Development Tools

IDEs
- Integrated development environments (IDEs), e.g. Bluej and Visual Studio...
  - help programmers focus on programming
  - by hiding details of underlying tools
- But
  - important to know differences between e.g. compile-time & run-time errors
  - important to know what details are being managed, e.g. make dependencies
  - want to gain better control sometimes
  - want to support additional tools

Manual development tools
- Alternatively, can make programmers know about and use all the tools that were packaged up in the IDE
  - more knowledge, understanding
  - more power (e.g. adding new tools)
  - more work on programmer's part

Unix tool suite

Structure of an IDE

Main Java development tools
- Your favorite text editor
- javac file.java...
  - compile one or more .java source files into corresponding .class compiled files
- java Class arg...
  - run compiled Java program
  - start in class Class with method
  - public static void main(String[] args)
    - typically, there's a .class, class compiled file
    - args array initialized with arg... from command line
- http://java.sun.com/j2se/1.4.1/docs/
Handling references to other classes

- One Java class can refer to many other Java classes
  - When compiling the first class, how does javac find the other classes, e.g. to check their types?
  - When running the main class, how does java find the other classes that the program references?
  - Can give them as extra javac arguments
  - What about standard Java library classes?
  - Don’t want to have to recompile every time
  - Can specify a classpath argument to javac

The classpath

- javac -classpath dirs file:java...
- java -classpath dirs Class arg...
  - Specified a series of directories in which to search for pre-compiled classes
  - dirs has the form path1:path2:path3:...
    - on cygwin, use ":" instead of ":" and "\" instead of "\"
  - (A class named Foo is compiled into a file named Foo.class)

CLASSPATH

- Instead of specifying -classpath to every javac and java command, can set the CLASSPATH environment variable instead
  - setenv CLASSPATH \$HOME/myClasses:$HOME/yourClasses
  - Do this in your .cshrc to "configure" your Java compilation and execution environment

Packages

- Java organizes classes into packages
  - E.g., java Jang, myApp.UI.windows
  - Each Java source file declares its package
    - E.g., "package myApp.UI.windows; ...
  - Packages correspond to directory hierarchies
    - E.g., the myApp/UI/windows directory contains the above .java source file
    - myApp should be found inside some directory in CLASSPATH

Archives

- Often want to put a collection of files together into a single file
  - tar is the standard Unix command to do this for regular files
  - Collections of compiled files are libraries
    - Id is the command that builds .a files from .o files
    - jar is the command for building Java .jar files
    - can contain .class files, Java files and anything else
      - E.g., jar of myStuff.jar *.java,.class
      - E.g., jar of myApp.jar myApp (myApp is a directory)
  - Can put a .jar file in the classpath
    - Will search the .jar file’s contents for matches

Standard libraries

- Every language has a set of standard things that every program should be able to access
  - Often called standard libraries
  - In Java, there’s a .jar file that contains all the .class files for the Java package
    - Implicitly added to the classpath
Debugging

- `jdb`
  - Starts up a Java debugger
  - Works best if used "javac -g ..." before
- Inside can run a program, set breakpoints, single-step through execution, and print out program state
  - If run under emacs, then emacs will show corresponding source lines where you are
  - Java's multiple threads makes this complicated

Debugger commands

- `run Class arg`
  - run class Class's main method, on args
  - good to set breakpoints first, if want to stop somewhere
- `stop in Class.method`
- `stop at Class.lineNumber`
  - set a break point at the start of a method or at a particular line in a source file
- `catch Err (e.g. java.lang.NullPointerException)`
  - stop if an instance of Err is thrown & not caught

More debugger commands

- `cont`
  - continue from a breakpoint
- `next`
  - continue to the next line in the current method
- `step`
  - continue to the next line, possibly in the callee or caller method

More debugger commands

- `where`
  - print out the current stack
- `print expr`
- `dump expr`
  - print out (short or long) description of result of evaluating `expr`
  - `expr` often a simple variable name, but can be as complex as a method call, too

Managing recompilation

- What happens if a source file is changed?
  - Possibly need to recompile all the files that referenced it
- How to do this?
  - IDE: built-in
  - So far: by hand
    - call `javac on out-of-date source files, maybe re-jar`
    - But: tedious error prone
  - Tool-based approach: make a tool for it!

make

- `make` is a great tool that manages any kind of building dependences
- A Makefile describes rules for when something depends on something else, and what to do to make it up-to-date
  - based on file modification times stored with every Unix file
- Invoking make then runs these rules to decide what, if anything, needs to be done to bring things up-to-date
Dependencies

- Makefile includes lines of the form
  target... : source...
  Means that each target depends on each source
  If any of the sources are modified, then all
  the targets are out-of-date
- Example:
  main.class: main.java

Actions

- For each dependency, can add an
  action to perform to bring the target(s)
  up-to-date
  Action is a series of shell command lines
  each line must start with a tab
  use /bin/sh syntax
- Example:
  main.class: main.java
  javac main.java

Invoking make

- make target...
  uses Makefile in current directory to bring one or
  more targets up to date, using their actions
  does nothing if all targets up to date
  if omit target arguments, then rebuild the first
  target in Makefile
  the default target

> make main.class
javac main.java
>

Controlling output

- By default, make prints out each action it
  performs
- Can disable printing an action by prefixing it
  with @
- Example:
  main.class: main.java
  @echo Compiling main.java...
  @javac main.java

> make main.class
Compiling main.java...
>

Dependency patterns

- Often have a simple rule over all files
  with certain naming patterns
  Can use % in the target and source
  Rule applies to any real targets and
  sources where % is replaced by the same
  thing on both sides
- Example:
  %-class: %-java
  Means that %-class depends on %-java

Actions for patterns

- Actions for dependency patterns need to
  have patterns too
  %@: the target
  @*: the sources
  $<: the first source
  $*: the thing matched by * in the rule
- Example:
  %-class: %java
  @echo "compiling class $* ($< to $@)"
  javac $<

CSE-480 - Only Oomhun 91
CSE-480 - Only Oomhun 92
CSE-480 - Only Oomhun 93
CSE-480 - Only Oomhun 94
CSE-480 - Only Oomhun 95
CSE-480 - Only Oomhun 96
Dependency trees

- One target can depend on another
target, ad nauseam
- Dependency rules form a DAG (directed
acyclic graph)
- Make figures out how to rebuild a
target by first making sure its sources
are up-to-date, which may cause make
to first rebuild them, recursively

Example dependency tree

```
% .class: % .java
  javac <
main.jar: main.class helper.class
  jar cvf $@ ^
install: main.jar
cp $< $(HOME)/bin
```

> make install
  javac main.java
  javac helper.java
  jar cvf main.jar main.class helper.class
cp main.jar /home/law/myLogin/bin

Makefile variables

- Can define variables in Makefiles, and use
them in rules and actions
  - `VARNAME` = `REPLACEMENT`
  - Referenced by `$(VARNAME)`
- Example:
  ```
  JAVAC_FLAGS = -g
  %.class: % .java
    @echo "compiling class "$"
    javac $(JAVAC_FLAGS) $<
  ```

Substitutions in make vars

- Can do replacements in variables
  - `$(VAR:pre%=post=)`
  - match each word in `VAR` against
    `pre%post`, where `%` can match anything
  - replace matches with `new`
    - if `new` contains `%`, substitute with what `%`
    matched
  - Good for adjusting extensions, prefixes

Examples of substitutions

```
SRCS = A.java B.java C.java
OBJS = $(SRCS: % .java = % .class)
default: $(OBJS)

INSTALL_DIR = $(HOME)/bin
INSTALLED_OBJS = \
  $(OBJS:% = $(INSTALL_DIR)/%)
$(INSTALL_DIR)/%.class: %.class
cp $< $@
install: $(INSTALLED_OBJS)
```

Make quiz

- Extend Makefile so that "make clean"
removes all .class files
- Add a rule so that I can say "make
  foo.java.ps", for any foo.java, to format my
  java source file using enscript-2r into a
  nicely formatted .ps file
- Add a rule to put all my .class files into a
  single .jar file
- Add a variable defining all the .java files in
  my application, and only clean, format, and
  archive those files