
# Build Tools

## Optimizing the Build Process

Randy J. Fortier
randy.fortier@uoit.ca
@randy_fortier

# Outline

- What are build tools?
- What can build tools do for me?
- An introduction to Gradle
- Incorporating 3rd party tools into the build cycle


# Build Tools

What are build tools?

Why use build tools?

# Build Tools

- A build tool is a software package that automates the build process
  - Make
  - Rake
  - Ant, NAnt
  - Maven
  - Grunt
  - Gradle

# make

*makefile:*

```
all: prog lib1 lib2

prog:
   g++ -Wall prog.cpp -o prog

lib1:
   g++ -Wall lib1.cpp -o lib1.o

lib2:
   g++ -Wall lib2.cpp -o lib2.o
```

# Rake

*rakefile:*

```
file 'prog' => 'prog.cpp' do
  sh 'g++ -Wall prog.cpp -o prog'
end
file 'lib1.o' => 'lib1.cpp' do
  sh 'g++ -Wall lib1.cpp -o lib1.o'
end
file 'lib2.o' => 'lib2.cpp' do
  sh 'g++ -Wall lib2.cpp -o lib2.o'
end
```

# Ant

*build.xml:*

```
<project>
  <target name="compile">
    <javac srcdir="src" destdir="build/classes"/>
  </target>
  <target name="jar">
    <mkdir dir="build/jar"/>
    <jar destfile="build/jar/MyApp.jar" basedir="build/classes">
      <manifest>
        <attribute name="Main-Class" value="pkg.MainApp" />
      </manifest>
    </jar>
  </target>
  <target name="run">
    <java jar="build/jar/MyApp.jar" fork="true" />
  </target>
</project>
```

# Maven

- Maven is quite a different build tool than make, rake, and ant
  - It handles libraries/dependencies
- Maven keeps track of a large repository of libraries that you can link to and use in your application
  - Dependencies are downloaded automatically
  - It is declarative, rather than imperative
  - We don’t necessarily declare the steps to do something, just declare that we want it done

# Maven

*pom.xml:*

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
         http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ca.uoit.myApp</groupId>
  <artifactId>myApp</artifactId>
  <version>1.0</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

# Maven Projects

- The pom.xml is called the Project Object Model
- The directory structure of your project should match:

```
myApp
   pom.xml
   src
      main
         java
      test
         java
```

# Grunt

- Declarative 'build' tool (like Maven)
  - Dependency management (e.g. libraries)
  - Automated tasks
- Used in Node.js projects (server-side JavaScript)

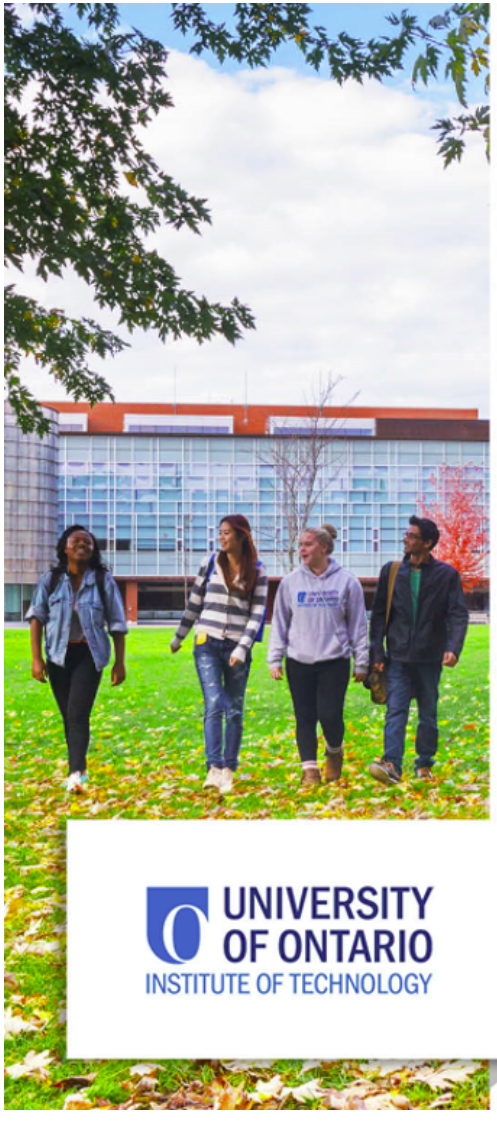
```
module.exports = function(grunt) {

  grunt.initConfig({
    jshint: {
      files: ['Gruntfile.js', 'src/**/*.js', 'test/**/*.js'],
      options: {
        globals: {
          jQuery: true
        }
      }
    },
    watch: {
      files: ['<%= jshint.files %>'],
      tasks: ['jshint']
    }
  });
```

# Bower

- Declarative 'build' tool (like Maven)
  - Dependency management
  - Automated tasks
- Client-side JavaScript

```
{
  "name": "myApp",
  "version": "1.0.0",
  "dependencies": {
    "jquery": "~2.1.4"
  },
  "private": true
}
```




# Build Tools

## Gradle Basics

# Gradle

- Gradle combines the best of Ant and Maven into a single tool:
  - The ability to create your own tasks (Ant)
  - The management of dependencies (Maven)
  - A declarative syntax (Maven)

# Gradle

- Gradle also other advantages:
  - It doesn’t use the cumbersome XML syntax that Ant and Maven use
  - Instead, Gradle uses Groovy
    - Groovy is a programming language, inspired by Java
    - Groovy is compiled into bytecodes, and can be run in the JVM
  - Writing custom tasks is easy

# Imperative vs. Declarative

- Imperative:
  1. Clean project
  2. Compile source code
  3. Put distribution into an archive
  4. ...
- Declarative:
  - Here is what I want to do

# Gradle Projects

- The directory structure is identical to a Maven project:

```
myApp
   build.gradle
   src
      main
         java
      test
         java
```

# Gradle Files

- settings.gradle
  - This file is optional
  - Properties (e.g. project name)
- gradle.build
  - Custom build tasks
  - Import plug-ins
  - Dependencies

# Custom Tasks

- Defining a task occurs in a Groovy closure (basically, a function):

```
task(sayHello) {
   doLast {
      println "Hello from a custom task!"
   }
}
```

# Dependent Tasks

- Tasks can require that other tasks be executed, first:

```
task(sayHello) {
    doLast {
        println "Hello from a custom task!"
    }
}

task(howAreYou, dependsOn: 'sayHello') {
    doLast {
        println "How are you?"
    }
}
```

# Dependencies

- Any libraries your project needs are called dependencies
- Like Maven, Gradle manages those dependencies for you
  - Gradle can even use Maven's repository, Maven Central
- This build script declares a dependency on Apache's Commons Validator

```
repositories {
    mavenCentral()
}

dependencies {
    compile group: 'commons-validator',
            name: 'commons-validator',
            version: '1.4.1'
}
```

# Plug-ins

- Gradle has many useful plug-ins:
  - java
  - android
  - scala
  - groovy

```
apply plugin: 'java'
```

# The 'java' Plug-in

- The 'java' plug-in adds several tasks:
  - classes: Generates classfiles
  - jar: Creates a JAR file
  - javadoc: Generate JavaDoc documentation from source files

```
$ gradle classes
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE

BUILD SUCCESSFUL

Total time: 3.913 secs
```

# Configuring Tasks

- Let's customize the 'jar' task that comes with the 'java' plug-in:

```
sourceCompatibility = 1.7
version = '1.0'
jar {
    manifest {
        attributes  'Main-Class': 'Test',
                    'Implementation-Title': 'Sample-Gradle',
                    'Implementation-Version': version
    }
}
```

# Task Types

- Tasks can also be assigned types
  - Task types indicate what Java class (if any) should implement the task
  - Task types, therefore, have an effect similar to inheritance

```
task(run, dependsOn: 'classes', type: JavaExec) {
    main = 'Test'
    classpath = sourceSets.main.runtimeClasspath
}
```

# Configuration

- Normally, goes in settings.gradle
- Can also put settings in gradle.properties

```
rootProject.name = 'Sample-Gradle'
```


# Build Tools

## Integrating external tools into the build

# Minification and Obfuscation

- Minification
  - Reduces the size of generated classfiles
- Obfuscation
  - Changes variable/function names to make the code harder to read if decompiled

# ProGuard

- Use ProGuard within Gradle:

```
buildscript {
   repositories {
      flatDir dirs: 'C:/Dev/proguard5.2.1/lib'
   }
   dependencies {
      classpath ':proguard'
   }
}

task (minify, dependsOn: 'jar', type: proguard.gradle.ProGuardTask)
   configuration 'proguard.cfg'

   injars 'build/libs/Sample-Gradle-1.0.jar'
   outjars 'build/libs/Sample-Gradle-1.0.min.jar'
}
```

# ProGuard

- Configure ProGuard:

```
-libraryjars  <java.home>/lib/rt.jar
-printmapping sample_gradle.map
-keep public class myapp.Test {
    public static void main(java.lang.String[]);
}
-keep public class myapp.data.Message { *; }
-dontwarn
```

# Wrap-Up

- In this section, we learned about:
  - What build tools are, and why should we use them
  - How to implement a Gradle build for a Java project
  - How to use external tools within our build:
    - e.g. obfuscation