Tools:

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

- **The Java Interpreter**
  
  \texttt{java [ options ] classname \langle args \rangle}

- **The Java Debugger**
  
  \texttt{jdb [ options ]}
  
  Type \texttt{help} or \texttt{?} 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:

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

Executing this program would print to the console the sentence “Hello.”.

A bit more complex:

```java
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");
    }
}
```

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, 9223372036854775807]

- **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 ...
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>assignment</td>
</tr>
<tr>
<td>+, + =</td>
<td>addition and string concatenation</td>
</tr>
<tr>
<td>−, − =</td>
<td>substraction</td>
</tr>
<tr>
<td>∗, ∗ =</td>
<td>multiplication</td>
</tr>
<tr>
<td>/, / =</td>
<td>division</td>
</tr>
<tr>
<td>%, % =</td>
<td>modulo</td>
</tr>
<tr>
<td>++</td>
<td>pre and post increment</td>
</tr>
<tr>
<td>−−</td>
<td>pre and post decrement</td>
</tr>
<tr>
<td>===</td>
<td>comparison, equality</td>
</tr>
<tr>
<td>!</td>
<td>boolean not</td>
</tr>
<tr>
<td>&gt;</td>
<td>comparison, greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>comparison, smaller than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>comparison, lesser-or-equal than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>comparison, greater-or-equal than</td>
</tr>
<tr>
<td>!=</td>
<td>comparison, not-equal</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>boolean AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>bitwise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>⊕</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>&lt;&lt;, &lt;&lt;=</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;=, &gt;&gt;</td>
<td>shift right</td>
</tr>
<tr>
<td>,</td>
<td>character delimiter</td>
</tr>
<tr>
<td>&quot;</td>
<td>string delimiter</td>
</tr>
<tr>
<td>;</td>
<td>statement terminator</td>
</tr>
<tr>
<td>,</td>
<td>separator</td>
</tr>
<tr>
<td>(</td>
<td>expression grouping</td>
</tr>
<tr>
<td>.</td>
<td>reference qualifier and decimal point</td>
</tr>
<tr>
<td>/*, */</td>
<td>comment delimiter</td>
</tr>
<tr>
<td>//</td>
<td>single line comment</td>
</tr>
</tbody>
</table>
and there are a few more...
Operator Precedence:

<table>
<thead>
<tr>
<th>Postfix Ops</th>
<th>[] . ((\text{exp})) (\text{exp}) + + (\text{exp}) − −</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix Ops</td>
<td>+ + (\text{exp}) − − (\text{exp}) − (\text{exp}) \sim (\text{exp}) !(\text{exp})</td>
</tr>
<tr>
<td>Casting</td>
<td>((\text{type}))(\text{exp})</td>
</tr>
<tr>
<td>Multiplication/Division</td>
<td>* / %</td>
</tr>
<tr>
<td>Addition/Subtraction</td>
<td>+ −</td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt; &gt;&gt; &gt;&gt;&gt;</td>
</tr>
<tr>
<td>Comparison</td>
<td>&lt; &lt;= &gt; &gt;= instanceof</td>
</tr>
<tr>
<td>Equality</td>
<td>== !=</td>
</tr>
<tr>
<td>Bitwise-And</td>
<td>&amp;</td>
</tr>
<tr>
<td>Bitwise-Xor</td>
<td>^</td>
</tr>
<tr>
<td>Bitwise-Or</td>
<td></td>
</tr>
<tr>
<td>Boolean And</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>Boolean Or</td>
<td></td>
</tr>
<tr>
<td>Conditional</td>
<td>\langle \text{bool-exp} \rangle \ ? \langle \text{true-value} \rangle : \langle \text{false-value} \rangle</td>
</tr>
<tr>
<td>Assignment</td>
<td>= += − = * = / = % = &gt;= &lt;&lt; &gt;= &gt;&gt; &gt;&gt;= &amp; = ^ =</td>
</tr>
</tbody>
</table>
Conditional executions:

- **if** (exp) statement else statement

  ex:

  ```java
  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

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

  can be rewritten

  ```java
  x = a < b ? 5 : 25;
  ```
\* switch ( exp ) {

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

---

**Escape characters:**

<table>
<thead>
<tr>
<th>Escape</th>
<th>Unicode</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>\u0008</td>
<td>BS</td>
<td>Backspace</td>
</tr>
<tr>
<td>\t</td>
<td>\u0009</td>
<td>HT</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>\n</td>
<td>\u000a</td>
<td>LF</td>
<td>Line feed</td>
</tr>
<tr>
<td>&quot;</td>
<td>\u0022</td>
<td></td>
<td>Double quote</td>
</tr>
<tr>
<td>'</td>
<td>\u0027</td>
<td></td>
<td>Single quote</td>
</tr>
<tr>
<td>\</td>
<td>\u005c</td>
<td></td>
<td>Backslash</td>
</tr>
</tbody>
</table>
Iteration keywords:

- **do statement while (exp);**
  
  ```java
  int i = 0;
  do System.out.println(i++);
  while (i < 10);
  ```

- **while (exp) statement**
  
  ```java
  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);
  }
  ```

- **for (initial stms; exp; steps) statement**
  
  ```java
  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:

```java
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.

```java
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.