Tools :

- The Java Compiler
  $\boxed{\text{javac [options] filename.java ...}}$
  `-depend`: Causes recompilation of class files on which the source files given as command line arguments recursively depend.
  `-O`: Optimizes code, slows down compilation, disables `-depend`.

- The Java Interpreter
  $\boxed{\text{java [options] classname } \langle\text{args}\rangle}$

- The Java Debugger
  $\boxed{\text{jdb [options]}}$
  Type `help` or `?` to get started.

On windows we have `Jcreator` and `netbeans` on our BSD systems.
(We also have `netbeans` on windows available, but it is not nearly as quick as `jcreator`.)

The simplest program:

```
public class Hello
{
    public static void main(String[] args)
        {
            System.out.println("Hello.");
        }
}
```

Hello.java

Executing this program would print to the console the sentence "`Hello.`".

A bit more complex:

```
public class Second
{
    public static void main(String[] input)
        {
            for (int i = input.length - 1; i >= 0; i--)
                System.out.print(input[i] + " ");
            System.out.print("\n");
        }
}
```

Second.java

Executing this program would print to the console the arguments in reverse order, each separated by one blank space.

Primitive data types :

- **`byte`** : 8 bit integer, [-128 , 127]
- **`char`** : 16 bit *unsigned* integer, [0 , 65536]. This character set offers 65536 distincts Unicode characters.
- **`short`** : 16 bit signed integer, [-32768 , 32767]
- **`int`** : 32-bit integer, [-2147483648 , 2147483647]
- **`long`** : 64 bit signed integer, [-9223372036854775807 , 9223372036854775806]

- **`float`** : 32-bit float, 1.4023984e–45 to 3.40282347e+38
- **`double`** : 64-bit float, 4.94065645841246544e–324 to 1.79769313486231570e+308

- **`boolean`** : such a variable can take on one of two values, **`true`** or **`false`**.

- **`null`** : special case . . .

Punctuation :

Symbol	Description
$=$	assignment
$+, +=$	addition $\boxed{\text{and string concatenation}}$
$-, -=$	substraction
$*, *=$	multiplication
$/, /=$	division
$\%, \%=$	modulo
$++$	pre and post increment
$--$	pre and post decrement
$==$	comparison, equality
$!$	boolean not
$>$	comparison, greater than
$<$	comparison, smaller than
$<=$	comparison, lesser-or-equal than
$>=$	comparison, greater-or-equal than
$!=$	comparison, not-equal
$\&\&$	boolean AND
$\vert\vert$	boolean OR
$\tilde{}$	bitwise negation
$\&$	bitwise AND
$\vert$	bitwise OR
$\hat{}$	bitwise xor
$<<, <<=$	shift left
$>>, >>=$	shift right
'	character delimiter
"	string delimiter
;	statement terminator
,	separator
()	expression grouping
.	reference qualifier and decimal point
$\boxed{/*, */}$	comment delimiter
//	single line comment

and there are a few more. . .

Operator Precedence :

postfix ops	[] . ($\langle exp\rangle$) $\langle exp\rangle$ ++ $\langle exp\rangle$ −−
prefix ops	++ $\langle exp\rangle$ −−$\langle exp\rangle$ −$\langle exp\rangle$ ~$\langle exp\rangle$!$\langle exp\rangle$
casting	($\langle type\rangle$)$\langle exp\rangle$
mult./div.	* / %
add./sub.	+ −
shift	<< >> >>>
comparison	< <= > >= **instanceof**
equality	== !=
bitwise-and	&
bitwise-xor	^
bitwise-or	|
boolean and	&&
boolean or	||
conditional	⟨*bool-exp*⟩ ? ⟨*true-value*⟩ : ⟨*false-value*⟩
assignment	=
op assignment	+= −= *= /= %=
bitwise assignment	>>= <<= >>>=
boolean assignment	&= ^= |=

Conditional executions :

- `if` `( exp )` statement `else` statement

  ex :

  ```
  if (input[i].equals("Hello"))
  {
      System.out.println("Hi!");
      System.out.println("Beautiful day, isn’t it ?");
  }
  else System.out.println("Impolite.");
  ```

- `?` : simplification for assignments

  ```
  if (a < b)
      x = 5;
  else
      x = 25;
  ```

  can be rewritten

  ```
  x = a < b ? 5 : 25;
  ```

- switch (exp) {

```
     case 1 :
          statement;
          break;
     case 2 :
          statement;
     case 3 :
          statement;
          break;
     default:
          statement;
          break;
}
```

Escape characters :

Escape	Unicode	Name	Description
\b	\u0008	BS	Backspace
\t	\u0009	HT	Horizontal tab
\n	\u000a	LF	Line feed
\"	\u0022	"	Double quote
\'	\u0027	'	Single quote
\\	\u005c	\	Backslash

Iteration keywords :

- <u>do</u> `statement while (exp);`
  ex :

```
int i = 0;
do System.out.println(i++);
while (i < 10);
```

- <u>while</u> `(exp) statement`
  ex :

```
public static void main(String arg[]) throws IOException
    {
     String temp, text = "";
     BufferedReader input;
     input = new BufferedReader(new InputStreamReader(System.in));

     while (! (temp = input.readLine()).equals("end."))
         text += temp;

     System.out.println(text);
    }
```

- <u>for</u> `(initial stms; exp; steps) statement`
  ex :

```
int j = 0;
for (int i = 0; i < 10; i++)
    j += i;

System.out.println(j);
```

Casting :

Casting is used to force an explicit conversion of data from one type to another. Depending on what type of data is being cast to what other type, the actual conversion can take place either at run time or at execution time. If there is no way that the conversion could be valid, the compiler generates an error.

ex:

```
int i;
float f;
f = (float) i;
/* however this is unnecessary since the compiler could
   have done the conversion automatically */

int a = 1, b = 2;
float f;
f = a / b;
/* would return 0,
   since the calculation is done in int */
f = (float)a / (float) b;
/* would return 1/2 */
```

Overloading :

Commonly refers to the capability of an operator to perform differently depending on its context within a program. Although a few operators are already overloaded, Java does not allow you to change these.

```
int a = 1, b = 2, c;
c = a + b;

/* c is now worth 3 */

String d = ‘‘d’’, e = ‘‘e’’, concat;
concat = d + e;

/* concat is now worth ‘‘de’’ */
```

However, as you’ll see later on, we can overload methods.