

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
```

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(): normalizes the output by adding a line break to
// the end
// 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 Math methods

```

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

```

```
Math.pow() // double → double
```

1.3.2 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 switch

3.1.1 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

3.1.2 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;  
    }  
}
```

3.1.3 Expression

- Can omit default clause in when either all the values of an enum are covered or no value is returned

3.2 for (x : y)

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

3.3 Flow Scoping

```
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
```

3.4 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 Objects, Interfaces, Records and Enums

4.1 Objects

4.2 Interfaces

4.3 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*.

4.4 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; }
}

```

```

// to obtain, e.g.

```

```

// instance methods, e.g.

```

```
// to obtain, e.g.
```

```
// instance methods, e.g.
```

```
// to obtain, e.g.
```

```
// instance methods, e.g.
```

```
// to obtain, e.g.
```

```
// instance methods, e.g.
```