at least one enum constant that has a class body, in which case the anonymous classes implicitly declared by the enum constants are its only permitted subclasses). However, enums (as well as records) may implement a sealed interface:

```
sealed interface I permits X, R{ }
enum X implements I{ }
record R(int value) implements I{ }
```

- While a sealed class is commonly extended by a subclass marked final, it can also be extended by a sealed or non-sealed subclass marked abstract.

5.2.4 Final and Immutable Classes

- Classes marked as final can't be extended. - Immutable classes do not include setter methods. They must be marked final *or* contain only private constructors.

5.3 Interfaces

- *Variables* are always public, static, final. No other modifiers are allowed.
- *Methods* can be either *default* (these are always public), static (always public), private, or private and static. Methods cannot be final nor protected. If there is no modifier, a method is implicitly public.
- Methods *with a body* must be declared *default, static or private.*
- *Default* methods are implicitly public. There is no modifier that can prevent a default method from being overridden in a class implementing an interface.
- Interfaces cannot have a constructor.
- If implementing two interfaces that define a default method with the same signature, classes (incl. abstract classes) have to override it explicitly, or their declaration will not compile. (I.e., this fails on compilation already, not on call. See example.)

  They can then access the inherited ones by calling Interface.super.method() - provided this happens in an instance method, since otherwise super() is not available.

```
public class App {
    interface Building {
        default Double getHeight() {
```

```
            return 1.0;
        }
    }
    interface Office {
        public default String getHeight() {
            return null;
    }
    // does not compile
    abstract class Tower implements Building, Office {}
```

- *Private* methods must contain a body.
- *Static* methods are only accessible with a qualifier. A static method can be invoked from other static or from default method. A static method cannot be overridden or changed in the implementation class. The static methods are implicitly public — there's no need to specify the public modifier.
- Interfaces may contain member type declarations (nested type). A member type declaration in an interface is implicitly static and public. It is permitted to redundantly specify either or both of these modifiers.

5.4 Records

Minimal example:

```
// parentheses are required
public record Crane(int numberEggs, String name) { }
```

Records may optionally have constructors.

5.4.1 Record Constructors

Constructors can be compact or long, as well as canonical or non-canonical.

Canonical or Non-canonical Canonical constructors are passed all of the fields mentioned in the record declaration. Non-canonical constructors may take less (no, even) or more arguments.

Long vs. Compact Both the long as well as the short constructor are *canonical constructors*, i.e., they operate on the exact record fields mentioned in the record declaration.

The following is a short constructor. Of those, a record can have at most one (since a short contructor uses the fields specified in the record declaration).

At the end of the compact record constructor, the compiler adds code that sets all the fields to the values passed in the constructor. E.g., if the constructor changes a passed value, the updated value will be given to respective field.

```java
public Crane { // no parens
    if (numberEggs < 0) throw new IllegalArgumentException();
    // could not be this.name in the compact constructor
    name = name.toUpperCase();
    // long form is automatically called here
}
```

And this is long constructor (there may be several):

```java
public Crane(int numberEggs, String name) {
    if (numberEggs < 0) throw new IllegalArgumentException();
    this.numberEggs = numberEggs;
    this.name = name;
}
```

Constructors may be overloaded:

```java
public record Crane(int numberEggs, String name) {
    public Crane(String firstName, String lastName) {
        // must be first call, must either call 1) another
            long constructor or 2) the short or 3) the
            default constructor
        this(0, firstName + " " + lastName);
}}
```

An example where both are present:

```java
public record Disco(int beats) {
    public Disco(String beats) {
        this(20);
    }
    public Disco {
        beats = 10;
    }
    public int getBeats() {
        return beats;
    }
    public static void main(String[] args) {
        // beats() is provided by default
        // calls default constructor
        System.out.print(new Disco(30).beats());
    }
}
```

Records may:

- implement interfaces:

```
public record Crane(int numberEggs, String name) implements
    Bird {}
```

- override a method:

```
public record BeardedDragon(boolean fun) {
    // overriding generated accessor
    @Override public boolean fun() { return false; }
    }
```

- have any access modifier.
- contain nested classes, interfaces, annotations, enums, and other records.
- contain instance *methods*.
- contain *static* fields, initializers and methods.

Records may not:

- extend other classes.
- be subclassed, since they are implicitly final.
- declare instance *variables* or instance *initializers*.

5.5 Enums

Enums can have constructors, methods, and fields. Methods may be abstract, while enum classes may not.

Example:

```
enum Flavors {
    VANILLA, CHOCOLATE, STRAWBERRY;
    // must declare values before members
    static final Flavors DEFAULT = STRAWBERRY;
}
```

Constructors are implicitly private. Example:

```java
enum Animals {
    // When an enum contains any other members, such as a
        constructor or variable, a semicolon is required:
    MAMMAL(true), INVERTEBRATE(Boolean.FALSE),
        BIRD(false),REPTILE(false),
    AMPHIBIAN(false), FISH(false) {public int swim() {
        return 4; }};

    final boolean hasHair;

    private Animals(boolean hasHair) {this.hasHair =
        hasHair;}

    public boolean hasHair() { return hasHair; }
    public int swim() { return 0; }
}
```

An Enum constructor can accept multiple values. Example:

```java
public enum Element {
    H("Hydrogen", 1, 1.008f),
    HE("Helium", 2, 4.0026f),
    // ...
    NE("Neon", 10, 20.180f);

    private Element(String label, int atomicNumber, float
        atomicWeight) {
        this.label = label;
        //
    }
}
```

Methods:

```java
// to obtain:
static <T extends Enum<T>> T valueOf(Class<T> enumClass,
    String name)

// instance method to get the ordinal (starts with 0)
final int ordinal()
```

5.6 Overriding, overloading and Hiding

5.6.1 Overloading

Overloading works also for pairs of primitive + wrapper, e.g.

```java
    public static void main(String... args) {
        System.out.println(new App().woof(5)); // 1
        System.out.println(new App().woof(Integer.valueOf(5)));
            // 2
    }
    public String woof(int bark) {
        return "1";
    }
    public String woof(Integer bark) {
        return "2";
    }
```

5.6.2 Overriding and Hiding

Methods and variables are considered separately.

- *Variables*

  Instance variables are never overridden. They are always determined by the reference type.

  The same goes for class variables. In the following example,the child class variable hides the parent's class variable; it is thus not a problem that the parent variable is final.

  ```java
  class A{
      final int fi = 10;
  }
  public class B extends A{
      int fi = 15;
  }
  ```

- *Methods*
  - Overridden and hidden methods can only have covariant return types. This also applies to implementing abstract methods.
  - When an instance method is private, no overriding takes place (so the child can do what it wants). In other words, the parent method is *hidden*.
  - Overriding replaces the method *regardless of the reference type.*
  - When a parent defines a with-arg constructor only, the child must call the parent constructor explicitly (it is not enough for both to take overridden the same arguments).

- When a child class defines a *static method with the same signature* as a static method in the parent class, then the child's method *hides* the one in the parent class.

  This is determined at *compile* time, as opposed to overriding of instance methods, which is resolved at runtime.

  The child *cannot* complement the parent's static method by a same-argument instance method.

5.7 Class loading and initialization

5.7.1 Class

First, static variables are created, then static initializers are run.

Static variables cannot access instance variables.

If any static variable is final, it must be initialized (either at declaration or in a static initializer).

5.7.2 Instance

Instance initialization blocks are invoked after the *parent class constructor* has been invoked (i.e., after the super() constructor call). An instance initializer can also access any static variables.

Beware: Variables newly created in the initializer block are in scope only there!

If an instance variable is final, it must be assigned a value when it is declared, either in an instance initializer or in a constructor.

5.7.3 Overall initialization order

1. static variables in order
2. static initializers in order
3. instance variables in order
4. call to *parent class constructor*
5. instance initializers in order
6. local variables created in constructor

Example:

```
abstract class Car {
    // 1st (on class load)
    static { System.out.print("1"); }
    public Car(String name) {
        super();
        // 3rd
```

```java
            System.out.print("2");
        }
        // 2nd
        // instance initializer, runs due to super() in BlueCar()
        { System.out.print("3"); } }
    public class BlueCar extends Car {
        // 4th
        { System.out.print("4"); }
        public BlueCar() {
            super("blue");
            // 5th
            System.out.print("5");
        }
        public static void main(String[] gears) {
            new BlueCar();
    } }
```

6 Collections

6.1 Collections class

This class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection.

Notes on methods:

```java
        // Collections.binarySearch() returns index
```

6.2 Arrays class

This class contains various static methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

Examples:

```java
        Arrays.length()

        Arrays.binarySearch()

        // takes array or individual elements
        // eturns a fixed-size list backed by the specified array
        // size has to remain the same
        // hanges made to the array will be visible in the returned
            list,
        // and changes made to the list will be visible in the array
```

```java
Arrays.asList()
// e. g.
Integer[] numbers = [1,2,3];
List<Integer> values = Arrays.asList(numbers);
List<String> stooges = Arrays.asList("Larry", "Moe",
    "Curly");

// takes two arrays and returns the index of the first item
    differing between the arrays.
Arrays.mismatch()

// compares every element in order, returns positive if
    first is bigger
Arrays.compare()
```

6.3 List

```java
// remove() is overloaded!
// boolean remove(Object obj) removes an object that matches
    the parameter.
// E remove(int index) removes (and returns) the element at
    the specified index.
var list = new LinkedList<Integer>();
list.add(3);
list.add(2);
list.add(1);
list.remove(2); // int, index
list.remove(Integer.valueOf(2)); //Integer, value

// List.of(): returns an unmodifiable list containing an
    arbitrary number of elements.
// can take individual elements or array
List.of()

// List.copyOf(): takes Collection and returns an immutable
    List
static <E> List<E> copyOf(Collection<? extends E> coll)
```

6.3.1 ArrayList

6.4 Maps

6.5 Sets

6.6 Queues

Queue methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation).

	Throws exception	Returns special value
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

```
// ---------------------- INSERTION ----------------------
// inserts the specified element
// returns true upon success, throws an
    IllegalStateException if no space is currently available
boolean add(E e)

// inserts the specified element
// returns true if the element was added to this queue, else
    false
boolean offer ()

// ---------------------- RETRIEVAL ----------------------
// retrieves, but does not remove, the head of this queue
// throws NoSuchElementException if this queue is empty
E element()

// retrieves, but does not remove, the head of this queue,
    or returns null if this queue is empty
E peek()

// ---------------------- REMOVAL ----------------------
// retrieves and removes the head of this queue
// throws NoSuchElementException if the queue is empty
E remove()

// retrieves and removes the head of this queue, or returns
    null if this queue is empty.
E poll()
```

6.7 Deque

A linear collection that supports element insertion and removal at both ends.

This interface defines methods to access the elements at both ends of the deque. Methods are provided to insert, remove, and examine the element. Each of these methods exists in two forms: one throws an exception if the operation fails, the other returns a special value (either null or false, depending on the operation).

Methods overview:

	First Element (Head)		Last Element (Tail)	
	Throws exception	Special value	Throws exception	Special value
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

This interface extends the Queue interface. When a deque is used as a queue, FIFO (First-In-First-Out) behavior results. Elements are added at the end of the deque and removed from the beginning. The methods inherited from the Queue interface are precisely equivalent to Deque methods as indicated in the following table:

Queue Method	Eqv. Deque Method
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Deques can also be used as LIFO (Last-In-First-Out) stacks. This interface should be used in preference to the legacy Stack class. When a deque is used as a stack, elements are pushed and popped from the beginning of the deque.

7 Generics

7.1 Declare

To declare (may leave out types on the right):

```java
Map<Long, List<Integer>> mapOfLists = new HashMap<>();
```

These all work:

```java
// Object assumed
var list = new ArrayList<>();
```

```java
// recommended:
var wash = new Wash<String>();
Wash<String> wash = new Wash<>();

// these compile, too:
var wash = new Wash<>();
Wash wash = new Wash();
Wash wash = new Wash<String>();
```

7.2 Wildcards

```java
// Unbounded Wildcard
// ?
List<?> a = new ArrayList<String>();

// Wildcard with upper bound
// ? extends type
List<? extends Exception> a = new
    ArrayList<RuntimeException>();

// Wildcard with lower bound
// ? super type
List<? super Exception> a = new ArrayList<Object>();

// e.g.
public List<? super Number> getList(){
    // could be any of
    return new ArrayList<Number>();
    return new ArrayList<Object>();
    return new ArrayList();
   }
```

Wildcards are only allowed in variable references (on left side), not in a class definition.

```java
// NOT allowed
class Fur<? extends Mammal> {}
```

Valid and invalid method declarations:

```java
// this is possible
public static <E> void swap(List<E> list, int src, int des);
```

```java
// if a type parameter appears only once in the method
    declaration, a wildcard is fine, too (and is preferable)
public static void swap(List<?> list, int src, int des);

// here a wildcard cannot be used:
private static <T extends Collection, U> U add(T list, U
    element)
// the return type is U!

// can't use T here (couldn't add anything)
public static void getExceptions(Collection<? extends Mammal)
```

7.3 When and why use lower bounds?

Consider:

```java
List<String> strings = new ArrayList<String>();
strings.add("tweet");

List<Object> objects = new ArrayList<Object>(strings);

// to a List<Object>, can't pass List<String> (need exact
    match)
public static void addSound(List<? super String> list)
    {list.add("quack");}
addSound(strings);
addSound(objects);

// to be able to do those...
coll.add(new RuntimeException());
coll.add(new Exception());
// need this:
public static void getExceptions(Collection<? super
    Exception>)
// whereas here, we could not add a broader type
public static void getExceptions(Collection<? super
    RuntimeException>)

// another example
List<? super IOException> exceptions = new
    ArrayList<Exception>();

// DOES NOT COMPILE because this could be a List<IOException>
exceptions.add(new Exception());

exceptions.add(new IOException());
exceptions.add(new FileNotFoundException());
```

```
// BUT this is ok:
List<? super Exception>
```

8 Streams, functional interfaces, and lambda functions

8.1 Lambda functions

Lambdas (as well as method references) use type inference. The parameter list may contain types, but does not have to. If var is used, it has to be used for all parameters.

Cmp. the following valid and invalid declarations.

```
// these are fine
Comparator<String> c1 = (j, k) -> 0;
Comparator<String> c2 = (String j, String k) -> 0;
 Comparator<String> c5 = (var j, var k) -> 0;

// these don't
Comparator<String> c3 = (var j, String k) -> 0;
Comparator<String> c4 = (var j, k) -> 0;

var dino = s -> "dino".equals(animal);
var dragon = s -> "dragon".equals(animal);
// does not compile, because var
var combined = dino.or(dragon);
```

Lambdas can reference any instance or static variable, as well as any lambda parameter. Local variables and method parameters must be *effectively final* to be accessible.

Lambda expressions cannot redeclare any local variable defined in an enclosing scope.

```
// does not compile
var p = new Hyena();
testLaugh(p, p -> true); // local variable already exists

// nor does this:
Set<?> s = Set.of("lion", "tiger", "bear");
Consumer<Object> consumer = s -> s.forEach(consumer);
```

8.2 Functional interfaces

A valid functional interface is one that contains a single abstract method, excluding any public methods that are already defined in the java.lang.Object class.

Beware: Types must match!

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T var1);
}

// So this will not compile:
Consumer<Object> c2 = String::new; // there is no new
    String(object)
```

8.3 Stream characteristics

ORed values from: ORDERED, DISTINCT, SORTED, SIZED, NONNULL, IMMUTABLE, CONCURRENT, SUBSIZED

If the stream is parallel, and the Collector is concurrent, and either the stream is unordered or the collector is unordered, then a concurrent reduction will be performed.

```
boolean isOrdered =
    stream.spliterator().hasCharacteristics(Spliterator.ORDERED);
```

8.4 Grouping/Collecting

Grouping operations return *boxed* numbers, Map, Optional, ...

8.4.1 Grouping vs. partitioning

groupingBy creates a Map<K, List<T>> as per the specified function, with optional *downstream collector* and optional *map type supplier*.

```
groupingBy(Function f);
groupingBy(Function f, Collector dc);
groupingBy(Function f, Supplier s, Collector dc);
```

partitioningBy creates a Map<Boolean, List<T>> as per the specified predicate, with optional further downstream collector. Note: there is no map type specifier!

```
partitioningBy(Predicate p);
partitioningBy(Predicate p, Collector dc);
```

8.4.2 collect() with downstream collector: Collectors.toMap(), Collectors.toSet(), Collectors.counting(), Collectors.mapping()

```
var ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, String> map = ohMy.collect(
    Collectors.toMap(
        String::length,
        k -> k,
        (s1, s2) -> s1 + "," + s2
    )
);

Map<Integer, Set<String>> map = ohMy.collect(
        Collectors.groupingBy(
        String::length,
        Collectors.toSet()
    )
);

Map<Integer, Long> map = ohMy.collect(
    Collectors.groupingBy(
        String::length,
        Collectors.counting()
    )
);
System.out.println(map); // {5=2, 6=1}

// specifying map type
TreeMap<Integer, Set<String>> map = ohMy.collect(
    Collectors.groupingBy(
        String::length,
        TreeMap::new,
        Collectors.toSet()
    )
);

// using another Collector in mapping
Stream<String> ohMy = Stream.of("lions", "tigers", "bears");
Map<Integer, Optional<Character>> map = ohMy.collect(
    Collectors.groupingBy(
        String::length,
        Collectors.mapping(s -> s.charAt(0),
            Collectors.minBy((a, b) -> a-b))
    )
);
System.out.println(map); //{5=Optional[b], 6=Optional[t]}

// teeing to return multiple results
record Separations(String spaceSeparated, String
```

```
        commaSeparated) {}
    var list = List.of("x", "y", "z");
    Separations result = list.stream()
        .collect(Collectors.teeing(
            Collectors.joining(" "),
            Collectors.joining(","),
            (s, c) -> new Separations(s, c)
            )
        );
    // Separations[spaceSeparated=x y z, commaSeparated=x,y,z]
```

8.4.3 Summary statistics

Collectors may also return summary statistic. Example:

```
Car[] cars = new Car[]{ new SUV("Ford", 5000), // ... new
   }

DoubleSummaryStatistics dss =
    Stream.of(cars)
    .filter(c->c instanceof SUV)
    .collect(Collectors.summarizingDouble(c-> ((SUV)c).milage));

// note: methods like getMax(), as opposed to max() etc. on
   primitive streams
System.out.println(dss.getMax());
```

8.5 Comparing and sorting

8.5.1 Using Comparator

Comparator is a functional interface with one non-default method (besides equals()), compare():

```
    compare(T o1, T o2);
```

Comparator returns a negative integer if the first argument is smaller, zero if both are the same, or a positive integer if the first argument is bigger. I.e., if (a-b) is negative, then a ist smaller than b.

Example:

```
    Comparator<Duck> byWeight = new Comparator<Duck>() {
        public int compare(Duck d1, Duck d2) {
            return d1.getWeight() - d2.getWeight();
```

```
    }
};
```

Methods:

```
// Compare by results of function that returns any Object
    (or primitive autoboxed into Object).
comparing(function)

// Compare by results of function that returns double.
comparingDouble(function)

// Compare by results of function that returns int.
comparingInt(function)

// Compare by results of function that returns long.
comparingLong(function)

// Sort using order specified by the Comparable
    implementation on object itself.
naturalOrder()

// Sort using reverse of order specified by Comparable
    implementation on object itself.
reverseOrder()

//Method chaining
reversed(), thenComparing(function),
    thenComparingDouble(function) ...
```

```
// syntax note: compare
// 1
s.sorted(Comparator.reverseOrder())

// 2
s.sorted(Comparator::reverseOrder) // does not compile! Why?

// sorted() takes a Comparator, i.e. a functional interface
    that takes 2 parameters and returns an int
```

```java
// while Comparator::reverseOrder is equivalent to (() ->
    Comparator.reverseOrder()), which is a
    Supplier<Comparator>
```

8.5.2 Using Comparable

Comparable is an interface with one method to be implemented, compareTo():

```java
public int compareTo(Sometype m)
```

8.6 Primitive streams

Primitive streams cannot take Double, Integer, etc. - there is no unboxing!

The other way round, boxing does happen! See:

```java
// this works:
IntFunction<Integer> f3 = s -> s; // int is boxed to Integer

// this does not:
IntFunction<Integer> f1 = (Integer f) -> f; // Integer is
    NOT unboxed to int
```

However, both IntSupplier and Supplier<Integer> *can* return a double!

```java
IntSupplier s = new MyIntSupplier();
int i = s.getAsInt();

// unboxing
Supplier<Integer> s = new MyIntSupplier();
int i = s.get();
```

Here is another auto-unboxing example. All these can be assigned to ToDoubleBiFunction:

```java
// unboxing
(Integer a, Double b) -> {int c; return b;}

(h,i) -> (long)h

(x,y) -> {int z=2; return y/z;}
```

8.6.1 Optional types

Nearly all statistics operations on primitive streams return *optional types*. Only sum() returns 0.

```
OptionalDouble opt = s.average()
opt.getAsDouble() // (may throw NoSuchElementException)

// max (Comparator) returns an Optional
mylist.stream().max(Comparator.comparing(a->a)).get();
```

8.7 Mapping, reducing, filtering

8.7.1 reduce()

If a combiner present, we get different results with parallel and serial streams.

```
var data = List.of(1,2,3);
int j = data.parallelStream().reduce(1, (a,b) -> a+b, (a,b)
    -> a+b); // 9
int f = data.stream().reduce(1, (a,b) -> a+b, (a,b) -> a+b);
    // 7
```

Reductions where no initial value is provided return an Optional:

```
List<Integer> ls = Arrays.asList(10, 47, 33, 23);
int max = ls.stream().reduce(Integer.MIN_VALUE, (a, b)->
    a>b?a:b).get()
```

8.7.2 flatmap()

E.g.,

```
integerList.stream().flatMapToInt(x -> IntStream.of(x))
integerList.stream().flatMapToDouble(x -> DoubleStream.of(x))
```

8.7.3 Mapping primitive streams to object streams or primitive streams

Examples:

```
// analogously for IntStream, LongStream
DoubleStream -> map() -> DoubleStream
```

```
// analogously for IntStream, LongStream
DoubleStream ->mapToObj()-> Stream<T>
```

Re. mapping functions:

```
// generic to all generic (no primitives involved): takes
    Function<T,R>

// generic to primitive:
ToDoubleFunction<T>, ToIntFunction<T>, ToLongFunction<T>

// primitive to primitive:
// e.g.

// DoubleStream -> map()-> generic: takes
DoubleFunction<R>

// DoubleStream -> mapToDouble()-> DoubleStream takes:
DoubleUnaryOperator

// DoubleStream -> mapToInt()-> IntStream: takes
DoubleToIntFunction

// DoubleStream -> mapToLong() -> LongStream takes:
     DoubleToLongFunction

// IntStream -> map[...]() -> generic/Int/Long/Double-Stream
    takes:
IntFunction<R>, IntUnaryOperator, IntToLongFunction,
    IntToDoubleFunction

// LongStream -> map[...]() ->
    generic/Int/Long/Double-Stream takes:
LongFunction<R>, LongUnaryOperator, LongToIntFunction,
    LongToDoubleFunction

// DoubleStream -> map[...]*( )->
    generic/Int/Long/Double-Stream: takes
DoubleFunction<R>, DoubleUnaryOperator, DoubleToIntFunction,
    DoubleToLongFunction
```

8.7.4 Chaining operations

E.g., andThen():

```
// andThen: for Consumer and Function
```

```
// Returns a composed BiConsumer that performs, in sequence,
    this operation followed by the after operation.
default BiConsumer<T,U> andThen BiConsumer<? super T, ?
    super U> after
```

8.8 Splitting up streams with Spliterator

```
var stream = List.of("bird-", "bunny-", "cat-", "dog-",
    "fish-", "lamb-", "mouse-");
Spliterator<String> originalBagOfFood = stream.spliterator();

// has the ones at the beginning
Spliterator<String> emmasBag = originalBagOfFood.trySplit();

// bird-bunny-cat
emmasBag.forEachRemaining(System.out::print);

// original now has the 4 ones at the end, jill gets dog and
    fish
Spliterator<String> jillsBag = originalBagOfFood.trySplit();

// dog-
// advance gets the first and removes it
jillsBag.tryAdvance(System.out::print);

// fish-
jillsBag.forEachRemaining
(System.out::print);

// lamb-mouse
originalBagOfFood.forEachRemaining(System.out::print);
```

9 IO

9.1 java.io

9.1.1 Serialization

To be serializable, a class must implement the Serializable marker interface. All members must either be serializable, as well, or must be declared transient (otherwise a NotSerializableException is thrown).

Methods and the exceptions they throw:

```
// ObjectInputStream
```

```
public Object readObject() throws IOException,
    ClassNotFoundException

// ObjectOutputStream
public void writeObject(Object obj) throws IOException
```

On deserialization, the constructor and any instance initializers *not* called. Instead, Java will call the *no-arg constructor of the first non-serializable parent class* it can find in the class hierarchy.

Static (!) as well as transient fields are ignored.

Values that are not provided are given their default Java value, such as null for String, or 0 for int values.

To read several objects from a file, we need to use an infinite loop to process the data, which throws an EOFException when the end of the I/O stream is reached. That's because, when calling readObject(), null and -1 do not have any special meaning, as someone might have serialized objects with those values.

```
var gorillas = new ArrayList<Gorilla>();
try (var in = new ObjectInputStream(new
    BufferedInputStream(new FileInputStream(dataFile)))) {
     while (true) {
          var object = in.readObject();
          if (object instanceof Gorilla g)
          gorillas.add(g);
     }
} catch (EOFException e) { // File end reached
}
return go
```

9.1.2 Abstract base classes

InputStream, OutputStream, Reader, Writer.

Beware: these are abstract classes, not interfaces!

9.1.3 Byte Streams

Classes: FileInputStream, FileOutputStream, BufferedInputStream (readAllBytes), BufferedOutputStream, PrintStream.

PrintStream methods: append(byte b),..., format(Locale l, String format, Object... args), void print(boolean b),...

9.1.4 Character Streams

FileReader, FileWriter, BufferedReader (readLine), BufferedWriter (write (line), newLine), PrintWriter.

PrintWriter methods: append(char c),..., format(Locale l, String format, Object... args), void print(boolean b),...

Beware: As opposed to BufferedInputStream, Buffered Reader does not have a method to read all lines!

9.1.5 Console

```
// to obtain (constructor is private)
Console console = System.console();

// always check that console is not null
Console console = System.console();
if (console != null)  String userInput = console.readLine();

// fields e.g.
Reader, Writer, PrintWriter

// methods e.g.
reader()
readLine() - this we wouldn't get from the Reader field!
// NOT: read() - we have to get the Reader first
reader.read()
```

9.1.6 Using mark()

```
// methods
public boolean markSupported()

// readLimit: instructs the I/O stream that we expect to
    call reset() after at most readLimit bytes.
// If our program calls reset() after reading more than 100
    bytes from calling mark(readLimit), it may throw an
    exception, depending on the I/O stream class.
public void mark(int readLimit)

public void reset() throws IOException

// returns number of values skipped
public long skip(long n) throws IOException

// how to use
public void readData(InputStream is) throws IOException {
    System.out.print((char) is.read()); // L

    if (is.markSupported()) {
```

```
            is.mark(100); // Marks up to 100 bytes
            System.out.print((char) is.read()); // I
            System.out.print((char) is.read()); // O
            is.reset(); // Resets stream to position before I
            }
            System.out.print((char) is.read()); // I
            System.out.print((char) is.read()); // O
            System.out.print((char) is.read()); // N
        }
```

9.2 java.nio

9.2.1 Constructing a path: Path.of, Paths.get

```
        Path zooPath1 = Path.of("/home/tiger/data/stripes.txt");
        Path zooPath2 = Path.of("/home", "tiger", "data",
           "stripes.txt");

        Path zooPath3 = Paths.get("/home/tiger/data/stripes.txt");
        Path zooPath4 = Paths.get("/home", "tiger", "data",
           "stripes.txt");
```

9.2.2 Getting individual parts: getName()

How it works:

- Indices for path names start from 0.
- Root (i.e. c:
) is not included in path names.
- is NOT a part of a path name.
- If you pass a negative index or a value greater than or equal to the number of elements, or this path has zero name elements, java.lang.IllegalArgumentException is thrown. It DOES NOT return null.

Example:

```
    // Path is "c:\\code\\java\\PathTest.java"

    p1.getRoot() // c:\\
    p1.getName(0) // code
    p1.getName(1) // java
    p1.getName(2) // PathTest.java
```

```
p1.getName(3) // IllegalArgumentException
```

9.2.3 Conversion to or back from java.io.File

```
File file = new File("rabbit");
Path nowPath = file.toPath();
File backToFile = nowPath.toFile();
```

9.2.4 Concatenation: Path resolve(Path other)

Resolves the given path against this path. Does not normalize!

```
// the input argument is appended onto the Path, e.g.

// with input a relative path:
Path path1 = Path.of("/cats/../panther");
Path path2 = Path.of("food");
System.out.println(path1.resolve(path2));
// /cats/../panther/food

// if input is absolute, return input
Path path3 = Path.of("/turkey/food");
path3.resolve("/tiger/cage");
// /tiger/cage
```

9.2.5 Constructing a relative path: Path relativize(Path other)

```
//requires that both path values be absolute or relative
// otherwise, an exception is produced at runtime
var path1 = Path.of("fish.txt");
var path2 = Path.of("friendly/birds.txt");
path1.relativize(path2);
// ../friendly/birds.txt

// Note: the file itself counts as one level!
path2.relativize(path1);
// ../../fish.txt
// => go up "plus 1"

// Relativization is the inverse of resolution.
// For any two normalized paths p and q, where q does not
    have a root component, we have
```

```
p.relativize(p.resolve(q)).equals(q)
```

9.2.6 toAbsolutePath

Resolves the path in an implementation dependent manner, typically by resolving the path against a file system default directory. For me, the current working directory is picked, and transformed to absolute.

9.2.7 Normalizing a path: normalize

```
var p1 = Paths.get("/pony/../weather.txt");
var p2 = Paths.get("/weather.txt");
p1.equals(p2); // false
p1.normalize().equals(p2.normalize()); // true
```

9.2.8 Resolve symlinks: toRealPath

```
ll horse/
schedule/
food.txt

ll zebra/
schedule/
food.txt

Paths.get("/zebra/food.txt").toRealPath(); // /horse/food.txt
Paths.get("../../food.txt").toRealPath()); // same --
    normalizes
```

9.2.9 Copying files: copy

```
// copy from file to file
Files.copy(Paths.get("book.txt"), Paths.get("movie.txt"),
    StandardCopyOption.REPLACE_EXISTING);
// copy from stream to file
try (var is = new FileInputStream("source-data.txt")) {
     Files.copy(is, Paths.get("/mammals/wolf.txt"));
}
// copy from file to stream
Files.copy(Paths.get("/fish/clown.xsl"), System.out);
```

```java
// copy a file into a directory
var file = Paths.get("food.txt");
var directory = Paths.get("/enclosure/food.txt"); // NOT
    /enclosure!
Files.copy(file, directory);
```

9.2.10 Comparing file content: isSameFile() and mismatch()

isSameFile() uses equals (maybe having to normalize).

mismatch() returns -1 if the files are the same; otherwise, it returns the index of the first position in the file that differs.

9.2.11 Reading files

```java
// reads the entire file into memory
// returns List<String>
Files.readAllLines(Paths.get("birds.txt"))
     .forEach(System.out::println);

// reads lazily
// returns Stream<String>
Files.lines(Paths.get("birds.txt"))
     .forEach(System.out::println);
```

9.2.12 java.nio.Files methods

e.g.,

- creation: createDirectories, createSymbolicLink, ...
- deletion: deleteIfExists, delete, ...
- other: newBufferedWriter, ...
- retrieve attributes: isDirectory, isRegularFile, isSymbolicLink, isHidden(), isReadable(), isWriteable(), isExecutable()

```java
// find
Stream<Path> find(Path start, int maxDepth,
    BiPredicate<Path, BasicFileAttributes> matcher,
    FileVisitOption... options) throws IOException

// walk
```

```java
public static Stream<Path> walk(Path start, int maxDepth,
    FileVisitOption... options) throws IOException
```

9.2.13 Using views for attribute retrieval

A view is a group of related attributes for a particular file system type.

```java
public static <A extends BasicFileAttributes> A
    readAttributes(
     Path path,
     Class<A> type,
     LinkOption... options
) throws IOException

var path = Paths.get("/turtles/sea.txt");   // needs a Path!
BasicFileAttributes data = Files.readAttributes(path,
    BasicFileAttributes.class);
System.out.println("Is a directory? " + data.isDirectory());
System.out.println("Is a regular file? " +
    data.isRegularFile());
System.out.println("Is a symbolic link? " +
    data.isSymbolicLink());
System.out.println("Size (in bytes): " + data.size()); //
    not length()!
System.out.println("Last modified: " +
    data.lastModifiedTime());
```

To modify attributes, use BasicFileAttributeView, not BasicFileAttributes:

```java
public static <V extends FileAttributeView> V
    getFileAttributeView(
     Path path,
     Class<V> type,
     LinkOption... options
) throws IOException

// step 1: Read file attributes, using BasicFileAttributeView
var path = Paths.get("/turtles/sea.txt");
// this uses BasicFileAttributeView, NOT BasicFileAttributes
BasicFileAttributeView view =
    Files.getFileAttributeView(path,
BasicFileAttributeView.class);
BasicFileAttributes attributes = view.readAttributes();

// step 2: Modify file last modified time
FileTime lastModifiedTime =
    FileTime.fromMillis(attributes.lastModifiedTime().toMillis()
    + 10_000);
```

```java
// BasicFileAttributeView instance method
//public void setTimes(FileTime lastModifiedTime,
//FileTime lastAccessTime, FileTime createTime)
view.setTimes(lastModifiedTime, null, null);
```

9.2.14 Common method arguments

```java
// Enums implementing (empty) interfaces, e.g.
public enum StandardCopyOption implements CopyOption {
    REPLACE_EXISTING,
    COPY_ATTRIBUTES,
    ATOMIC_MOVE;
    private StandardCopyOption() {}}
```

10 Exception handling

10.1 Multi-catch

```java
// DOES NOT COMPILE
catch(Exception1 e | Exception2 e | Exception3 e)

// DOES NOT COMPILE
catch(Exception1 e1 | Exception2 e2 | Exception3 e3)

// this works
catch(Exception1 | Exception2 | Exception3 e)
```

10.2 try-with-resources

Only classes that implement the AutoCloseable interface can be used in a try-with-resources statement.

Resources are closed in reverse order of how they were created.

Resources can be declared in advance, provided they are final or effectively final.

```java
// simple syntax
try (FileInputStream is = new FileInputStream("myfile.txt"))
    {}

// if we have several resources, a semicolon between them is
    needed
try (
```

```
        var in = new FileInputStream("data.txt");
        // the one at the end is optional
        var out = new FileOutputStream("output.txt");) {}
```

If the close() method also throws an exception, this is added as *suppressed*. In other words: If exceptions are thrown from both the try block and the try-with-resources statement, the exception from the try block thrown; the exception thrown from the try-with-resources block is added as suppressed.

Example:

```
public class JammedTurkeyCage implements AutoCloseable {

    public void close() throws IllegalStateException {
        throw new IllegalStateException("Cage door does not
            close");
    }

    public static void main(String[] args) {
        try (JammedTurkeyCage t = new JammedTurkeyCage()) {
            // primary
            throw new IllegalStateException("Turkeys ran
                off");
            // here close is called, we get an
                IllegalStateException
            // this is added as suppressed
        } catch (IllegalStateException e) {
            // primary
            System.out.println("Caught: " + e.getMessage());
            for (Throwable t: e.getSuppressed())
            // the one from close()
            System.out.println("Suppressed:
                "+t.getMessage());
}}}
// Output
// Caught: Turkeys ran off
// Suppressed: Cage door does not close
```

Another example:

```
try (JammedTurkeyCage t = new JammedTurkeyCage()) {
    // primary exception
    // not caught
    throw new RuntimeException("Turkeys ran off");
    // close is called, get IllegalStateException
    // this is added as suppressed
} catch (IllegalStateException e) {
    System.out.println("caught: " + e.getMessage());
    for (Throwable t: e.getSuppressed())
```

```
            // the one from close()
            System.out.println("Suppressed: "+t.getMessage());
    }
    // Output
    // Exception in thread "main" java.lang.RuntimeException: //
       Turkeys ran off
    // Suppressed: java.lang.IllegalStateException:Cage door
       does not close
```

Note: Programmer-provided catch and final blocks are run after automatic ones.

11 Internationalization

11.1 Resource bundles

Once a resource bundle has been selected, only properties along a *single hierarchy* will be used.

11.2 Using Locales

```
    Locale l2 = new Locale.Builder()
        .setRegion("US")
        .setLanguage("en")
        .build();

    // formatting with a Locale: withLocale
    dtf.withLocale(locale).format(dateTime));
```

11.3 Formatting

11.3.1 Formatting Dates

Using DateTimeFormatter:

```
    LocalDate date = LocalDate.of(2022, Month.OCTOBER, 20);
    LocalTime time = LocalTime.of(11, 12, 34);
    LocalDateTime dt = LocalDateTime.of(date, time);

    System.out.println(date.format(DateTimeFormatter.ISO_LOCAL_DATE));
    System.out.println(time.format(DateTimeFormatter.ISO_LOCAL_TIME));
    System.out.println(dt.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

    // custom format
    // a     a.m./p.m.           AM, PM
    // z   Time zone name        Eastern Standard Time, EST
```

```
// Z   Time zone offset      -0400
var f = DateTimeFormatter.ofPattern("MMMM dd, yyyy 'at'
    hh:mm");

// For formatting dates
DateTimeFormatter.ofLocalizedDate(FormatStyle dateStyle)
// For formatting times
DateTimeFormatter.ofLocalizedTime(FormatStyle timeStyle)
// For formatting dates and times
// FormatStyle: possible values are SHORT, MEDIUM, LONG, and
    FULL.
DateTimeFormatter.ofLocalizedDateTime(FormatStyle dateStyle,
    FormatStyle timeStyle)
DateTimeFormatter.ofLocalizedDateTime (FormatStyle
    dateTimeStyle)
```

To format, one can use both dates/time classes themselves and the formatter.

The date/time classes contain a format() method that will take a formatter, while the formatter classes contain a format() method that will take a date/time value:

```
var dateTime = LocalDateTime.of(2022, Month.OCTOBER, 20, 6,
    15, 30);
var formatter = DateTimeFormatter.ofPattern("MM/dd/yyyy
    hh:mm:ss");
System.out.println(dateTime.format(formatter));
System.out.println(formatter.format(dateTime));
```

11.3.2 Formatting Numbers and Currencies

```
// currency
String income = "$92,807.99";
var cf = NumberFormat.getCurrencyInstance();
double value = (Double) cf.parse(income);
System.out.println(value); // 92807.99

// number
// public DecimalFormat(String pattern)
// #  Omit position if no digit exists for it.     $2.2
// 0  Put 0 in position if no digit exists for it.  $002.20
```

12 Concurrency

12.1 Thread states

Note: Calling interrupt() on a thread in the TIMED_WAITING or WAITING states causes the main() thread to become RUNNABLE again, triggering an InterruptedException.

Calling interrupt() on a thread already in a RUNNABLE state doesn't change the state.

12.2 Interface Runnable

```
// def.
@FunctionalInterface public interface Runnable {void run();}

// to start, call start, not run
new Thread(() -> System.out.print("Hello")).start();

// example
Runnable printInventory = () -> System.out.println("Printing
    zoo inventory");
new Thread(printInventory).start();
```

12.3 Interface Callable

```
// def.
@FunctionalInterface public interface Callable<V> {V call()
    throws Exception;}
```

12.4 Concurrency API

```
ExecutorService service =
    Executors.newSingleThreadExecutor();
try {
    service.execute(printInventory);
} finally {
    // doesn't implement AutoCloseable!
    service.shutdown();
}

// isShutdown() -- no longer accepts new
// isTerminated() -- is shut down
```

```
service.awaitTermination(1, TimeUnit.MINUTES); // after
    shutdown
```

Comparing execute() and submit(). Excute():

```
// def.
void execute(Runnable command)  / no returns!
```

Submit():

```
// def.
// pass Runnable, get Future
Future<?> submit(Runnable task)

// pass Callable<T>, get Future<T>
<T> Future<T> submit(Callable<T> task)

// pass collection of Callables
// waits for all tasks to complete
<T> List<Future<T>> invokeAll(Collection<? extends
    Callable<T>> tasks)

// pass collection of Callables
// waits for at least one task to complete
<T> T invokeAny(Collection<? extends Callable<T>> tasks)

// get result
// for Runnable: always null!
ExecutorService service =
    Executors.newSingleThreadExecutor();
try {
     Future<?> result = service.submit(() -> {
          for(int i = 0; i < 1_000_000; i++) counter++;
     });
     // Returns null for Runnable!
     result.get(10, TimeUnit.SECONDS);
     System.out.println("Reached!");
} catch (TimeoutException e) {
     System.out.println("Not reached in time");
} finally {
     service.shutdown();
}}

// for Callable, this returns something
try {
     Future<Integer> result = service.submit(() -> 30 + 11);
```

```
        System.out.println(result.get());
    }
```

12.5 volatile

Ensures that only one thread is modifying a variable at one time and that data read among multiple threads is consistent. But operations are not atomic!

13 Modules and services

13.1 Modules

13.1.1 Module types

A *named* module has the name inside the module-info.java file and is on the module path

An *automatic* module appears on the module path but does not contain a module--info.java file. Exports all files to other modules on module path.

A *unnamed* module appears on the classpath. It exports no files to other modules.

Note: Code on the classpath can access the module path. By contrast, code on the module path is unable to read from the classpath.

13.1.2 Migration

13.2 Services

A service is composed of an interface, any classes the interface references, and a way of looking up implementations of the interface. The implementations are not part of the service.

Artifact	Part of the service	Directives required
Service provider interface	Yes	exports
Service provider	No	requires, provides
Service locator	Yes	exports, requires,uses
Consumer	No	requires

13.2.1 Interface

```
    public interface Tour {
        String name();
        int length();
        Souvenir getSouvenir();
    }

    // module-info.java
    module zoo.tours.api {
```

```
        exports zoo.tours.api;
    }
```

13.2.2 Service Locator

A service locator can find any classes that implement a service provider interface. Luckily, Java provides a ServiceLoader class to help with this task. You pass the service provider interface type to its load() method, and Java will return any implementation services it can find.

```
     public static List<Tour> findAllTours() {
        List<Tour> tours = new ArrayList<>();
        ServiceLoader<Tour> loader =
            ServiceLoader.load(Tour.class);
        // implements Iterable
        // can also use findFirst() to get first found
        for (Tour tour : loader)
        tours.add(tour);
        return tours;
    }

    // module-info.java
    module zoo.tours.reservations {
        exports zoo.tours.reservations;
        requires zoo.tours.api; // for compilation
        uses zoo.tours.api.Tour; // to use service
    }
```

ServiceLoader methods:

```
    public static <S> ServiceLoader<S> load(Class<S> service)
    public Stream<Provider<S>> stream() { ... }

    // see
    ServiceLoader.load(Tour.class).stream().map(Provider::get)
```

13.2.3 Consumer

```
    List<Tour> tours = TourFinder.findAllTours();
```

13.2.4 Service Provider

```
public class TourImpl implements Tour {...}

module zoo.tours.agency {
    requires zoo.tours.api;
    provides zoo.tours.api.Tour with
        zoo.tours.agency.TourImpl;
```

13.3 Migration

13.3.1 Top-down

If A.jar depends on B.jar B.jar depends on C.jar, in the top down approach we directly make A.jar modular by including a module-info and adding a `requires B;` clause.

We create an automatic module for B.jar by simply placing it on module-path (instead of the classpath).

We leave C.jar on the classpath so that B.jar may access it.

13.3.2 Bottom-up

While modularizing an app using the bottom-up approach, you need to convert lower level libraries first. Thus, bottom up approach is possible only when the dependencies are modularized already.

Effectively, when bottom-up migration is complete, every class/package of an application is put on the module-path. Nothing is left on the classpath.