

- [Plugincentral^[18]]

NetBeans

The NetBeans IDE is a free, open-source Integrated Development Environment for software developers. The IDE runs on many platforms including Windows, Linux, Solaris, and the MacOS. It is easy to install and use straight out of the box. The NetBeans IDE provides developers with all the tools they need to create professional cross-platform desktop, enterprise, web and mobile applications.

More info can be found at <http://www.netbeans.org/products/ide/>

BlueJ

BlueJ is an IDE that includes templates and will compile and run the applications for you. BlueJ is often used by classes because it is not necessary to set classpaths. BlueJ has it's own sets of Library's and you can add your own under preferences. That sets the classpath for all compilations that come out of it to include those you have added and the BlueJ libraries.

BlueJ offers an interesting GUI for the creation of packages and programs. Classes are represented as boxes with arrows running between them to represent inheritance/implementation or if one is constructed in another. BlueJ adds all those classes (the project) into the classpath at compile time.

BlueJ Homesite^[19]

Kawa

Kawa was developed by Tek-Tools. It is basically a Java editor which does not include wizards, and GUI tools. It is best suited to experienced Java programmers in small and midsized development teams.

The latest version is 4.0, you can download it.^[20] For more info. see Kawa from tek-tools^[21]. See a javaworld article^[22]

It looks that there is no new development for Kawa.

Ant

For comprehensive information about all aspects of Ant, please see the Ant Wikibook.

Ant is a build management tool designed to replace make as the tool for automated builds of large Java applications. Like Java, and unlike make, Ant is designed to be platform independent.

Building a Java application requires certain tasks to be performed. Those tasks may include not only compiling the code, but also copying files, packaging the program into a jar file, running tests and so on. Some of these tasks may depend upon others having been done previously (not creating a jar unless the program has been compiled, for instance). It might also be a good idea to not execute all tasks every time the program is compiled -- e.g. to only compile changed source files. **Ant makes all of these things easy.**

The tasks and their dependencies are defined in a build.xml file, generally kept in the root directory of the java project. Ant parses this file and executes the tasks therein. Below we give an example build.xml file.

Ant tool is written in Java and is open source, so it can be extended if there is a task you'd like to be done during the build that is not in the pre-defined tasks list. It is very easy to hook your ant task code to the other tasks: your code only needs to be in the classpath, and the Ant tool will load it at runtime. For more information about writing your own Ant tasks, please see the project website at <http://ant.apache.org/>.

Example build.xml file.

```
<?xml version="1.0"?>

<project name="ExampleApp" basedir="." default="main">

    <property name="source.dir"      value="source" />
    <property name="libraries.dir"    value="libraries" />
    <property name="build.dir"        value="build" />
    <property name="classes.dir"      value="${build.dir}/classes"
/>
    <property name="dist.dir"         value="${build.dir}/dist" />
    <property name="main-class"
value="com.example.ExampleApp"/>

    <path id="classpath">
        <fileset dir="${libraries.dir}" includes="**/*.jar"/>
    </path>

    <target name="clean">
        <delete dir="${build.dir}"/>
    </target>

    <target name="compile">
        <mkdir dir="${classes.dir}"/>
        <javac srcdir="${source.dir}" destdir="${classes.dir}"
classpathref="classpath" />
        <copy todir="${classes.dir}">
            <fileset dir="${src.dir}" excludes="**/*.java" />
        </copy>
    </target>

    <target name="build" depends="compile">
        <mkdir dir="${dist.dir}"/>
        <copy todir="${dist.dir}/lib" flatten="true">
            <path refid="classpath" />
        </copy>
        <path id="dist.classpath">
            <fileset dir="${dist.dir}/lib" includes="*.jar" />
        </path>
        <manifestclasspath property="dist.manifest.classpath"
jarfile="${dist.dir}/${ant.project.name}.jar">
            <classpath refid="dist.classpath" />
        </manifestclasspath>
        <jar destfile="${dist.dir}/${ant.project.name}.jar" >
            <zipfileset dir="${classes.dir}" />
            <manifest>
                <attribute name="Class-Path"
value="${dist.manifest.classpath}"/>
```

```
        <attribute name="Main-Class" value="${main-class}"
/>
    </manifest>
</jar>
</target>

<target name="run-build" depends="build">
    <java jar="${dist.dir}/${ant.project.name}.jar"
fork="true">
        <classpath>
            <path refid="classpath"/>
            <path
location="${dist.dir}/${ant.project.name}.jar"/>
        </classpath>
    </java>
</target>

<target name="run" depends="compile">
    <java classname="${main-class}" >
        <classpath>
            <path refid="classpath"/>
            <pathelement location="${classes.dir}" />
        </classpath>
    </java>
</target>

<target name="clean-build" depends="clean,build"/>

<target name="main" depends="clean,run"/>

</project>
```

The JIT compiler

The standard JIT compiler runs *on demand*. When a method is called repeatedly, the JIT compiler analyzes the bytecode and produces highly efficient machine code, which runs very fast. The JIT compiler is smart enough to recognize when the code has already been compiled, so as the application runs, compilation happens only as needed. As Java applications run, they tend to become faster and faster, because the JIT can perform runtime profiling and optimization to the code to meet the execution environment. Methods or code blocks which do not run often receive less optimization; those which run often (so called *hotspots*) receive more profiling and optimization.
