Objectives

- Eclipse
- Build tools
  - Make
  - Ant
  - Maven

Review

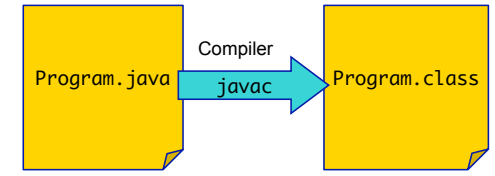
- What are tools meant to do?
  - Why do we create them?
- Where can we apply tools to the software life cycle?

Eclipse: IDE

- Integrated Development Environment
  - Developing
  - Compiling
  - Running
  - Debugging
  - Packaging

Java Programming Language

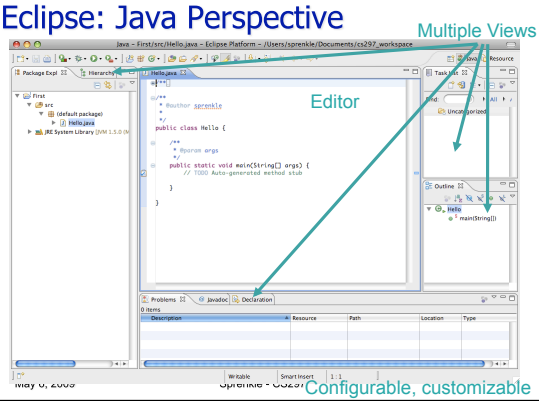
- Entirely object-oriented
- Similar to Python



Written in Java Programming Language

Bytecode: machine code for a virtual CPU

Eclipse: Java Perspective



Eclipse Warm Up

- Make a new Java project called *First*
- Create a new class called Hello
- Create a new `jar` file
  - `jar` is a Java archive, similar to a tape archive (`tar`)
    - Includes all the `class` files needed to run the application/for the library
  - Export
    - Java → Jar file

More using Eclipse

- Import
  - Existing projects into workspace
  - Select archive file:
    - /home/courses/cs297/handouts/day7/jotto.jar
  - Should give the name "Jotto" to the project
- Run Jotto
  - Click the play button on the project, on jotto, or on jotto.Jotto
- Like Lingo
  - Start a new game

Eclipse Plugins

- Allow you to customize your Eclipse
- Provide new perspectives, views for Eclipse
- Download and install the plugins you want

BUILD TOOLS

Distributing Software

- Pieces typically distributed:
  - Binaries/Bytecode
  - Required libraries
  - Data files
  - Documentation

  artifacts
- Typically packaged in an archive:
  - e.g., tgz, jar, zip, rpm
- May need all of these or some subset of them

Related Tools: make

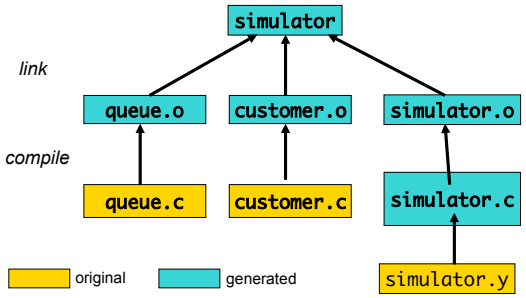
- **make**: A program for building and maintaining computer programs
  - Developed at Bell Labs around 1978 by Stu Feldman
    - Now Google VP Engineering
    - Past President of ACM
- Instructions stored in a special format file called a **makefile**

`http://www.gnu.org/software/make/`

make Features

- Contains the build instructions for a project
  - Automatically updates files based on a series of *dependency rules*
  - Supports multiple configurations for a project
- Only re-compiles necessary files after a change (conditional compilation)
  - Major time-saver for large projects
  - Uses timestamps of the intermediate files
- Typical usage: executable is updated from object files which are in turn compiled from source files

Dependency Graph



Example Makefile

```
# Example Makefile
CC=g++
CFLAGS=-g -Wall -DDEBUG
OBJECTS=customer.o simulator.o queue.o

simulator: $(OBJECTS)
        $(CC) $(CFLAGS) -o simulator $(OBJECTS)
simulator.o: simulator.c
        $(CC) $(CFLAGS) -c simulator.c

customer.o: customer.c
        $(CC) $(CFLAGS) -c customer.c
…
clean:
        rm $(OBJECTS) simulator
```

Variables

Dependencies

Rules/ Targets

Must be a tab

Commands

Running:

```
$ make
$ make clean
$ make -f other_makefile
```

By default looks for `makefile`

Related Tools: Apache Ant

- Java-based build tool
- Similar to `make`

Make	Ant
Shell-based, makefile	Java, XML config files

`http://ant.apache.org/`

XML: eXtensible Markup Language

- Looks similar to HTML
  - HTML's stricter brother
- Designed to structure, store, and transport data
  - Text file → PORTABLE!
- Made up of *nested* elements
  - Hierarchy of data
- Schema
  - *Define* your own tags, tag nesting, tag attributes

XML Example

```
<email>
   <to>you@somewhere.org</to>
   <from>me@here.org</from>
   <subject>Reminder</subject>
   <message>Don't forget me this
weekend!</message>
</email>
```


XML Example

Root element

child elements

```
<email>
   <to>you@somewhere.org</to>
   <from>me@here.org</from>
   <subject>Reminder</subject>
   <message>Don't forget me this
weekend!</message>
</email>
```

Close every element you open

XML Example

attribute

```
<imdb>
        <movie category="comedy">
                <title lang="en">Juno</title>
                <title lang="es">La joven vida de Juno</title>
        </movie>
        <movie category="comedy">
                <title lang="en">Chicken Run</title>
                <title lang="de">Hennen Rennen</title>
        </movie>
</imdb>
```


Ant buildfile: `build.xml`

- Starts with XML version:

```
<?xml version="1.0" encoding="UTF-8"?>
```

- Root element: `project`
  - `name` attribute: name of the project
  - `default` attribute: default *`target`*
  - `basedir` attribute: directory to run from

```
<project name="Hello World"
default="Hello" basedir=".">
```

 In Eclipse

Ant `target`

- Target: has a name, set of tasks to execute
- Can specify which targets to execute
  - If no target given, use project's default
- Can depend on other targets
- Examples:
  - Compile
  - Distribute
    - Needs compile

Closes open tag

```
<target name="compile"/>
<target name="jar"
depends="compile"/>
```


Example Ant target

What does this do?

```
<target name="compile"
      description="Compile the source code">
   <mkdir dir="build/classes"/>
   <javac srcdir="src"
          destdir="build/classes"
          debug="on">
      <include name="**/*.java"/>
      <classpath refid="build.class.path"/>
   </javac>
</target>
```

 `build-replay.xml`

Ant `property`

- Like a variable: defines a name and its value:
  - `<property name="vname" value="vvalue" />`
- To use property, use `${name}`
- In `build.xml` in Eclipse, add two properties:
  - HelloText=Hello
  - WorldText=World

Ant in Eclipse

- Add two new targets
- First:
  - Use ctl-space to auto-complete

```
<target name="Hello">
    <echo>${HelloText}</echo>
</target>
```

- Second: use Eclipse's design view
- Run file

See similarities to Ant?

```
# Example Makefile
CC=g++
CFLAGS=-g -Wall -DDEBUG
OBJECTS=customer.o simulator.o queue.o

simulator: $(OBJECTS)
        $(CC) $(CFLAGS) -o simulator $(OBJECTS)
simulator.o: simulator.c
        $(CC) $(CFLAGS) -c simulator.c

customer.o: customer.c
        $(CC) $(CFLAGS) -c customer.c
…
clean:
        rm $(OBJECTS) simulator
```

Variables

Dependencies

Rules/ Targets

Commands

Running:

```
$ make
$ make clean
$ make -f other_makefile
```

By default looks for `makefile`

Apache Maven

- Maven: Yiddish word meaning *accumulator of knowledge*
- For building and managing any Java-based project

`http://maven.apache.org/`

Maven Philosophy: Convention Over Configuration

- Maven's location assumptions:
  - source code: ${basedir}/src/main/java
  - Resources: ${basedir}/src/main/resources
  - Tests: ${basedir}/src/test
- Other assumptions:
  - Want to produce a JAR file in ${basedir}/target
  - Compile byte code to ${basedir}/target/classes

How does this philosophy help us?

Maven Philosophy: Convention Over Configuration

How does this philosophy help us?

- Ant-based builds *define* locations
  - No built-in idea of where source code or resources are
  - **User** has to supply this information → more work for us!!

Could be for any project:

```
<target name="compile"
      description="Compile the source code">
   <mkdir dir="build/classes"/>
   <javac srcdir="src"
          destdir="build/classes"
          debug="on">
      <include name="**/*.java"/>
      <classpath refid="build.class.path"/>
   </javac>
</target>
```


Maven Philosophy: Convention Over Configuration

- Beyond location conventions…
- **Core plugins** apply a common set of conventions for compiling source code, packaging distributions, generating web sites, and many other processes
  - Example: similar to Ant compile target
- Little effort:
  - Put source in the correct directory
  - Maven handles the rest

Consequences of Convention Over Configuration

- Users may feel forced to use a particular methodology or approach
- Most defaults can be customized
- Can create custom plugins for your requirements

Maven Build Lifecycle

- Defined by a list of *build phases*
- Example build phases
  - `compile` - compile the source code of the project
  - `test` - test the compiled source code using a suitable unit testing framework
  - `package` - take the compiled code and package it in its distributable format, such as a JAR
- When execute a phase, executes life cycle's previous phases first, in order
  - E.g., calling package would execute compile and then test

Maven Build Lifecycle

- 3 built-in build lifecycles
  - `default` lifecycle handles project deployment
  - `clean` lifecycle handles project cleaning
  - `site` lifecycle handles the creation of project's site documentation

Eclipse Maven Plugin

- Two plugins available

Summary: Build Tools

- Automate process of building various "artifacts" from your source code
  - Examples: compile, distribute (jars), documentation, commercial_version, …
- Why is there more than one build tool?
- What are the similarities and differences between make, ant, and maven?

Running Discussion Questions

- Why does the tool exist? What is its purpose?
- What can the tool do?
- What can't the tool do?
  - Because it hasn't been done? Because of current technology limitations? Or some other limitations?
  - If because it hasn't been done, what can we need to do to change that?

For Next Time

- Read "Source Code Exploration Using Google"
  - Summary on Sakai