Self-Learning Material #1:

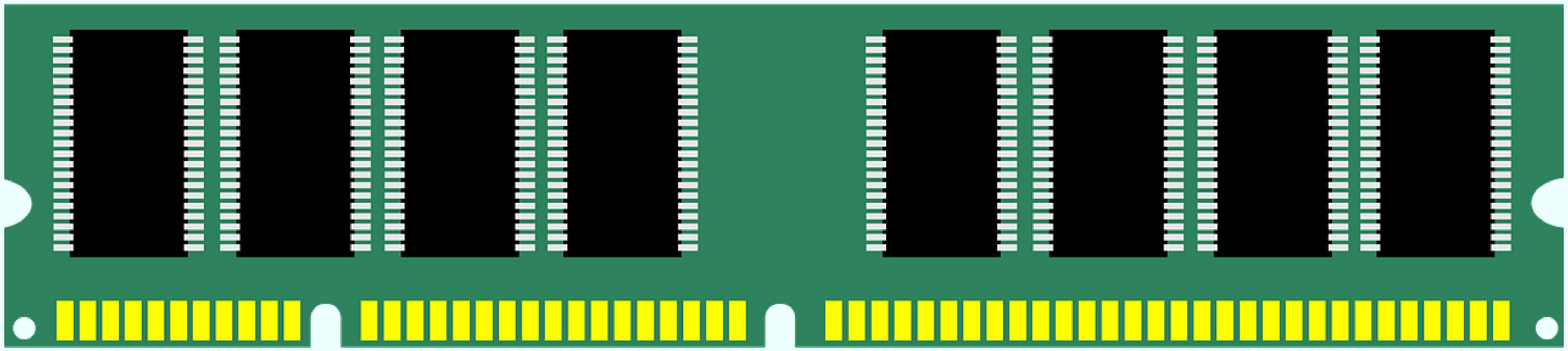
Variables and Operators

By Kevin 2022/2023

Variables

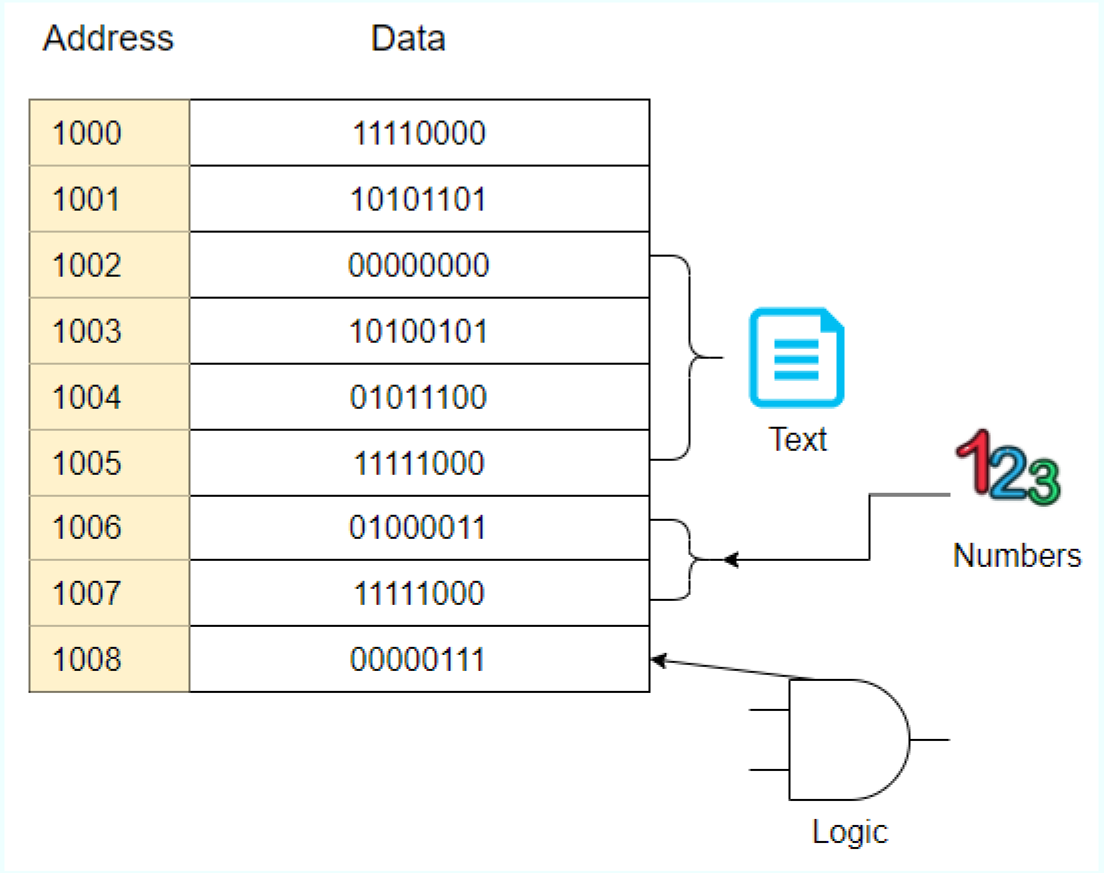
Computer Memory

- Data is digitalized and stored in a computer with a pre-defined format.
- It is a number, more precisely a binary number, for any digital contents - number/text/image/video.
- But How?



Computer Memory

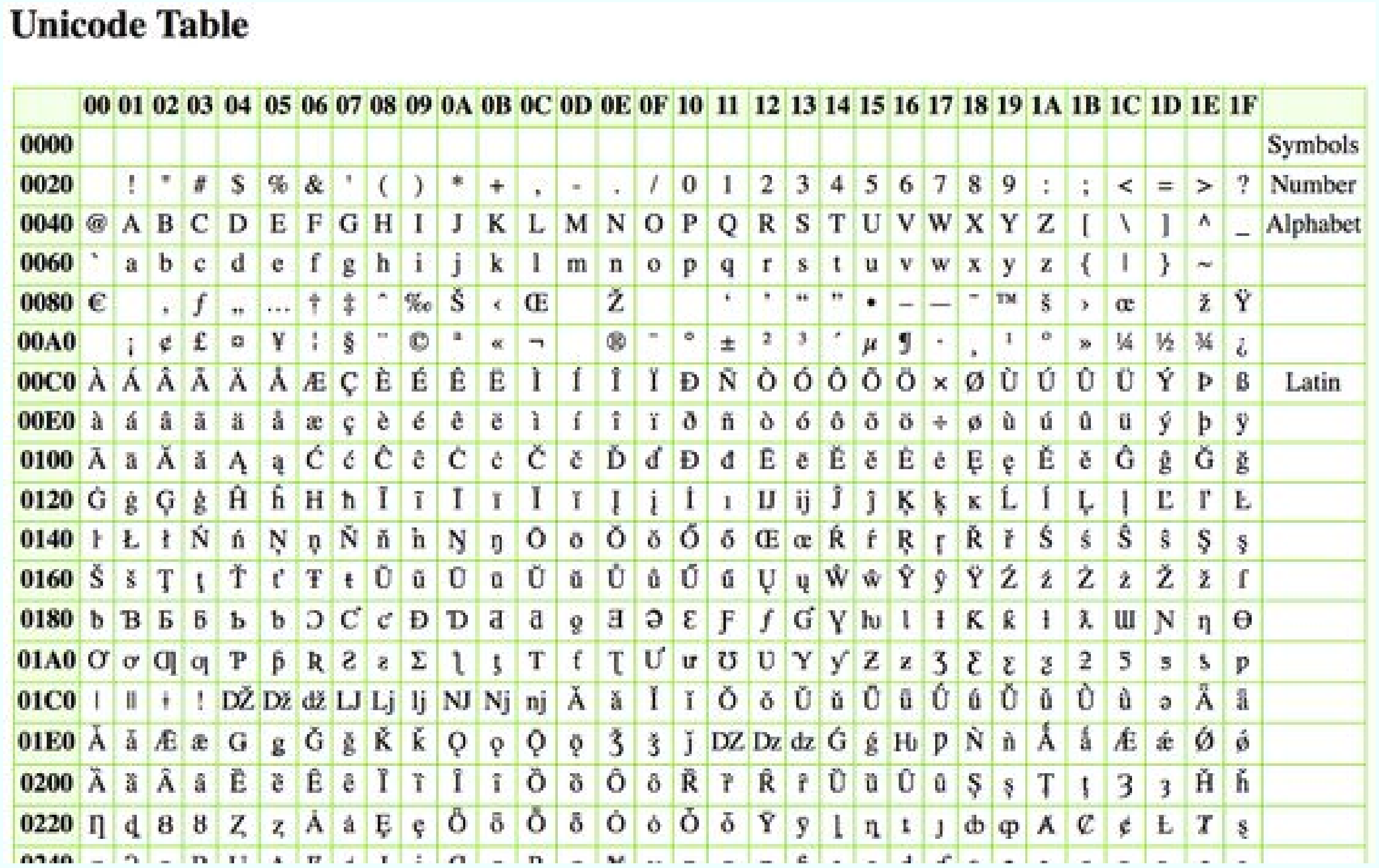
- Each binary digit is call a **bit**.
- 8 bits are grouped into a **byte**.
- Depends on the size of the data, each piece of data is stored using a number of bytes.
- Some data use more (a paragraph, images, videos...)
- Some data use less (a small integer, the logic true or false)



Address	Data
1000	11110000
1001	10101101
1002	00000000
1003	10100101
1004	01011100
1005	11111000
1006	01000011
1007	11111000
1008	00000111

Storing Text

- Text also known as **Char**acter or **String** (multiple characters)
- Every character is coded as a number (called Unicode code)
- Each Unicode character consumes two bytes.
- E.g `comp2026` is stored as `0x63`,`0x6F`, `0x6D`,`0x70`, `0x32`, `0x30`, `0x32`, `0x36`



Unicode Table

	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
0000																																	Symbols
0020		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?	Number
0040	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	Alphabet
0060	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~		
0080	€		‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ		Ž			‘	’	“	”	•	–	—	˜	™	š	›	œ		ž	Ÿ	
00A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿	
00C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß	Latin
00E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ	
0100	Ā	ā	Ă	ă	Ą	ą	Ć	ć	Ĉ	ĉ	Ċ	ċ	Č	č	Ď	ď	Đ	đ	Ē	ē	Ĕ	ĕ	Ė	ė	Ę	ę	Ě	ě	Ĝ	ĝ	Ğ	ğ	
0120	Ġ	ġ	Ģ	ģ	Ĥ	ĥ	Ħ	ħ	Ĩ	ĩ	Ī	ī	Ĭ	ĭ	Į	į	İ	ı	Ĳ	ĳ	Ĵ	ĵ	Ķ	ķ	ĸ	Ĺ	ĺ	Ļ	ļ	Ľ	ľ	Ŀ	
0140	ŀ	Ł	ł	Ń	ń	Ņ	ņ	Ň	ň	ŉ	Ŋ	ŋ	Ō	ō	Ŏ	ŏ	Ő	ő	Œ	œ	Ŕ	ŕ	Ŗ	ŗ	Ř	ř	Ś	ś	Ŝ	ŝ	Ş	ş	
0160	Š	š	Ţ	ţ	Ť	ť	Ŧ	ŧ	Ũ	ũ	Ū	ū	Ŭ	ŭ	Ů	ů	Ű	ű	Ų	ų	Ŵ	ŵ	Ŷ	ŷ	Ÿ	Ź	ź	Ż	ż	Ž	ž	ſ	
0180	ƀ	Ɓ	Ƃ	ƃ	Ƅ	ƅ	Ɔ	Ƈ	ƈ	Ɖ	Ɗ	Ƌ	ƌ	ƍ	Ǝ	Ə	Ɛ	Ƒ	ƒ	Ɠ	Ɣ	ƕ	Ɩ	Ɨ	Ƙ	ƙ	ƚ	ƛ	Ɯ	Ɲ	ƞ	Ɵ	
01A0	Ơ	ơ	Ƣ	ƣ	Ƥ	ƥ	Ʀ	Ƨ	ƨ	Ʃ	ƪ	ƫ	Ƭ	ƭ	Ʈ	Ư	ư	Ʊ	Ʋ	Ƴ	ƴ	Ƶ	ƶ	Ʒ	Ƹ	ƹ	ƺ	ƻ	Ƽ	ƽ	ƾ	ƿ	
01C0	ǀ	ǁ	ǂ	ǃ	Ǆ	ǅ	ǆ	Ǉ	ǈ	ǉ	Ǌ	ǋ	ǌ	Ǎ	ǎ	Ǐ	ǐ	Ǒ	ǒ	Ǔ	ǔ	Ǖ	ǖ	Ǘ	ǘ	Ǚ	ǚ	Ǜ	ǜ	ǝ	Ǟ	ǟ	
01E0	Ǡ	ǡ	Ǣ	ǣ	Ǥ	ǥ	Ǧ	ǧ	Ǩ	ǩ	Ǫ	ǫ	Ǭ	ǭ	Ǯ	ǯ	ǰ	Ǳ	ǲ	ǳ	Ǵ	ǵ	Ƕ	Ƿ	Ǹ	ǹ	Ǻ	ǻ	Ǽ	ǽ	Ǿ	ǿ	
0200	Ȁ	ȁ	Ȃ	ȃ	Ȅ	ȅ	Ȇ	ȇ	Ȉ	ȉ	Ȋ	ȋ	Ȍ	ȍ	Ȏ	ȏ	Ȑ	ȑ	Ȓ	ȓ	Ȕ	ȕ	Ȗ	ȗ	Ș	ș	Ț	ț	Ȝ	ȝ	Ȟ	ȟ	
0220	Ƞ	ȡ	Ȣ	ȣ	Ȥ	ȥ	Ȧ	ȧ	Ȩ	ȩ	Ȫ	ȫ	Ȭ	ȭ	Ȯ	ȯ	Ȱ	ȱ	Ȳ	ȳ	ȴ	ȵ	ȶ	ȷ	ȸ	ȹ	Ⱥ	Ȼ	ȼ	Ƚ	Ⱦ	ȿ	

Storing Numbers

- Apparently we can be more efficient when we just store numbers
- Represent a decimal number into binary format:

$$2026 = 111\ 1110\ 1010_{(2)}$$

- This requires two bytes.

0000 0111	1110 1010

- Need more bytes if we want to store larger number like 100,000,000.

👨‍🏫 💬 What is the binary format of 1013 and 4052?

Java Built-in Types

- Java has a few built-in data types
  - For manipulating numbers (integers or fractional numbers)
  - For manipulating characters
  - For manipulating booleans
- Built-in data types are also known as...
  - **Primitive data types**
  - Base types

Numbers in Java

- Integers and decimals numbers work differently.
- Java use the following numerical primitive types to store numbers

For integer

- `byte` (8 bits)
- `short` (16 bits)
- `int` (32 bits)
- `long` (64 bits) - the larger possible number can be

For decimal numbers

- `float` (32 bits)
- `double` (64 bits) - more precise a number can be

Integer

- In Java we can have integers of different sizes
- We usually use `int` to represent integers

Types	Size	Range
`byte`	8 bits	-128 to 127 (-2^7 to 2^7-1)
`short`	16 bits	-32768 to 32767 (-2^{15} to 2^{15}-1)
`int`	32 bits	-2147483648 to 2147483647 (-2^{31} to 2^{31}-1)
`long`	64 bits	-9223372036854775808 to 9223372036854775807 (-2^{63} to 2^{63}-1)

Decimal Numbers

- `float` and `double` to support different precision requirements of decimal numbers
- `double` requires more storage and more computation than `float`

Types	Size	Range
`float`	32 bits	$\pm$ 1.40129846432481707e-45 to 3.40282346638528860e+38
`double`	64 bits	$\pm$ 4.94065645841246544e-324 to 1.79769313486231570e+308

Character

- `char` is a type to represent ONE character
- `char` is 16-bits long, encoded by **unicode**
- Each char must be either: enclosed by single quotes `'a'` or represented by a number < 65535

Examples of char

- `'a'`, `'A'`
- `'$'`
- `'K'`, `'e'`, `'v'`, `'i'`, `'n'`
- `'\u6D78'` (the character `浸`)
- `'\u0060'` (the character `` ` ``)
- 100, without single quote `'` (the character `d` which has the unicode `'\u0064'`)

Boolean

- `boolean` is a type to represent the logics `true` and `false`.
- In a computer, we need only one bit to store a true/false value.
- More convenient than storing that as a char `'T'`/`'F'` or an int `0`/`1`.
- Boolean is used in condition (`if-else`) and loops.

Example:

```java
boolean enrolled = true;
boolean fail = (mark < 35);
```

```java
while (fail) {
  retake();
}
```

String

- `String` is a little different than the other type.
- `String` starts with an upper case
- `String` is used to represent any length of characters.
- Each string **must be enclosed** by a pair of double-quotes `"Kevin"`.
- `String` is also encoded by **unicode**.

Example of strings

- "Kevin"
- "Hello World!"
- "\u6D78 ~~!"

Escape Character

- Apart from using unicode encoding (e.g. `\u0060`), we can also represent certain character using escape characters `\`.

Code	Output	Example	Example Output
`\"`	" Double quote	`"I \"know\" Java"`	I "know" Java
`\\`	\ Backslash	`"True\\False"`	True\False
`\'`	' Single quote	`"I don't know"` or `"I don\'t know"`	I don't know
`\n`	new line symbol	`"So... \nNew line!"`	So... New line!
`\t`	tab symbol. Fill with spaces until tab stop	`"A\tBcd"`	A Bcd

Declaring Variables

- Declare **variables** for storing data
- Each variable must have a type (`int`, `float`, etc..)
- Computer memory would be allocated for the variable based on the size of the data type
- A variable must be **declared** before use

```
int numOfApples; // an integer
double temperature; // a double to store decimal number
short aLittleCounter; // an integer ranged from -32768 to 32767
char grade; // a character storing A, B, C, D, F
```

- You can also declare multiple variables in a single line, separated by `,`

```
int i,j; //i and j are integer
double a,b,c,d,e; //all are doubles
int a, double b; //invalid "," shd be replaced by ";"
```

Declaring Variables

- Optionally assign an initial value when declare a variable

```java
int val1 = 2026;
int val2 = 2007;
int sum = val1 + val2;
int val3; //without an initial value
val3 = 50; //assign a new value to val3
val3 = val3 - 20; //assign a new value to val3
```

Variables Naming

The naming of a variable must

- contain only letters `[a-zA-Z]`, digits `[0-9]`, and the underscore character `_`
- not begin with digits
- not be the same as reserved words (words rendered in *blue* in IntelliJ, e.g. `if`, `void`).
- Variables are **case sensitive**, i.e., `apple` is not the same as `AppLe`

Valid Variable Names	**Invalid Variable Names**
smallTree	2026COMP
BIG_HOUSE	COMP-2026
COMP2026	Kevin Wang
_2026	*do*
DO	

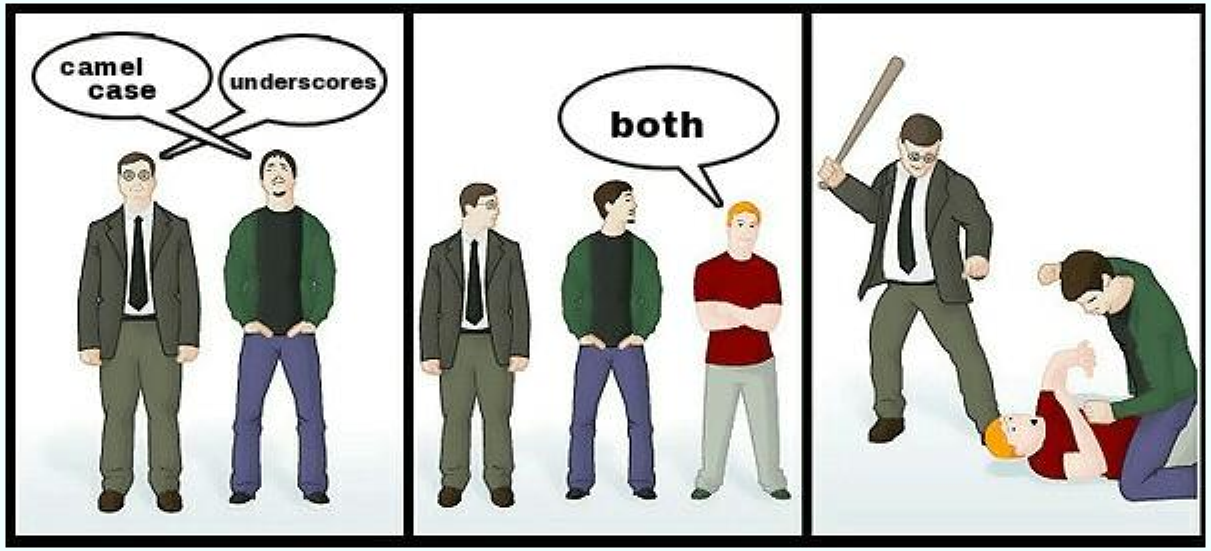
Java Reserved words (non-exhausting)

- abstract
- *assert*
- boolean
- break
- byte
- case
- catch
- char
- class
- continue
- ~~const~~
- default
- do
- double
- else
- *enum*
- *exports*
- extends
- false
- final
- finally
- float
- for
- ~~goto~~
- if
- implements
- import
- instanceof
- int
- interface
- long
- *module*
- *native*
- new
- null
- package
- private
- protected
- public
- *requires*
- *record
- return
- *sealed*
- short
- static
- *strictfp*
- super
- switch
- synchronized
- this
- throw
- throws
- *transient*
- true
- try
- *var*
- void
- *volatile*
- while
- *yield*

Words in *italic* are not taught in this course. Words that are ~~stroked~~ has no function.

Variables Naming - Not compulsory

- Variable should always starts with lower case.
- Class name should always starts with upper case.
- Constant should be all upper case, separated by underscore.
- Two different major camps in variable naming: **Camel** (`numberOfRounds`) vs **Snake** (`number_of_rounds`)



https://www.reddit.com/r/ProgrammerHumor/comments/79ww91/camelcase_vs_underscores/

Common Mistake - Declaring Variables

- A variable must be declared before use
- A variable should match with the type it is declared

```
newValuable = 5; //error! newValuable is not declared
int sum;
sum = 69.45; //error! sum is declared as int
```

- You must assign a value to a variable before referencing it.

```
int val;
int sum = val + 2; //error, the value of val is unknown
```

Summary

- Data types for numbers: `int`, `float`, `double`, `short`, `byte`
- Data type for character: `char`
- Data type for logic: `boolean`
- Class for strings: `String`
- Variable naming rules and conventions.

Operators

Operators

Java has a few types of operators

- Arithmetic Operators: `+` `-` `*` `/` `%` `++` `--` `()`
- Assignment Operators: `=` `+=` `-=` `*=` `/*` `%=`
- Relational Operators: `>=` `==` `<=` `>` `<` `!=`
- Logical Operators: `!` `||` `&&`
- Bitwise Operators: `~` `|` `&` `^` `<<` `>>` `>>>`
- Conditional Operators: `? :`

Arithmetic Operators

Operator	Meaning	Example	Result
`()`	Parentheses	`3 * (1 + 2)`	9
`-`	Negation	`-5`	-5
`*`	Multiplication	`4 * 2`	8
`/`	Decimal Division	`9.3 / 3`	3.1
`/`	Integer Division	`9 / 2`	4
`%`	Modulus/Remainder	`47 % 4`	3
`+`	Addition	`4.3 + 5`	9.3
`-`	Subtraction	`34 - 10`	24

🧑‍🏫 💬 Recommendation: Keep a space between operators except parentheses. It looks nicer! `5+3*4` ➡️ `5 + 3 * 4`

Arithmetic Operators

Examples:

- `(((10.4)))` evaluates to `10.4`
- `7 % 5` evaluates to 2
- `9 / 2` evaluates to 4
- `9 / 2 * 2` evaluates to 8
- numOne + numOne evaluates to twice the value of numOne
- `2 (3 + 4)` is invalid
- `2 / (99/100)` gives an error
- `2 ^ 3` this compiles, but it is not 8
- `2 ** 3` gives an error

Assignment Operators

- An assignment operator **assigns** a value to a variable.
- Left hand side must be a variable.

```
a = 10 + 5;
5 + 10 = a; //incorrect
5 = 10 - a; //incorrect
a - 5 = 10; //incorrect
a + b = 10; //incorrect
```

Assignment Operators

- Shorthand operators
- Note: no space between `<op>` and `=`

Operators	Meaning
`a += b;`	`a = a + b;`
`a -= b;`	`a = a - b;`
`a *= b;`	`a = a * b;`
`a /= b;`	`a = a / b;`
`a %= b;`	`a = a % b;`

Pre/Post increment/decrement

- Post-increment `i++` is a short hand of `i = i + 1`
- similarly post-decrement `i--` means `i = i - 1`
- `++` and `--` can also be placed in front of a variable (pre-increment/decrement)
- `++i` also does `i = i + 1`
- `--i` also does `i = i - 1`

Difference?

- Pre-increment: adds 1 first and use this value
- Post-increment: use this value and adds 1 later

Pre/Post-increment/decrement

Pre-increment

```java
int i = 10;
int j = ++i;
```

- adds 1 first and use `i`'s value
- `i = 11`, `j = 11`

Post-increment

```java
int i = 10;
int j = i++;
```

- use `i`'s value first and adds 1 later
- `j = 10`, `i = 11`

Pre/Post-increment/decrement

Pre-decrement

```
int i = 10;
int j = --i;
```

- minus 1 first and use `i`'s value
- `i = 9`, `j = 9`

Post-decrement

```
int i = 10;
int j = i--;
```

- use `i`'s value first and minus 1 later
- `j = 10`, `i = 9`

Pre/Post-increment/decrement

- What does this mean?

```java
int a = 5;
int b = 4;
int c = a+++b;
```

- Post-increment/decrement has a higher precedence than a pre-increment/decrement operator.
- This means

```java
int c = (a++) + b;
//c = 9, a = 6
```

Good Practices

- Theoretically speaking `i++` runs faster than `i = i + 1`
- `i++` looks nicer.
- Should not include more than one Pre/post-increment in the same line.
- Do not cascade assignment operator except for `=`. i.e.

```
int a, b, c;
a = b = c = 10; //OK
a += b = c += a; //discourage
a += b = c += b++; //can be problematic
```

Relational Operator

- Relational operators compare *any two numbers* and *generates boolean results*

Operator	Meaning
`a > b`	true if `a` is **greater than** `b`
`a >= b`	true if `a` is **greater than or equal to** `b`
`a < b`	true if `a` is **smaller than** `b`
`a <= b`	true if `a` is **smaller than or equal to** `b`
`a == b`	true if `a` is **same as** `b`
`a != b`	true if `a` is **not the same as** `b`

- The result evaluated is either `true` or `false`.

Logical Operators

- True and false logic can be combined using logical operators `&&` (and), `||` (or), `!` (not).

Operator	Meaning
`b1 && b2`	true if `b1` and `b2` are true (both `b1`, `b2` are booleans)
`b1 || b2`	true if `b1` or `b2` are true (both `b1`, `b2` are booleans)
`!b1`	true if `b1` is false

- Details of logical operators:

`b1`	`b2`	`b1 && b2`	`b1 || b2`	`!b1`
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Short Circuit of Logical Operators

- When evaluating the logical operators, there could be **short circuit**!

When evaluating `b1 && b2` (read as b1 and b2),

- if `b1` is false, the whole expression will be false anyway
- therefore `b2` will not be evaluated.

When evaluating `b1 || b2` (read as b1 or b2),

- if `b1` is true, the whole expression will be true anyway
- therefore `b2` will not be evaluated.

Example

- Try the following, true or false?

```java
boolean bool1 = (3 == 2) && (2 < 3);
boolean bool2 = (!bool1) || (5.6 >= 8);
boolean bool3 = !(bool1 && bool2);
```

- Based on the short circuit evaluation, we've got

```java
int i = 1/0; // error
boolean b1 = (3 == 2) && (1/0 == 5); // ok
boolean b2 = (3 >= 2) && (1/0 == 5); // error
boolean b3 = (3 == 2) || (1/0 == 5); // error
boolean b4 = (2 == 2) || (1/0 == 5);
```

Arithmetic Operators for char

- char is internally stored as a 16-bits integer
- We can perform +/- over char with some restrictions

```java
char num = '8';
int a = num - '0'; //a = 8 integer
```

char	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
unicode value	48	49	50	51	52	53	54	55	56	57

- `num - '0'` effectively performs 56 - 48.
- Can also be used for comparison

```java
if (input > 'a'  && input < 'z')
   System.out.println("input is lower case");
```

Arithmetic Operators for String

- We can use + over strings to combine two strings.

```java
String a = "abc";
String b = "def";
String c = "ghi";
String d = a + " " + b + c;
//d = "abc defghi"
```

- It is also possible to combine an integer with a string too

```java
String a = "abc";
int num = 10;
String d = a + num;
```

Relational Operators for String

- To compare a string against another string, do not use `==`
- Instead, we need to use `string.equals()`

```java
String s1 = "ab";
s1 = s1 + "c";
if (s1 == "abc")
  System.out.println("It is abc. But this line never shown.");
```

A correct way to do it

```java
String s1 = "ab";
s1 = s1 + "c";
if (s1.equals("abc"))
  System.out.println("It is abc. It works this time!");
```

Bitwise Operators

- Bitwise operators operates on ***bit level***.
- Assume `a = 0b0010; b = 0b0110`

Operator	Meaning	Result
`~a`	Bitwise complement operation	0b1101
`a | b`	Bitwise or operation	0b0110
`a & b`	Bitwise and operation	0b0010
`a ^ b`	Bitwise exclusive-or operation (1 if the bits are different)	0b0100
`a << n`	Bitwise left shift operation on a for n position. ($\times 2^n$)	a << 2 = 0b1000
`a >> n`	Bitwise right shift operation on a for n positions (fills the top bits with the left most bit, that is, the sign bit. $\div 2^n$)	b >> 2 = 0b0001

Bitwise Operators

The following program checks if the variable a,b,c,d contains all 1,2,3,4 each in any order.

```
int a,b,c,d;
//assign values to a,b,c,d; assume they are between 0 to 8
int result = 1 << a;          //line 1
result = result ^ (1 << b);//line 2
result = result ^ (1 << c);//line 3
result = result ^ (1 << d);//line 4
boolean is1234 = (result == 16 + 8 + 4 + 2);
```

- e.g. `a` = 4, `b` = 3, `c` = 1, `d` = 2. The value of result in each line:

Lines	Expression	Expression's value	`result` in binary	`result` in decimal
1	`1 << a`	0b10000	0b10000	16
2	`1 << b`	0b01000	0b11000	16 + 8
3	`1 << c`	0b00010	0b11010	16 + 8 + 2
4	`1 << d`	0b00100	0b11110	16 + 8 + 2 + 4

Conditional Operators

- Works like `=if` function in Excel
- `cond ? a : b` is a very special operator that produces value depends on the condition `cond`.
- If `cond` is true, the value of this expression is a
- If `cond` is false, the value of this expression is b

```java
//set fanSpeed = 50 when it is hot
int fanSpeed = temperature > 38.9 ? 50 : 20;

//get a F if mark lower than 35
char grade = mark < 35 ? 'F' : 'P';
```

Conditional Operators

- You can even cascade the conditional operators

```
char grade = mark > 80 ? 'A' : (mark > 70 ? 'B' : (mark > 40 ? 'C' : 'F' ));
```

- Used when you don't bother to write a `if-else` statement.

Operator Precedence

Remember the following basic rules:

1. Always do what is inside the bracket first
2. Then, evaluate `i++` and `i--`
3. Then, evaluate `++i` and `--i`
4. Then, evaluate multiplication/division
5. Then, evaluate addition/subtraction
6. Always evaluate the expression from left to right
7. AND is higher than OR

Operator Precedence

Precedence	Operators
1	`i++`, `i--`
2	`++i`, `--i`, `-i`, `!i`, `~i`
3	`*`, `/`, `%`
4	`+`, `-`
5	`<<`, `>>`
6	`<`, `>`, `<=`, `>=`, `==`, `!=` relational
7	`&`,`^`, `|` bitwise AND/OR/XOR
8	`&&` AND
9	`||` OR
10	`? :` conditional operator
11	assignments `=`, `+=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`

Operator Precedence

Example

1. `2` ➡ 2
2. `2 * 3` ➡ 6
3. `2 + 4 * 3 - 7` ➡ 7
4. `4 / 2` ➡ 2
5. `10 % 3` ➡ 1
6. `(2 + 3) * (11 / 12)` ➡ 0
7. `(6 + 4) * 3 + (2 - (6 / 3))` ➡ 30
8. `1 + 0 % 9` ➡ 1
9. `99 % 9` ➡ 0

Summary

- Arithmetic Operators: `+` `-` `*` `/` `%` `++` `--` `()`
- Assignment Operators: `=` `+=` `-=` `*=` `/*` `%=`
- Relational Operators: `>=` `==` `<=` `>` `<` `!=`
- Logical Operators: `!` `||` `&&`
- Bitwise Operators: `~` `|` `&` `^` `<<` `>>` `>>>`
- Conditional Operators: `? :`

Numeric Literal and Type Casting

Numeric Literal

- Numbers can be represented in the basic form: `1`, `2`, `3.1415926`, `-468`...
- Representing lengthy number can cause clerical mistakes

```
int population = 1444812274; //how many digits are there??
```

- You cannot separate a number by space or `,`.

```
1 444 812 274; //error!
1,444,812,274; //error!
```

- Java supports representing a lengthy number separated by `_`: such as `9_999`, `1_000_000`, or even `2_0_2_6`.

Numeric Literal

- By default, any decimal numeric literal is considered as a **double**.
    - `0.1` is a double even it stores only 1 digit after the decimal space!
- Assigning a double literal to float will cause error.
- To explicit state a decimal number is a float, we add `f` after the number

```
float roughPi = 3.14f;
```

- Similarly, by default integers is considered as `int` by default.
- To explicit state a long literal, we add `L` after the number

```
long longNumber = 500L; //be careful!
```

Other Base number

- Java also support typing binary number and hexadecimal integers directly.
- Binary number has a prefix of `0b`: `0b1101` is the same as `13`, `0b0001` is same as `0b1` which is 1.
- Hexadecimal number has a base of 16. Used rather frequently in computer hardware.
- Each digit of a hex number takes a value from the set [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f].
- To represent a hex number in Java, add the prefix `0x`: `0x10` is the same as `16` and `0b10000`. `0xFF` is the same as `255` and `0b1111_1111`.
- The digit `a`,`b`,`c`,`d`,`e`,`f` can be lower case or upper case.

Casting

- Type casting changes the data type of a value from its normal type to some other type.

Two type of casting:

- Widening (automatic): changes a smaller type to a bigger/more precise type
  - byte ➡️ short ➡️ char ➡️ int ➡️ long ➡️ float ➡️ double
- Narrowing (manual): changes a bigger/more precise type to a smaller type
  - double ➡️ float ➡️ long ➡️ int ➡️ char ➡️ short ➡️ byte

Widening

```java
float f = 1.2345f; //to specify a number literal as float, add f after it
double d;
d = f;
```

- The value 1.2345 will be stored in double without any precision lost.
- No problem will happen for sure.

```java
int i = 439234;
long l;
l = i;
```

- The variable `l` has a type `long` which support a larger range than `int`.
- No problem will happen for sure.

Narrowing

```
double d = 1.23456;
float f;
f = d; //error!
```

- The assign has an error because it is possible that some digits in `d` can't be stored in `f`
- **Lost of precision**

```
long l = 123456789;
int i;
i = l; //error!
```

- It is possible that `l` has a value large than what `int` can support (± 2147483647)

Narrowing

- You can suppress the error by casting if you are sure the value are compatible

```java
double d = 1.23456;
float f;
f = (float) d; //casting
```

```java
long l = 123456789;
int i;
i = (int) l; //casting
```

- Both examples compile

Narrowing

- However, what happen if the value is *incompatible*?

```
double d = 1.23456789123456789;
float f = (float) d;
System.out.println(d + ":" + f);
```

```
1.2345678912345668:1.2345679
```

- Things get worst for integer

```
int i = 1234567;
short s = (short) i; //short support -32768 to 32767
System.out.println(i + ":" + s);
```

```
123456:-7616
```

Type casting from a char to an int

```java
char symbol = '3';
System.out.println((int) symbol);
```

- You may think this will output 3.
- Actually it prints 51, the Unicode code for `'3'`.

```java
char symbol = '3';
int x = symbol * 10;
System.out.println(x);
```

- Similarly, this **DOES NOT** give 30

Type casting from a char to an int

- The proper way to convert a char digit to int is by subtraction

```java
char symbol = '3';
int digit = symbol - '0';
```

- How about `11`?
  - Remember char only contains a single character, impossible
  - Converting a string (which support multiple characters) to int will be done by another method

Unicode	Char
48	`0`
49	`1`
50	`2`
51	`3`
52	`4`
53	`5`
54	`6`
55	`7`
56	`8`
57	`9`

Summary

- Numerical Literacy
- Type widening/narrowing
- Casting
- Converting char to digit