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.
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
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.

```
// 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.
LocalDateTime atDate(LocalDate date)

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

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

// returns copy
LocalTime withHour(int hour)
LocalTime withMinute(int minute) //etc.
```

1.1.3 LocalDateTime

```
// 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.

```
// 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)
```

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)
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()
```

1.2 String and StringBuilder

1.2.1 String

```
// 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 if missing at end.
// indent(0) does not change the indentation, but still adds a
    normalizing line break
System.out.println(phrase.indent(0).length());

// 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");

// format string
var quotes = """
\"The Quotes that Could\" // could remove both backslashes
\"\"\"                   // could remove 2 backslashes
""";

// 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)
```

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 or double.

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

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:

```
Boolean.valueOf("8").booleanValue() // false
Character.valueOf('x').byteValue(); // does not compile
Double.valueOf("9_.3").byteValue(); // NumberFormatException
Long.valueOf(128).byteValue(); // - 128
```

2 Operators

2.1 Kinds

2.1.1 Logical Operators

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

2.1.2 Bitwise Operators: Logical

```
&
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
```

The bitwise NOT or complement operator is equivalent to negation of each bit in the input value. This will result in a negative number one smaller, i.e., obtain -x-1 from x.

Steps: First we need to find its 2's complement, and then convert the resultant binary number into a decimal number.

1. write 6 in binary: 0000 0110
2. take complement: 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

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

To 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!

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

2.1.3 Bitwise operators: arithmetic

```
<< // shift left (signed)
>> shift right (signed)
>> shift right (unsigned)

12 << 2: 48 // *2^n
12 >> 2: 3 // 1100 -> 0011 (pos.: fill with 0)
-12 >> 2: -3 //                (neg..: fill with 1)
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	^	
Logical inclusive OR	|	
Conditional OR	||	
Ternary operator	e1 ? e2 : e3	Right-to-left
Assignment operators	=, +=, -=, *=, /=, %=, &=, ^, |, <<= , >>=, >>>=	Right-to-left
Ternary operator	e1 ? e2 : e3	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 Variable declarations and initialization topics

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

3.3 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 -¿ no newline!
- Space at end of line is ignored
- \s yields two spaces
- \n yields additional line break

Example:

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

3.4 var

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

var cannot be used in a multiple-variable assignment.

3.5 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.

3.6 switch

3.6.1 General

- Supports: 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

3.6.2 Statement

- The value of a case 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;
}}
```

3.6.3 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";}
case 30 -> "Kelly";
default -> "Unassigned";
```

3.7 for (x : y)

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

3.8 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
```

3.9 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 Mixed notes on functions

4.2.1 compose()

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

5 Classes, Interfaces, Records and Enums

5.1 Classes

5.1.1 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.

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

5.1.2 Sealed Classes

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

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

Parent and child must be in same package. If they are in same file or the extension is nested, no permits are needed.

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

Note:

- We can have sealed interfaces (permitting both extensions and implementations). - 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.

```
        // extends clause missing; it is Friendly that does not
            compile!
        sealed class Friendly extends Mandrill permits Silly {}
        final class Silly {}
```

5.1.3 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.

5.2 Interfaces

Variables are always publics, static, final.

Methods can be either *default* (these are always public), static (always public), private, or private and static. There are no protected members.

Non-static metho47.3 concatenation: 47.4 relativize():

47.5 normalize

47.6 : Resolves symbolic links. Normalizes path. Interacts with the file system - throws exception if not exists! - we have a symbolic link from /zebra to /horse. - current working directory is /horse/schedule ds with a body must be explicitly marked *private* or *default*.

Note:

- There is no modifier that can prevent a default method from being overridden in a class implementing an interface!

- If implementing two interfaces with conflicting signatures, *default* methods have to override it explicitly. They can then access the inherited ones by calling Interface.super.method().

- *Static* methods are only accessible with a qualifier.

5.3 Records

Minimal example:

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

Records may optionally have constructors.

A (short) constructor (at most one):

```
public Crane { // no parens
    if (numberEggs < 0) throw new IllegalArgumentException();
    name = name.toUpperCase();
    // long form is automatically called here
}
```

A long constructor:

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

Constructors may be overloaded:

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

They can also implement interfaces:

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

Overriding a method:

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

Records may be package access or public. They may contain methods, nested classes, interfaces, annotations, enums, and other records.

Records can't be subclassed, since they are implicitly final.

Records *cannot* declare instance *variables* (as opposed to instance *methods*) or instance *initializers*.

5.4 Enums

Enums can have constructors, methods, and fields. Example:

```
enum Flavors {
    VANILLA, CHOCOLATE, STRAWBERRY;
    static final Flavors DEFAULT = STRAWBERRY;
}
```

Constructors are implicitly private. Example:

```
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; }
}
```

5.5 Overriding and overloading

5.5.1 Overloading

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

```
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";
}
```

5.5.2 Overriding

Overridden and hidden methods can only have covariant return types. This also applies to implementing abstract methods.

When a parent method is private, no overriding takes place (so the child can do what it wants).

Note:

- Overriding replaces the method *regardless of the reference type*. - There is no overriding of instance *variables*! Instance variables are always determined based on the reference type!

5.6 Class loading and initialization

5.6.1 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.

5.6.2 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 any instance variable is final, it must be initialized.

5.7 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

6 Streams

6.1 Stream characteristics

6.1.1 Ordered

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);
```

6.2 Grouping/Collecting

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

6.2.1 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);
```

6.2.2 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]
```

7 IO

7.1 java.io

7.1.1 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
```

7.1.2 Abstract base classes

InputStream, OutputStream, Reader, Writer.

Beware: these are abstract classes, not interfaces!

7.1.3 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),...

7.1.4 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!

7.1.5 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()
```

7.1.6 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
}
```

7.2 java.nio

7.2.1 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");
```

7.2.2 Conversion to or back from java.io.File

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

7.2.3 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");
// /turkey/food
```

7.2.4 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)
```

7.2.5 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.

7.2.6 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
```

7.2.7 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
```

7.2.8 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);

// 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);
```

7.2.9 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.

7.2.10 Reading files

```
// 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);
```

7.2.11 java.nio.Files methods

e.g.,

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

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

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

7.2.12 Using views for attribute retrieval

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

```
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:

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

7.2.13 Common method arguments

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

8 Exception handling

8.1 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.

```
// 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.

9 Internationalization

9.1 Resource bundles

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

10 Concurrency

10.1 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.

10.2 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();
```

10.3 Interface Callable

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

10.4 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());
        }
```

10.5 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!