COMP 250

Lecture 3

binary numbers (continued),
Java primitive types,
ascii, Unicode
casting

Wed. Jan. 12, 2022
Addition in binary

The addition and multiplication algorithms from lecture 1 are based on operators +, *, /, %. These algorithms work for any base. (See lecture notes.) For example,

\[
\begin{array}{c}
11010 \\
+ 01111 \\
\hline
\end{array}
\quad 26
\quad 15
\begin{array}{c}
\hline
\end{array}
\begin{array}{c}
\quad + 15 \\
\hline
\quad 41
\end{array}
\]
Addition in binary

The addition and multiplication algorithms from lecture 1 are based on operators +, *, /, %. These algorithms work for any base. (See lecture notes.) For example,

carry  111100
    11010          26
+ 01111          + 15
 101001          41
Addition in binary

Fun example:

\[
\begin{array}{c}
11111111111 \\
+ \quad 00000000001 \\
\hline
100000000000 \\
\end{array}
\]

Let’s use the example above to ask a fundamental question about binary number representations (next slide).

youtube video of odometer at 100,000
Q: How many bits $N$ do we need to represent a positive integer $m$ in binary?

$$m = \sum_{i=0}^{N-1} b_i 2^i$$

What do we mean by “need”? We mean using as few bits as possible, such that $b_{N-1} = 1$ and $b_i = 0$ for $i \geq N$.

For example, we consider representations like $(11010)_2$ but not $(0000011010)_2$.
Q: How many bits $N$ do we need to represent a positive integer $m$ in binary?

Assuming we are using as few bits as possible, suppose:

$$m = (b_{N-1} \ldots b_4 b_3 b_2 b_1 b_0)_2$$

The smallest that $m$ can be is the $N$ bit number:

$$(100000 \ldots 000000)_2 = ?$$

The largest that $m$ can be is the $N$ bit number:

$$(111111 \ldots 111111)_2 = ?$$
Q: How many bits $N$ do we need to represent a positive integer $m$ in binary?

Assuming we are using as few bits as possible, suppose:

$$m = (b_{N-1} \ldots b_4 b_3 b_2 b_1 b_0)_2$$

The smallest that $m$ can be is the $N$ bit number:

$$(100000 \ldots 000000)_2 = 2^{N-1}$$

The largest that $m$ can be is the $N$ bit number:

$$(111111 \ldots 111111)_2 = 2^N - 1$$
From the previous slide: \[ 2^{N-1} \leq m < 2^N \]

Take the log (base 2) of each of the three terms:

\[ N - 1 \leq \log_2 m < N \]

The inequality is still correct since the log function is strictly increasing.

From here, we can show:

\[ N = \text{floor}(\log_2 m) + 1 \] where “floor” means “round down”.
<table>
<thead>
<tr>
<th>$m$ (decimal)</th>
<th>$m$ (binary)</th>
<th>$N = \lfloor \log_2 m \rfloor + 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>undefined</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Exact powers of 2 shown in red.
ASIDE: How are numbers represented in a computer?

How are integers represented (both positive and negative)?

How are fractional numbers represented?

Surprisingly, the answers do not depend on the computer or language. Rather, there is a standard format that is used by all computers. (For fractional numbers, the format is the IEEE 754 standard.)

The technical details are covered in detail in COMP 273 — see my lecture notes for that course if you are curious. I will say a bit about how integers are represented a few slides from now.
Java primitive types

- byte
- short
- int
- long
- float
- double
- boolean
- char

integer values
fractional ("real") numbers
true or false
One character

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

These are reserved keywords.
Java variables: type declaration

We can declare a primitive type variable as follows.

```java
int i;
double x;
```

In order to use it, we need to assign it a value.

```java
i = 3;
x = 4.75;
```

We can declare and assign a value in a single statement.

```java
boolean b = false;
```

false is 00000000
true is 00000001
Java primitive types: what do they encode?

The number of bits used for each data type is fixed.

The bits can encode a particular set of values.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Size</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8-bits</td>
<td></td>
</tr>
<tr>
<td>short</td>
<td>16-bits</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>32-bits</td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>64-bits</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>32-bits</td>
<td></td>
</tr>
<tr>
<td>double</td>
<td>64-bits</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>1-bit</td>
<td></td>
</tr>
<tr>
<td>char</td>
<td>16-bits</td>
<td></td>
</tr>
</tbody>
</table>
Java primitive types: what do they encode?

The number of bits used for each data type is fixed.

The bits can encode a particular set of values.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Size</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8-bits</td>
<td>{-2^7, ..., 2^7 - 1}</td>
</tr>
<tr>
<td>short</td>
<td>16-bits</td>
<td>{-2^{15}, ..., 2^{15} - 1}</td>
</tr>
<tr>
<td>int</td>
<td>32-bits</td>
<td>{-2^{31}, ..., 2^{31} - 1}</td>
</tr>
<tr>
<td>long</td>
<td>64-bits</td>
<td>{-2^{63}, ..., 2^{63} - 1}</td>
</tr>
<tr>
<td>float</td>
<td>32-bits</td>
<td>COMP 273/ECSE 222</td>
</tr>
<tr>
<td>double</td>
<td>64-bits</td>
<td>COMP 273/ECSE 222</td>
</tr>
<tr>
<td>boolean</td>
<td>1-bit</td>
<td>{true, false}</td>
</tr>
<tr>
<td>char</td>
<td>16-bits</td>
<td>later today</td>
</tr>
</tbody>
</table>

As I mentioned on slide 12, this uses 8 bits, but only the last bit matters.
N bit integers

Recall the concept of modulus: like a circle, if you keep walking forward, you get back to where you started. With modulus operator, the circle is $0, 1, \ldots, 2^N - 1$.

e.g. $N = 8$

- Java represents “signed” integers. The values on the circle go from $0, 1, \ldots, 2^{N-1} - 1$, and then jump back to $-2^{N-1}, -2^{N-1} + 1, \ldots, -1$.

e.g. $N = 8$

(inline diagram: circle with values 0 to 255 and their negative counterparts, illustrating the concept of modulus and its application in Java for signed integers.)
N bit integers

Recall the concept of modulus: like a circle, if you keep walking forward, you get back to where you started. With modulus operator, the circle is 0, 1, ..., $2^N - 1$.

e.g. $N = 32$

Java represents “signed” integers. The values on the circle go from 0, 1, ..., $2^{N-1} - 1$, and then jump back to $-2^{N-1}$, $-2^{N-1} + 1$, ..... -1.

e.g. $N = 32$
(int)

16
“Overflow” and “Underflow” e.g. int

Variables of type int store integer values from $-2^{31}$ to $2^{31} - 1$.

\[ -2^{31} = -2147483648 \quad \Leftarrow \quad \text{min int value} \]
\[ 2^{31} - 1 = 2147483647 \quad \Leftarrow \quad \text{max int value} \]

Example of overflow:

```java
int x = 2147483647; \quad (\text{max})
System.out.println(x+1);
```

\[ \Rightarrow \text{prints} \quad -2147483648 \quad (\text{min}) \]

Example of underflow:

```java
int y = -2147483648; \quad (\text{min})
System.out.println(y-1);
```

\[ \Rightarrow \text{prints} \quad 2147483647 \quad (\text{max}) \]
Floating Point

• In Java, fractional numbers are represented using “floating point”, similar to scientific notation. e.g. $6.022149 \times 10^{23}$

• The type can be either float (32 bits) or double (64 bits).

• All standard arithmetic operations (+, -, *, /) can be done on floating point.

• Java distinguishes between 1 and 1.0. If you write .0 after an integer, it will be represented as a double. If you want to represent a float, see below.

  ```java
  int x = 3.0;  // X
  int x = 3;   // ✓
  double x = 3.0;  // ✓
  float y = 3.0;  // X
  float y = 3.0f; // ✓
  ```
Floating point approximation (round off)

The value of $1/3.0$ is an approximation only.

More surprising perhaps, the value of $1/10.0$ is also an approximation only.

The reason is that computers only represent sums of powers of 2, *including negative powers of 2* (1/2.0, 1/4.0, 1/8.0, etc).

Here is an interesting example:

```java
System.out.println( 0.1 + 0.1 + 0.1 + 0.1 + 0.1 +
                     0.1 + 0.1 + 0.1 + 0.1 + 0.1 );
```

It prints out 0.9999999999999999 rather than 1.0
We can declare and initialize a variable of type `char` as follows:

```c
char letter = 'a';
```

- Character literals appear in single quotes.

  (A ‘literal’ is a particular character, string, or number.)

Character literals can only contain a single character. If you put two characters inside quotes, then it is not a character but rather it is a string.
Escape Sequences

An escape sequence is a sequence of characters that represents a special character. In Java, escape sequences are two characters and the first is a backslash.

Examples:

\n says to start a new line (e.g. when printing)
" or ' represent quotation marks
\t represents a tab
\\ represents a backslash

Escape sequences are legal characters. e.g. char c = '\n';
The `char` data type is two bytes (16 bits).

Think of them as numbered from 0 to $2^{16} - 1$ (65,535).
A common notation is ‘\u----’ where the – places are hexadecimal digits.

i.e. the values of a `char` range from ‘\u0000’ to ‘\uffff’

The first $2^7 = 128$ of them correspond to the ASCII characters (next slide).
### ASCII Table

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[NULL]</td>
<td>32</td>
<td>20</td>
<td>[SPACE]</td>
<td>64</td>
<td>40</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>'</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>[START OF HEADING]</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>[START OF TEXT]</td>
<td>34</td>
<td>22</td>
<td>*</td>
<td>66</td>
<td>42</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>[END OF TEXT]</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>[END OF TRANSMISSION]</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>[ENQUIRY]</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>[ACKNOWLEDGE]</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>f</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>[BEL]</td>
<td>39</td>
<td>27</td>
<td>'</td>
<td>71</td>
<td>47</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>g</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>[BACKSPACE]</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>h</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>[HORIZONTAL TAB]</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>i</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>[VERTICAL TAB]</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>k</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>[FORM FEED]</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>l</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>[CARRIAGE RETURN]</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>m</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>[SHIFT OUT]</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>n</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>[SHIFT IN]</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>o</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>[DATA LINK ESCAPE]</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>p</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>[DEVICE CONTROL 1]</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>[DEVICE CONTROL 2]</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>r</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>[DEVICE CONTROL 3]</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>s</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>[DEVICE CONTROL 4]</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>t</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>[NEGATIVE ACKNOWLEDGE]</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>u</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>[SYNCHRONOUS IDLE]</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>v</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>[ENG OF TRANS. BLOCK]</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>w</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>[CANCEL]</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>x</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>[END OF MEDIUM]</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>y</td>
</tr>
<tr>
<td>26</td>
<td>2A</td>
<td>[SUBSTITUTE]</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>z</td>
</tr>
<tr>
<td>27</td>
<td>2B</td>
<td>[ESCAPE]</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>\</td>
<td>123</td>
<td>7B</td>
<td>(</td>
</tr>
<tr>
<td>28</td>
<td>2C</td>
<td>[FILE SEPARATOR]</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>\</td>
<td>124</td>
<td>7C</td>
<td>]</td>
</tr>
<tr>
<td>29</td>
<td>2D</td>
<td>[GROUP SEPARATOR]</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td>I</td>
<td>125</td>
<td>7D</td>
<td>)</td>
</tr>
<tr>
<td>30</td>
<td>2E</td>
<td>[RECORD SEPARATOR]</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>~</td>
<td>126</td>
<td>7E</td>
<td>~</td>
</tr>
<tr>
<td>31</td>
<td>2F</td>
<td>[UNIT SEPARATOR]</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>'</td>
<td>127</td>
<td>7F</td>
<td>(DEL)</td>
</tr>
</tbody>
</table>
The `char` data uses Unicode which is an international standard.

Unicode is a superset of ASCII: the numbers 0-127 map to the same characters both in ASCII and Unicode.

Unicode provides the fonts for many languages. It also encodes emoji’s.

ASIDE: *there’s more to Unicode than this:* you can expand beyond $2^{16}$ symbols by having pairs of `char` where the first one is essentially an escape character.
When we say that ASCII and Unicode are “codes”, we mean that each character or symbol is represented by a sequence of bits.

But we know sequences of bits also represent numbers!

In Java, we can perform arithmetic with char values e.g.:

```java
char c = 'a';    //  97
int  k = c + 1;  //  98
```
Comparing Chars

We can also compare char values using operators ==, <, >, >=, <= which essentially compares their code values.

```java
char letter0 = 'g';
char letter1 = 'k';
System.out.println( letter0 < letter1 ); // prints true
System.out.println( 'g' < 'G' ); // prints false
System.out.println( '%' >= '&' ); // prints false
```
Type casting

Convert the values of variables from one type to another using **type casting**.

```java
double y = 4.56;
int n = (int) y;    // value of n is ??????

int x = 3;
double z = (double) x;    // value of z is ??????
```
Type casting

Convert the values of variables from one type to another using type casting.

double y = 4.56;
int n = (int) y; // value of n is 4 (rounds down)

int x = 3;
double z = (double) x; // value of z is 3.0

As we will see next, an explicit cast is unnecessary here.
Primitive type conversion – wider vs. narrower

```
<table>
<thead>
<tr>
<th>type</th>
<th>number of bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>64</td>
</tr>
<tr>
<td>float</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>64</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
</tr>
<tr>
<td>char</td>
<td>16</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
</tr>
<tr>
<td>byte</td>
<td>8</td>
</tr>
</tbody>
</table>
```

wider

implicit cast

“Wider” usually (but not always) means more bits.

narrower

explicit cast is needed

char and short are special ... see later.
Examples of widening & narrowing

```java
int i = 3;
double d = 4.2;

d = i; // widening ("implicit casting") → stores value 3.0

i = d; // compile time error
i = (int) d; // narrowing ("explicit casting") → stores value 4
```
Examples of widening & narrowing

```java
int i = 3;
double d = 4.2;

d = 5.3 * i;  // widening by "promotion"
             // (the casting here happens when the * operation is performed)

byte k = 127;
System.out.println(k + 1);  // widening by "promotion" (to integer)
                          // (the casting here happens when the + operation is performed)

Output: 128
```
Recall: Overflow and Underflow e.g. byte

How is 127 represented as a byte?

(01111111)_2 = 127

What happens if we add 1?

(1000 0000)_2 = −128
Overflow and Underflow e.g. byte

Recall that variables of type `byte` store values between $-2^7$ and $2^7 - 1$, that is, $-128$ and $127$.

**Overflow:**

```java
byte k = 127;
System.out.println(k+1);
System.out.println((byte) (k+1));
```

**Output:**

128 \hspace{0.5cm} \text{(widening by promotion)}
-128 \hspace{0.5cm} \text{(cast, narrowing)}

**Underflow:**

```java
byte j = -128;
System.out.println(j-1);
System.out.println((byte) (j-1));
```

**Output:**

-129 \hspace{0.5cm} \text{(widening by promotion)}
127 \hspace{0.5cm} \text{(cast, narrowing)}
Examples of widening & narrowing char

```java
char first = 'a';  // 97
char second = (char) (first + 1);
```

`first` is automatically converted into an `int` when performing `first + 1`, which evaluates to 98. (widening by promotion)

This `int` value is cast to `char` (narrowing), and 'b' is stored in `second`. 
It seems Eclipse and IDEA don’t require an explicit down cast. Literals treated differently.

```java
int i = 7;
char c1 = i; // compiler error
char c2 = 10; // no compiler error
```
ASIDE: Examples with char and short

char c = 'q';
short s = 2;  // allowed
int i = 3;

s = i;       // compile time error
s = (short) i; // allowed

s = c;       // compile time error
s = (short) c; // allowed

c = s;       // compile time error
c = (char) s; // allowed
Examples with `float` and `double`

```plaintext
double y = 1/4;  // assigns value 0.0 to y

double x = 1;    // legal, but considered bad style

float y = 3.0;   // compiler error

float y = (float) 3.0;  // narrowing

float z = 3.0f;    // ✓
```

Coming up…

**Lectures**

Fri. Jan 14
Java Overview (JRE, JDK, ...)

Next week
arrays, strings, objects & classes

**Homework (TODO)**

- w3schools Tutorial (this week!)
- Install either Eclipse or IntelliJ. (this week!)
- Content -> tutorials.
  Tutorial (tomorrow)
- TA office hours