CS1 Exam Notes: Andrew Gold

Start from the beginning:

Von Neumann Architecture:

- Control Unit } CPU
- Arithmetic Unit }
- Memory
- Input/Output interface

- stored the program concept, unable to be changed easily.
- sequential execution of instructions

(1) Control Unit
(2) Arithmetic Unit
(3) Memory
(4) In/Out interface
- program set stored
- sequential execution

Execution Cycle:

1) Fetch instruction
2) Decode instruction
3) Execute instruction

(1) Fetch instruct
(2) Decode instr
(3) Execute

Memory:

- Stores and retrieves instructions, data,
- Consists of circuits that represent cells that are capable of storing N bits.
  - Each cell has an address (such as 1011) and content (N-bit architecture, 8, 16, 32, 64 bit)

Internal Memory:

- RAM
- ROM

- each cell has a unique address, accessed in a few nanoseconds (1 nsec = 10^{-9} sec)

External Memory:

- USB "Flash Drives" } Direct Access
- SSD's }
- Hard disks (HDD) } Pseudo-Direct Access
- Optical disks }

Binary Representation: Ones and Zeroes

- Information is stored using voltage levels
- Using decimals requires 10 distinct levels
- Much cheaper to use 2 (binary) but requires more components.

1·000 bytes = 10^3 bytes
1·000·000 = 10^6

bit :	0 or 1		
1 byte :	8 bits		
1 KB :	$2^{\cancel{10}\,3}$ (1024) bytes		
1 MB :	$2^{\cancel{20}\,6}$ bytes		
1 GB :	$2^{\cancel{30}\,9}$ bytes		
1 TB :	$2^{\cancel{40}\,12}$ bytes	1·000·000·000·000	

Leftmost bit represents the sign: 0 (+) and 1 (-)

0	1	0	1	0	1	1	0		$= +86_{10}$
1	1	0	1	0	1	1	0		$= -86_{10}$

Arithmetic Overflow:

- when the arithmetic result requires more than the available number of bits.

Text in Binary:

- ASCII uses 8 bits (1 byte per char)
- UNICODE uses 16 bits (2 bytes per char)

Machine Language:

- Language used for instructions inside the computer:
  - input / output
  - moving data between RAM and Registers
  - Arithmetic and Logic
  - Comparisons and Conditional Outcomes

machine-specific programming language

☆ **Assembler:** A human-readable representation of machine language. Can differ for each specific processor

- Programming assembler instructions is extremely tedious and mundane.
  - Machine-Specific
  - Manual management of every step
  - Microscopic view of tasks
- only used for ultra-high performance requirements of small subroutines.

High-Level Programming:

- each statement can correspond to many machine instructions
- Closer to "natural language" descriptions.

Assembly: → High-Level Langaage

```
load x          z = x+y
add y
store z
halt
```

Source Code: Computer files containing high-level programming language statements.

- Can be compiled and executed (or maybe interpreted directly)

Editors: Source code files are text files.

- contain only ASCII/UNICODE characters
- Different than eg. Word file.

- **Programming Editors:** edit text files, supply syntax highlighting.

Source Code -> Running Program:

Program Interpretation:

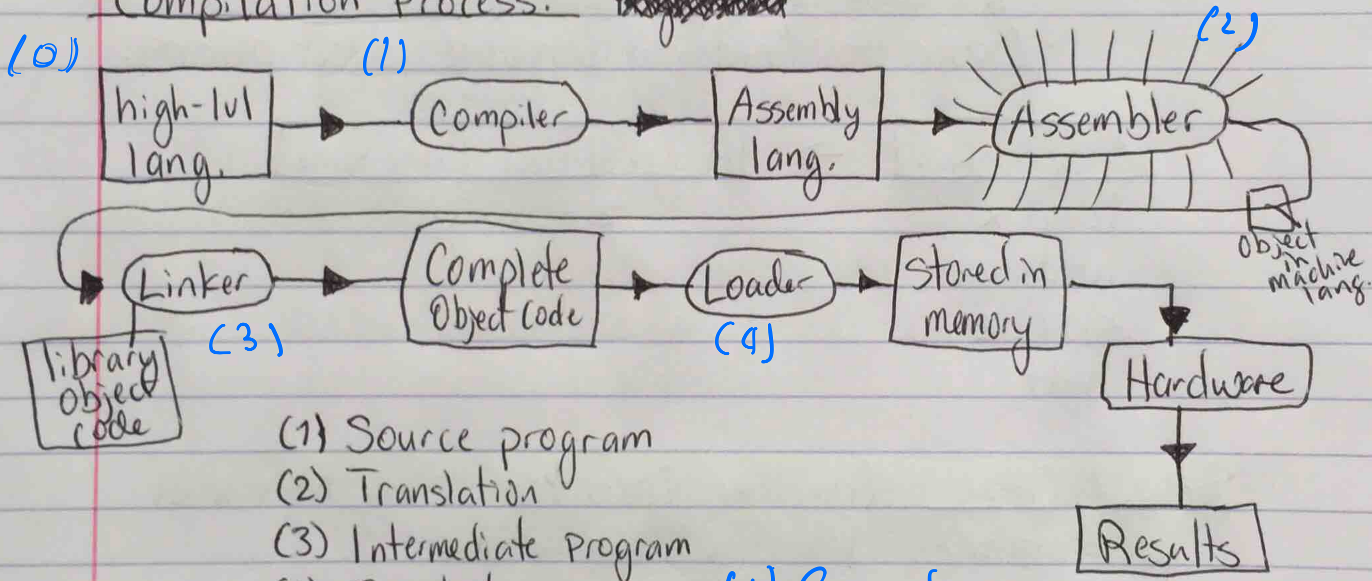
- Process of executing a program by another called "interpreter"

Program Compilation:

- Converting high-level language into machine language.
- Done by the "compiler."

- Compiler operates differently for each OS.

Compilation Process: [redacted]



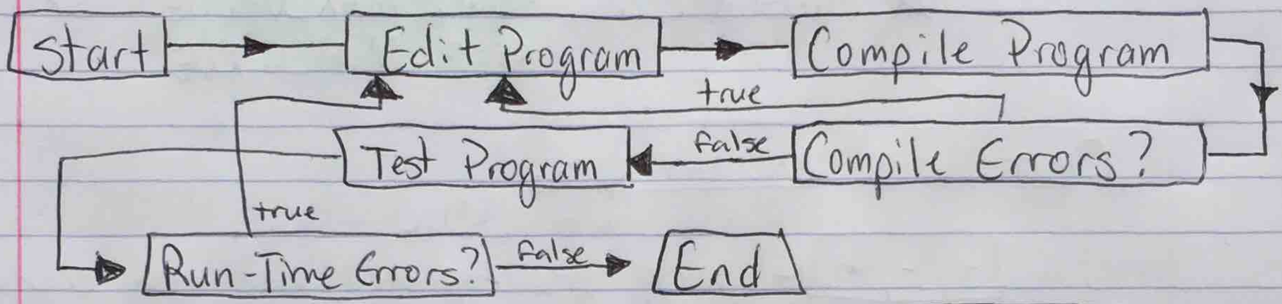
(1) Source program
(2) Translation
(3) Intermediate program
(4) Translation
(5) Object Program
(6) Loading
(7) Execution
(8) Results

(1) Compiler → source code to Assembly
(2) Assembler → Assembly to Object machine la
(3) Linker → Add libraries
(4) Loader → store in memory

Java: Object oriented, interpreted, architecture-neutral, portable, multi-purpose language made by Sun Microsystems in 1995.

Errors:
- Syntax errors are violations of the programming language.
- Logical errors cause a program to behave in an unintended manner. (human error)

Edit-Compile-Test Loop (debugging):



Java Notes:

Variables: Use a "name" or variable to reference memory locations.

Contain:
- Name
- Type
- Value (content)
- Lifetime

Rules:
- Letters, numbers, and underscore (_) can be used.
- First character may not be number.
- CASE SENSITIVE

Guidelines: Be informative, start with a lower-case letter for names, use CamelCase.

(Next Page)...

Data Types

Whole # or Integers:

- byte - 8 bits - (-128 - 127)
- short - 16 bits - (-32768 - 32767)
- Int - 32 bits - (-2,147,483,648 - 2,147,483,647)
- long - 64 bits - (too long to write - quintillions)

☆ accessible as <var>.MAX_VALUE or
<var>.MIN_VALUE

Operations on Integers:

+, -, * - work as expected, but can cause over/underflow.

/, (÷) - of two ints results in an int.

% - modulo, results in remainder of int division.

Floating Point Types: Scientific Notation

example: (double) = 13452300 = $0.134523 * 10^8$
= .134523E8

- float - 32 bits
- double - 64 bits

} very precise.

Operations work normally, but using multiple types results in most expressive type.

Boolean: 1 bit, binary, "True" or "False."

Boolean Operations on next page:

Boolean Operations: &&, ||, ! (and, or, not)

a	b	a&&b	A||b	!a	!b
T	T	T	T	F	F
T	F	F	T	F	T
F	T	F	T	T	F
F	F	F	F	T	T

others: <, >, <=, >=, ==, !=

Char Variables: set of all characters in the UNICODE table.

- Expressions always convert char values and variables to integers

☆ Strings: Not a primitive data type, rather are objects of a class.

- Represents a sequence of characters
  - "Andrew"
  - "qwerty"
  - "101.5"

"quotes" are needed!

- To use a variable, you must declare it.

  - intx= 5;
  - boolean k = False;

Expressions: Syntactically correct combinations of variables, constants, operators, and invocations, that evaluate to a single value.

Precedence: order of operators.

1) *, /, % : all are equal to (1)
2) +, − : all are equal to (2), come after (1)

(use parentheses to be structured!)

Narrowing Data Types:

- **casting operator** reduces a value of an expression to a smaller (less expressive) type.

ie: (int) (3.14 * 5) [redacted]

- Assignment (=) != Equality (==)

Constants: a fixed variable, value cannot be changed.
- "final"

eg. final int BOIL = 100;
↑
use caps

Variable Lifetime:

- local variables exist during the execution of the block in which they are created/declared.

block:
- uses braces { }
- groups statements.
- defines lifetime of variables.

Methods: Groups related statements that re-occur together in a task.

have a:
- Name
- input (parameters)
- output (return value)
- body (instructions)

$$\text{public static } \overbrace{\text{int}}^{\text{return type}}\ \overbrace{\text{sum}}^{\text{name}}\ (\underbrace{\text{int x, int y, int z}}_{\text{parameters}})$$

body{ . . . . -
. . . . - - - - -

$$\underbrace{\text{return sum;}}_{\text{return statement}}$$

Overloading: defining a method with the same name but different parameter counts or types.

Calling a Method:

- "invoking" or calling a method executes the body of the method and allows the collecting of results
- Factual parameter values have to match the type of the method definition (or be able to widen)
- Pass-by-Value: values are copied.
- Collecting the result is done by assigning the method expression to a suited variable.

Pass-by-Value: Copies an value of a variable, and applies any changes to that copy only. (copy is "disconnected")

Local Variable:
- Can be defined inside method
- Last only as long as execution (1x) or the surrounding block { }

☆ Side Effects: When something is unexpectedly altered.
- Not always bad, but must be aware of.
- ex: writing to Sys.out.Prtln changes the terminal, could have consequences.

☆ Constructor: A special method
- Creates a new instance of a class
  - possibility of some initialization!
- Has no return type, returns an instance of the class
- name must = name of class
- can be overloaded! Different constructors use the same name but with diffent parameters.

Floating Point Comparisons:

Solution: allow some error...
for doubles, use ~~1E-14~~ (10^{-14})

Arrays: a collection of data items, of all the same variable type, packaged under a single identifier.

- Arrays are objects! -

- must be declared AND created!

example:
```
(1) int[] numbers;
(2) numbers = new int[10];
              or
    numbers = new int[length*2-1]
```

if length is defined!

- Arrays use an index to search through the data. array[0] is the first position in the array. array[array.length-1] is the last position.

- Arrays can be instantiated!

```
double[] ar = {1.0, 2.0, 3.0, 4.0};
```

- Arrays of objects: each index holds a reference to an object.

Array length issues: arrays cannot have their length altered after instantiation.

(next page)

… to avoid array issues:

(1) create an array larger than the reasonable expectation of size.

(2) replace with an array of correct length:

```
int[] numbers = new int [0];
int[] newnums = new int[numbers. length + 1];
    for (int i = 0; i < numbers.length; i++)
        newnums [i] = numbers [i];
numbers = new nums;                    //creates a new referance
numbers [numbers. length-1] = ...;
```

(3) replace with a larger array only when necessary:

- use an if/else condition

Updating indexes:

(1) nums [i++] / [i--]
(2) nums [++i] / [--i]

(1) uses current value, then updates.
(2) updates, then uses new value.

☆ Copying an array:

```
for (int i = 0; i < nums. length; i++)
    new nums [i] = nums[i];
```

Pass-by-Value:

- if you create a variable within a method, ie:

```
public static int doSomething (int a) {
   ...
}
        int x = ...;
        doSomething(x); // this will copy the value
//of x and "a" will also now refer to that same value.
```

- Array references can be passed as parameters:

```
public static int doSomething (int[] a) {
  ...
}
  int[] x = ...;
  doSomething(x);
```

The reference to array **x** will be copied to the same location in "a".

☆ Side Effects! - a[i] (for above array) inside method doSomething will ALSO change the content of "x".

☆ Pass-by-Value analogy:

Java is a Pass-by-Value language (as opposed to pass-by-reference).

Pass-by-Reference is like me sharing a website URL with you. We both hold the URL, and they point to the same website. Any changes to the website, both of us will see on our own computers.

BUT!

Pass-by-Value is different:

Pass-by-Value is like me going to a website and printing out on a sheet of paper what I see. Any subsequent changes to the website won't appear on the paper, and any changes you make to the paper won't change the contents of the website.

Objects: an "entity" within the program that has (can have) an internal state, and set of abilities (methods that can be called upon).

- Objects are referenced as variables
- Objects can be used as parameters, or by parameters

- **Instance Variables:** data stored within object
  - lifetime is the same as the object.

- **Data encapsulation:** Hiding or Protecting data inside the object.
  - hides complexity.
  - reduces bugs.

- Local and Parameter Variables belong to a method.
- Instance Variables belong to an object.
  - initialized to a default value, unlike local variables.

Getters and Setters: are used to provide access to variables in an object.

- Instance variables should be PRIVATE!

Implicit Parameter: The object on which the method is invoked.

Recursion: Amethod that "loops" upon itself.
must have a: Base Case (stopping criterion)
Self Invocation (working towards base case)

Tabling: Goal: Calculate every result only once.

- use an array to store results
- fill in results as computed
- constructed on first call to method
- Pass along array as an extra parameter.

Type Widening: when combining two variables in an expression, the most expressive is chosen as a resulting type.

example:

(int) vs. (double) ⇒ double
(int) vs. (float) ⇒ float
(short) vs. (long) ⇒ long
(short) vs. (int) ⇒ int
(byte) vs. (short) ⇒ short
otherwise ⇒ byte