MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

Software Design and Construction 159.251

Organising Code

Amjed Tahir
a.tahir@massey.ac.nz

Original author: Jens Dietrich

References

[CC] Robert Martin:
Clean Code: A Handbook of Agile Software Craftsmanship.
Prentice Hall 2009.

[EJ] Joshua Bloch:
Effective Java Second Edition.
Sun Micro 2008.

Summary

- documentation
- case study: reverse engineering UML with yDoc
- formatting
- generating code, templating
- case study: velocity
- naming artefacts
- annotating artefacts
- case study: reflection
- measuring code size and complexity
- project layouts

Comments

- the purpose of the comment is to explain code
- however, code should be self-explanatory – means that it should explain itself (and therefore self-documenting)
- adding comments to self-documenting code creates redundancies (and the evils of duplication)

Legal Comments

- often required by law or corporate rules
- in open source software, copyright and license used should be documented
- copyright court cases could come down to figuring out whether a programmer knew what the legal status of a piece of source code was
- see http://goo.gl/kk0zh for a famous copyright court case: Oracle vs Google about the use of Java in Android
- legal comments should be placed on top of each source unit
- source unit = compilation unit (java file), but also other text files like xml configuration files etc

Legal Comments Example

```
/*
 * Licensed to the Apache Software Foundation (ASF) under one or more
 * contributor license agreements.  See the NOTICE file distributed with
 * this work for additional information regarding copyright ownership.
 * The ASF licenses this file to You under the Apache License, Version 2.0
 * (the "License"); you may not use this file except in compliance with
 * the License.  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
...
 */
package org.apache.log4j;
import org.apache.log4j.spi.Filter;
...
```

in org.apache.log4j.Appender

Reasons to Comment

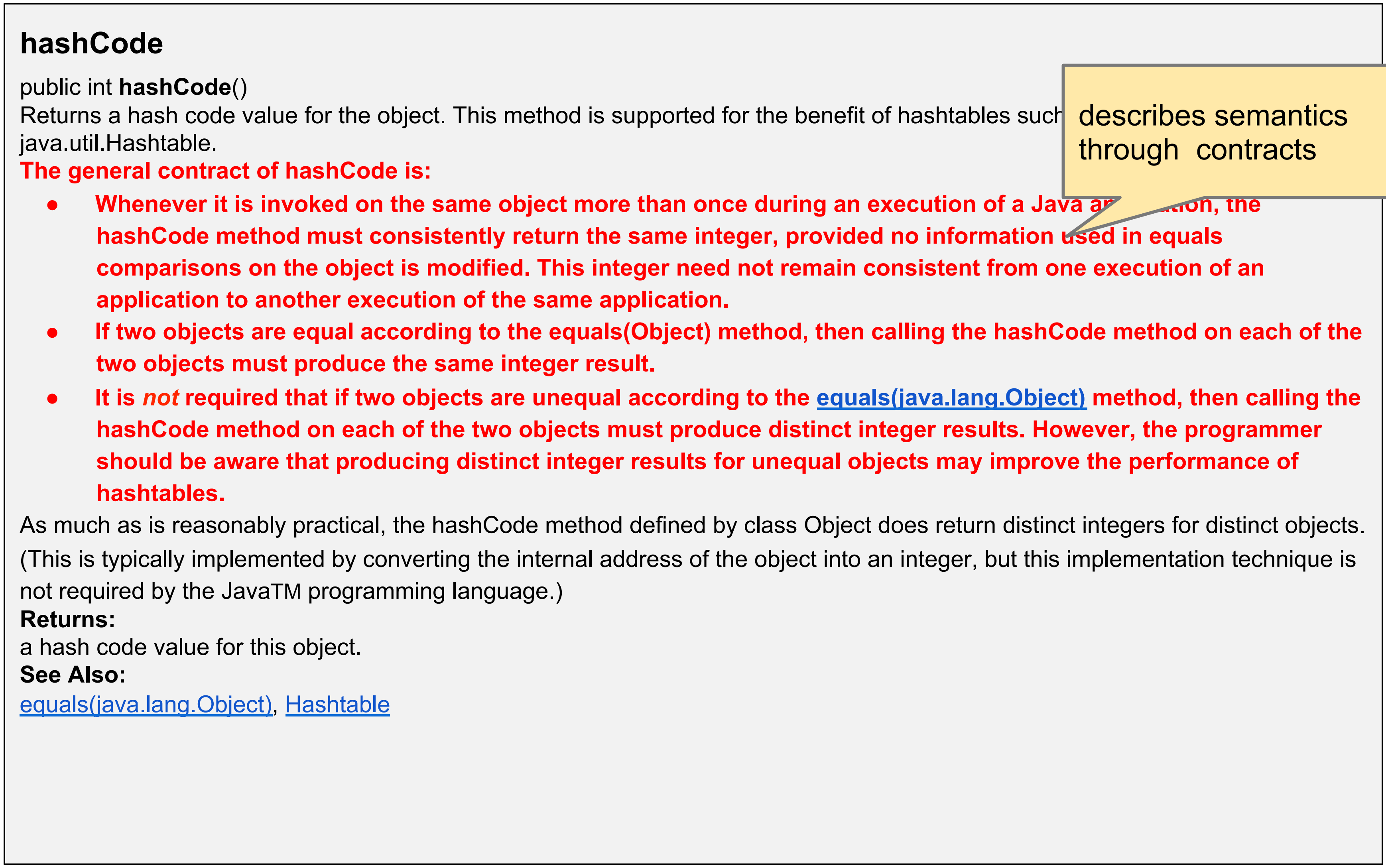
- Comments are used to explain
  - **intent** if names (of classes, methods etc) cannot express this
  - **semantics** (=**meaning**) if names cannot express it
- this could include pre- and post conditions, but it is preferable to use code for this as well (e.g., assert statements)

Example: Comment Intention

```
public void StressTest {
      /**
       * Create a list of test objects large enough to
       * trigger garbage collection when passed to the
    * method to be tested.
       */
      private List<Object> createLargeList() {
      ...
   }
   ...
}
```

describes the intention of this method, and the semantics (meaning) of large in this context

Example: Describing Semantics



hashCode

public int **hashCode**()
Returns a hash code value for the object. This method is supported for the benefit of hashtables such
java.util.Hashtable.
The general contract of hashCode is:

- **Whenever it is invoked on the same object more than once during an execution of a Java ap ation, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.**
- **If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.**
- **It is *not* required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.**

As much as is reasonably practical, the hashCode method defined by class Object does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the JavaTM programming language.)
Returns:
a hash code value for this object.
See Also:
equals(java.lang.Object), Hashtable

describes semantics through contracts

http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#hashCode()

Example: Describing Semantics

```
public class MyClass implements Cloneable {
        /**
         * Clones the object, returns an instance of MyClass.
         * This is a deep copy!
         */
    public Object clone() {
        ...
    }
}
```

useful information that is not obvious (from the method name)!

Example: Describing Semantics

```
public interface TaxCalculator{
         /**
          * Calculates the income tax for non-negative incomes.
      * @param income a non-negative value
      * @throws IllegalArgumentException if income is negative
      * @return a value between 0 and 100
          */
    public double calculateIncomeTax(double income);
}
```

precondition

postconditions

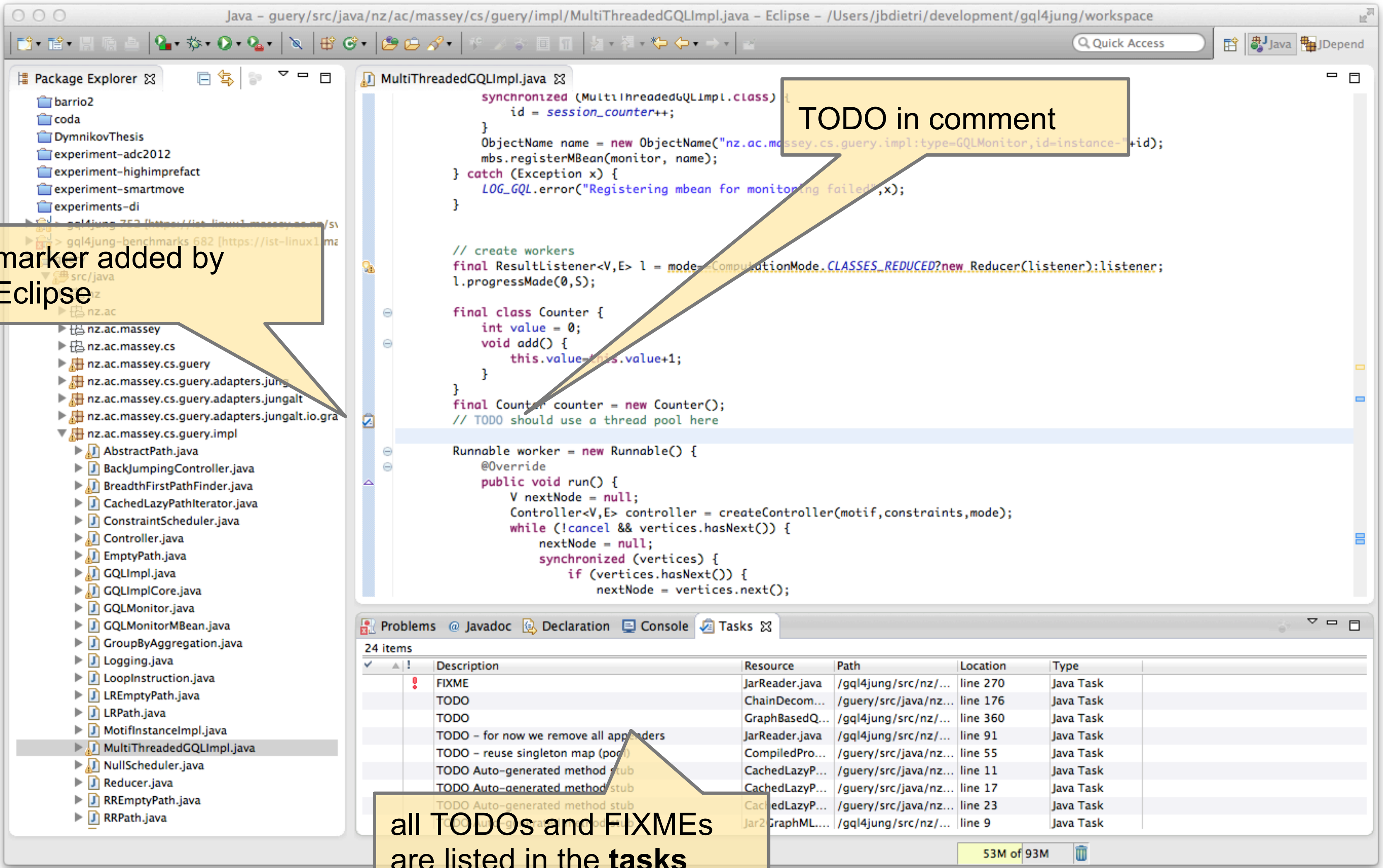
Redundant Comments

- history comments list changes (what, who, why) in code - this is better done in versioncontrol systems (SVN, GIT etc)!
- the same applies to author and version information
- some version control systems* can replace tags during commit: e.g., in "version: $revision$", the variable $revision$ would be replaced by the revision number generated by the version control system

Use Comments to Managing Technical Debt

- common practise: use **TODO** and **FIXME** in code to document issues and improvements in the code.
- while this is useful, it should not replace the use of issue (bug) tracking systems!
- IDEs like Eclipse pick up these tags, and manage them - e.g., the are highlighted in the editor and listed in the task view

Example: TODOs and FIXMEs in Eclipse



Java – guery/src/java/nz/ac/massey/cs/guery/impl/MultiThreadedGQLImpl.java – Eclipse – /Users/jbdietri/development/gql4jung/workspace

Package Explorer

- barrio2
- coda
- DymnikovThesis
- experiment-adc2012
- experiment-highimprefact
- experiment-smartmove
- experiments-di
- gql4jung 752 [https://ist-linux1.massey.ac.nz/s
- gql4jung-benchmarks 682 [https://ist-linux1.ma
- src/java
- nz
- nz.ac
- nz.ac.massey
- nz.ac.massey.cs
- nz.ac.massey.cs.guery
- nz.ac.massey.cs.guery.adapters.jung
- nz.ac.massey.cs.guery.adapters.jungalt
- nz.ac.massey.cs.guery.adapters.jungalt.io.gra
- nz.ac.massey.cs.guery.impl
  - AbstractPath.java
  - BackJumpingController.java
  - BreadthFirstPathFinder.java
  - CachedLazyPathIterator.java
  - ConstraintScheduler.java
  - Controller.java
  - EmptyPath.java
  - GQLImpl.java
  - GQLImplCore.java
  - GQLMonitor.java
  - GQLMonitorMBean.java
  - GroupByAggregation.java
  - Logging.java
  - LoopInstruction.java
  - LREmptyPath.java
  - LRPath.java
  - MotifInstanceImpl.java
  - MultiThreadedGQLImpl.java
  - NullScheduler.java
  - Reducer.java
  - RREmptyPath.java
  - RRPath.java

MultiThreadedGQLImpl.java

```
                synchronized (MultiThreadedGQLImpl.class) {
                    id = session_counter++;
                }
                ObjectName name = new ObjectName("nz.ac.massey.cs.guery.impl:type=GQLMonitor,id=instance-"+id);
                mbs.registerMBean(monitor, name);
            } catch (Exception x) {
                LOG_GQL.error("Registering mbean for monitoring failed",x);
            }


            // create workers
            final ResultListener<V,E> l = mode==ComputationMode.CLASSES_REDUCED?new Reducer(listener):listener;
            l.progressMade(0,S);

            final class Counter {
                int value = 0;
                void add() {
                    this.value=this.value+1;
                }
            }
            final Counter counter = new Counter();
            // TODO should use a thread pool here

            Runnable worker = new Runnable() {
                @Override
                public void run() {
                    V nextNode = null;
                    Controller<V,E> controller = createController(motif,constraints,mode);
                    while (!cancel && vertices.hasNext()) {
                        nextNode = null;
                        synchronized (vertices) {
                            if (vertices.hasNext()) {
                                nextNode = vertices.next();
```

TODO in comment

marker added by Eclipse

Problems | Javadoc | Declaration | Console | Tasks

24 items

✓	!	Description	Resource	Path	Location	Type
	!	FIXME	JarReader.java	/gql4jung/src/nz/...	line 270	Java Task
		TODO	ChainDecom...	/guery/src/java/nz...	line 176	Java Task
		TODO	GraphBasedQ...	/gql4jung/src/nz/...	line 360	Java Task
		TODO – for now we remove all appenders	JarReader.java	/gql4jung/src/nz/...	line 91	Java Task
		TODO – reuse singleton map (pool)	CompiledPro...	/guery/src/java/nz...	line 55	Java Task
		TODO Auto-generated method stub	CachedLazyP...	/guery/src/java/nz...	line 11	Java Task
		TODO Auto-generated method stub	CachedLazyP...	/guery/src/java/nz...	line 17	Java Task
		TODO Auto-generated method stub	CachedLazyP...	/guery/src/java/nz...	line 23	Java Task
		[illegible]	Jar2GraphML....	/gql4jung/src/nz/...	line 9	Java Task

53M of 93M

all TODOs and FIXMEs are listed in the **tasks view**

Generating Documentation

- tools that generate documentation from comments found in source code and (optional) additional information
- different output formats, popular choices are HTML (many hyperlinked documents) and PDF (for printing)
- **examples**: javadoc, ndoc, jsdoc, doxygen

- most tools work only for one particular programming languages (Java, Python etc),
  - Doxygen supports multiple languages
   Mainly for C++, but supports other popular programming languages (such as C, Objective-C, C#, PHP, Java, Python)

javadoc

- original document generator for the Java
- generates linked HTML documents by default, but this can be changed by configuration
- interprets tags: annotated source code elements such as @version, @author, @exception etc
- has a built-in plugin model:
    - **doclets** can be used to inspect parsed code and comments and generate documents
    - **taglets** are an API to add custom tags

javadoc (ctd)

- for each class, one HTML page is created
- internal links (anchors) are generated for class members
- overview pages for packages and indices are generated
- many libraries have their generated javadoc pages online, including the JDK itself

javadoc Example

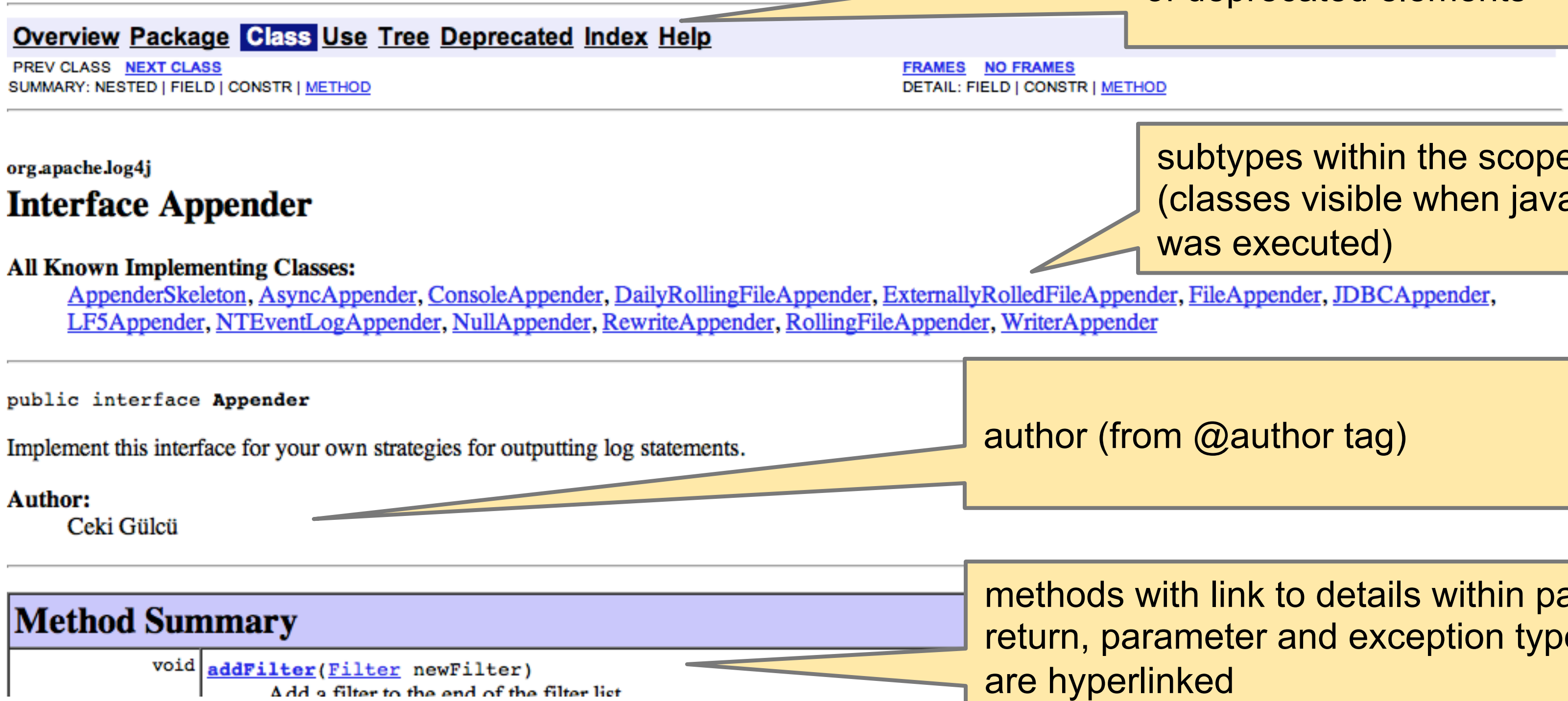
```
package org.apache.log4j;
import org.apache.log4j.spi.Filter;
import org.apache.log4j.spi.ErrorHandler;
import org.apache.log4j.spi.LoggingEvent;

/**
   Implement this interface for your own strategies for outputting log
   statements.
   @author Ceki G&uuml;lc&uuml;
*/
public interface Appender {
  /**
     Add a filter to the end of the filter list.
     @since 0.9.0
   */
  void addFilter(Filter newFilter);
```

@author tag - the author

@version tag - the version when this method was added

javadoc Example (ctd)



Overview Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.apache.log4j
Interface Appender

All Known Implementing Classes:
AppenderSkeleton, AsyncAppender, ConsoleAppender, DailyRollingFileAppender, ExternallyRolledFileAppender, FileAppender, JDBCAppender, LF5Appender, NTEventLogAppender, NullAppender, RewriteAppender, RollingFileAppender, WriterAppender

```
public interface Appender
```

Implement this interface for your own strategies for outputting log statements.

Author:
Ceki Gülcü

Method Summary

```
void addFilter(Filter newFilter)
```

links to index and overview pages, and lists of deprecated elements

subtypes within the scope (classes visible when javadoc was executed)

author (from @author tag)

methods with link to details within page, return, parameter and exception types are hyperlinked

http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/Appender.html

javadoc Important Tags

@author	the author
@version	the current version of this class
@since	the version when this class/method/field was added
@see	explicit reference to another model element, a link will be created
@param	describes a method parameter
@return	describes a method return type
@exception or @throws	describe an exception that is thrown by this method
@deprecated	deprecated code

javadoc Integrations

- command line (command javadoc is in the jdk bin folder)

- IDE
- **<u>To generate</u>** - Eclipse: Project > Generate javadoc
  This will then show under %project%\doc directory.
- **<u>To export</u>**- Eclipse: right click on project > Export > Java > javadoc

- ANT: built-in [javadoc]() task

Running javadoc from a terminal

- run javadoc without parameters for full list of options

```
javadoc options package_or_class_list
```

- main options:
  - -d destination folder
  - -sourcepath list of source path folders
- options can be put into a file (e.g. options.txt), and javadoc can be invoked as follows:

```
javadoc @options.txt package_or_class_list
```

Application: UML Reverse Engineering

- usually, code is generated from UML models
- yWorks(previously known as yDoc) is a doclet that allows to **reverse** this: UML class diagrams are generated from code
- the diagrams are navigable: visual elements representing artefacts within the scope can be clicked to navigate
- yWorks is a commercial product, but there is a free community edition
- source code: https://bitbucket.org/jensdietrich/oop-examples/src/1.0/java2uml/

Example: javadoc config file for yWorks

```
-d uml
-docletpath <yuml path>/lib/ydoc.jar:<yuml path>/
resources/:./bin
-doclet ydoc.doclets.YStandard
-sourcepath ./src
-classpath bin
-umlautogen
org.apache.log4j
org.apache.log4j.chainsaw
org.apache.log4j.config
org.apache.log4j.helpers
...
```

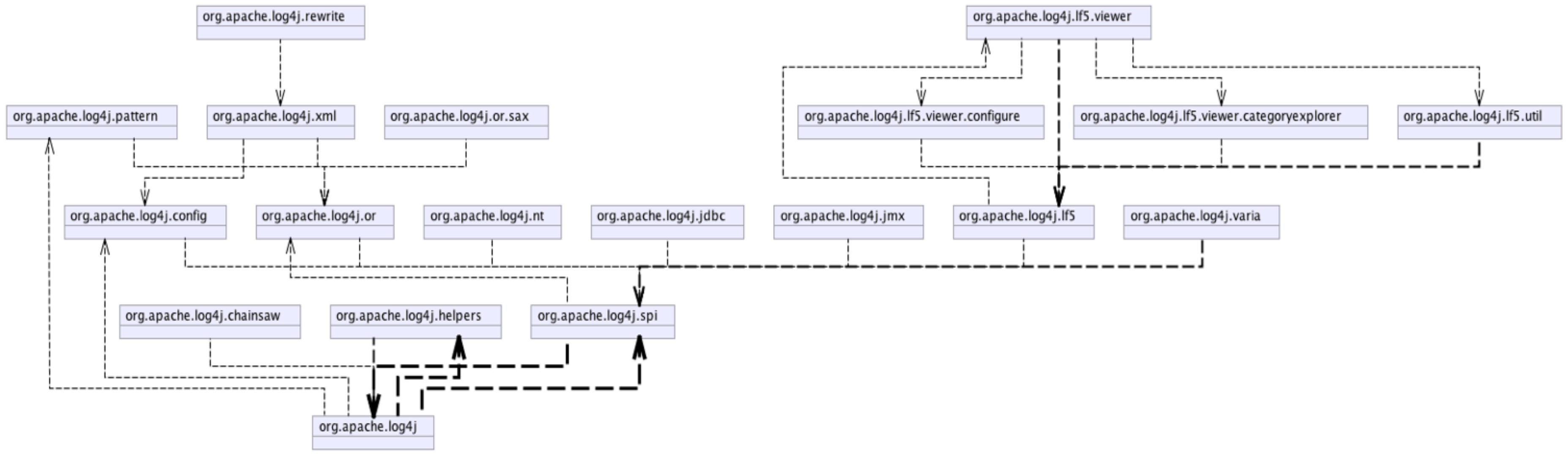
<yuml path> .. folder where yuml is installed

use yuml doclet

generate uml!

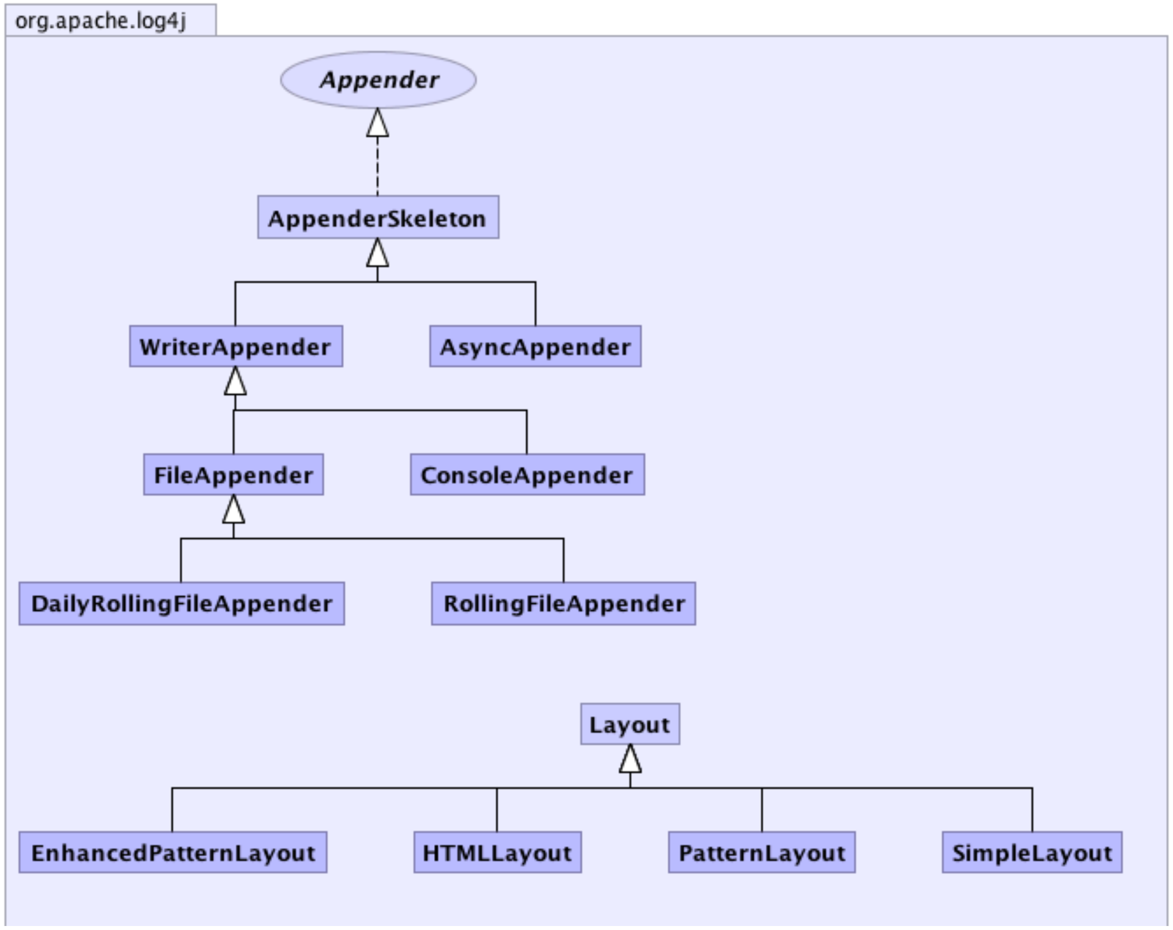
list of packages for which to generate javadoc starts here

Example: yWorks-generated Diagram for log4j (overview)



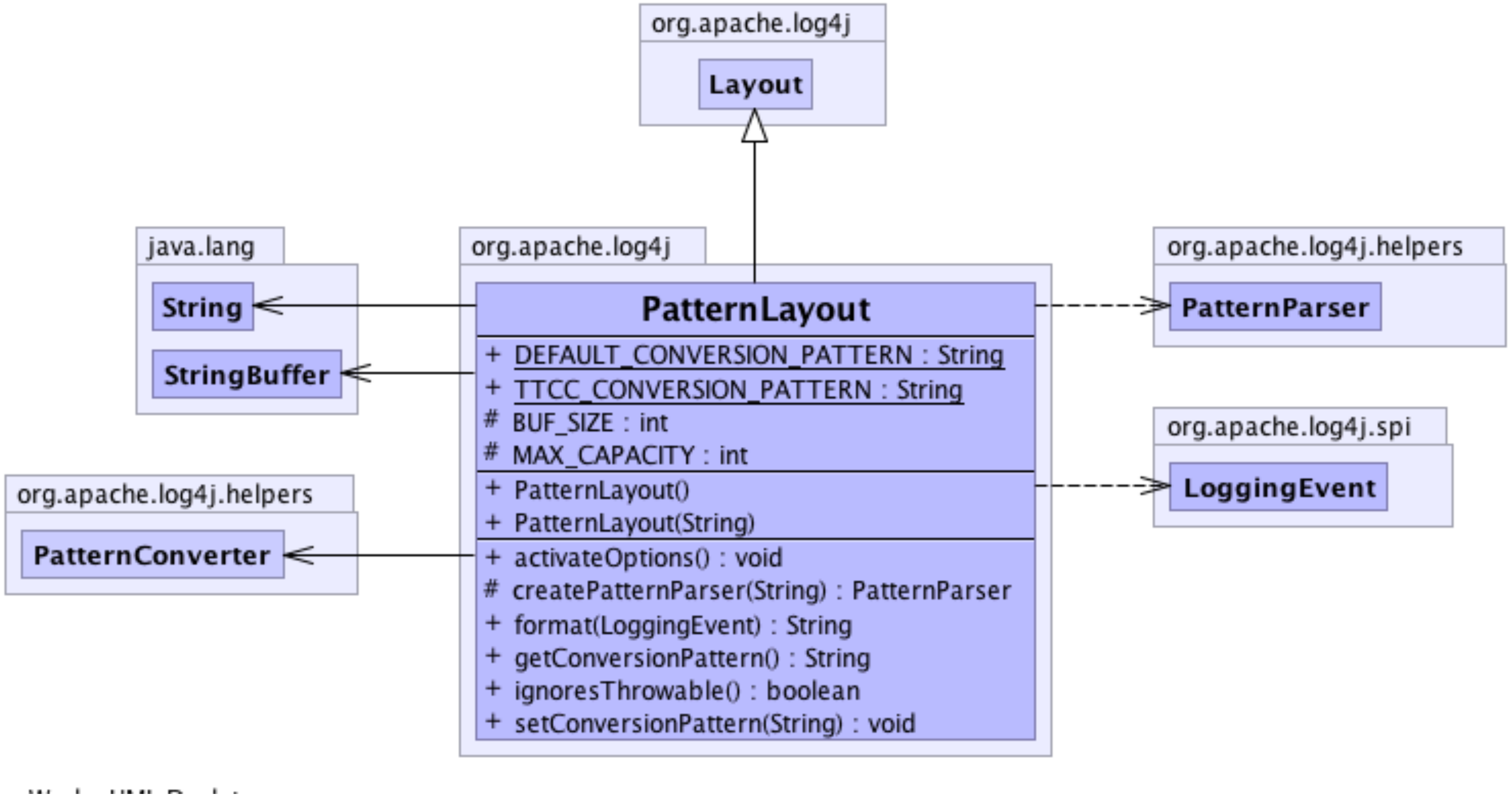
log4j packages and their relationships - diagram generated with yWorks

Example: yWorks-generated Class Diagram for package org.apache.log4j



types in org.apache.log4j and their relationships (figure is incomplete) - diagram generated with yWorks

Example: yWorks-generated Class Diagram for PatternLayout



org.apache.log4j
Layout
java.lang
String
StringBuffer
org.apache.log4j
PatternLayout
+ DEFAULT_CONVERSION_PATTERN : String
+ TTCC_CONVERSION_PATTERN : String
# BUF_SIZE : int
# MAX_CAPACITY : int
+ PatternLayout()
+ PatternLayout(String)
+ activateOptions() : void
# createPatternParser(String) : PatternParser
+ format(LoggingEvent) : String
+ getConversionPattern() : String
+ ignoresThrowable() : boolean
+ setConversionPattern(String) : void
org.apache.log4j.helpers
PatternParser
org.apache.log4j.spi
LoggingEvent
org.apache.log4j.helpers
PatternConverter

class PatternLayout and its relationships - diagram generated with yWorks

XDoclet

- xdoclet is an (older) project that uses doclets and taglets to add meta data
- example: annotate classes with information how to map them to relational database tables
- made redundant by the Java annotation API (to be discussed)

```
/**
 * @jmx.mbean
 *     name="BlogManager"
 */
public class BlogManager implements BlogManagerMBean {
  BlogFacadeHome home;
...
```

xdoclet defines the meaning of additional tags, this is used to generate code or to provide additional services at runtime

Example from: Craig Walls and Norman Richards: XDoclet in Action. Manning, 2003.

Formatting Code

- layout code to improve readability and better **communicate** the structure of code
- code is a tree like structure (the so-called Abstract Syntax Tree (AST))
- code formatting should reflect this
- **vertical formatting**: use empty lines to separate concepts, restrict size of artefacts
- **horizontal formatting**: manage lengths of lines, use **indentation**
- many code editors use tree controls (collapse and expand branches) to hide parts of the tree.
- try to make similar methods and function together (close to each other).

Vertical and Horizontal Formatting

```
public class EvenNumberChecker {
        public static boolean isEven(int i) {
                if (i%2==0) {
                        return true;
                }
                else {
   -                    return false;
                }
        }

        public static boolean isOdd(int i) {
                return !isEven(i);
        }
}
```

many IDEs have controls to collapse/expand branches

vertical formatting:
an empty line is used to separate methods

horizontal formatting:
tabs/white spaces are used to indent code to reveal logical structure

Generated AST (with PMD)

```
<?xml version="1.0" encoding="UTF-8"?>
<CompilationUnit beginColumn="1" beginLine="1" comments="[]" endColumn="1" endLine="14">
    <TypeDeclaration beginColumn="1" beginLine="1" endColumn="1" endLine="14">
        <ClassOrInterfaceDeclaration abstract="false" beginColumn="8" beginLine="1"  ..>
            <ClassOrInterfaceBody beginColumn="32" beginLine="1" endColumn="1" endLine="14">
                <ClassOrInterfaceBodyDeclaration anonymousInnerClass="false" ..
                                    ...
```

- PMD can generate an AST file
- the AST is encoded using XML
- XML can be displayed in a web browser to show its hierarchical structure

Formatting Conditionals

```
if (i%2==0)
System.out.println("even number");
System.out.println("do something else");
```

bad: it is not clear whether the second println is executed when i is even!

```
if (i%2==0)
	System.out.println("even number");
System.out.println("do something else");
```

better: indentation makes this clear

Formatting Conditionals (ctd)

```
if (i%2==0)
      System.out.println("even number");
      System.out.println("do something else");
```

very bad: misleading, this suggests that the second println is executed when i is even!

```
if (i%2==0) {
      System.out.println("even number");
}
System.out.println("do something else");
```

best: explicit {} used to define scope of conditional !

Formatting Rules

```
if (i%2==0)
{
        System.out.println("even number");
}
System.out.println("do something else");
```

```
if (i%2==0) {
        System.out.println("even number");
}
System.out.println("do something else");
```

- Which style is better?

Formatting Rules

```
if (i%2==0)
{
	System.out.println("even number");
}
System.out.println("do something else");
```

GNU style

```
if (i%2==0) {
	System.out.println("even number");
}
System.out.println("do something else");
```

BSD style

- for languages with C-like syntax (C,C++,C#, Java, JavaScript etc), two standard rule sets exist: GNU and BSD
- it is easy to add your own rules, in Eclipse: Preferences > Java > Code Style > Formatter

Code Beautifiers

- aka **pretty printers**
- tools that can apply formatting rules to code
- based on AST built from code
- easy to implement
- in Eclipse:
  - source > correct indentation
  - source > format code

Code Uglifiers ?!

- main idea: pretty printing uses additional characters, remove those to compact source code

- use case: source code is interpreted, and has to be send over the network: JavaScript!
- uglifiers are therefore mainly compression tools (i.e., try to make the program shorter)
- in javascript, they are sometimes called **JS compilers**
- uglifiers remove all unnecessary characters incl. comments
- examples: google closure compiler, YUI compressor

Uglified Code Example

```
(function(a,b){function cy(a){return f.isWindow(a)?
a:a.nodeType===9?a.defaultView||a.parentWindow:!1}function
cu(a){if(!cj[a]){var b=c.body,d=f("<"+a
+">").appendTo(b),e=d.css("display");d.remove();if(e==="non
e"||e===""){ck||
(ck=c.createElement("iframe"),ck.frameBorder=ck.width=ck.he
ight=0),b.appendChild(ck);if(!cl||!
ck.createElement)cl=(ck.contentWindow||
ck.contentDocument).document,cl.write((f.support.boxModel?"
<!doctype html>":"")
+"<html><body>"),cl.close();d=cl.createElement(a),cl.body.a
ppendChild(d),e=f.css(d,"display"),b.removeChild(ck)}
cj[a]=e}return cj[a]}function ...
```

- this is jquery-1.7.2.min.js - uglified version of the popular jquery JavaScript library
- compression ratio is : 0.375 :
  - jquery-1.7.2.min.js .. 95KB
  - jquery-1.7.2.js .. 253KB

Code Obfuscators

- closely related to uglifiers
- main goal is not to compress code, but to make it less readable and difficult to reverse engineer
- often used on compiled code, try to make reverse engineering difficult to protect intellectual property (IP)
- uses identifier (variable, class and method names) renaming
- removes comments and line number information
- good free obfuscators: yGuard, ProGuard

Generating Code With Eclipse

- most IDEs have a comprehensive set of **code generators**
- Eclipse uses **templates** to generate classes, methods and code snippets
- these templates can be customised
- code generation is used to **automate** and **speed up** coding

What (not) to expect

- some code patterns are sophisticated and guarantee:
  - **correctness (semantics)**
  - **reasonable performance** - this is sometimes difficult to achieve with manual coding
- example: **<u>generate</u>** ***equals*** and ***hashCode***
  - generated code honors contract (if objects are equal hashcode must be the same) - this guarantees that objects will work with hash maps and sets
  - in general, low probability of hash code collisions with generated code - reasonable performance
- setters and getters comply with standard naming patterns, and tools using them will work (e.g., XML persistency, UI builders)
- **not guaranteed: optimised performance**

Standard Method Generators in Eclipse

- method generators are in the source menu of the context menu
- standard set:
  - override/implement method
  - getters and setters
  - delegate method
  - `hashCode` and `equals`
  - `toString`
  - constructor using fields
  - constructor using superclass
- additional plugins may add more generators

Generating Code .. Example

```
public class Person {
       private int id = 0;
       private String name = "";
}
```

start here: simple class with fields, **manually** created

Example: Generating Constructors

```
public class Person {
        private int id = 0;
        private String name = "";
    public Person(int id, String name) {
        super();
        this.id = id;
        this.name = name;
    }
   public Person() {
    super();
    }
}
```

- apply "generate constructors using fields"
- note: class should still have overloaded constructor without parameter to comply with "Java Bean" standard required for tools like XML decoder or UI builder

Example: Generating Setters & Getters

```
public class Person {
        private int id = 0;
        private String name = "";
        ..
        public int getId() {
                return id;
        }
        public void setId(int id) {
                this.id = id;
        }
        public String getName() {
                ...
}
```

apply "generate setters and getters"

Example: Generating `toString`

```
public class Person {
        private int id = 0;
        private String name = "";
        ..
        @Override
        public String toString() {
                return "Person [id=" + id + ", name=" + name + "]";
        }
}
```

warning: this code is relatively slow as it uses string concatenation. A better implementation would use a StringBuffer!
This used to be a big problem in earlier Java releases

- apply "generate toString"
- facilitates debugging
- consider **`System.out.println(new Person(42,"James Gosling"))`**
- with *inherited* `toString()` this will print (something like):
  `Person@6345e044`
- with generated method, this will print:
  `Person [id=42, name=James Gosling]`

Example: Generating `equals`

```
public class Person {
          private int id = 0;
          private String name = "";
          ..
          @Override
          public boolean equals(Object obj) {
                    if (this == obj) return true;
                    if (obj == null)return false;
                    if (getClass() != obj.getClass()) return false;
                    Person other = (Person) obj;
                    if (id != other.id) return false;
                    if (name == null) {
                              if (other.name != null) return false;
                    } else if (!name.equals(other.name)) return false;
                    return true;
          }
}
```

if obj and this is the same object, they are equal

null is not equal to this object

if obj is of a different type, they cannot be equal (comparing "apples and oranges"!)

now we know that obj is of type Person, so we can safely cast

if name doesn't match, objects are not equal

if id doesn't match, objects are not equal

all conditions that would mean that objects aren't equal have been checked - object are equal!

generated together with hashCode to ensure consistency!

Example: Generating `hashCode`

```
public class Person {
        private int id = 0;
        private String name = "";
        ..
        @Override
        public int hashCode() {
                final int prime = 31;
                int result = 1;
                result = prime * result + id;
                result = prime * result + ((name == null) ? 0 : name.hashCode());
                return result;
        }

}
```

this code uses a heuristics to ensure that different objects are likely to have different hash codes
this means that they will be stored in different buckets in hash maps!

Code Snippets in Eclipse

- code snippets allow to generate common code patterns
- list: Preferences > Java > Editor > Templates
- templates are activated/triggered by typing the template name followed by ctrl-space

Example: merge arrays

```
String[] arr1 = new String[]{"one","two","three"};
String[] arr2 = new String[]{"four","five","size"};

String[] arr12 = new String[arr1.length +
arr2.length];
System.arraycopy(arr1, 0, arr12, 0, arr1.length);
System.arraycopy(arr2, 0, arr12, arr1.length,
arr2.length);
```

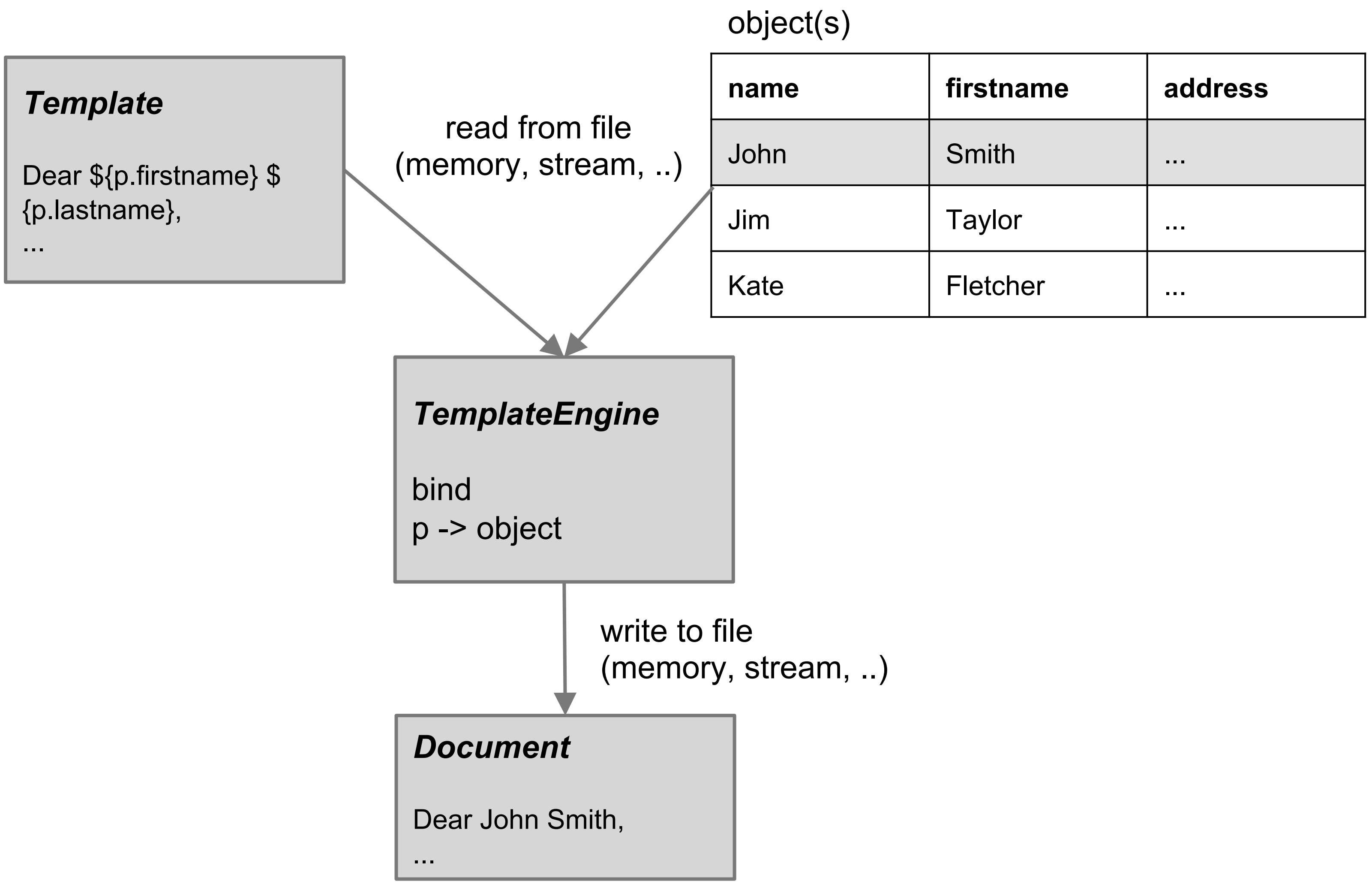
code generated by template:
type **arraymerge ctrl+space**

Inside Code Templates

```
${array_type}[] ${result:newName(array1)} = new ${array_type}[$
{array1:array}.length + ${array}.length];
System.arraycopy(${array1}, 0, ${result}, 0, ${array1}.length);
System.arraycopy(${array}, 0, ${result}, ${array1}.length, ${array}.length);
```

- the template definition defines the code to be generated with some **variables**
- variable syntax: ${name}
- when the actual code is generated, these variables are replaced ("**bound**","**instantiated**") by identifiers (variable names) found in the code
- the general term for this technique is **templating**

Template Engines



Template
Dear ${p.firstname} ${p.lastname},
...
read from file (memory, stream, ..)
object(s)
TemplateEngine
bind
p -> object
write to file (memory, stream, ..)
Document
Dear John Smith,
...

name	firstname	address
John	Smith	...
Jim	Taylor	...
Kate	Fletcher	...

Template Engines Example

- office packages (MS Office, Libre/Open Office): “mail merge” feature
- general purpose template engines for Java:
  - velocity
  - mvel
  - stringtemplate
- server page technology such as JSP and ASP is based on templates
- applications:
  - generate code
  - generate reports
  - generate web pages

Example: Velocity - template

```
To $student.firstName $student.name ,

Our records show that you are enrolled in the following
degree:

      $student.degree

with the major(s):

#foreach( $major in $student.majors )
      $major
#end
```

references the student variable and its properties

supports procedural elements such as loops and conditionals

source code: https://bitbucket.org/jensdietrich/oop-examples/src/1.0/templating/

Example: Velocity - API

```
Student s = ... ;
```

the object that will be used to instantiate the template

```
VelocityContext context = new VelocityContext();
context.put( "student", s);
```

bind the variable "student" to s

```
Template template = Velocity.getTemplate("letter.vm");
```

load template (parsing, caching)

```
FileWriter out = new FileWriter("letter1.txt");
template.merge( context, out );
```

instantiate template

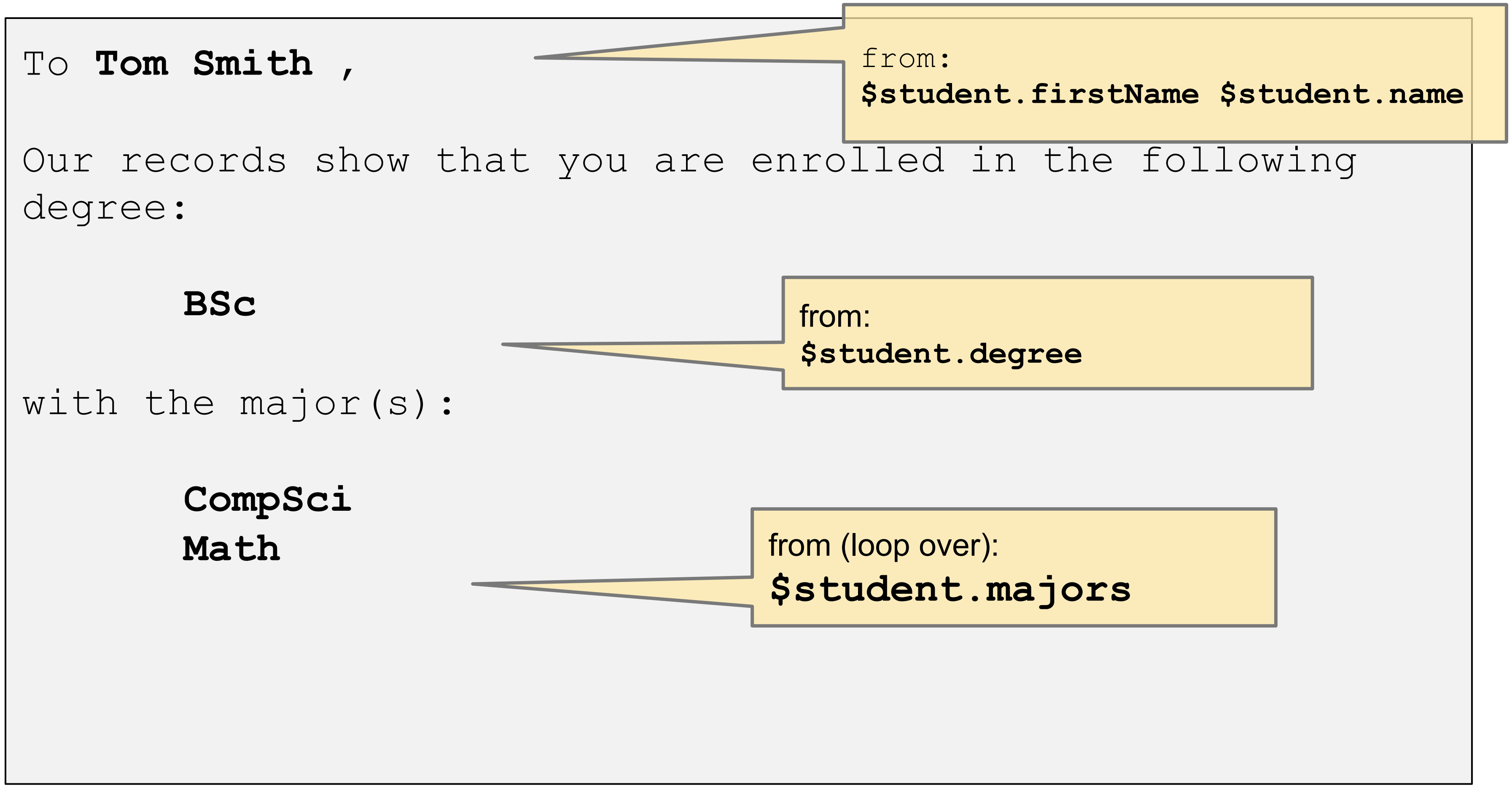
```
out.close();
```

clean up

template file

Vm is a velocity file extension. See here https://fileinfo.com/extension/vm

Example: Velocity - result



```
To Tom Smith ,

Our records show that you are enrolled in the following
degree:

      BSc

with the major(s):

      CompSci
      Math
```

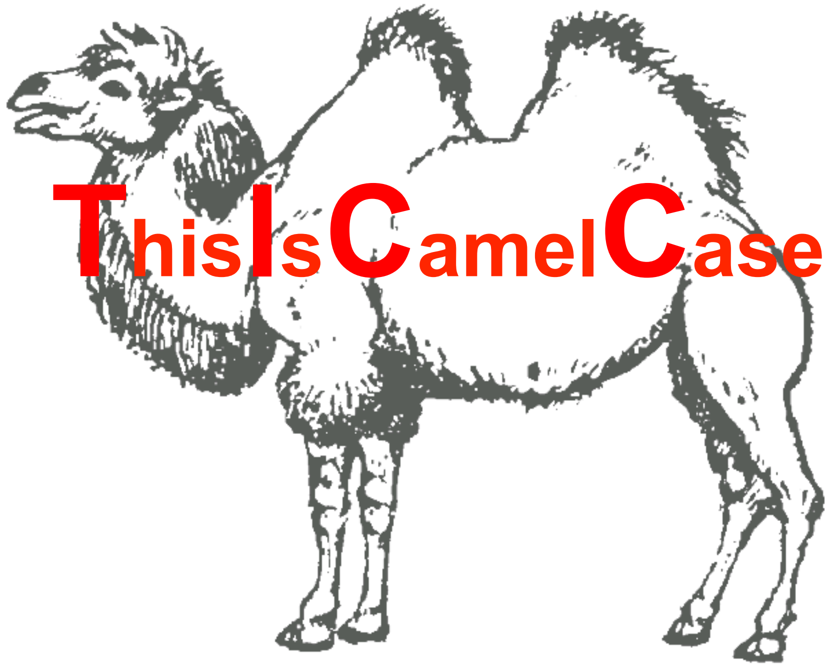
from:
$student.firstName $student.name
from:
$student.degree
from (loop over):
$student.majors

output: letter1.txt

Naming Artefacts

- names should reveal the intention
- often, with good descriptive names, no comment is needed
- there is usually no reason to worry about long names for performance (space) reasons
- in rare cases where this is a concern (networking!), uglifiers or obfuscators can be used to shorten names before code is deployed
- short non-descriptive names ("x","i") should only be used for local variables (e.g., to control loops)
- compound names are more descriptive

Camel Case



- syntax used for compound names
- merged into one identifier, but structure should remain recognisable, to avoid: **thisisnotcamelcase**
- made popular by Smalltalk
- used in Java, replaces the older convention used in C using underscores: **this_is_not_camel_case**

Grammar and Artefacts

- the principles of OO are aligned with the grammar of the English languages
    - classes and objects = nouns
    - methods = verbs
    - fields = attributes or nouns if another class is referenced
- the naming of the respective artefacts should reflect this: a class "Student" (noun) with a method "enroll" (verb) and a property "name" (noun)
- example: Comparator, and not Compare
- example: Iterator, and not Iterate
- sometimes, classes with only a main method can be treated like functions, and names can reflect this

Names and Types

- names and types should be consistent
- examples:
  - **Student student**, and not **studentInfo**
  - **List<Student> students** or **studentList** or **listOfStudents**, and not **student** or **studentset**

Naming Conventions

- naming conventions should be used
- some naming conventions add semantics (meaning) to code
- tools may depend on this meaning
- this is called "convention over configuration" (coc)
- coc killer application: RubyOnRails
- example: the JavaBean model: http://www.oracle.com/technetwork/java/javase/index-137642.html

Convention over Configuration

Convention over configuration (also known as **coding by convention**) is a software design paradigm which seeks to decrease the number of decisions that developers need to make, gaining simplicity, but not necessarily losing flexibility. The phrase essentially means a developer only needs to specify unconventional aspects of the application. ...

http://en.wikipedia.org/wiki/Convention_over_configuration

The JavaBean Model (simplified)

- JavaBeans are simple Java classes (aka plain old Java objects - POJOs)
- properties have matching pairs of setters (set*) and getters (get*) (aka accessors and mutators)
- JavaBean classes must have public constructors without parameters
- advanced:
  - naming patterns for events / event sources
  - support for property change events
  - BeanInfo classes to attach additional information to beans

A Bean (aka Plain Old Java Object)

```
public class Lecturer {

        public Lecturer() {
                super();
    }
        private java.util.Date dob = null;
        ..

        public java.util.Date getDob() {
                return dob;
        }

        public void setDob(java.util.Date dob) {
                this.dob = dob;
        }
        ..
}
```

public constructor without parameters (can be omitted only if no other constructor is defined)

matching pairs of setters and getters: types and names must match

Reflection

- reflection is the ability of a computer program to **access** and **modify** its structure and behaviour **at runtime**
- Java has built-in reflection facilities: classes like java.lang.Class and java.lang.reflect.Method
- the JavaBean model facilitates reflection
- utility: java.beans.Introspector
- *full discussion in other papers (159.707)*

Reflection Examples

```
public static void main(String[] args) throws Exception
        Lecturer s = new Lecturer();
        inspect(s);
}
public static void inspect(Object obj) throws Exception
        BeanInfo beanInfo = Introspector.getBeanInfo(obj.ge        s());
        PropertyDescriptor[] properties = beanInfo.getPrope    escriptors();
        for (PropertyDescriptor property:properties) {
                System.out.print(property.getName());
                Method getter = property.getReadMethod();
                Object value = getter.invoke(obj,new O
                System.out.print(" = ");
                System.out.println(value);
        }
}
```

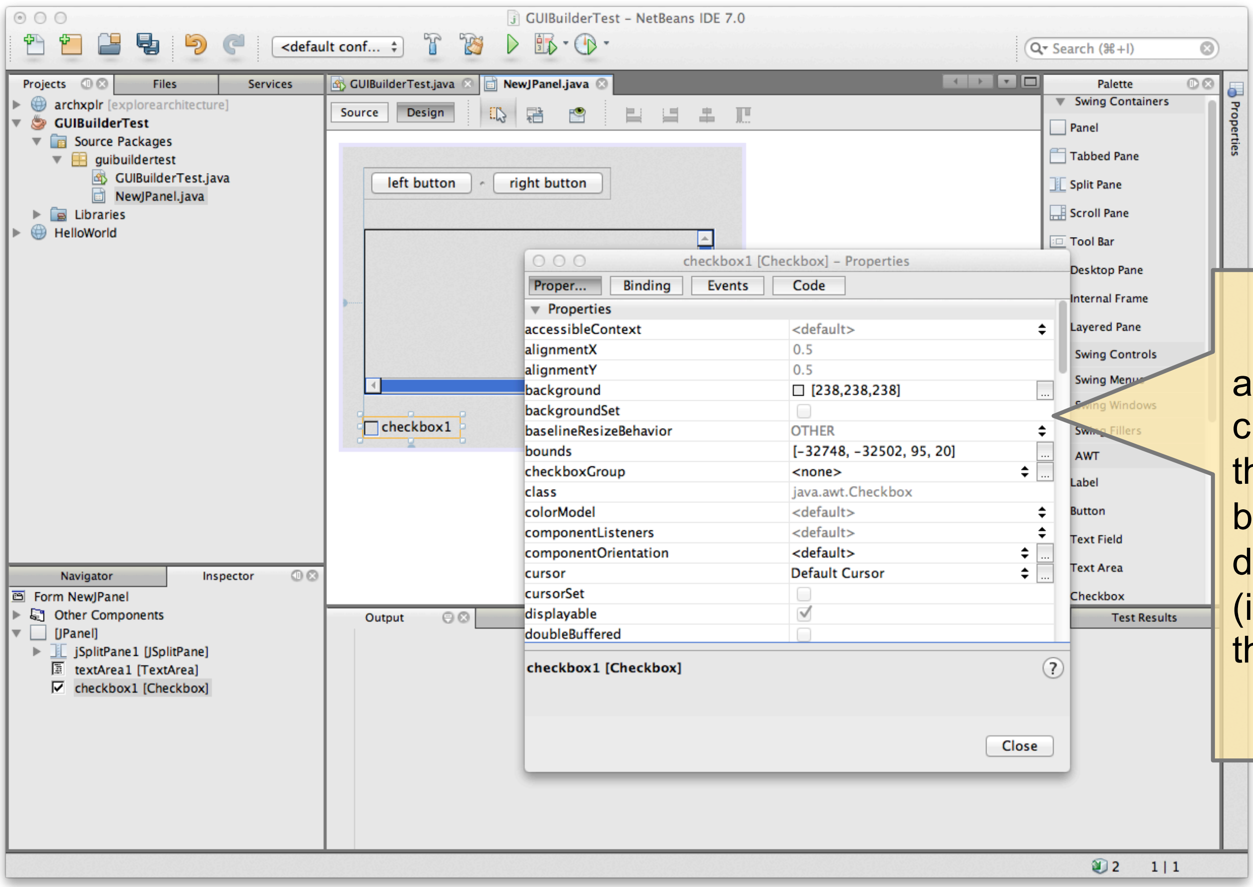
get info for all properties - this will analyse getters/ setters

this is a reference to the get method !

execute the get method without parameters

source code: https://bitbucket.org/jensdietrich/oop-examples/src/1.0/dynamic/

Application of Reflection: UI Builder



Properties

accessibleContext	<default>
alignmentX	0.5
alignmentY	0.5
background	[238,238,238]
backgroundSet	
baselineResizeBehavior	OTHER
bounds	[-32748, -32502, 95, 20]
checkboxGroup	<none>
class	java.awt.Checkbox
colorModel	<default>
componentListeners	<default>
componentOrientation	<default>
cursor	Default Cursor
cursorSet	
displayable	✓
doubleBuffered	

checkbox1 [Checkbox]

a **property sheet** is used to customise components
this is built dynamically based on the properties discovered, and setters (invoke) are used to **modify** the values of properties

- UI builders can customise components
- this must be kept dynamic: new components can be added to pallete at runtime

Application of Reflection: Persistency

- Java has two built-in mechanisms to serialises objects: binary and XML serialisation
- use cases: save objects to files, networking
- XML serialisation is based on reflection
- basic idea: read object properties one by one, encode in XML and write to file
- when object is read from file, the class name is read and instantiated:
  Class.forName("classname").newInstance()
- this requires the constructor without parameters (and will fail if it is missing and the convention has been violated!)
- source code:
  https://bitbucket.org/jensdietrich/oop-examples/src/1.0/dynamic/

XML Encoder

```
Student s = new Student();
s.setName("Max");
s.setFirstName("Dietrich");
XMLEncoder encoder = new XMLEncoder(
    new FileOutputStream("student.xml")
);
encoder.writeObject(s);
encoder.close();
```

object written (serialised) to student.xml

XML Decoder

```
XMLDecoder decoder = new XMLDecoder(
    new FileInputStream("student.xml")
);
Student obj = (Student)decoder.readObject();
decoder.close();
```

object read from student.xml

a **cast** is necessary: this is a general-purpose utility and therefore returns instances of Object

Generated XML

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.6.0_33" class="java.beans.XMLDecoder"
    <object class="reflection.Student">
        <void property="dob">
            <object class="java.util.Date">
                <long>1344223448903</long>
            </object>
                </void>
        <void property="firstName">
            <string>Bill</string>
        </void>
        <void property="name">
            <string>Clinton</string>
        </void>
    </object>
</java>
```

the class name can be used to create the class (**Class.forName()**) and then instantiate it (**newInstance()**)

properties are listed here: invoke on the property setters can be used to set the respective values

Annotating Artefacts

- how can we add **metadata** to classes, methods, fields etc
- example for meta data use: object-relational mapping
- consider a class Student that is to be mapped to a table STUDENTS
- this mapping has to associate:
    - the class with a table
    - the properties of the class with a column
    - instances will be mapped to rows

How to Configure ORM: Code (inline)

```
public class Student {
                private String name = "";
                public String getColumnForName() {
                                return "NAME";
        }
                public String getTable() {
                return "STUDENTS";
        }
}
```

- add mapping info directly to class (e.g., add method getTableName()).
- problem:
  - this in **invasive**, the class now depends on the database
  - does not separate concerns
  - difficult to add fine grained info (e.g., column data types)

How to Configure ORM: Config Files

```
public class Student {
            private String name = "";
}
```

```
<type_mapping>
            <class>Student</class>
            <table>STUDENTS</table>
            <attribute_mapping>
                        <property>name</property>
                        <column type="varchar2(30)">NAME</column>
                        ...
```

- manage mapping info in config file (e.g., xml)
- problem: difficult to keep config file and code consistent

How to Configure ORM: Comments

```
// map to STUDENTS
public class Student {
		// map to NAME(varchar2(30))
		private String name = "";
}
```

- manage mapping info in comments
- problem:
  - difficult to automatically process this info, requires structured comments (like XDoclets)
  - is not checked by compiler

How to Configure ORM: Annotations

```
@ORM.TableMapping(table="STUDENTS")
public class Student {
		@ORM.ColumnMapping(column="NAME",type="varchar2",length=30)
		private String name = "";
}
```

- manage mapping info in annotations
- annotations are types, checked by the compiler
- annotations can be read by the compiler, and via reflection
- annotations add semantics (meaning) to code

Defining Annotations

- when defining an annotation, annotations are used:
  - **@Retention** - whether the annotation can be seen at runtime (via reflection)
  - **@Target** - which element (class, field, method) is annotated

```
package ORM
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface ColumnMapping {
     String name();
     String type();
     int length;
}
```

will be available at runtime

used to annotate methods

properties

Using Annotations - Examples

- retention policy: runtime
  - modern ORMs like hibernate use annotations, e.g. the annotations defined in javax.persistence
  - annotations are used to map server-side scripts ("servlets") to URLs - this replaces an older mechanism using configuration files (web.xml)
- retention policy: source
  - @override is an annotation used when overriding methods - it is checked by the compiler to ensure that the overridden method exists
  - @SuppressWarnings is used to suppress compiler warnings for the annotated element

Measuring Code Complexity

- several metrics have been suggested to measure the complexity of code
- high complexity - code is difficult to understand, and therefore hard to maintain
- complex code is error-prone
- easiest way to assess complexity: measure size (lines of code)

Project Structure

- folder structure for Java projects
- standardising this structure facilitates working in teams
- some tools use special templates that pre-define this structure
- in Eclipse, the structure can easily be changed, in particular by setting source folders

A Simple Project Structure for Java/ Eclipse

<root>

bin	compiled classes (*.class)
src	source code (*.java)
lib	libraries (*.jar) in build path
build	output folder for build script
doc	documentation
.project	(Eclipse only) project settings
.classpath	(Eclipse only) build path

note: when sharing projects using a repository (GIT, SVN, etc), the project metadata files (.classpath and .project) can be shared as well - this makes it easier to share project settings (such as changed build paths if a library has been added)

The Maven Project Layout

src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/main/filters	Resource filter files
src/main/assembly	Assembly descriptors
src/main/config	Configuration files
src/main/scripts	Application/Library scripts
src/main/webapp	Web application sources
src/test/java	Test sources
src/test/resources	Test resources
src/test/filters	Test resource filter files
src/site	Site
LICENSE.txt	Project's license
NOTICE.txt	Notices and attributions required by libraries that the project depends on
README.txt	Project's readme

https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html