2009
YEAR

SOFTENG 254

The University of Auckland | New Zealand

Software Engineering
The University of Auckland

Quality Assurance More Tools

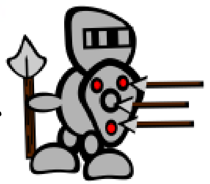
Part II - Lecture 8

Once upon a time. . .

. . . a project was started in a fortress, to automate some business process.

The team was motivated and the architect was bright.

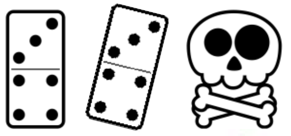
Everyone was busy writing code, but no documentation was in sight.



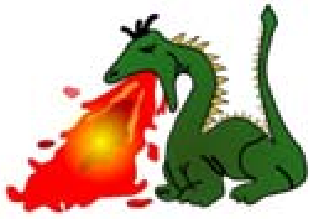
The architect spent sleepless nights, worked with the team in endless fights.

And then the time came of deployment, for the first customer's enjoyment. . .

Some requirements were fulfilled, but not all.

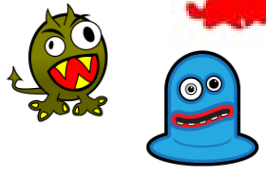


Changing code in one place made other code fall.



The system had problems under real load,

And the code was full of bugs and bloat.

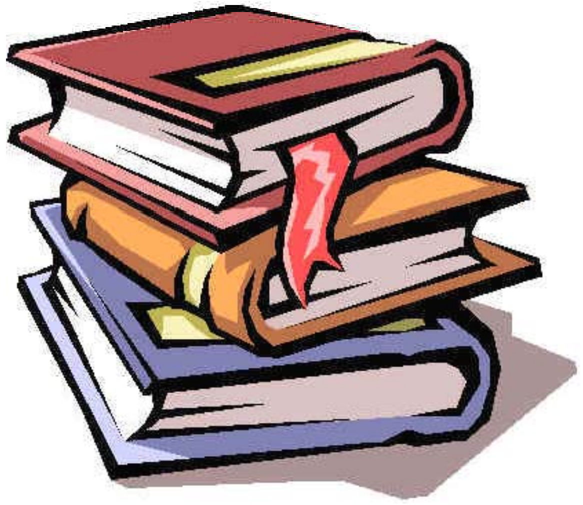


Read more at http://vidhujoshua.blogspot.co.nz/2009/05/analysis-of-failed-software-project.html

Today's Outline

- JavaDoc
- The ANT Build Tool
- Source Code Formatting with Eclipse

JavaDoc



The guy who knows about computers is the last person you want to have creating documentation for people who don't understand computers.
(Adam Osborne)

JavaDoc

- Tool that generates **HTML documentation** from Java source code
- Industry standard for documenting Java APIs
- Idea: developers put **special comments** starting with /** that contain documentation in front of classes, fields and methods
- JavaDoc comments are structured by **JavaDoc tags**, which are keywords that begin with an @ sign

```
/**
  * Divides two integer numbers
  * @author Christof Lutteroth
  * @param x Dividend
  * @param y Divisor
  * @return x divided by y
  * @throws ArithmeticException if y==0
  */
int div(int x, int y) { return x/y; }
```

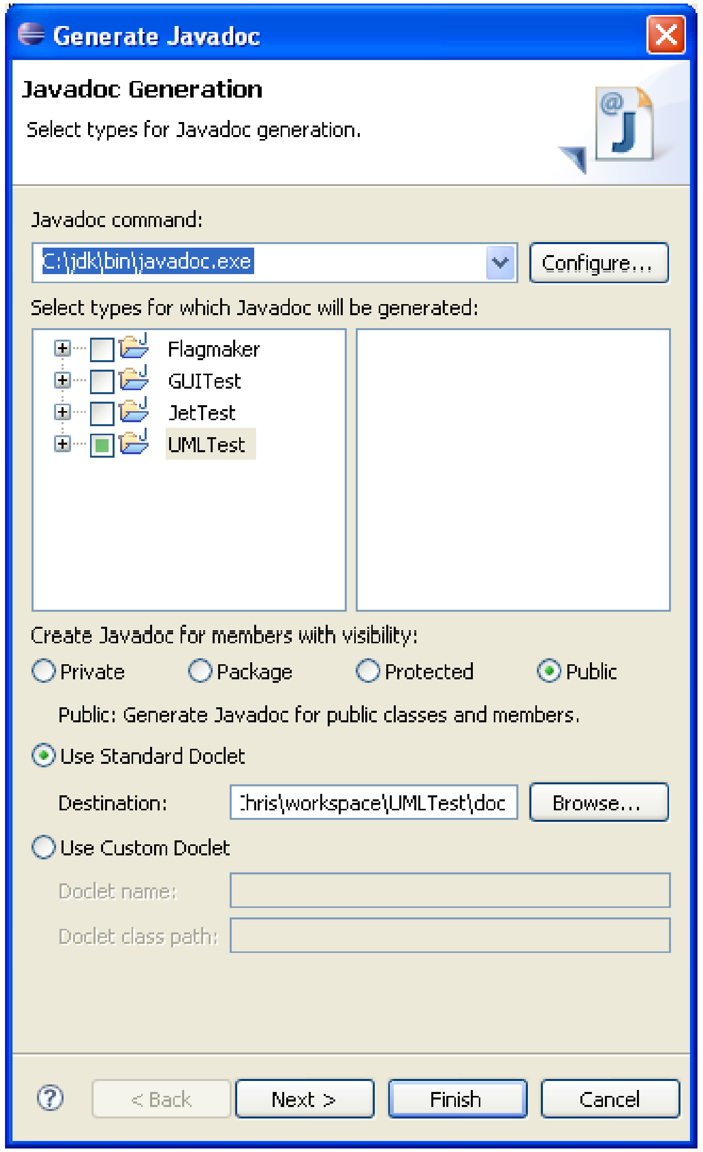
JavaDoc Tags

@author *name*	Specifies developer name
@version *number*	Add version number to a class or method
@param *name description*	Add description for a method parameter
@return *description*	Add a description for the method return value
@throws *className description*	Describes an exception that a method might throw (synonym to **@exception**)
@see *reference*	Adds a reference to something else
@since *text*	Add a comment since when a class/field/method exists
@deprecated	Marks a method as deprecated
{@link *package.class#member label***}**	Add a link to the documentation of some other class/field/method

JavaDoc in Eclipse

- Eclipse has auto-insertion feature for JavaDoc comments
  1. Move cursor into line before type/method/field
  2. type /** and press enter

- Generating the documentation
  1. From the menu: *Project -> Generate Javadoc…*
  2. Select location of **`javadoc.exe`**
  3. Select folder for documentation; typically **/doc** in project folder
  4. Many other settings about appearance of documentation…
  5. Click finish



Generate Javadoc

Javadoc Generation

Select types for Javadoc generation.

Javadoc command: C:\jdk\bin\javadoc.exe

Select types for which Javadoc will be generated:

- ☐ Flagmaker
- ☐ GUITest
- ☐ JetTest
- 🟩 UMLTest

Create Javadoc for members with visibility: Private Package Protected [✓ Public]

Public: Generate Javadoc for public classes and members.

[✓ Use Standard Doclet]

Destination: :hris\workspace\UMLTest\doc

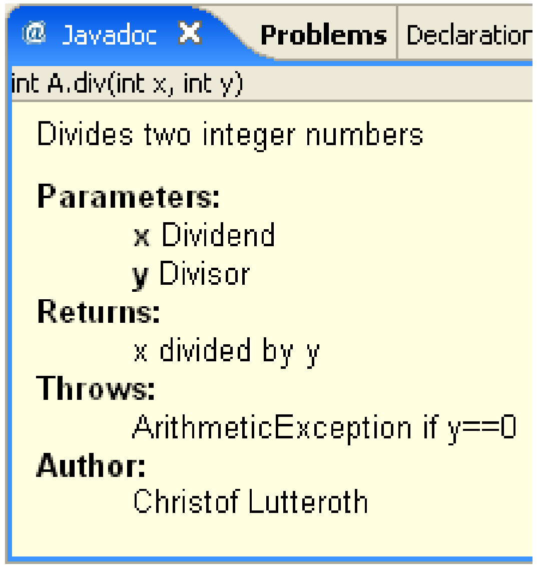
Use Custom Doclet

Doclet name:

Doclet class path:

View JavaDoc in Eclipse

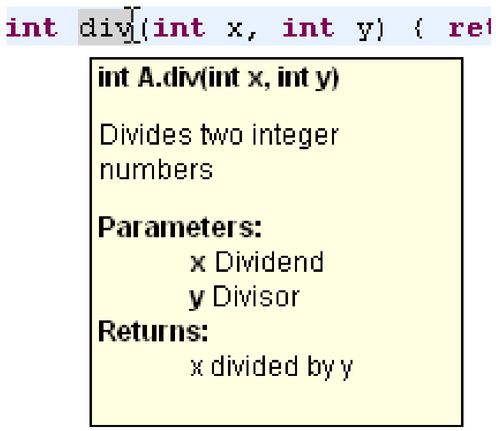
1. In JavaDoc view when identifiers are selected by double-clicking on them
2. In tooltip when hoovering mouse pointer over identifier
3. In help view under section "Java help" when cursor is on identifier



int A.div(int x, int y)

Divides two integer numbers

Parameters:
x Dividend
y Divisor
Returns:
x divided by y
Throws:
ArithmeticException if y==0
Author:
Christof Lutteroth

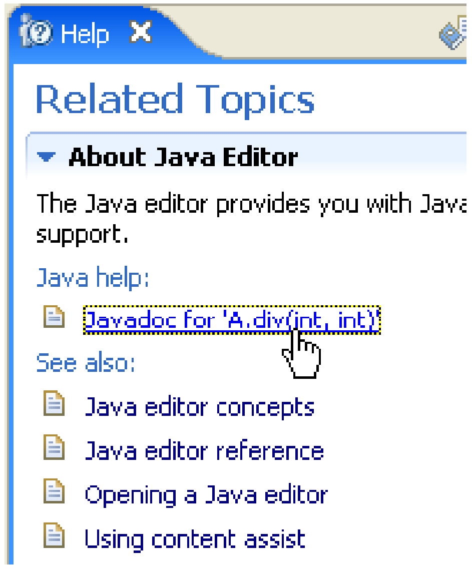


```
int div(int x, int y) { ret
```

int A.div(int x, int y)

Divides two integer numbers

Parameters:
x Dividend
y Divisor
Returns:
x divided by y



Related Topics

About Java Editor

The Java editor provides you with Java support.

Java help:
Javadoc for 'A.div(int, int)'

See also:
Java editor concepts
Java editor reference
Opening a Java editor
Using content assist

The ANT Build Tool

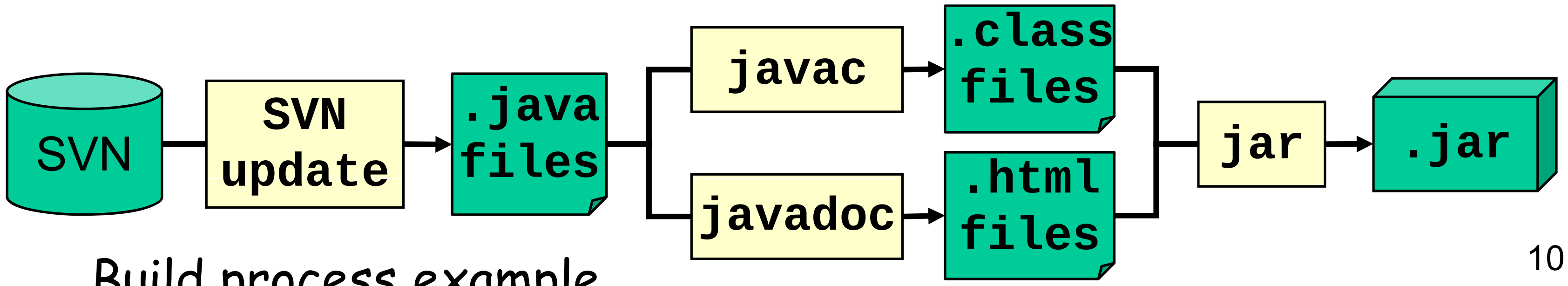
Ants can carry more than
50 times their body weight.

The Build Process

Software Engineering
The University of Auckland

The generation of **end-user artefacts** (executable programs, documentation, packaged files) **from developer artefacts** (source code, models, …)

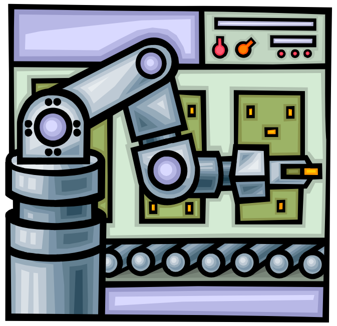
- Can involve many complex steps
  - Getting latest stable source code from a VCS (e.g. SVN)
  - Compilation of source code files, linking (e.g. **`javac`**)
  - Running tests (e.g. JUnit)
  - Generation of documentation (e.g. JavaDoc)
  - Packaging (e.g. **`jar`**)
  - Deployment (e.g. copying package to a server with ftp)
  - Clean up (e.g. deleting old or redundant files)
- Different build processes for different product variants (e.g. "enterprise" and "home" versions)



Build process example

Build Tools

- Build tools **automate** the build process
  - Build process is documented/specified in a build script
  - Much **faster** than manual build
  - Helps to perform builds **exactly the same** each time (less mistakes)
  - Can be used to manage **different build processes**
  - Helps **close the gap** between the development, integration, test, and production environments
- Orchestrate the build process;
  usually **invoke other tools** for doing the work
- Can be **triggered by other tools**,
  e.g. for nightly builds or continuous integration



Targets and Dependencies

- The different steps in a build process are called targets ("building a target" refers to the outcome of a step)
- Usually there are dependencies between targets, e.g.
  - Get latest version *before* compiling source code
  - Compile source code *before* packaging
- Targets have to be built following the dependencies
- Target dependencies are **transitive** (if A→B and B→C then A→C)
- Build process can be **optimized** by executing targets only when necessary (e.g. recompile class only if **`.java`** file has changed)
- Example: dependency between targets in C++ project

$$\begin{tikzcd}
 & \texttt{calc} \arrow[dl] \arrow[dr] & & \\
\texttt{main.o} \arrow[d] & & \texttt{math.o} \arrow[d] \arrow[dr] & \\
\texttt{main.cpp} & & \texttt{math.cpp} & \texttt{math.h}
\end{tikzcd}$$

x→y = x depends on y

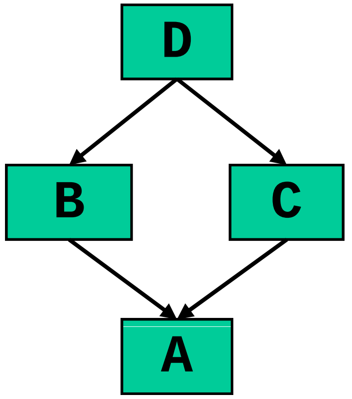
The ANT Build Tool

- Platform-independent, open-source scripting tool for automating build processes
- Uses **XML files** to describe the build process and its dependencies (default script name: **`build.xml`**)
- Implemented in **Java** and primarily intended for use with Java; de facto standard
- **Solves portability problems** of older build tools (e.g. make)
  - ANT provides built-in functionality for many tasks
  - Built-in functions are guaranteed to behave (nearly) identically on all platforms
- An ANT script defines a project with targets
  - Target: set of tasks you want to be executed
  - Task: piece of code that can be executed
- When starting ANT, you can select the target(s) to be executed

Projects and Targets

- **`<project>`** is the top level element; has three optional attributes:
  - **`name`**: the name of the project
  - **`default`**: the default target (when no target is chosen)
  - **`basedir`**: the base directory for relative paths
- **`<target>`** attributes:
  - **`name`**: the name of the target (mandatory)
  - **`depends`**: list of targets that it depends on (optional)
  - **`description`**: short target description (optional)
- In the example: A is executed first, then B, then C, and finally D

```
<?xml version="1.0"?>
<project name="DependencyDemo">
  <target name="A"/>
  <target name="B" depends="A"/>
  <target name="C" depends="A"/>
  <target name="D" depends="B,C"/>
</project>
```


$$\begin{tikzcd} & D \arrow[dl] \arrow[dr] & \\ B \arrow[dr] & & C \arrow[dl] \\ & A & \end{tikzcd}$$

Tasks

- Task: piece of code that can be executed and can have multiple attributes (arguments) and sub-tags
- ANT comes with over 80 built-in tasks; many more available
- Invoking a task:
  <*name attribute1*="*value1*" *attribute2*="*value2*" … />

```
<?xml version="1.0"?>
<project name="Hello" default="compile">
<target name="compile" description="compile .java files">
  <mkdir dir="classes"/>
  <javac srcdir="." destdir="classes"/>
</target>  <!-- This is an XML comment -->
<target name="jar" depends="compile"
  description="create a jar file for the application">
  <jar destfile="hello.jar">
    <fileset dir="classes" includes="**/*.class"/>
    <manifest>
    <attribute name="Main-Class" value="HelloProgram"/>
  </manifest> </jar> </target>
</project>
```

More Tasks

- **File tasks**: **<copy *file tofile*>**, **<delete *file*>**, **<mkdir *dir*>**, **<touch *file*>**, **<get *src dest*>**
- **Java tasks**: **<java *classname*>**, **<javac *srcdir destdir*>**, **<javadoc *sourcefiles destdir*>**, **<junit …>**
- **Packaging**: **<jar *destfile basedir*>**, **<zip *destfile basedir*>**, **<unzip *src dest*>**
- **Misc tasks**: **<echo *message*>**, **<exec *command*>**, **<mail …>**
- **Add your own tasks** with **<taskdef *name classname*>**:

```
…
public class MyPrintTask extends Task {
  private String msg;
  // setter for attribute "message"
  public void setMessage(String msg) { this.msg = msg; }
  public void execute() throws BuildException {
    System.out.println(msg);
} }
```

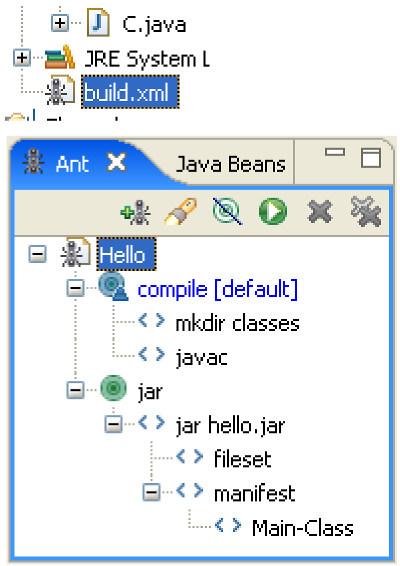
Properties

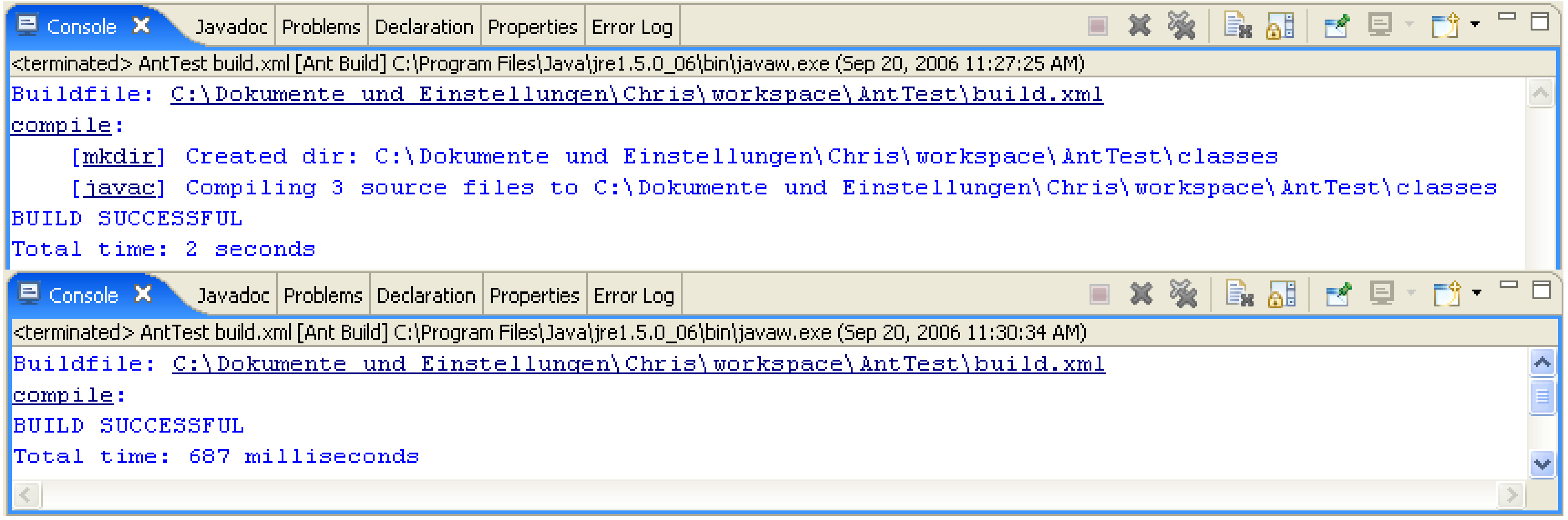
- **Property**: case-sensitive name associated with a value
  - immutable: once it is set it cannot be changed
  - may be used in the value of task attributes by placing the property name between **`${`** and **`}`** in the attribute value
- **`<property name="foo.x" value="bar"/>`** sets the property **`foo.x`** to the value **`bar`** (for files: **`location`** instead of **`value`**)
- Many **built-in properties**, e.g. **`basedir`**, **`ant.file`**, **`java.class.path`**, **`os.name`**, **`os.version`**, **`file.separator`**
- Targets can be **conditionally** executed with special attributes:
  - **`if`**: executes target only if a property is set
  - **`unless`**: executes target only if property is not set

```
<?xml version="1.0"?>
<project name="MyProject">
  <property name="classdir" location="classes"/>
  <target name="compile">
    <javac srcdir="." destdir="${classdir}"/> </target>
  <target name="workaround-code" if="system-has-bug"/>
  <target name="normal-code" unless="system-has-bug"/>
</project>
```

ANT and Eclipse

1. Create a text file **build.xml** in the main folder of your project
2. Open Ant view and add the build file by dragging it into the view
3. Double-click target to execute it





```
<terminated> AntTest build.xml [Ant Build] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Sep 20, 2006 11:27:25 AM)
Buildfile: C:\Dokumente und Einstellungen\Chris\workspace\AntTest\build.xml
compile:
    [mkdir] Created dir: C:\Dokumente und Einstellungen\Chris\workspace\AntTest\classes
    [javac] Compiling 3 source files to C:\Dokumente und Einstellungen\Chris\workspace\AntTest\classes
BUILD SUCCESSFUL
Total time: 2 seconds
```

```
<terminated> AntTest build.xml [Ant Build] C:\Program Files\Java\jre1.5.0_06\bin\javaw.exe (Sep 20, 2006 11:30:34 AM)
Buildfile: C:\Dokumente und Einstellungen\Chris\workspace\AntTest\build.xml
compile:
BUILD SUCCESSFUL
Total time: 687 milliseconds
```

Best Practices

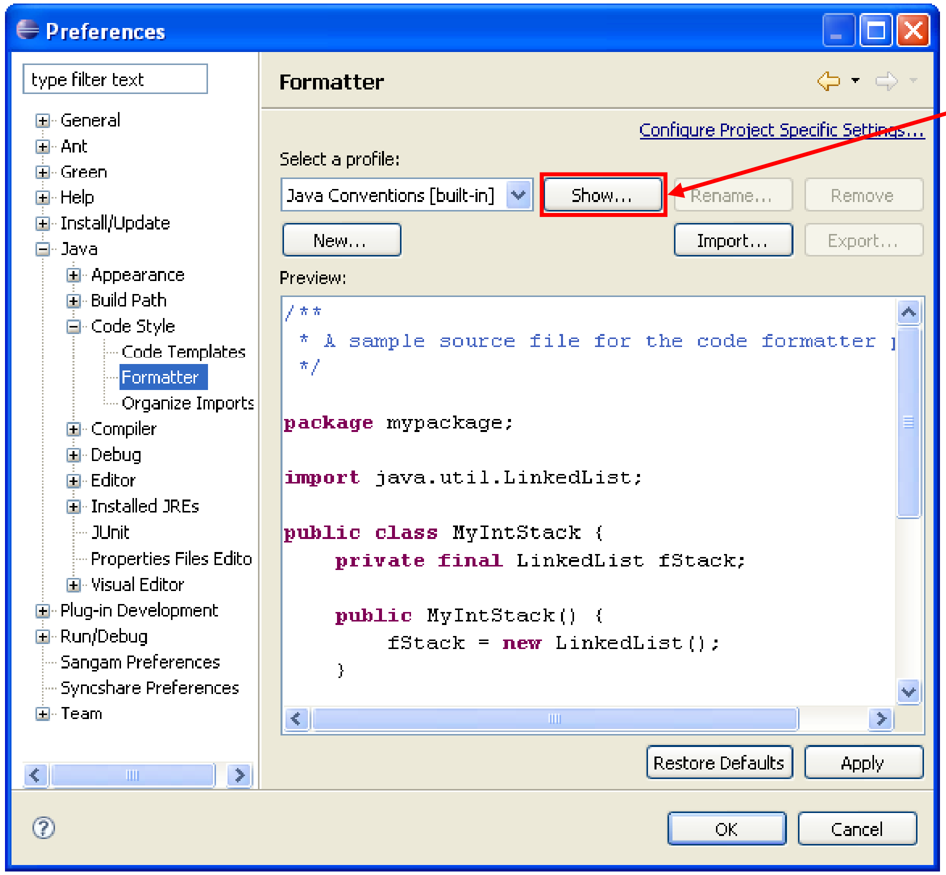
1. **Use Simple Targets**
   - Each target should do a single well defined job
   - Targets that do *too much* make the build harder to maintain and should be split up into multiple targets with dependencies between them
2. **Standardize Target Names**
   Makes it easier to understand and switch between build files
3. **Use Properties for Configurability**
   Properties should be defined for:
   - Any information that needs to be configured
   - Any information that might change
   - Any information that is used in more than one place

Source Code Formatting with Eclipse

Man is a strange animal.
He generally cannot read the handwriting on the wall until his back is up against it.
(Adlai E. Stevenson)

Source Code Formatting with Eclipse

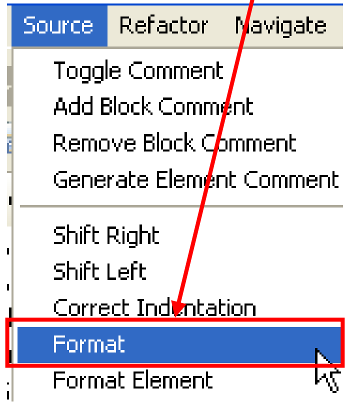
- Most projects use a coding style standard (e.g. see XP practice)
- Helps to read code, e.g. indentation follows code structure
- Choose/define/customize a coding style profile with *Window -> Preferences -> Java -> Code Style -> Formatter*



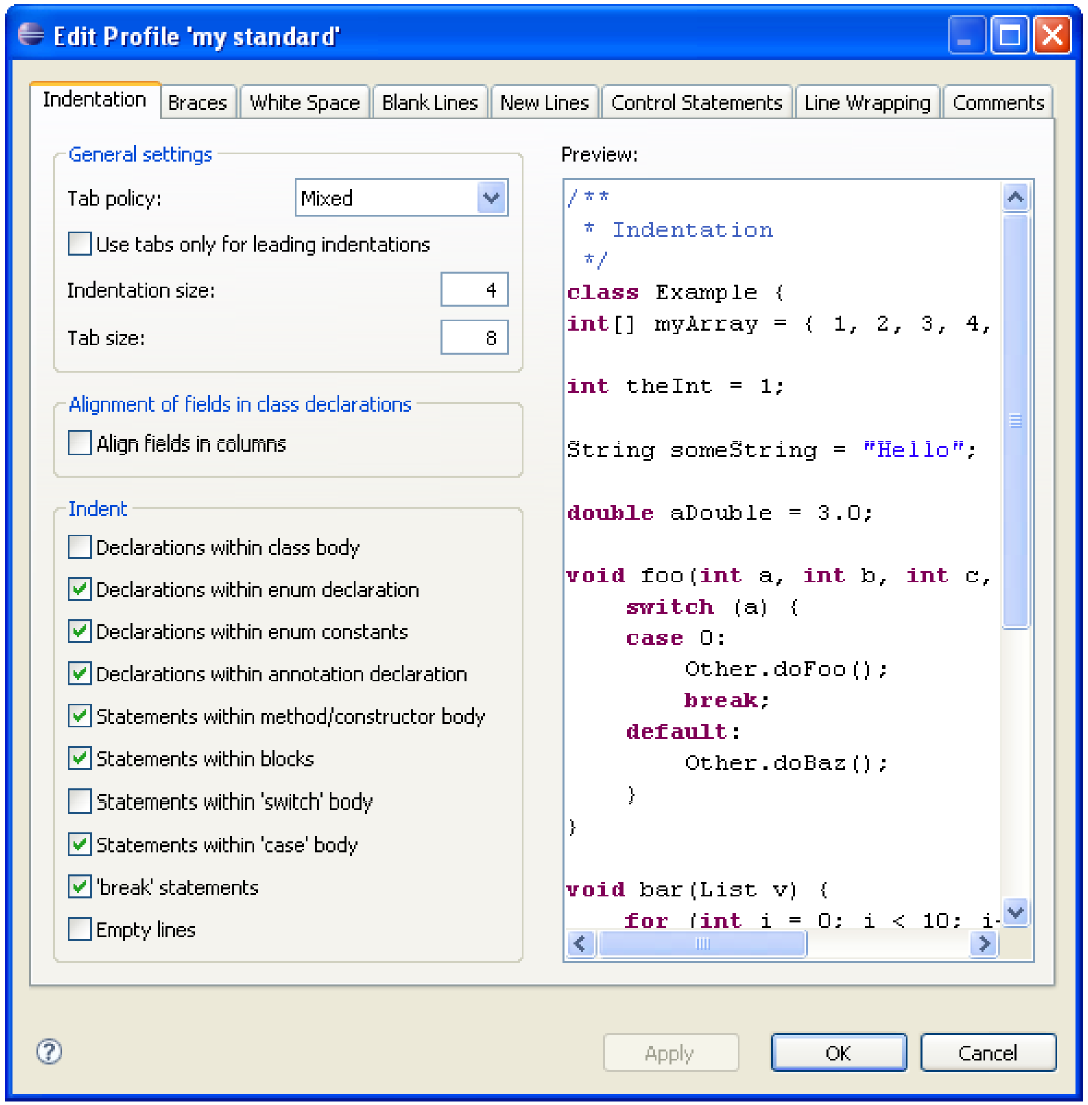
```
/**
 * A sample source file for the code formatter p
 */

package mypackage;

import java.util.LinkedList;

public class MyIntStack {
    private final LinkedList fStack;

    public MyIntStack() {
        fStack = new LinkedList();
    }
```

Show / change coding style profile

Apply coding style profile to editor by selecting code and choosing *Source->Format*



Source Code Formatting with Eclipse



General settings

Tab policy: Mixed

☐ Use tabs only for leading indentations

Indentation size: 4

Tab size: 8

Alignment of fields in class declarations

☐ Align fields in columns

Indent

☐ Declarations within class body

✓ Declarations within enum declaration

✓ Declarations within enum constants

✓ Declarations within annotation declaration

✓ Statements within method/constructor body

✓ Statements within blocks

☐ Statements within 'switch' body

✓ Statements within 'case' body

✓ 'break' statements

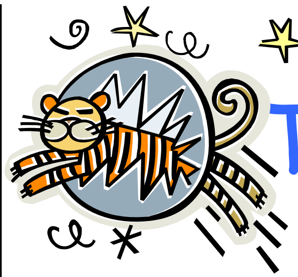
☐ Empty lines

Preview:

```
/**
 * Indentation
 */
class Example {
int[] myArray = { 1, 2, 3, 4,

int theInt = 1;

String someString = "Hello";

double aDouble = 3.0;

void foo(int a, int b, int c,
    switch (a) {
    case 0:
        Other.doFoo();
        break;
    default:
        Other.doBaz();
    }
}

void bar(List v) {
    for (int i = 0; i < 10; i+
```

Full control over

- Indentation
- Placement of braces
- Whitespace
- Blank lines
- New lines
- Control statements
- Line wrapping
- Comments



Today's Summary

- Documentation tools like **JavaDoc** generate API documentation from source code annotations
- Build tools like **Ant** automate the build process
  - Manage different build configurations with **build scripts**
  - **Tasks** are pieces of code defining what is done
  - **Targets** define sets of tasks that belong together
  - Targets can **depend** on other targets
  - **Properties** can be used to configure a build script
- Source code **formatting** can be used to enforce coding style guides automatically

Quiz

1. How does JavaDoc generate an API documentation from source code?

2. What are the advantages of using a build tool?

3. How do build scripts work? Explain targets, tasks and properties.