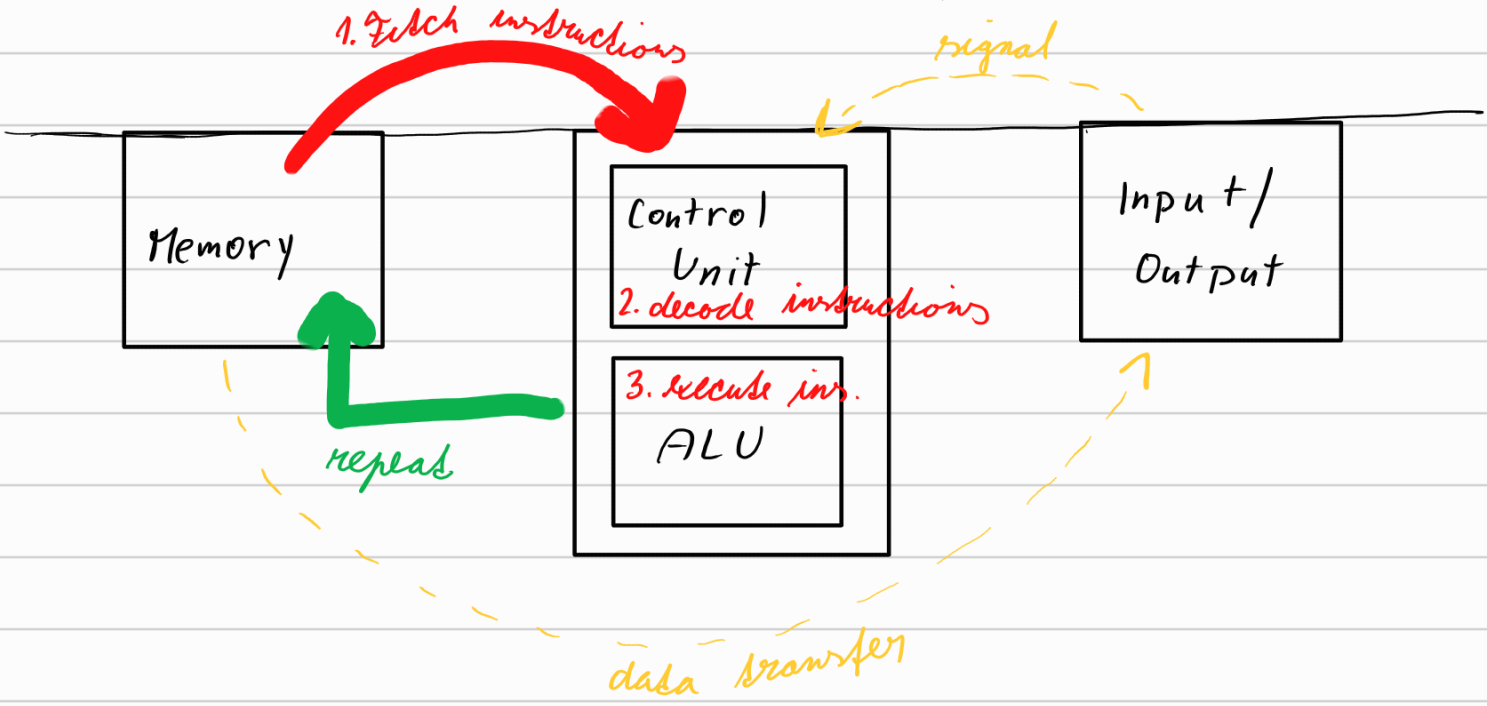
Computer = a machine that reads and executes programmable code (source code)

Data types = classifications that identify one or more types of (variables and objects) data that perform tasks
- primitive = boolean, char, byte, short, int, long, float, double
- non-primitive = String, array, arraylist, ...

(float and double should NOT be used for EXACT values, because of the LOSS of precision)

Von Neumann Architecture

- CPU → heart of the PC, performs program control and data processing → performs arithmetic and logical operations and fetches data from the memory unit



Algorithm = step-by-step problem solving procedure, set of instructions used by machines to solve some problems in a finite number of steps
= always has an input as parameters, methods to calculate solution and output

to calculate solution and output

Execution Cycle

1. memory unit fetches instructions
2. the Control Unit decodes instructions → becomes readable to machine
3. the ALU executes the instructions as operations → stored back in Memory Unit

Assembly Language

= human readable representation of machine language which differs for each specific processor → language that machine can interpret as a set of instructions via an assembler

Assembler = abstract program that converts human readable assembly language to machine language.
= reconstructs a source file written in Java assembly language into a readable, executable Java class file

Assembly Programming = complex, requires microscopic view of tasks, data movement must be managed manually and is machine specific

Programming Environment = IDE are used to write, test and debug programs

High-Level Programming Languages

= more complex human-understandable language used in source code, where each language statement corresponds to multiple specific machine instructions

{assembly}

load x, add y, store z, halt

{High-Level Prog. lang.}

$z = x + y$

Source code = collection of computer files that contain HLP statements
= can be compiled and executed by using assembler followed by a compiler

Program Compilation = process of converting a high-lvl, human-interpreted and sophisticated language program into machine-readable language
= the program is executed at once before it is executed by compiler

Program Interpretation = process of executing the program by an interpreter (virtual machine)

Interpreter = a computer program that converts all code line-by-line into machine code during the program's execution
= this machine code generated is called BYTE CODE

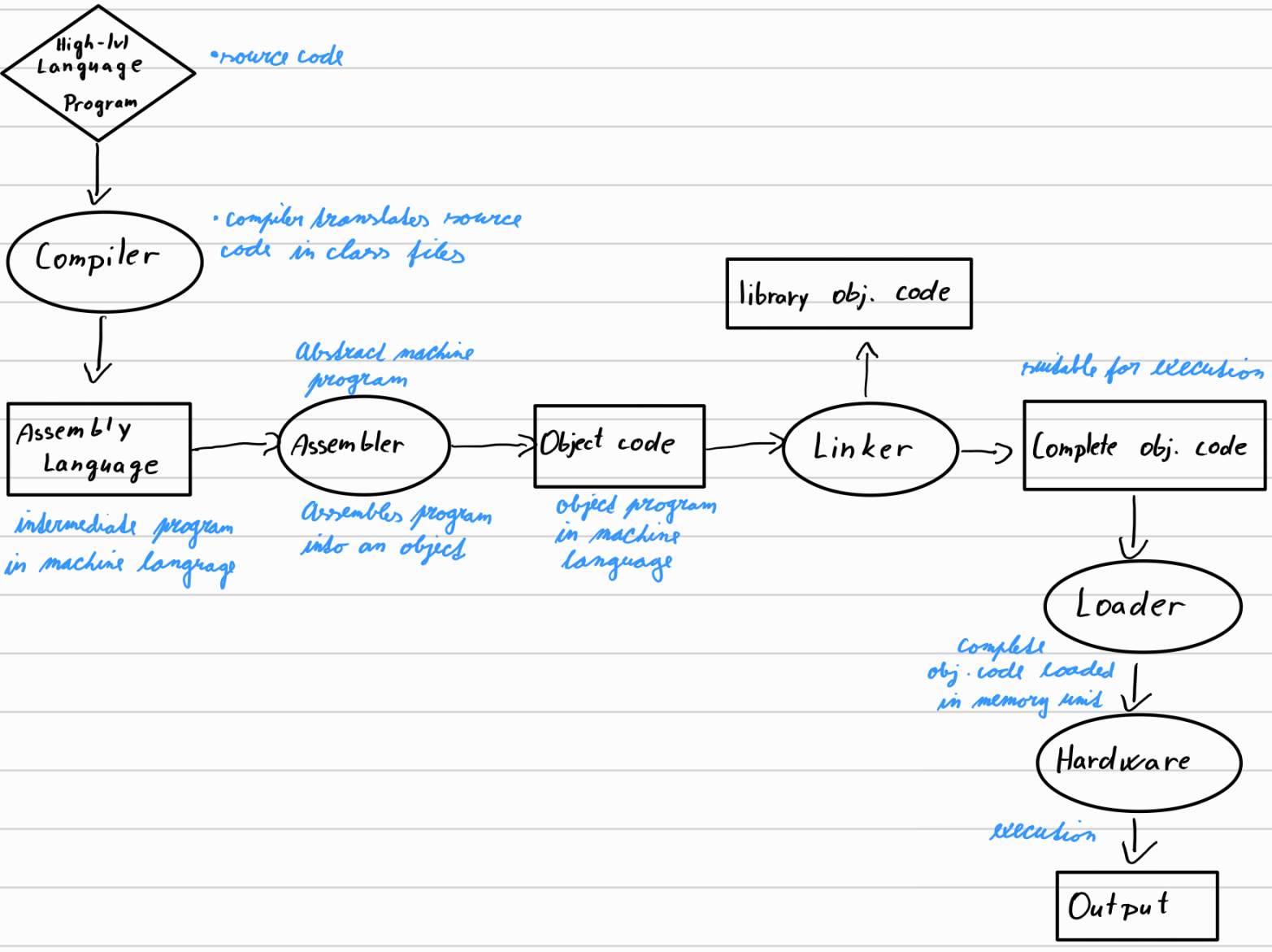
Class file = a file that contains a set of instructions for the Java Virtual Machine (JVM)

Linker = a system program that brings together separately compiled modules of program into a language program valid for execution

Compiler = program that reads high-lvl language program, in form of source code, and translated it into machine-readable language, in form of class files containing a set of sophisticated instructions

that are to be executed by the CU of a PC

The Compilation and Execution Process



Variables = data values use to share information allocated in memory unit : memory location that associates/allows linking of name containing some value, to reference the memory location → symbol references memory location, fetches its value

= use to store information for a computer to refer or modify based on the set of instructions issued

= 4 properties: name, type, value, lifetime ({} blocks)

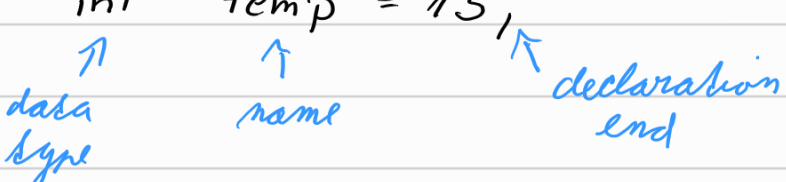
Variable Name

- Rules: → all letters and "_" can be used

→ 1st char. cannot be number
→ case sensitive
→ name should be informative
→ camel Case (start with lowercase)

Variable Declaration

- so we a variable, we must declare it a initialize it
- eg. `int temp;` (declared but NOT initialized)
  `int temp = 15;` (declared and initialized)
- we must specify data type:



- variables within methods must always be initialized

Variable Expressions = syntactically correct combination of variables, constants, operations, method invocations and values that evaluate to a single value

Expression Types

- Type widening → most expressive input data type will be an output data type
  (double) /(int) => (double)
- Type narrowing → conversion of more expressive type of value to a less expressive type via a manual command
  (int) <double> = <int>

= the variable type has to always match the expression type (type widening is automatic)

Casting = assigning a value of one basic type to

another type

- Widening Casting = automatic conversion of a smaller type to a larger type:
  byte → short → char → int → long → float → double
- Narrowing Casting = manually converting a larger type to a smaller type
  (int) in = 5.28; (will results in in = 5)

"final" variable = type of variable that allows use of name for a memory location with a fixed value
= "final" is use to fix a variables' value in Java to be a constant (value cannot be changed by program once initialized)
= can help to remove errors (magic numbers) if value is accidently modified
* (magic numbers = value that appears in code without explanation"

- eg.
  final int PRICE = 30;

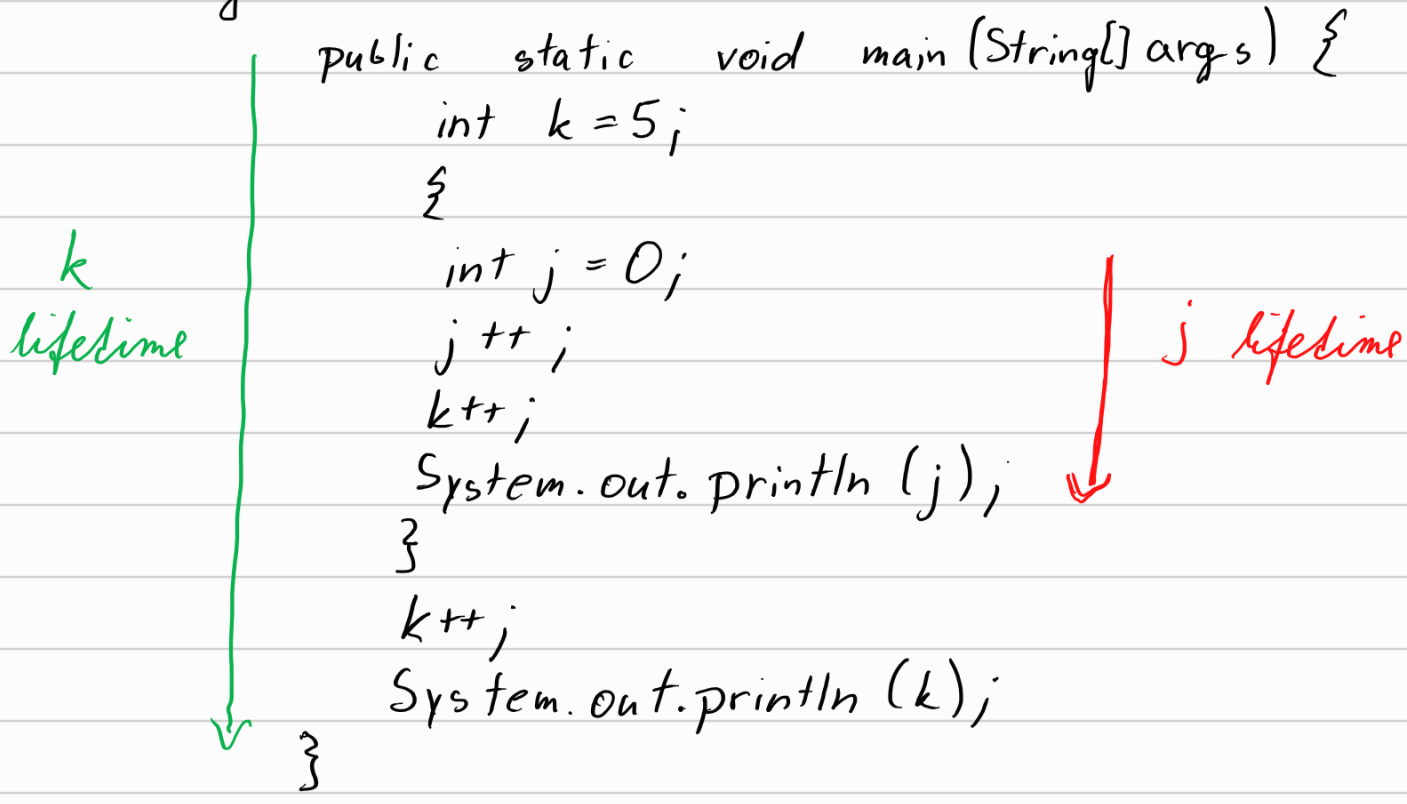
Variable lifetime and Variable Scope

= different types of variables store specific values in different locations in program, where some programming languages require variables to be allocated
= variable scope represents a part of code where the variable is visible (accessible). This is directly linked with the concept of "local" and "global" variables. The scope of variable starts with its declaration and finishes with the end of the block (loop, method...) where it is declared (usually represented by {})
= variable lifetime is for how long the variable is

accessible during the execution of the code (from its declaration until it is destroyed)

= when writing code it's important to know the scope of each variable in order to be sure that you can access to them (read and write). It also allows us to re-use variables' names, as long as their scope does not overlap

= eg.



Types of Variables (depending on scope)

- instance variable - a variable declared inside a class and outside all methods and blocks, where the scope is throughout the entire class except for static methods and where lifetime holds until object stays in the memory
- class / global variable - a static keyword, variable, declared inside class and outside of all method blocks, where term 'static' is used to share the same variable or method of a given class, and the variable belongs to the class itself rather then the instance
- local variable - a variable that is not declared as

an instance of a class, but rather methods, loops, if-statements bordered by blocks
(parameters in a method are a special case of local variables)

Methods = functions that are used to prevent duplication of code, where each function executes a set of instructions
= method has: input, name, output, body of code
= like variables they must have a data type (except void - not returning anything)
= executes a block of code with parameters to solve some code used elsewhere
(= Java doesn't care about name of method or parameters or variables, as long as they match up size-wise.)
(= the order of methods in Java doesn't matter)

The main method:

```
public static void main (String[] args){ }
```

- public = access modifier where Java runtime executes method
- static = stores the class into memory via JVM and calls main method
- main = a special keyword name → when executed this is the 1st observed method
- void = doesn't return anything
  = the only possible way to extract data from a void method is via print or special boolean method
  = if we put return into void method we will get

if we put return into void method we will get
2 errors: "compilation failed" and
"incompatible types: unexpected return value"

Declaring a Method

- define a type of class (public, private)
- define a data type of method (void or not)
- name a method
- define parameters

(in Java the program always expects the main method first)

Return type

- return <return.expression> used only for getter methods
- return type ends the execution of the method and retrieves the last uploaded expression as the output
- the expression type must always match to the return type
- in Java, the return method will terminate and exit the method

Parameters

<type> <method name> (<type> <name>) { }

parameters

- inputs of any method, an expected value required for method to compute the correct output
- treated as local variables

Method overloading = occurance in the code, where multiple called methods have the same name, but different parameter counts or types

(method is identified by the unique combination of the name and its parameters)

eg.

```
public static int add(int x, int y) {
    return x+y;
}

public static int add(double x, double y){
    return (int) (x+y);
}
```

Calling a Method = will execute the method body and allow it to retrieve outputs

- the factual parameters must match method type parameters during method definition
- the values of the method or parameters are copied via pass-by-value

Pass-by-value

- Java is a language program that utilizes this feature
- objects handles defined as references are copied and thus passed by value
- if variable is passed onto a method as a parameter, method receives the copy of the parameter's value (if method changes the value of the copy received from parameter → change is NOT reflected on original value = copy of values doesn't affect real objects value)
- Arrays and objects are pass-by-reference, thus pass-by-value is passing a copy of reference
- eg.

```
public static void main(String[] args) {
    int num = 1;
    changeValue(num);
```

→ assigns copy of num to changeValue()

```
        System.out.println(num);
    }
    public static void changeValue(int i){
        i=2;
    }
```
* Output: 1

Pass-by-value:

```
b=10;
a=5;
b=5;
print(a)*5;
print(b)*10;
```

Pass-by-reference

```
b=10;
a=5;
b=5;
print(a)*5;
print(b)*10;
```

Control flow = the order of method calls, instructions and statements are executed or evaluated when program is running

eg.

```
var x=1;
if(x==1) {
    window.alert("x is equal 1");
}
else{
    window.alert("x is not equal to 1");
}
```

control flow statement