1. **Operators**: special symbol ls that perform specific operations on 2 or 3 **operands** and return a result.
2. **Operator precedence**: operators with higher precedence are evaluated before those with lower precedence, while those of equal precedence in the same expression are evaluated.
   1. **Prefix and Postfix** (++x, x++, etc.)
   2. **Unary** (++x –x +x –x ~!): only have 1 operator.
      1. + or –: positive/negative.
      2. ++ or --: increment/decrement.
      3. ! : logical complement; inverts Boolean value.
   3. **Arithmetic**
      1. **Multiplicative** (\* / %)
      2. **Additive** (+ -)
         1. C: concatenation of strings requires use of stringcat()
         2. Java: concatenation of strings can use +.
   4. **Bitwise shift** (<< >> >>>)
   5. **Equality** (== !=) and **Relational** (< > <= >= instanceof)
      1. C: comparison of strings by ASCII requires use of stringcmp()
      2. Java: string comparisons can be made.
         1. Letter cses
            1. Use “.” operator to view methods
            2. Ignore case, all upper/lowercase, etc.
   6. **Bitwise operators** [left 🡪 right]
      1. AND (&)
      2. XOR (^)
      3. OR (|)
   7. **Logical**
      1. **Conditional**
         1. AND (&&)
         2. OR (||)
         3. NOT (!)
         4. **Short-circuiting behavior**: second operand is only evaluated if needed.
      2. **Ternary** (?:): shorthand for if-then-else statement, with 3 operands.
         1. Read as…
            1. if the result of the condition is true

result coded between ? and : is executed.

* + - * 1. else

result coded between : and ; is executed

* 1. **Assignments** and **compound assignments** (= += -= \*= /= %=, etc…) [right 🡪 left]
  2. **instanceof**
     1. Compares object to specified type to test if object is…
        1. Instance of a class
        2. Instance of subclass
        3. Instance of a class that implements a particular interface.
     2. Using an inner class
        1. In the outer class, create instances of inner class.
        2. Declaration is in the main() method of class.
        3. Defining an inner class
           1. **Inner classes**: classes defined within a class.

Private to the class they are defined in.

* + - * 1. Example

//defining inner class

static class Parent {}

static class Child extends Parent implements MyInterface {}

interface MyInterface {}

* + 1. Checking instanceof: checks each object.
       1. Class Child extended class Parent. Therefore…
          1. obj2 (of class Child) is an instanceof extended class Parent.

Parent obj1 = new Parent();

* + - * 1. obj2 is an instanceof inner class Child.

Used Child() constructor.

* + - * 1. obj2 is an instanceof inner interface MyInterface because the inner class Child implemented it.
      1. Example
         1. System.out.printIn(“obj 1 instanceof Parent: ” + (obj1 instanceof Parent));

1. **Action listener**: listens for where user is clicked.
2. I/O
   1. Standard I/O: 2 systems that don’t make much of a difference to humans.
      1. Common methods
         1. System.out.print(): no line break
         2. System.out.printIn(): includes line break.
      2. **Unbuffered Streams**: each read/write request is handled directly by underlying OS.
         1. Less efficient system
         2. Request triggers disk access, network activity, or other expensive operations.
         3. **Byte streams**: performs I/O of 8-bit bytes.
            1. All byte streams are descended from classes InputStream and OutputStream.
            2. Always close streams when no longer needed.
            3. Error: file cannot be opened.

Stream variable never changes from initial NULL.

Ensure that each stream variable contains an object reference before invoking close.

* + - * 1. **Try-catch block**: allows handling of exceptions in file I/O to prevent program crashing.

try {

in = new FileInputStream (“… .txt”);

out = new FileOutputStream (“… .txt”);

int c;

while ((c = in.read()) != -1) {

out.write(c);

}

}

catch (Exception ex) {

System.out.printIn(“Error opening files”);

}

* + - 1. **Character stream I/O**: translates internal format to local character set (usually 8-bit ASCII or 16-bit chars).
         1. Descended from classes Reader and Writer.
      2. **Line-based I/O**: based on a **line**: string of characters with a **line terminator**.
         1. **Line terminators** (on text files)

carriage-return/line-freed sequence (“\r\n”)

single carriage-return (“\r”)

single line-feed (“\n”)

* + 1. **Buffered Streams**: read data from **buffer** (memory area).
       1. Write data to a buffer.
          1. inputStream: new BufferedReader(new FileReader(“text.txt”));
          2. outputStream: new BufferedWriter (new FileWriter(“output.txt”));
       2. 4 Classes used to wrap unbuffered streams
          1. BufferedInputStream and BufferedOutputStream: create buffered byte streams.
          2. BufferedReader and BufferedWriter: create buffered character streams.
       3. **Flushing**: write out a buffer at critical points, without waiting for it to fill.
          1. **autoflush**: specified by optional constructor argument; supported by some buffered output classes.
          2. **flush**: manual method of flushing.
       4. **Scanning**
          1. Scanner: object type useful for breaking down formatted input into tokens and translating tokens according to their data type.

Default: uses white space to separate tokens.

NOT a stream, but close to indicate being done with its underlying stream.

useDelimiter(): used to specify a different token separator with a regular expression.

Treated as simple string values

Also supports

Primitive types (not char)

BigInteger

BigDecimal

Numeric values can use 1000s separators correctly.

Separators and decimal symbols are locale-specific.

Not all scanning would work in all locales without specifying US locale.

* + - * 1. Code

Scanner s = null;

try {

s = new Scanner(new BufferedReader(new FileReader(“Scan.txt”)));

while (s.hasNext()){

System.out.println(s.next());

}

} finally {

if (s != null) {

s.close();

}

}

* + - * 1. Formatting

Stream objects are instanceof…

PrintWritter: character stream class.

PrintStream: byte stream class (most common classes used)

System.out

System.err

print and println

Format individual values in standard way.

Output a single value after converting value using appropriate toString method.

format: method which formats almost any number of values based on a *format string*, with options for more precise formatting.

**Format specifiers**: embedded in static text of a format string.

Specifiers

d: integer

f: floating point

n: line terminator (platform-specific)

x: integer as HEX

s: any value as a string

tB: integer as locale-specific month name

Many conversions

**Precision**

float: mathematical precision.

s: max width of formatted value; right-truncated if needed.

**Width**: minimum width of formatted value; left-padded with blanks.

**Flags**: additional formatting options

+: always have (+)

0: 0 is padding character

-: pad on right

,: format number with locale-specific thousands separators

Some flags cannot be used with certain other flags or conversions.

**Argument Index**: specify explicitly a designated argument, or < to match same argument as previous specifier.

* 1. File I/O: stores and organizes files on some form of media in a retrievable manner.
     1. Hierarchical structure: tree structure with root node at the top and directories (*folders*).
     2. Path
        1. Absolute: contains root element and complete directory list.
        2. Relative: needs to be combined with another path in order to access a file.
     3. Why Files?
        1. Variables and arrays store data temporarily; need a method to make data persistent.
           1. Files: sequential stream of bytes.
           2. Databases
        2. Methodology for identifying end of a file
           1. End-of-file
           2. Total byte count
     4. java.io Package
     5. JFileChooser: displays dialog to allow user to select a file or directory.
     6. **Relative pathing**: “..” used within string name to manipulate files in directories, without knowing the absolute directory (from the root).
        1. new String(“../data/BoggleData.txt”)
     7. Interface
        1. Cannot have instance variables
        2. Can have constant
           1. Java naming conventions: capital letters.