Example Uses of Java Reflection

Explained Simply

CiaranMcHale.com
Complexity explained simply

License

1. Basic uses of Java reflection

Ant

- Ant reads build (compilation) instructions from an XML file
- Ant is hard-coded to know how to process top-level elements
  - `property`, `target`, `taskdef` and so on
- Each Ant task (used inside `target` elements) is a plug-in
  - See example Ant build file on the next slide for examples of tasks
- Many task plug-ins are bundled with the Ant distribution (`jar`, `javac`, `mkdir`, ...)
  - A properties file provides a mapping for *task-name* → *class-that-implements-task*
- Users can use `taskdef` to tell Ant about user-written tasks
  - See example on the next slide

Example Ant build file

```
<?xml version="1.0"?>
<project name="example build file" ...>
    <property name="src.dir" value="..."/>
    <property name="build.dir" value="..."/>
    <property name="lib.dir" value="..."/>

    <target name="do-everything">
        <mkdir dir="..."/>
        <mkdir dir="..."/>
        <javac srcdir="..." destdir="..." excludes="..."/>
        <jar jarfile="..." basedir="..." excludes="..."/>
        <foo .../>
    </target>

    <taskdef name="foo" classname="com.example.tools.Foo"/>
</project>
```

Auto-completion in a text editor

- Some Java editors and IDEs provide auto-completion
  - Example: you type "`someObj.`" and a pop-up menu lists fields and methods for the object's type
- The pop-up menu is populated by using Java reflection

JUnit

- JUnit 3 uses reflection to find methods whose names start with “test”
- The algorithm was changed in JUnit 4
  - Test methods are identified by an annotation (Annotations were introduced in Java 1.5)
  - Reflection is used to find methods with the appropriate annotation

Spring

- Below is an extract from a Spring configuration file:

```
<?xml version="1.0"?>
<beans ...>
  <bean id="employee1"
    class="com.example.xyz.Employee">
    <property name="firstName" value="John"/>
    <property name="lastName" value="Smith"/>
    <property name="manager" ref="manager"/>
  </bean>

  <bean id="manager"
    class="com.example.xyz.Employee">
    <property name="firstName" value="John"/>
    <property name="lastName" value="Smith"/>
    <property name="manager" ref="manager"/>
  </bean>

  ...
</beans>
```

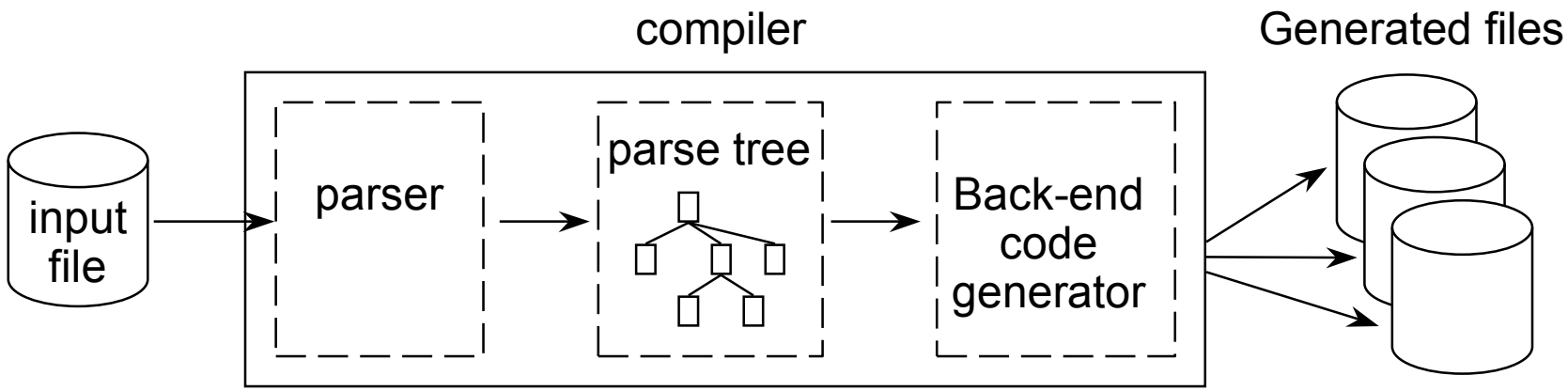
Spring (cont')

- Spring uses reflection to create an object for each `bean`
  - The object's type is specified by the `class` attribute
- By default, the object is created with its default constructor
  - You can use `constructor-arg` elements (nested inside `bean`) to use a non-default constructor
- After an object is constructed, each `property` is examined
  - Spring uses reflection to invoke `obj.setXxx(value)`
    - Where `Xxx` is the capitalized name of property `xxx`
  - Spring uses reflection to determine the type of the parameter passed to `obj.setXxx()`
  - Spring can support primitive types and common `Collection` types
  - The `ref` attribute refers to another `bean` identified by its `id`

2. Code generation and bytecode manipulation

Code generators

- Most compilers have the following architecture



- Java's reflection metadata is conceptually similar to a parse tree
- You can build a Java code generation tool as follows:
  - Do *not* write a Java parser. Instead run the Java compiler
  - Treat generated .class files as your parse tree
  - Use reflection to navigate over this "parse tree"

Code generators (cont')

- Compile-time code generation in a project:
  - Use technique described on previous slide to generate code
  - Then run Java compiler to compile generated code
  - Use Ant to automate the code generation and compilation
- Runtime code generation:
  - Use techniques described on previous slide to generate code
  - Then invoke a Java compiler *from inside* your application:
    - Can use (non-standard) API to Sun Java compiler
      - Provided in `tools.jar`, which is shipped with the Sun JDK
    - Or can use Janino (an open-source, embeddable, Java compiler)
      - Hosted at www.janino.net
  - Finally, use `Class.forName()` to load the compiled code

Uses for runtime code generation

- Runtime code generation is used…
- By JSP (Java Server Pages)
  - To generate servlets from .jsp files
- By IDEs and debuggers
  - To evaluate Java expressions entered by user

Uses for Java bytecode manipulation

- Compilers:
  - Write a compiler for a scripting language and generate Java bytecode
    - Result: out-of-the-box integration between Java and the language
  - Groovy (groovy.codehaus.org) uses this technique
- Optimization:
  - Read a .class file, optimize bytecode and rewrite the .class file
- Code analysis:
  - Read a .class file, analyze bytecode and generate a report
- Code obfuscation:
  - Mangle names of methods and fields in .class files
- Aspect-oriented programming (AOP):
  - Modify bytecode to insert “interception” code
  - Generate proxies for classes or interfaces
  - Spring uses this technique

Tools for bytecode manipulation

- Example open-source projects for bytecode manipulation:
  - ASM (http://asm.objectweb.org/)
  - BCEL (http://jakarta.apache.org/bcel/)
  - SERP (serp.sourceforge.net)
- CGLIB (Code Generation LIBrary):
  - Built on top of BCEL
  - Provides a higher-level API for generating dynamic proxies
  - Used by other tools, such as Spring and Hibernate

3. Summary

Summary

- A *lot* of tools use Java reflection:
  - Plugins to extend functionality of an application (Ant)
  - Auto-completion in Java editors and IDEs
  - Use naming conventions of methods to infer semantics (JUnit test methods)
  - Tie components together (Spring)
  - Compile-time code generation
  - Runtime code generation
    - Generate proxies
    - Generate servlets from a markup language (JSP)
  - Evaluate Java expressions entered interactively by a user