CSE 15L Fall 2018

Final Review

Soheil

Nate

Gerard

Jacalyn

Aabjeet

Julia

Unix Commands and Shell Scripting

Piping/Filtering

What is a pipe used for??

A pipe is used to redirect the output of one command to the input of another.

What does a pipe look like?

A pipe is a vertical bar ‘|’
Note: This is the same symbol used in ‘or’ ‘||’

Show me an Example.

```
ls -l | grep Apr  ls | wc
man ksh | grep “history” | wc -l
```

Shell Scripting

- Lines starting with # are comments, but the first line #! is not a comment; it indicates the location of the shell that will be run
- Quote characters
  - “ double quote: if a string is enclosed in “ “ the references to variables will be replaced with their values
  - ‘ single quote: taken literally
  - ` back quote: treated as command
    - echo “Date is:” `date`
- chmod is used to change the permissions so we can run our script

Let’s go to the command line and try it out for ourselves!

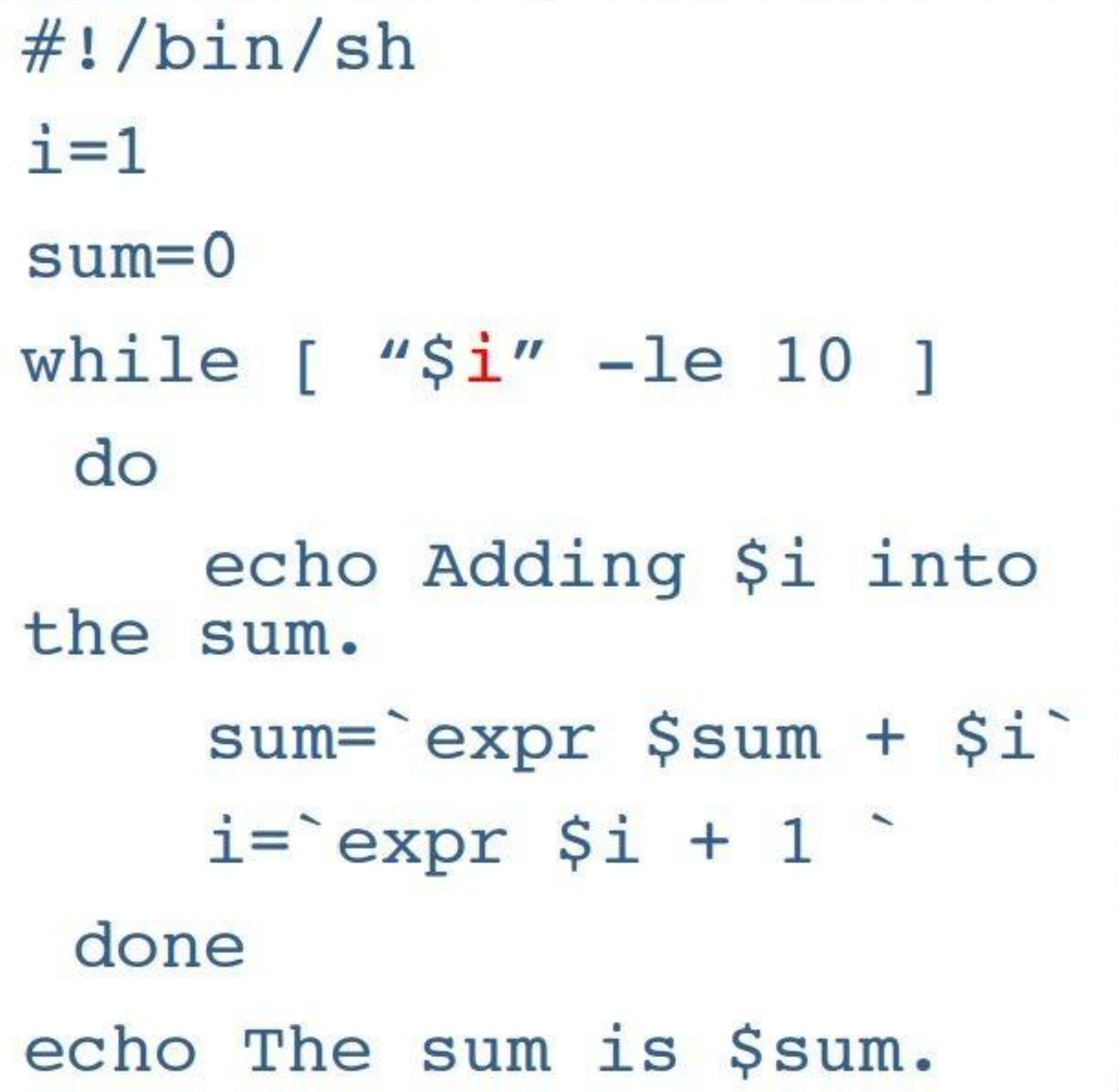
Loops

- Instead of using braces {} to control logic flow and statement blocks, shell uses terminating words:
  - if, then/ fi
  - case / esac
  - for, do, done
  - while, do, done

when you get downvoted on Stack Overflow for telling someone to read the man page
WHY ARE YOU BOOING ME? I'M RIGHT.

Let’s go to the command line and try it out for ourselves!

What does this chunk of code do?



```
#!/bin/sh
i=1
sum=0
while [ “$i” -le 10 ]
 do
     echo Adding $i into
the sum.
     sum=`expr $sum + $i`
     i=`expr $i + 1 `
 done
echo The sum is $sum.
```

Let’s go to the command line and try it out for ourselves!

Declarations

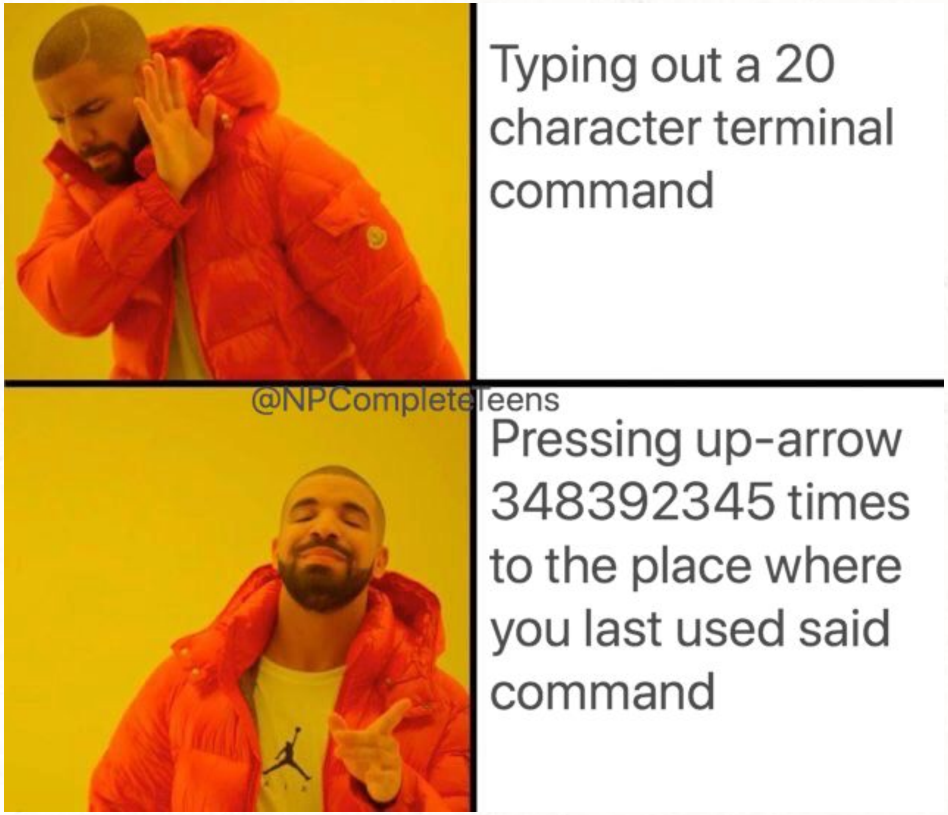
Are these two different?

1) y=5
2) y = 5

when you run something from command line in Linux but it fails

Frequently used Bash commands

- ls
- cat / more / less
- grep / cut [options]
- sort
- source ~/.bashrc
- cd / mkdir
- pwd
- man
- cp / scp / mv
- touch
- rm / rmdir
- diff
- uniq
- finger
- tar [options]
- ps
- kill [options]
- head / tail [options]


Typing out a 20 character terminal command
@NPCompleteTeens
Pressing up-arrow 348392345 times to the place where you last used said command

Vim

When you finally exit vim
You know, I'm something of a scientist myself

Common vim commands to know

Different types of modes: can you name the 3?

EXCUSE ME SIR,
DO YOU HAVE A MOMENT TO TALK ABOUT OUR LORD AND SAVIOR, VIM?
© Alamy
made on imgur

How to navigate:

How to search forwards/backwards:

Be able to describe in detail at least 10 unique vim commands.

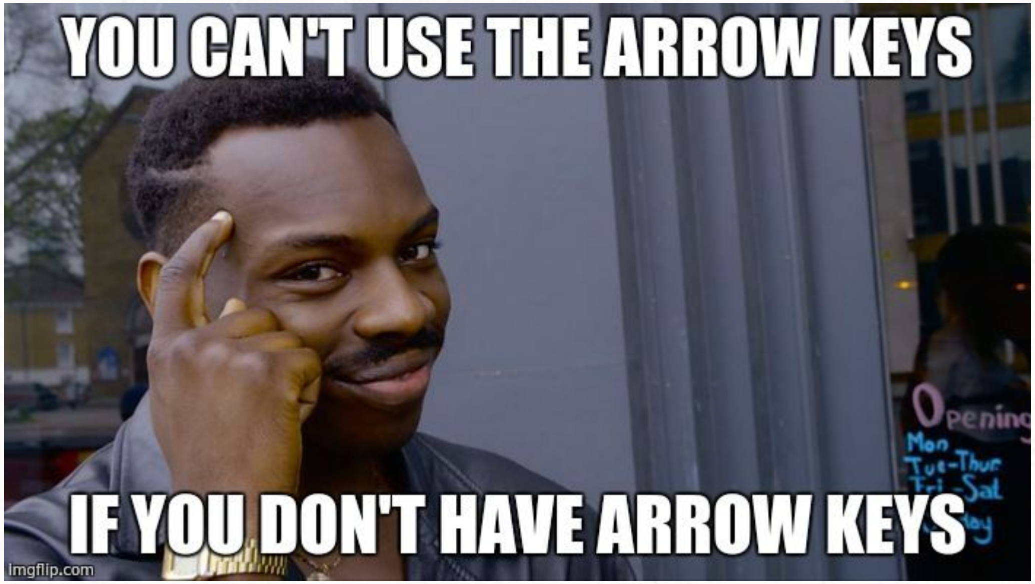
CAPITAL AND LOWERCASE VERSIONS ARE NOT CONSIDERED UNIQUE ON EXAM

Vim Modes

Mode:	Enter with:	Description:
Normal (command)	<esc>, <ctrl-c>	For navigation and manipulation of text. This is the mode that vim will usually start in.
Insert	aiocs (AIOCS)	For inserting new text.
Visual	v, V, <ctrl-v>	For navigation and manipulation of text selections.

Vim Navigation

- ***h, j, k, l***
- gg, G
- w, e, b
- ^, 0, $



Search and replace with reg\(ular \)\?ex\(pression\)\?

- Difference between / vs ?, n vs N
- :[range]s/[pattern]/[replacement]
- :[range]s/[pattern]/[replacement]/gc

ex: :%s/hello/bye/gc

Miscellaneous

- operator + motion (cw, de, y$, etc)
- repeat with number (c2w, 4de, y6y, etc)
- .vimrc (what is it? what can you do with one?
- save and quit

TDD & Unit Testing

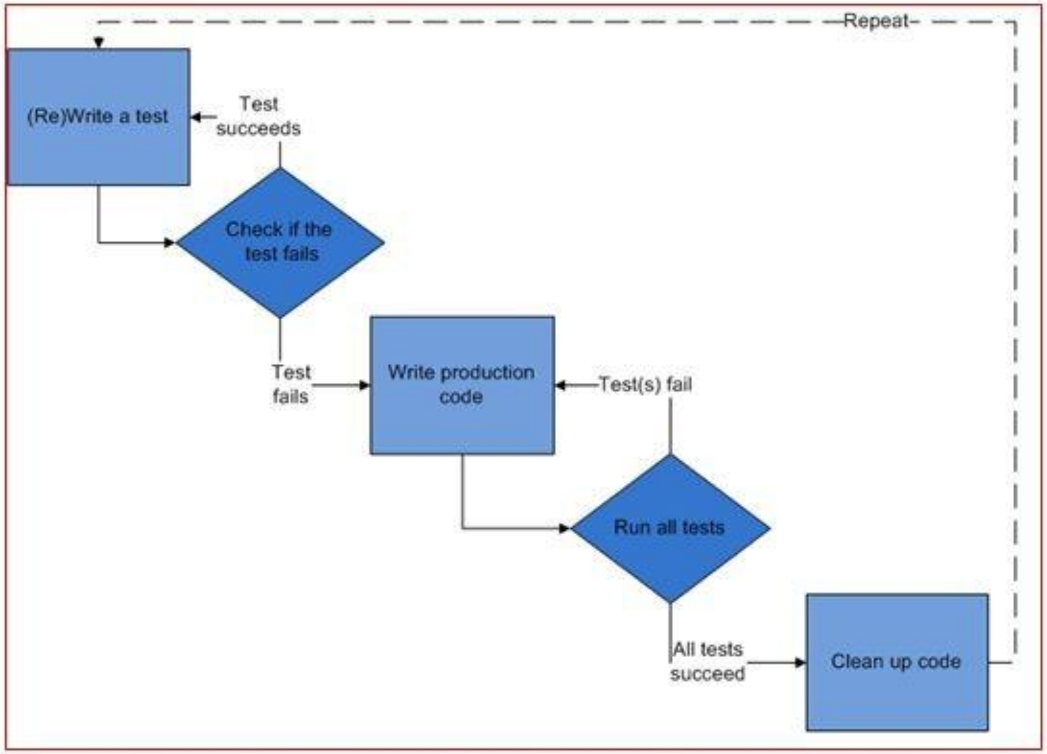
Me: I haven't tested this code, it definitely won't work first try
code doesn't run first try
Me:

Test-Driven Development

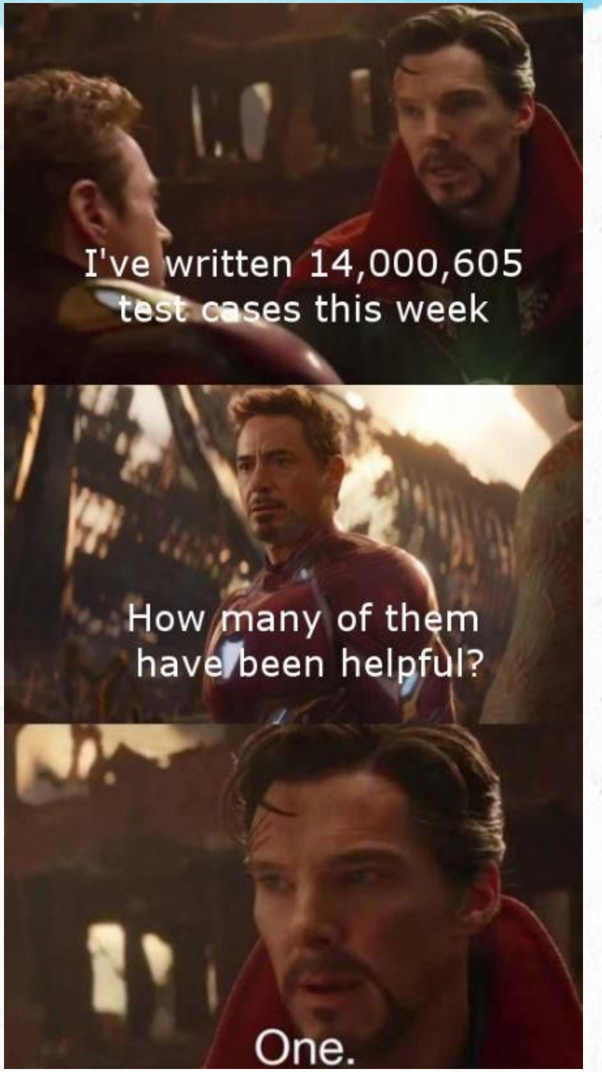
- In TDD, tests are written before software.
- You must understand requirements first!
- **Regression testing**
    - Everytime you change code, run original tests!
    - Make sure old features work after adding new ones.

Test-Driven Development

Understand the requirements before writing the code!



Needs to be followed by software integration testing!


I've written 14,000,605 test cases this week
How many of them have been helpful?
One.

Unit Testing

- What is a **unit**?
  - Usually a single method.
- Pros for unit testing
  - Better code functionality
  - Concise, goal driven code
  - Increased productivity
- Cons for unit testing
  - Doesn’t test full software.
  - Takes a long time to write all those tests.
  - Developer writes both the code and tests

Strategies for Unit Testing

- Identify and prioritize testing of
  - core functionality
  - corner cases for exceptions
  - special input values
  - commonly used functionality
- Test related functionality as test suites
- Test both positive and negative paths

JUnit

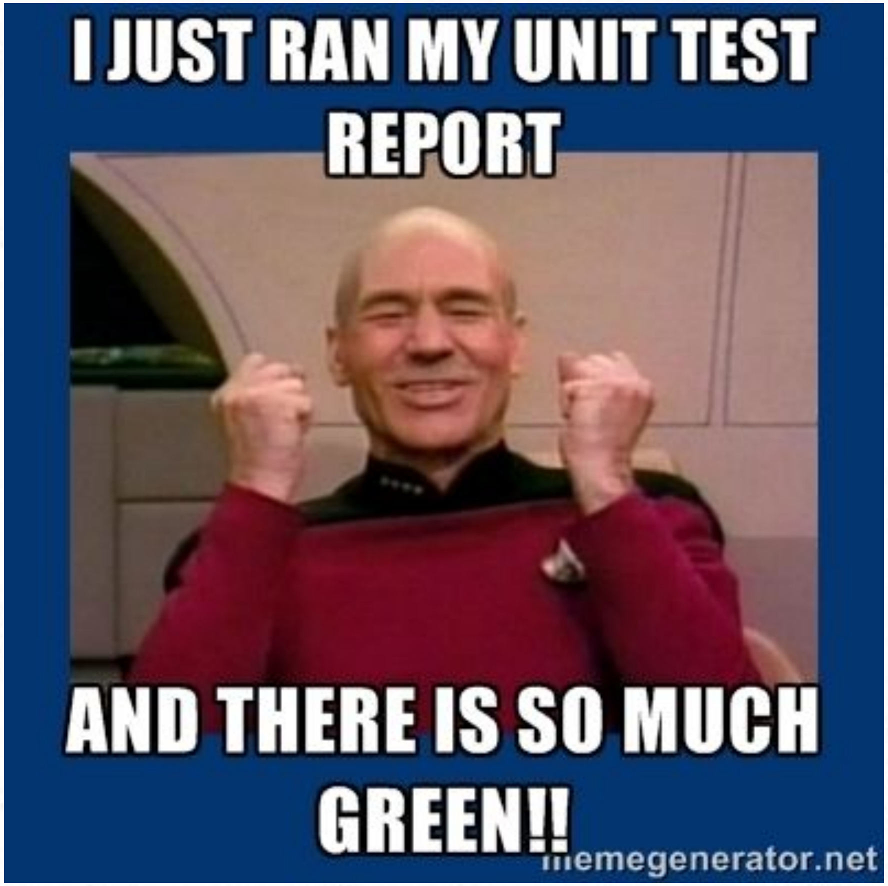
- JUnit is a widely used framework for unit testing in Java.
- Makes testing standardized and easy (relatively easier) to implement.
- Testing whole suites at once.

How many units should we test if we have 3 methods and 7 constructors?

JUnit

- Tests pass when they return without failing or without throwing exceptions (that are not caught).
- Failure happens when JUnit assertion is incorrect.

Test Driven Development Cons and Pros



Test Driven Development Cons and Pros

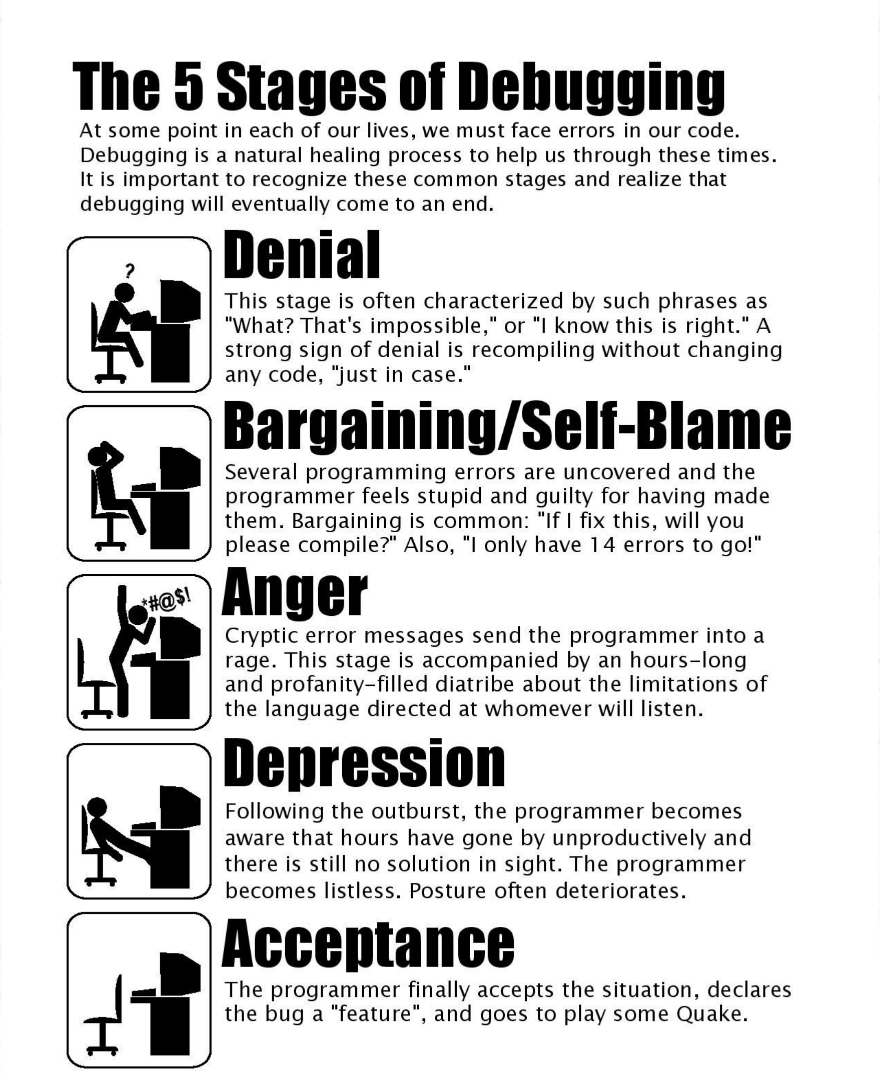
Pros:

- Increased productivity
- Results in modular and extensible software
- Leads to better code functionality and cleaner interfaces
- Leads to concise goal-driven code

Cons:

- Does not test the full software
- Requires team buy-in
- Same developer coding and testing a feature
- Can consume too much time

Debugging, GDB & Valgrind



The 5 Stages of Debugging

At some point in each of our lives, we must face errors in our code. Debugging is a natural healing process to help us through these times. It is important to recognize these common stages and realize that debugging will eventually come to an end.

Denial

This stage is often characterized by such phrases as "What? That's impossible," or "I know this is right." A strong sign of denial is recompiling without changing any code, "just in case."

Bargaining/Self-Blame

Several programming errors are uncovered and the programmer feels stupid and guilty for having made them. Bargaining is common: "If I fix this, will you please compile?" Also, "I only have 14 errors to go!"

Anger

Cryptic error messages send the programmer into a rage. This stage is accompanied by an hours-long and profanity-filled diatribe about the limitations of the language directed at whomever will listen.

Depression

Following the outburst, the programmer becomes aware that hours have gone by unproductively and there is still no solution in sight. The programmer becomes listless. Posture often deteriorates.

Acceptance

The programmer finally accepts the situation, declares the bug a "feature", and goes to play some Quake.

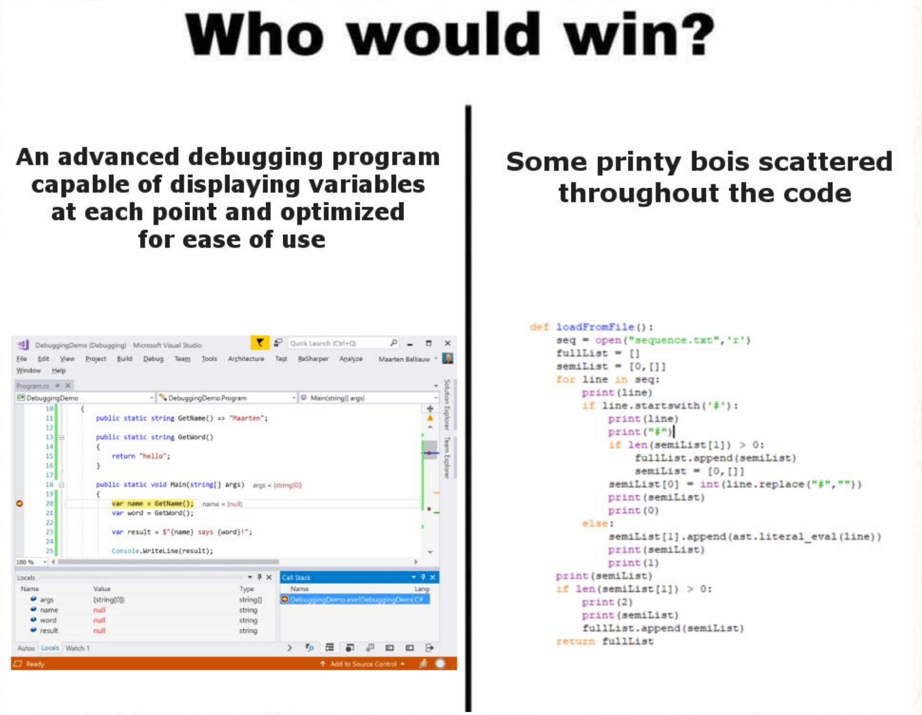
Debugging

- Debugging is NOT algorithmic
- Basic steps:

1. Understand the system
   - context, software, tools
2. Identify the problem
   - What is happening? What is not normal?
3. Reproduce the problem
   - Know how this problem is happening
4. Diagnose the cause of the problem
   - Make and confirm a hypothesis
5. Fix the problem
   - And don't introduce new bugs
6. Reflect and learn from the problem (and your fix)
   - Can you improve testing/design so you don't get more similar bugs?

Tools For Debugging

- Diagnostic output (stdout/stderr, logging, profiling, etc.)
- gdb
- Visual VM (profiling)
- Valgrind
- Many more!



Who would win?

An advanced debugging program capable of displaying variables at each point and optimized for ease of use

Some printy bois scattered throughout the code

```
    {
        public static string GetName() => "Maarten";

        public static string GetWord()
        {
            return "hello";
        }

        public static void Main(string[] args)  args = {string[0]}
        {
            var name = GetName();  name = (null)
            var word = GetWord();

            var result = $"{name} says {word}!";

            Console.WriteLine(result);
```

Locals

Name	Value	Type
args	{string[0]}	string[]
name	null	string
word	null	string
result	null	string

Call Stack

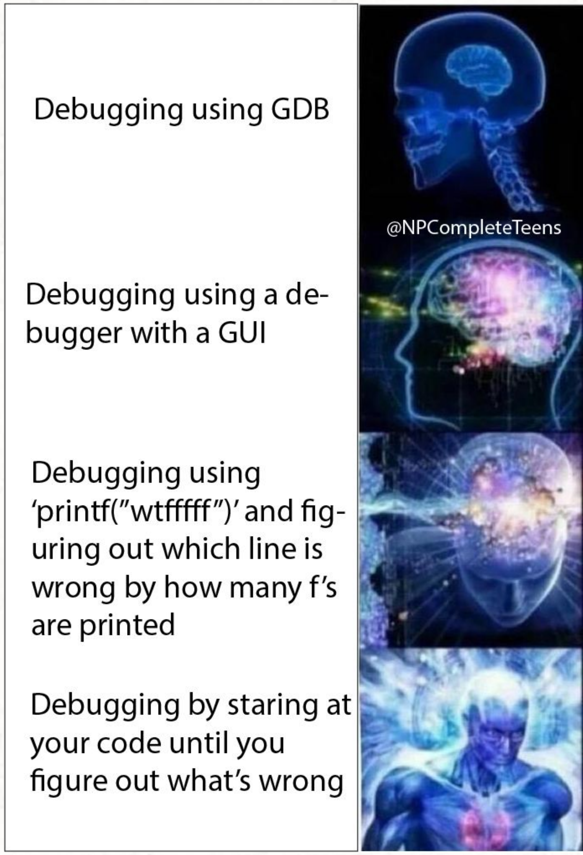
Name	Lang
DebuggingDemo.exe!DebuggingDemo	C#

```
def loadFromFile():
    seq = open("sequence.txt",'r')
    fullList = []
    semiList = [0,[]]
    for line in seq:
        print(line)
        if line.startswith('#'):
            print(line)
            print("#")
            if len(semiList[1]) > 0:
                fullList.append(semiList)
                semiList = [0,[]]
            semiList[0] = int(line.replace("#",""))
            print(semiList)
            print(0)
        else:
            semiList[1].append(ast.literal_eval(line))
            print(semiList)
            print(1)
    print(semiList)
    if len(semiList[1]) > 0:
        print(2)
        print(semiList)
        fullList.append(semiList)
    return fullList
```

GDB Cheatsheet

In the GDB console:

- **run** (to run your program)
- **break x** (where x is the name of your function in your program, line number)
- **next** (executes one more line, without stepping into the function if called)
- **continue** (when the program has stopped, it resumes execution)
- **step** (executes one more line, stepping into a function if called)
- **print x** (where x is an expression that can involve constants and variables)
- **quit** (to quit out of gdb)


Debugging using GDB
@NPCompleteTeens
Debugging using a debugger with a GUI
Debugging using 'printf("wtfffff")' and figuring out which line is wrong by how many f's are printed
Debugging by staring at your code until you figure out what's wrong

Difference between step vs. next

```
void foo() {
  for ( int i = 0; i < v.size(); i++ ) {
    print(“foo”)
  }
}
int main() {
→ foo();
  return 1;
}
```

Difference between step vs. next

```
    void foo() {
       for ( int i = 0; i < v.size(); i++ ) {
         print(“foo”)
       }
    }
    int main() {
       foo();
→      return 1;           (AFTER NEXT)
    }
```

Difference between step vs. next

```
    void foo() {
      for ( int i = 0; i < v.size(); i++ ) {
        print(“foo”)
      }
    }
    int main() {
→     foo();
      return 1;
    }
```

Difference between step vs. next

```
void foo() {
→ for ( int i = 0; i < v.size(); i++ ) {        (AFTER STEP)
      print(“foo”)
    }
}
int main() {
    foo();
    return 1;
}
```

Valgrind output

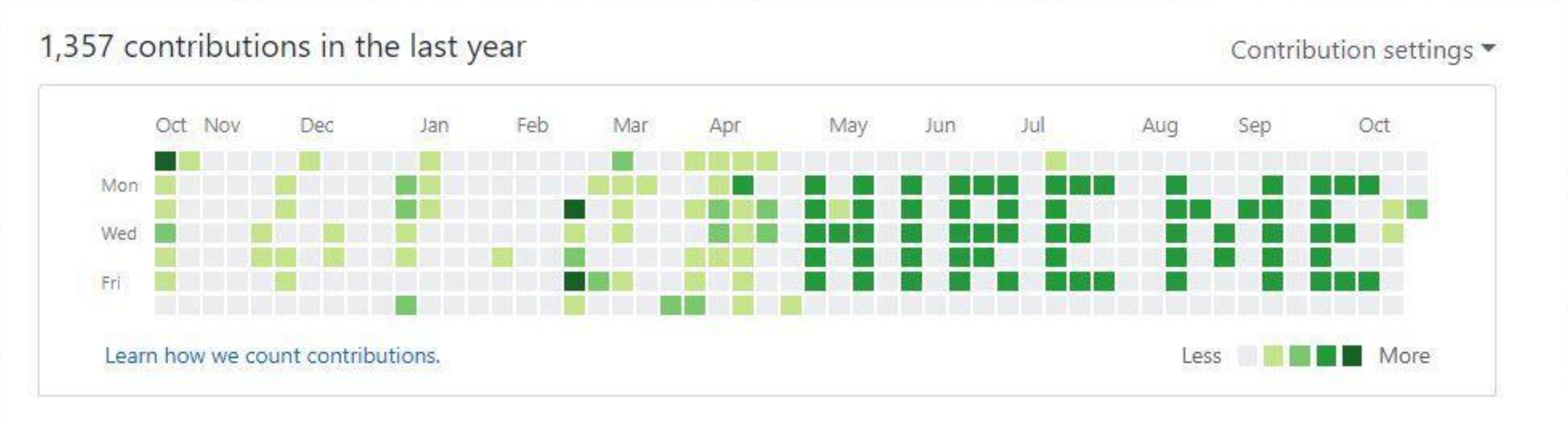
```
==15640==
==15640== HEAP SUMMARY:
==15640==  in use at exit: 10 bytes in 5 blocks
==15640== total heap usage: 5 allocs, 0 frees, 10 bytes allocated
==15640==
==15640== LEAK SUMMARY:
==15640==   definitely lost: 10 bytes in 5 blocks
==15640==   indirectly lost: 0 bytes in 0 blocks
==15640==   possibly lost: 0 bytes in 0 blocks
==15640==   still reachable: 0 bytes in 0 blocks
==15640==      suppressed: 0 bytes in 0 blocks
==15640== Rerun with --leak-check=full to see details of leaked memory
```

- What is this warning telling you, how might you resolve this?

Valgrind output (no leaks)

```
==18957==
==18957== HEAP SUMMARY:
==18957==  in use at exit: 0 bytes in 0 blocks
==18957== total heap usage: 5 allocs, 5 frees, 10 bytes allocated
==18957==
==18957== All heap blocks were freed -- no leaks are possible
==18957==
==18957== For counts of detected and suppressed errors, rerun with: -v
==18957== ERROR SUMMARY: 28 errors from 15 contexts (suppressed: 12 from 8
```

Git



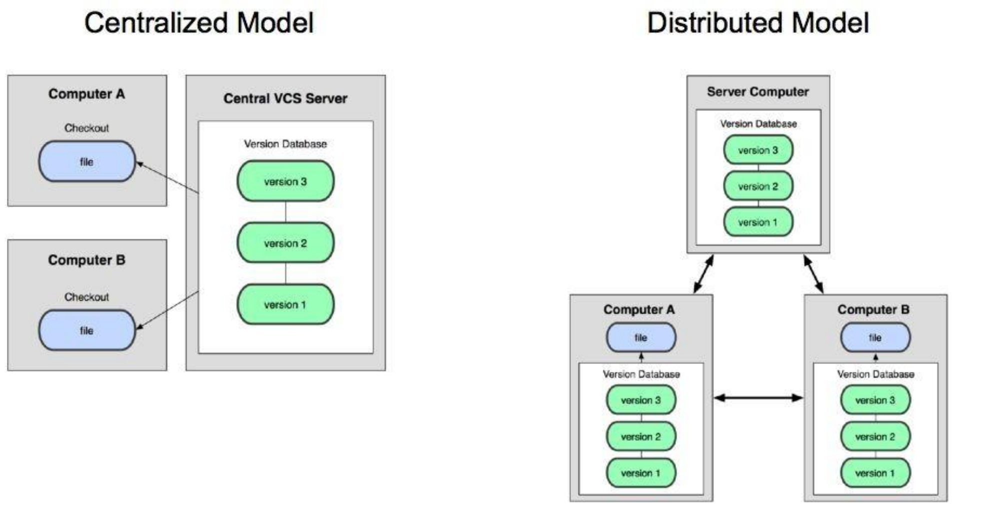
1,357 contributions in the last year

Learn how we count contributions.

“Add me on Instagram”
“Add me on LinkedIn”
“Add me on GITHUB”

What is Git?

Git is a distributed version control system.

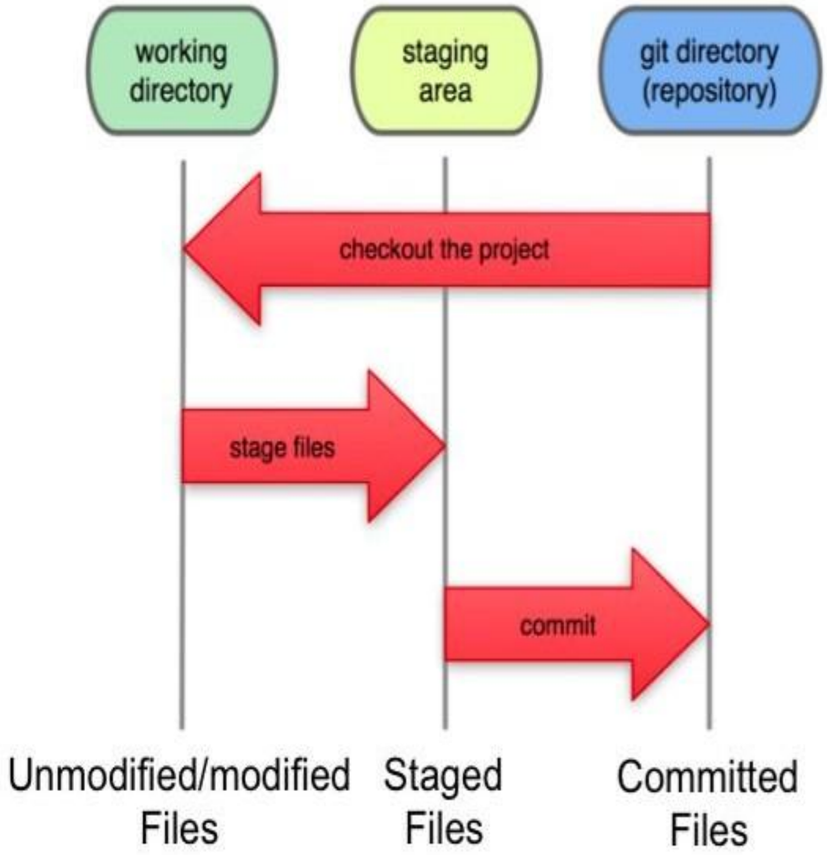


(CVS, Subversion, Perforce)

(Git, Mercurial)

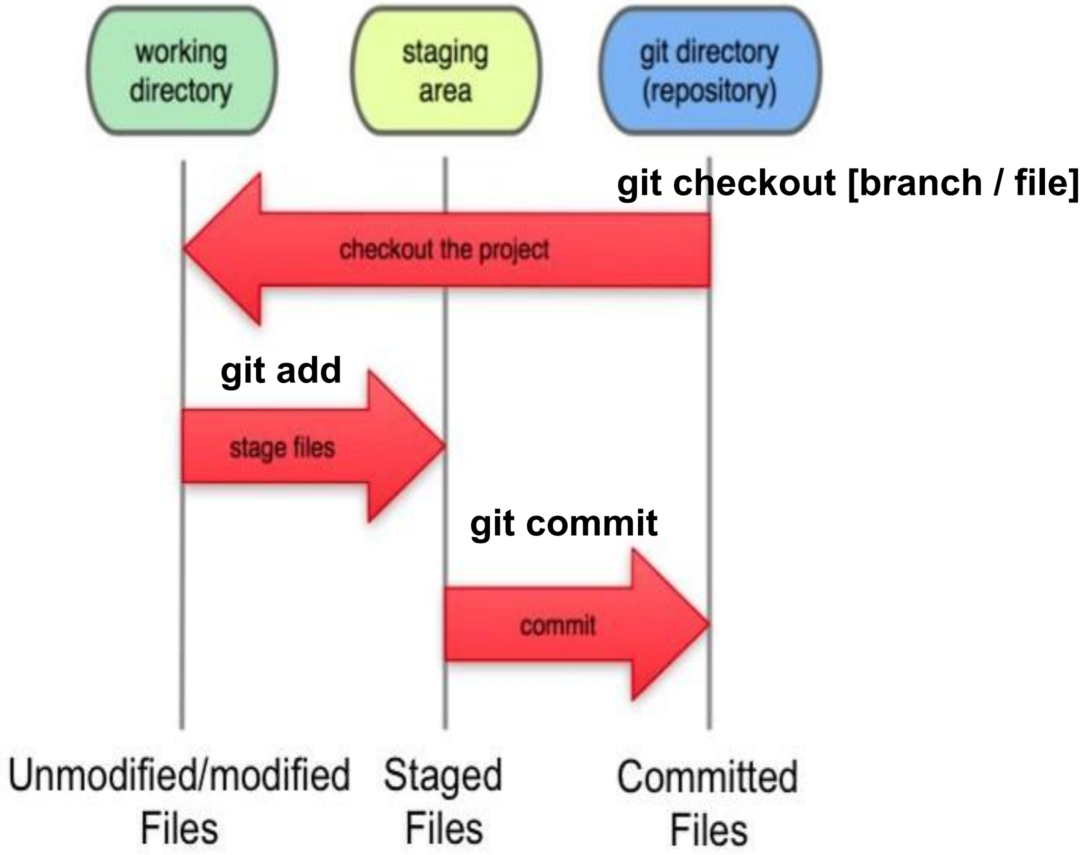
Result: Many operations are local

Local git project layout



Note: working directory sometimes called the “working tree”, staging area sometimes called the “index”.

Local git project layout



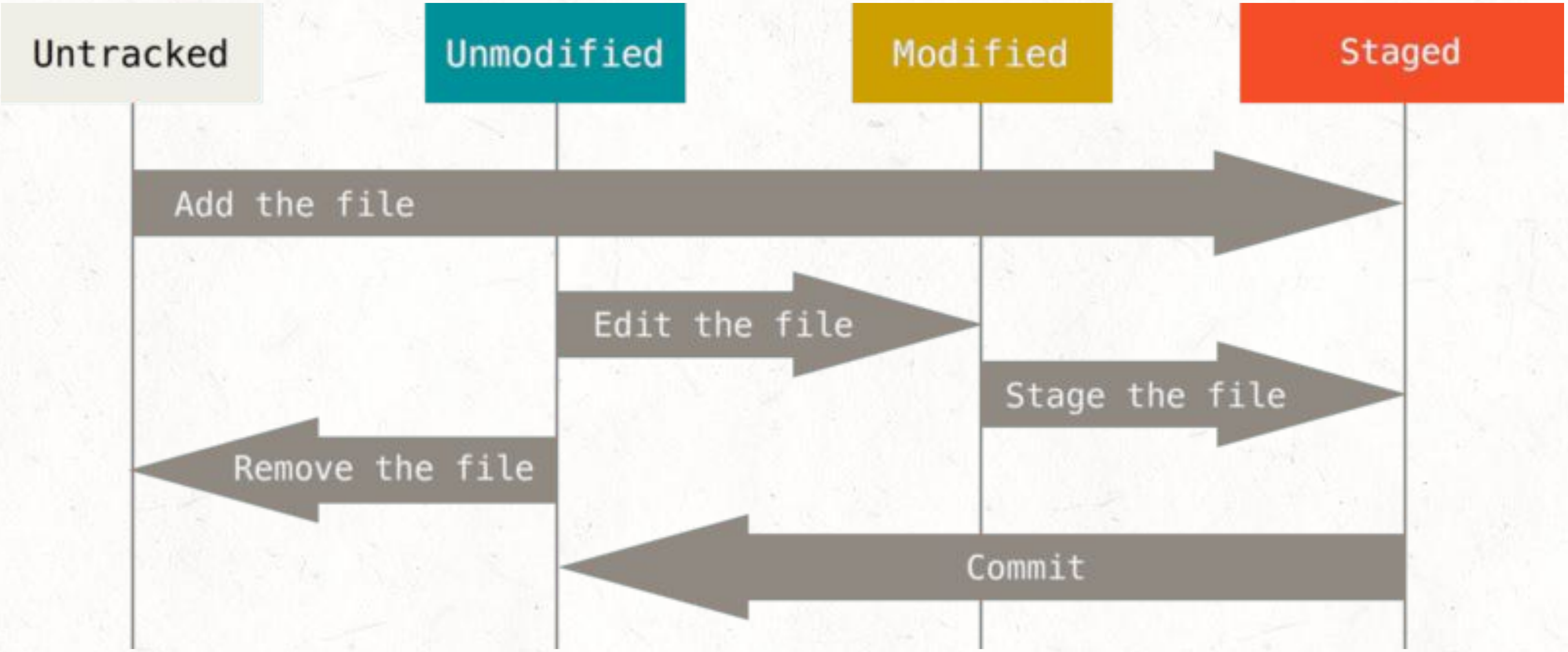
git checkout [branch / file]

git add

git commit

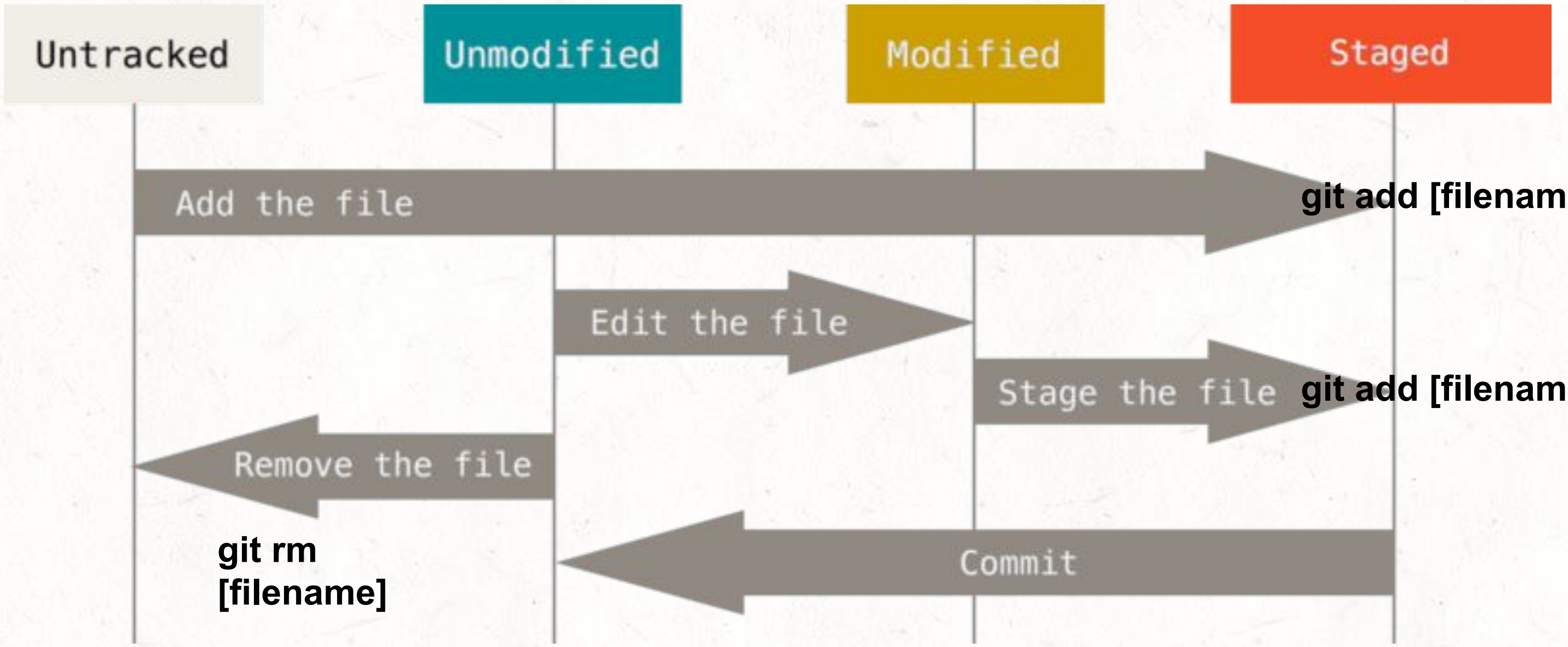
Note: working directory sometimes called the “working tree”, staging area sometimes called the “index”.

Git file lifecycle



Git file lifecycle

Modified relative to repo



git add [filename]

git add [filename]

git rm
[filename]

Git commands

git init

Initialize a new git repo git add <file>

Add/Stage a new file to your repo git

commit -m “message”

Commit staged changes to your repo

Git commands

git status

Show the status of files in the directory

git log (NOT git hist - this is just an alias)

Log of all the commits made to the repo

git diff

File differences for unstaged, modified files

git remote add <remote name> **NONLOCAL**

adding a remote server

git pull (fetch + merge) **NONLOCAL**

pull changes from a remote server

git push **NONLOCAL**

push changes to a remote server

Git commands

```
git branch <branchname>
```

Create a new branch

```
git merge <branchname>
```

Merge branch with current branch

git checkout <branchname>

Create a new branch

git checkout <filename>

Restore file from repository

Makefiles

Makefiles - The Overview

Questions you should be able to answer

- What is the point of a Makefile?
- What is the format of a Makefile?
- How do we define variables in Makefiles?
- How do we call make in subdirectories?

Makefiles - The Format

Dependencies can be **files** or **targets**

Basic Structure:

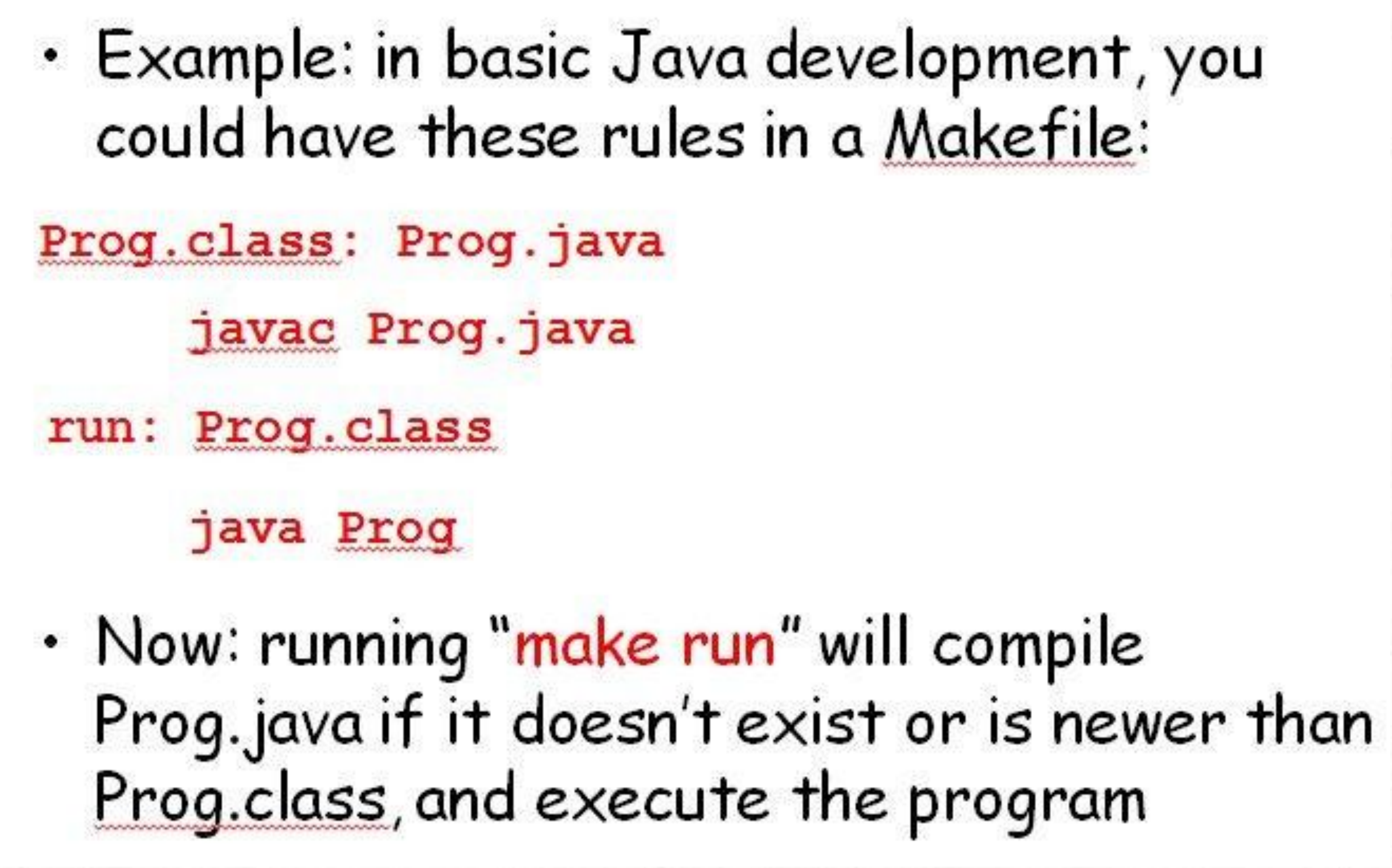
```
target: dependencies
	action
```

Note: Each action must be tab-indented

Sample makefile

```
lunch:
        make sandwich
        make clean
bread.baked:
    bake bread
butter.made:
    make butter
sandwich: bread.baked butter.made
        cut bread
        spread butter
clean:
        eat sandwich
```

Makefile - The Example



- Example: in basic Java development, you could have these rules in a Makefile:

```
Prog.class: Prog.java
        javac Prog.java
run: Prog.class
        java Prog
```

- Now: running "make run" will compile Prog.java if it doesn't exist or is newer than Prog.class, and execute the program

Calling another Makefile

To execute the Makefile of subfolder lib, from inside the parent directory Makefile, use:

```
make -C lib/ target
```

replace target with the target name (ie. new, clean etc)

Ant & XML

Ant - Another Neat Tool

- Tool for automated software builds - Very useful in industry with Java dev.
- Similar to makefiles, but specifically for Java
- Uses XML as its format to describe the building process and its dependencies
- By default the XML file is named **build.xml**
- property names are like variables…
    - You can obtain the value of a property name using the following syntax - **${property_name}**
- How to create a variable?

    <property name=”public_dir” location=”~/usr/public” />

- you can also append property names to strings …
    - assuming “public_dir” is set above
        - “${public_dir}/gary” would have the value “~/usr/public/gary”

Ant - Another Neat Tool

- **Important things that you should know for the final.**
  - What happens if we do not specify a target when running ant?
    - for example, in Makefiles, it’ll run the default target. If there are no default target then it will run the first target, this process is same for ant.

  - How do you access the values in a property name?

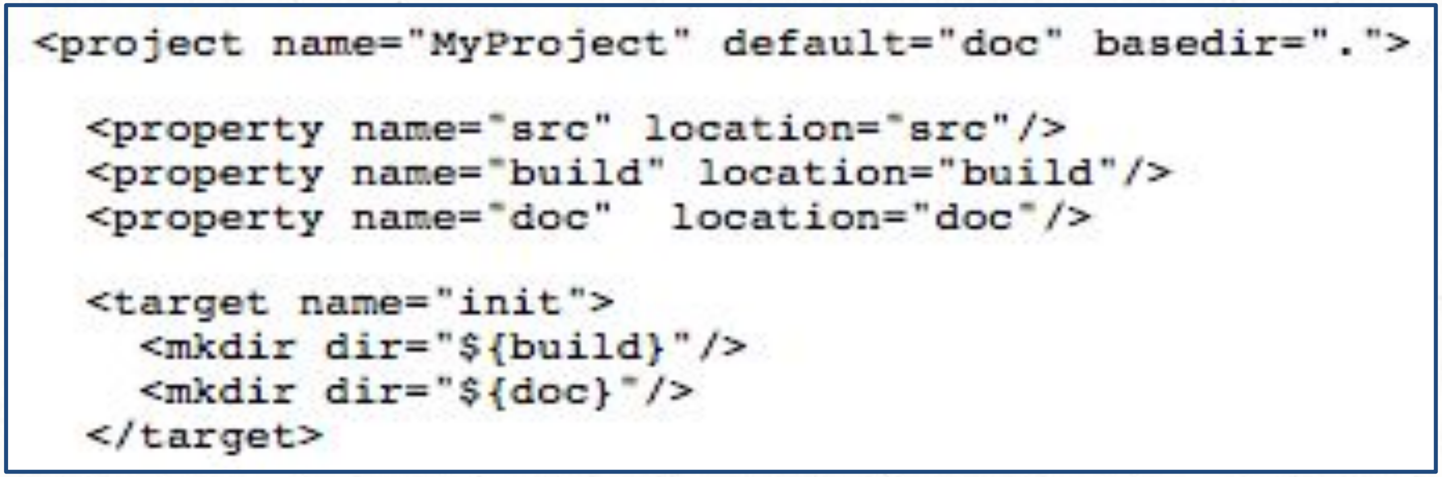
  - How do you write property names?
  - At least 1 target
    - Target **contains a name** and optionally **depends**
      - **description**
        - Each target contains multiple tasks, which are actions that need execution
        - There can be dependencies between targets

XML Example

What is the default target?

```
<project name =”myproj” default=”init” basedir=”.”>
        <property name=”n” location=”nate” />
        <property name=”g” location=”gerard” />
        <property name=”j” location=”jacalyn” />
        <target name=”init1”>
                //xml stuff
        </target>

        <target name=”init2”>
                //xml stuff
        </target>
</project>
```

XML Example

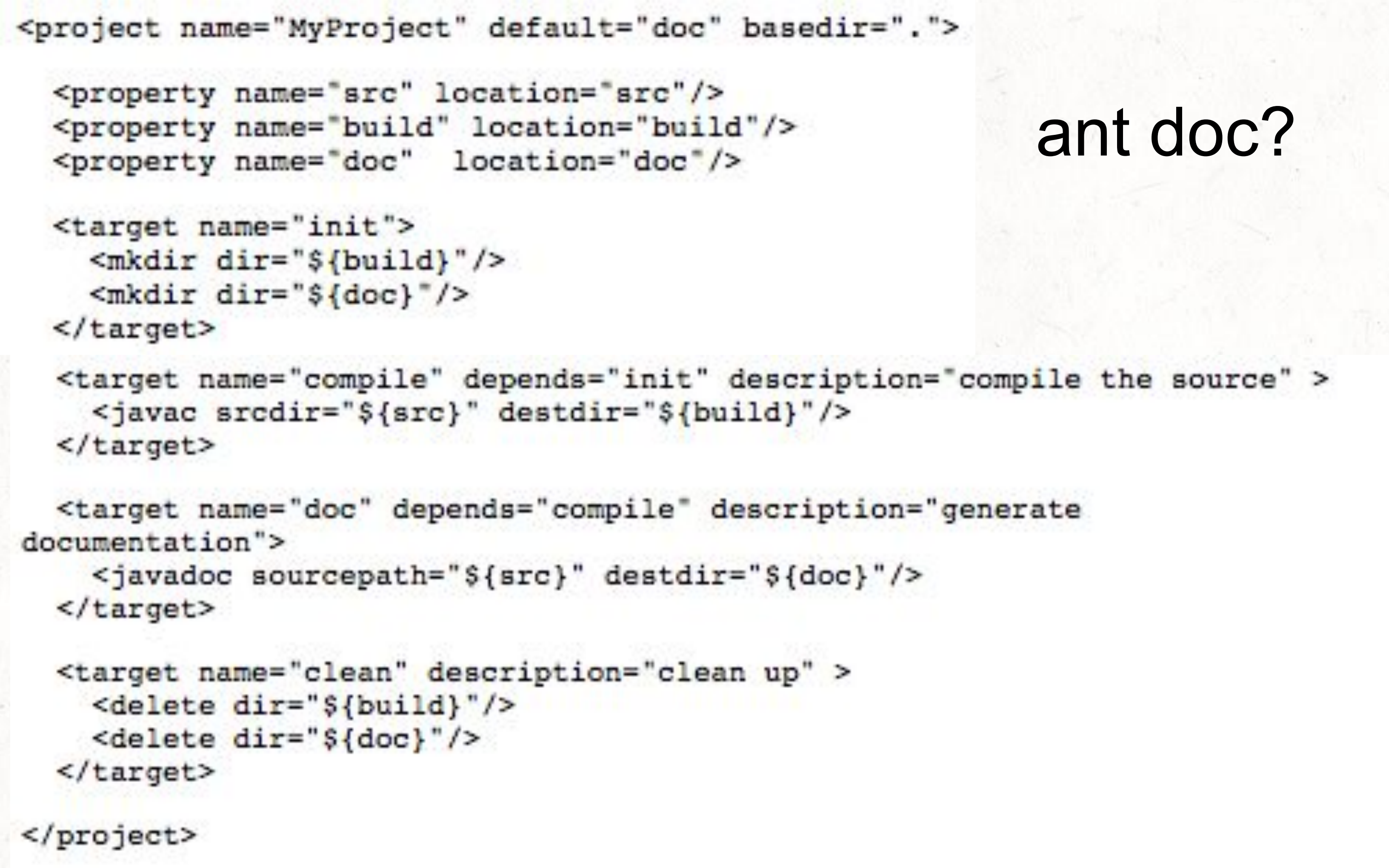


```
<project name="MyProject" default="doc" basedir=".">

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="doc"  location="doc"/>

  <target name="init">
    <mkdir dir="${build}"/>
    <mkdir dir="${doc}"/>
  </target>
```

```
src=”src”
build=”build”
doc=”doc”
init:
	mkdir $(build)
	mkdir $(doc)
```

Makefile
equivalent

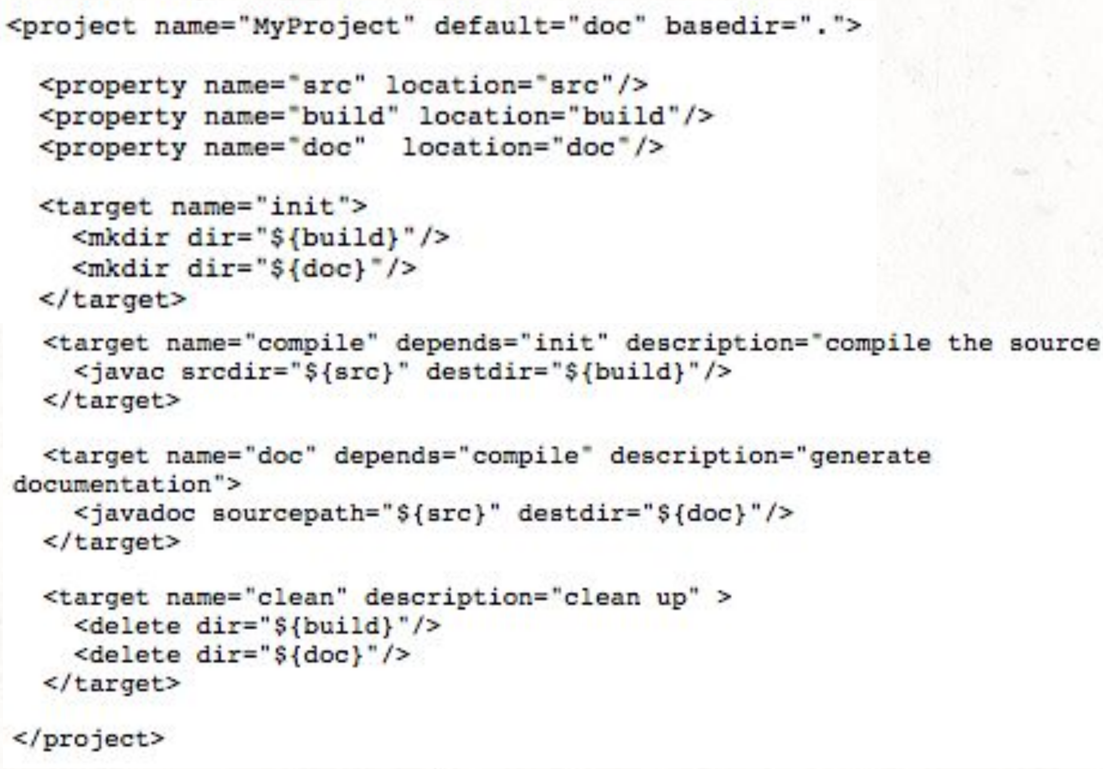
XML Example



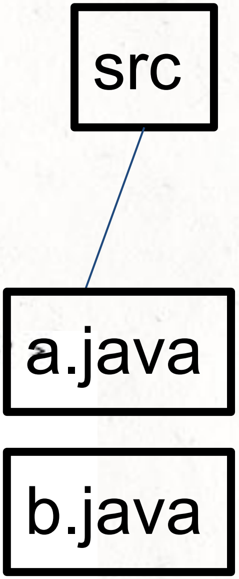
```
<project name="MyProject" default="doc" basedir=".">

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="doc"  location="doc"/>

  <target name="init">
    <mkdir dir="${build}"/>
    <mkdir dir="${doc}"/>
  </target>

  <target name="compile" depends="init" description="compile the source" >
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="doc" depends="compile" description="generate
documentation">
    <javadoc sourcepath="${src}" destdir="${doc}"/>
  </target>

  <target name="clean" description="clean up" >
    <delete dir="${build}"/>
    <delete dir="${doc}"/>
  </target>

</project>
```

ant doc?

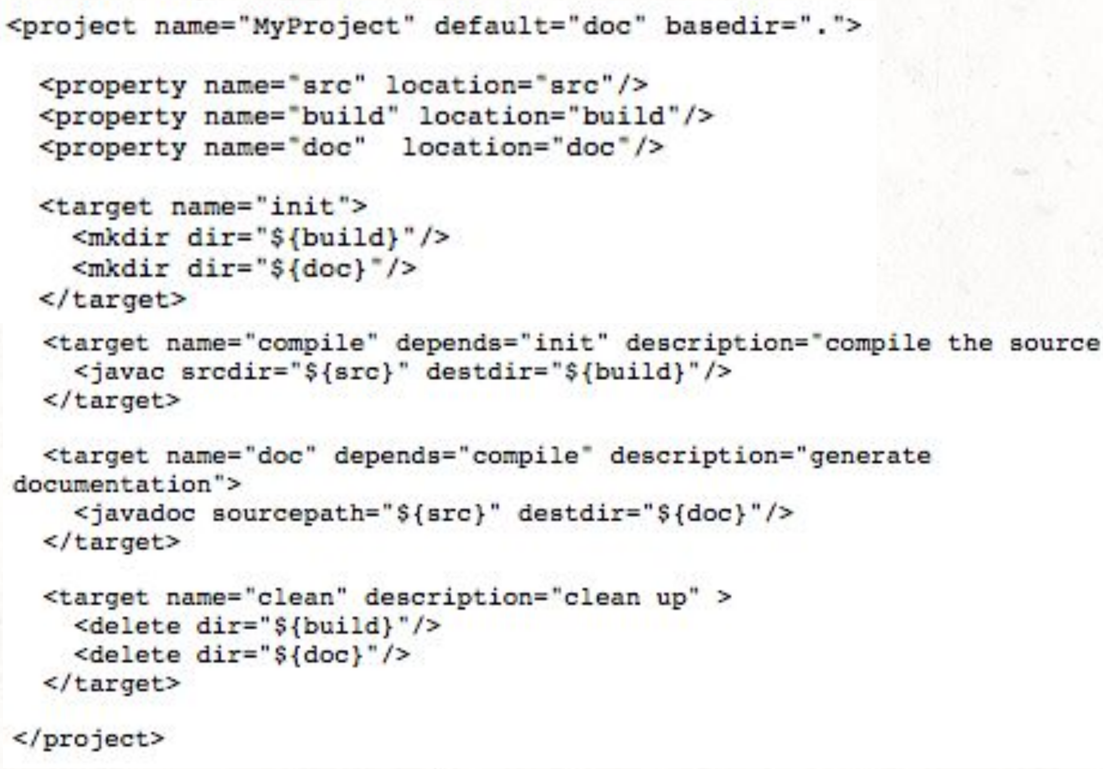
XML Example



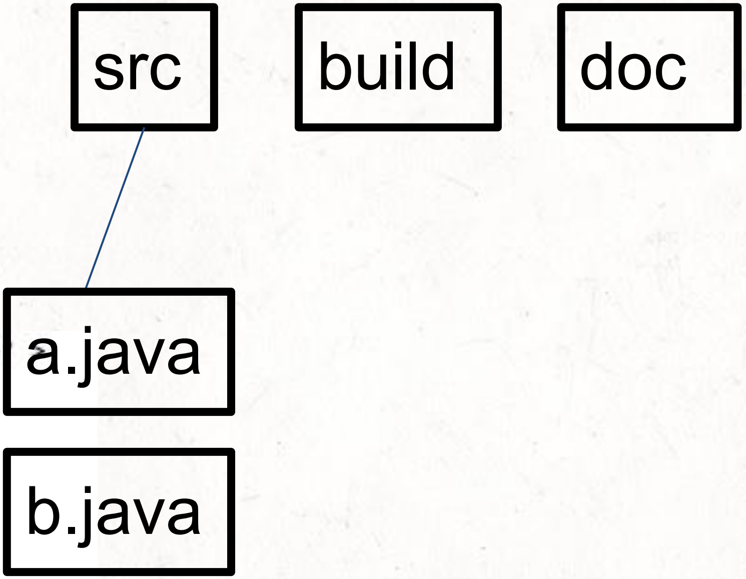
```
<project name="MyProject" default="doc" basedir=".">

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="doc"  location="doc"/>

  <target name="init">
    <mkdir dir="${build}"/>
    <mkdir dir="${doc}"/>
  </target>

  <target name="compile" depends="init" description="compile the source">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="doc" depends="compile" description="generate
documentation">
    <javadoc sourcepath="${src}" destdir="${doc}"/>
  </target>

  <target name="clean" description="clean up" >
    <delete dir="${build}"/>
    <delete dir="${doc}"/>
  </target>

</project>
```



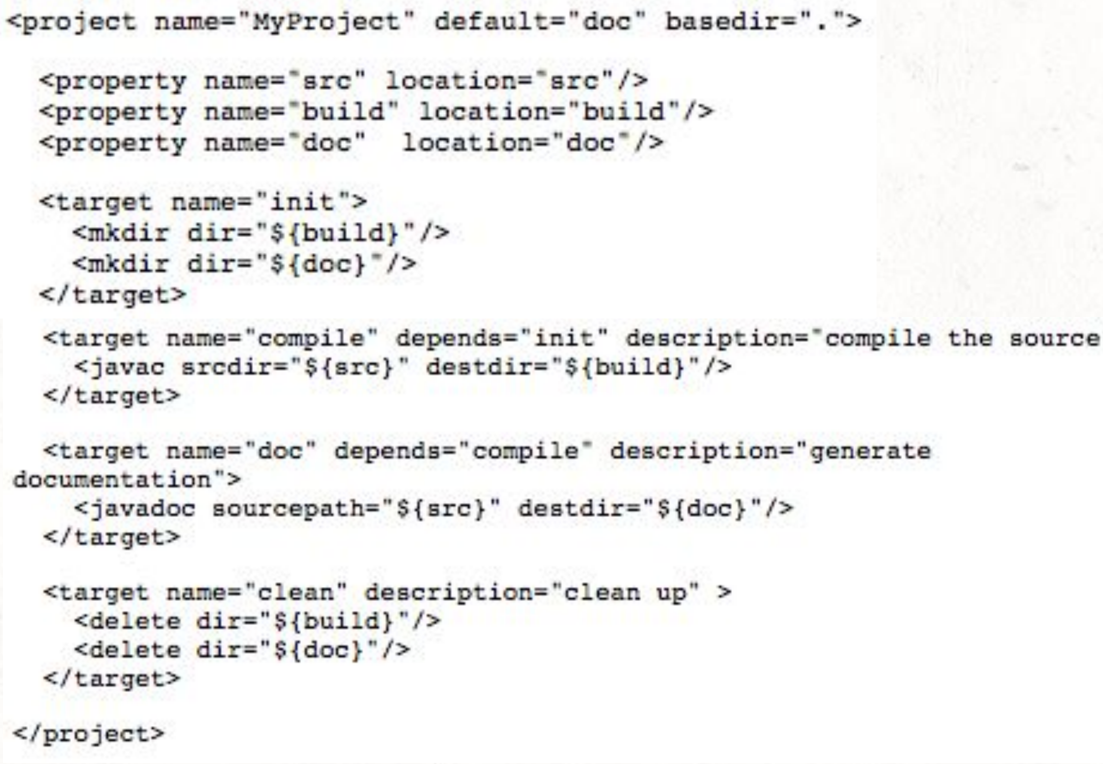
XML Example



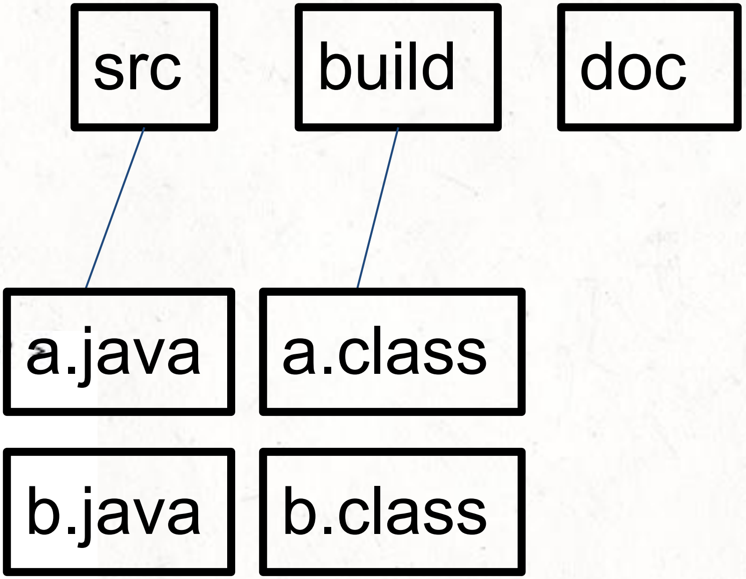
```
<project name="MyProject" default="doc" basedir=".">

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="doc"  location="doc"/>

  <target name="init">
    <mkdir dir="${build}"/>
    <mkdir dir="${doc}"/>
  </target>

  <target name="compile" depends="init" description="compile the source"
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="doc" depends="compile" description="generate
documentation">
    <javadoc sourcepath="${src}" destdir="${doc}"/>
  </target>

  <target name="clean" description="clean up" >
    <delete dir="${build}"/>
    <delete dir="${doc}"/>
  </target>

</project>
```



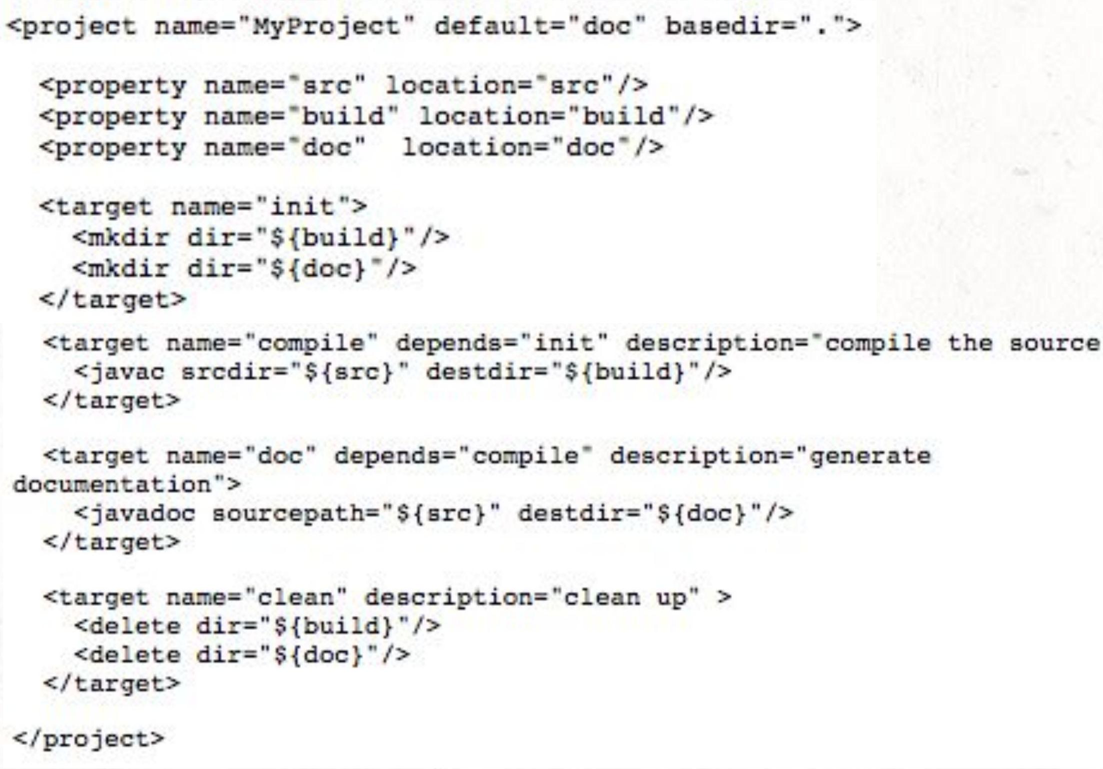
XML Example



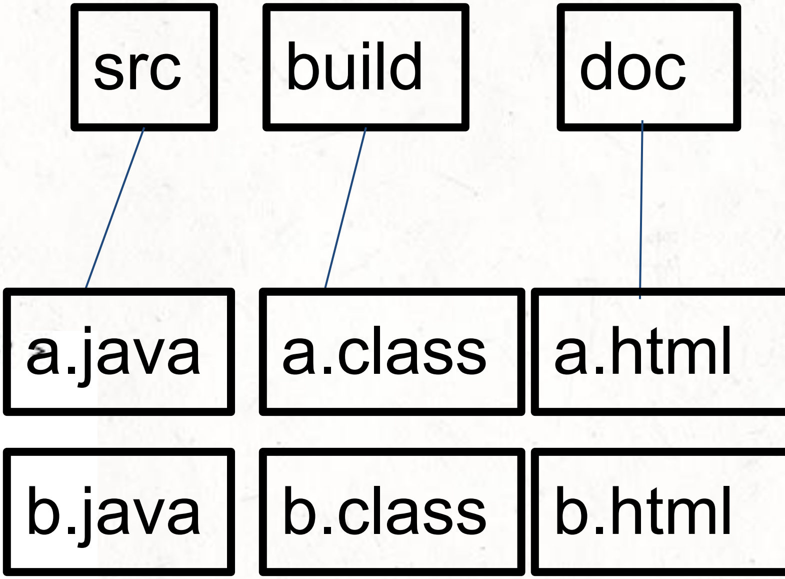
```
<project name="MyProject" default="doc" basedir=".">

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="doc"  location="doc"/>

  <target name="init">
    <mkdir dir="${build}"/>
    <mkdir dir="${doc}"/>
  </target>

  <target name="compile" depends="init" description="compile the source
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="doc" depends="compile" description="generate
documentation">
    <javadoc sourcepath="${src}" destdir="${doc}"/>
  </target>

  <target name="clean" description="clean up" >
    <delete dir="${build}"/>
    <delete dir="${doc}"/>
  </target>

</project>
```



XML Example



```
<project name="MyProject" default="doc" basedir=".">

  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="doc"  location="doc"/>

  <target name="init">
    <mkdir dir="${build}"/>
    <mkdir dir="${doc}"/>
  </target>

  <target name="compile" depends="init" description="compile the source
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="doc" depends="compile" description="generate
documentation">
    <javadoc sourcepath="${src}" destdir="${doc}"/>
  </target>

  <target name="clean" description="clean up" >
    <delete dir="${build}"/>
    <delete dir="${doc}"/>
  </target>

</project>
```



src

build

doc

a.java

a.class

a.html

b.java

b.class

b.html

Java Logging Framework (Lab 8)

Logging

- Logging means:
  - automatically recording and diagnostic output from a program

Logging

- Examples
  - A web server could log IP address of incoming http requests
  - A mail server could log basic info about each email received etc.

Logging

We can use a logging framework!

ex. Java’s logging framework

```
import java.util.logging
```

The Level Class

The Level class contains constants that:

1) specify the importance level of log messages

2) control which log records are actually logged

The Level Class - Constants

Level.SEVERE

Level.____________

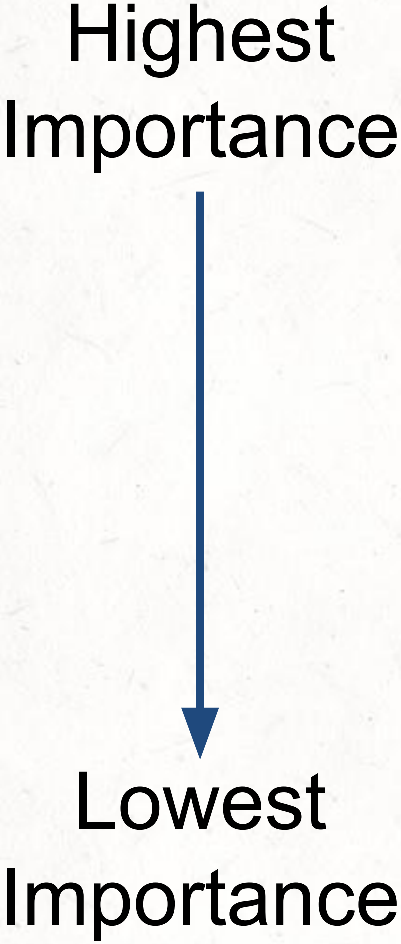
Level.____________

Level.____________

Level.____________

Level.____________

Level.FINEST



Highest Importance

Lowest Importance

The Level Class

Level.SEVERE

Level.WARNING

Level.____________

Level.____________

Level.____________

Level.____________

Level.FINEST

Highest Importance

↓

Lowest Importance

The Level Class

Level.SEVERE

Level.WARNING

Level.INFO

Level.___________

Level.___________

Level.___________

Level.FINEST

Highest Importance

↓

Lowest Importance

The Level Class

Level.SEVERE

Level.WARNING

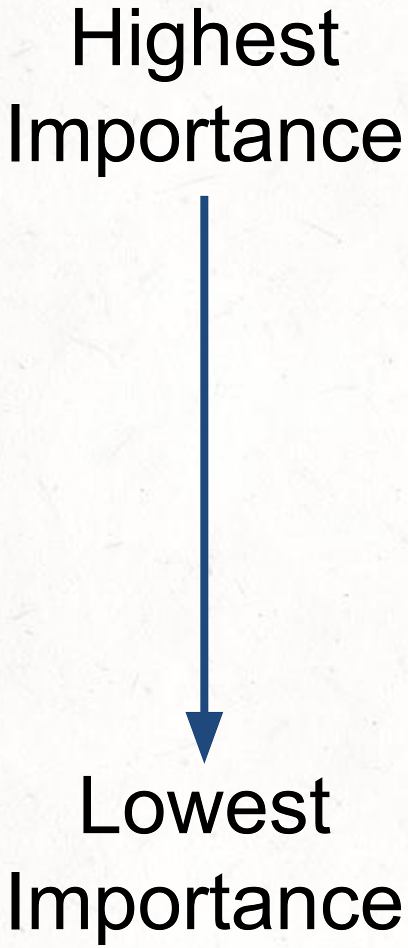
Level.INFO

Level.CONFIG

Level.__________

Level.__________

Level.FINEST



Highest Importance

Lowest Importance

The Level Class

Level.SEVERE

Level.WARNING

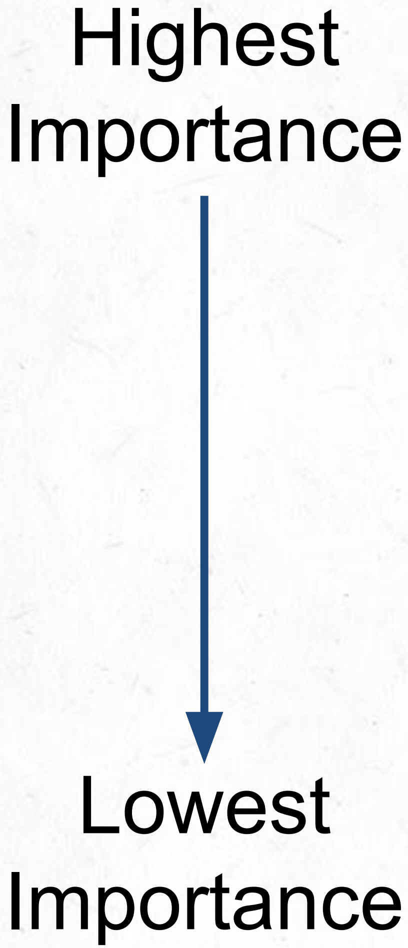
Level.INFO

Level.CONFIG

Level.FINE

Level.__________

Level.FINEST



Highest Importance

Lowest Importance

The Level Class

Level.SEVERE

Level.WARNING

Level.INFO

Level.CONFIG

Level.FINE

Level.FINER

Level.FINEST

Highest Importance

Lowest Importance

The Level Class

- Level.SEVERE
- Level.WARNING
- Level.INFO
- Level.CONFIG
- Level.FINE
- Level.FINER
- Level.FINEST

Highest Importance

Lowest Importance

The Level Class

Two more!

`Level.____`	log every message
`Level.____`	ignore every message

The Level Class

Two more!

`Level.ALL`	log every message
`Level.___`	ignore every message

The Level Class

Two more!

`Level.ALL`	log every message
`Level.OFF`	ignore every message

logging properties file

Look in Lab 8 for a sample properties file. (It’s called logging.properties)

```
….
//Change this line to change what
//gets printed
.level=<level>
….
```

How to run with a properties file

```
java -Djava.util.logging.config.file=logging.properties [MainClass]
```

logging properties file

Look in Lab 8 for a sample properties file. (It's called logging.properties)

```
....
//Change this line to change what
//gets printed
.level=<level>
....
```

How to run with a properties file

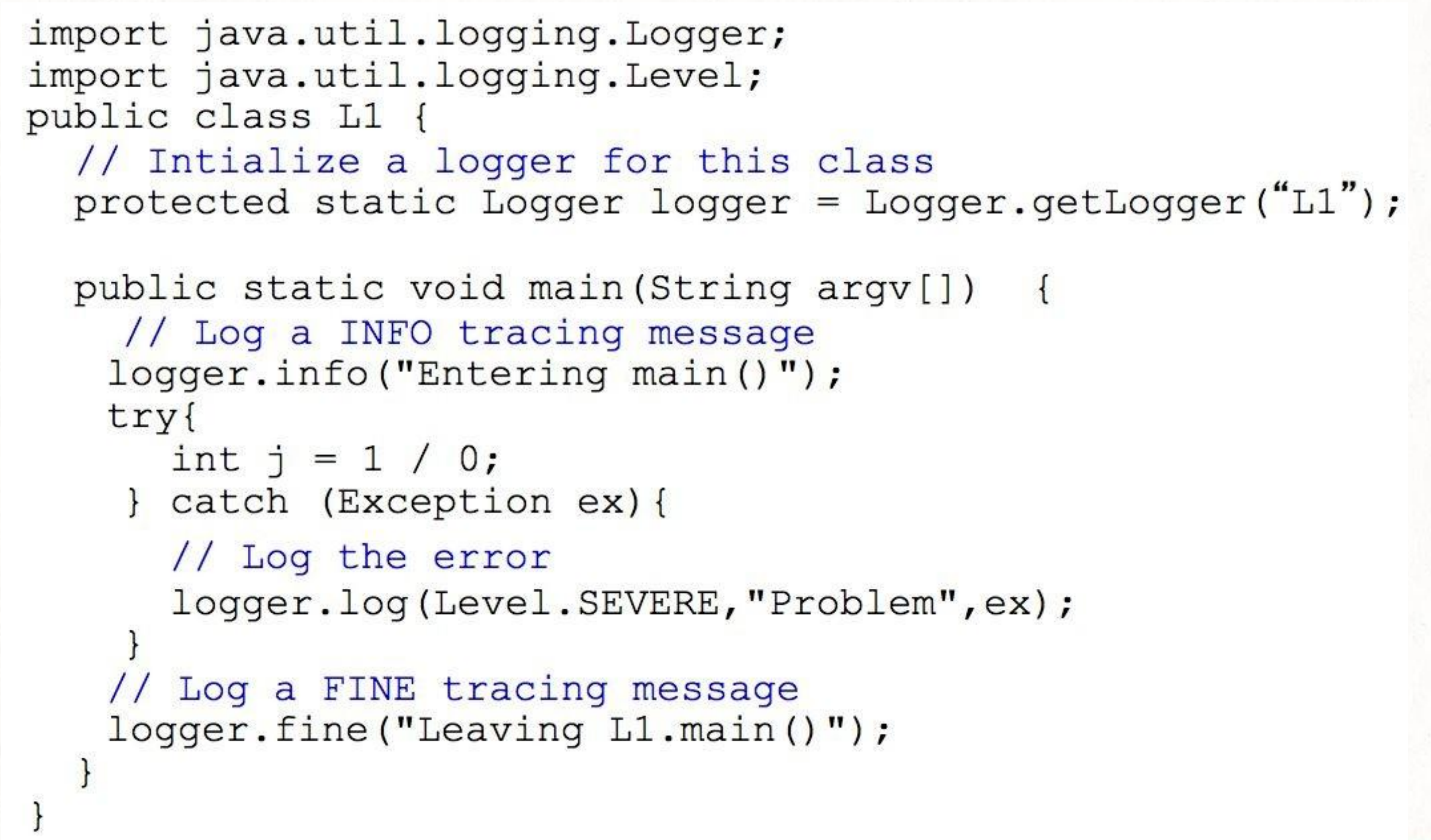
```
java -Djava.util.logging.config.file=logging.properties [MainClass]
```

When is a message logged ?

For the message to be logged:

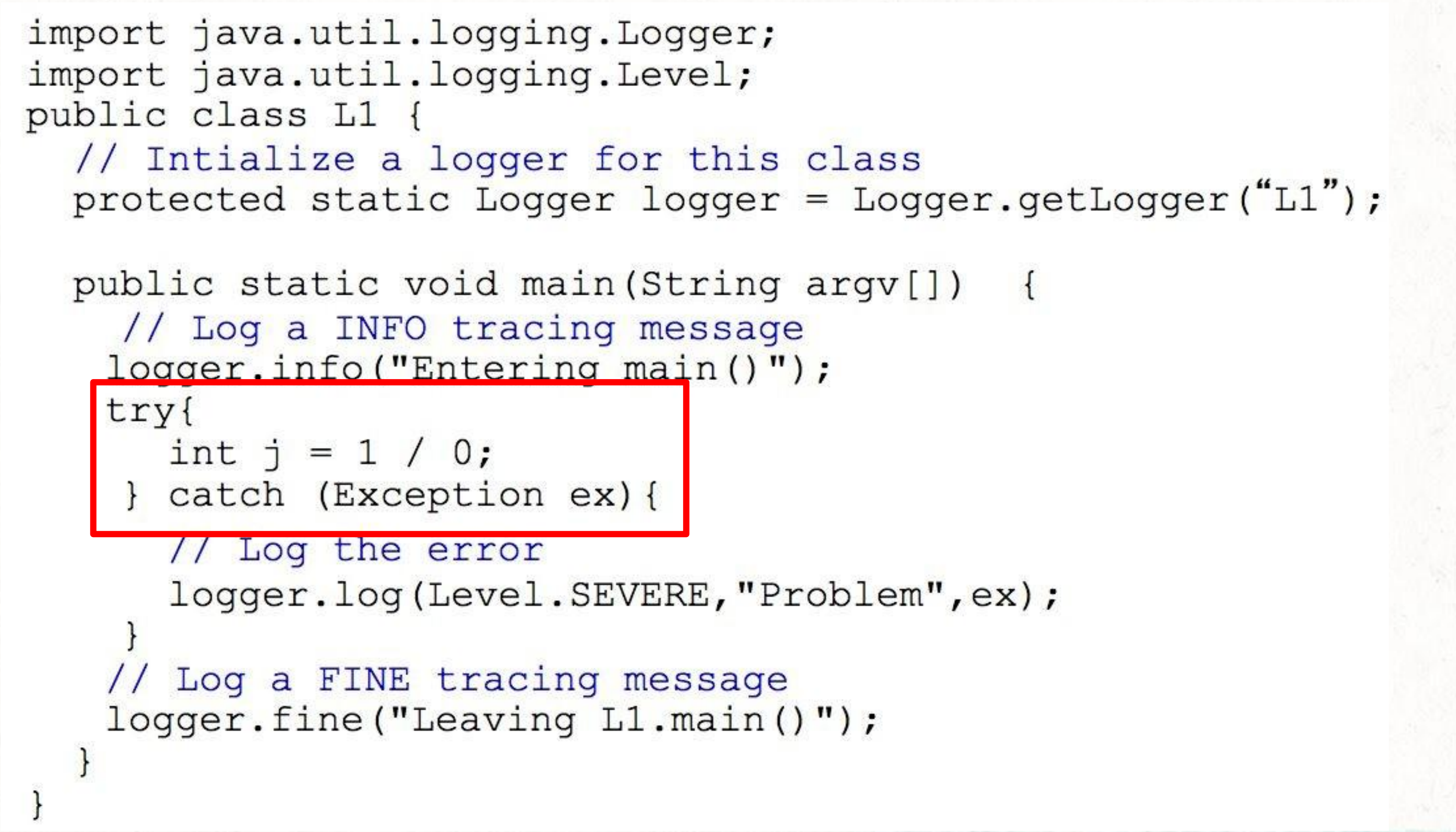
Level of the message ≥ Level of the logger

Example Logger



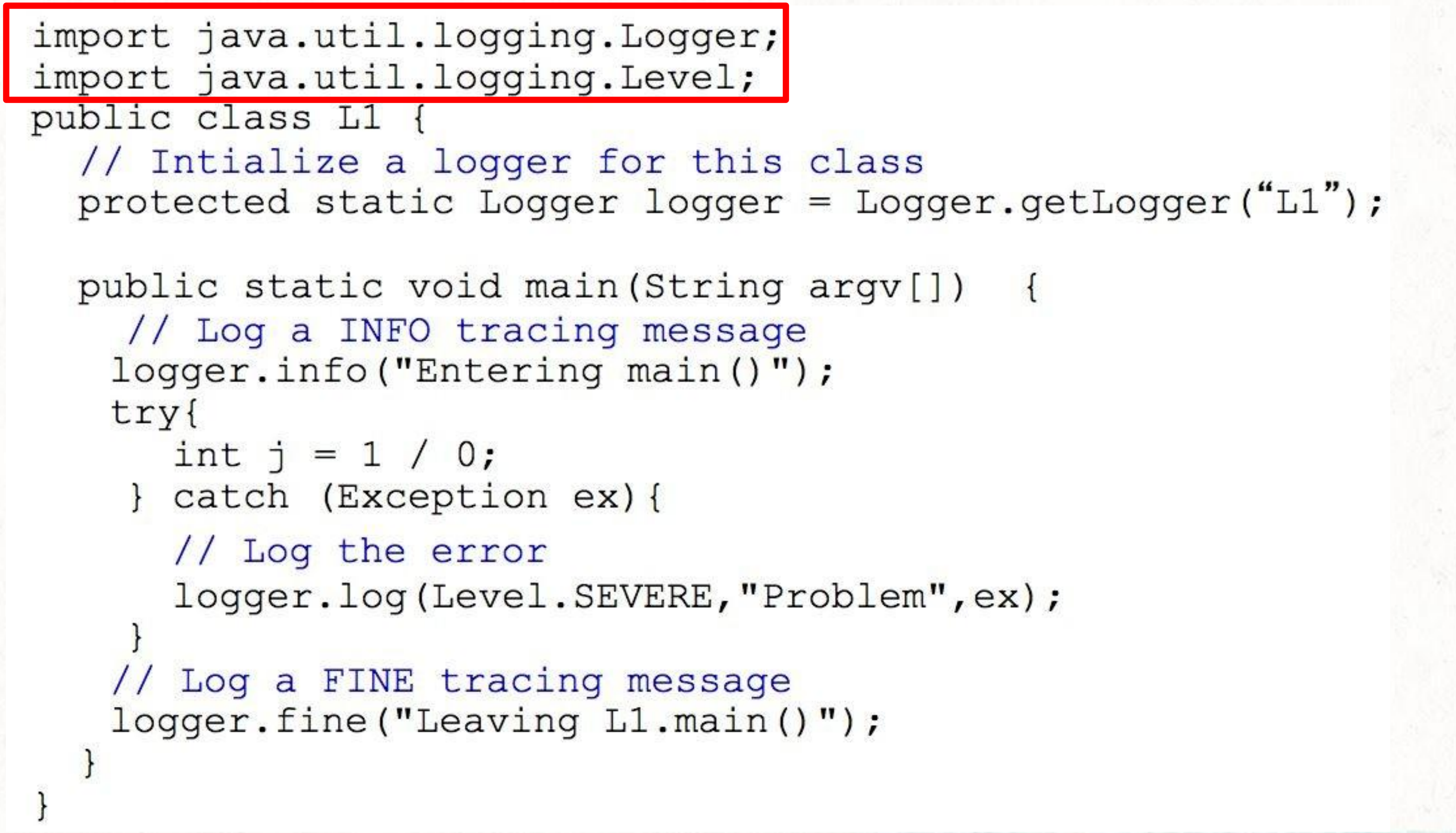
```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
  // Intialize a logger for this class
  protected static Logger logger = Logger.getLogger(“L1”);

  public static void main(String argv[])  {
    // Log a INFO tracing message
   logger.info("Entering main()");
   try{
      int j = 1 / 0;
    } catch (Exception ex){
      // Log the error
      logger.log(Level.SEVERE,"Problem",ex);
    }
   // Log a FINE tracing message
   logger.fine("Leaving L1.main()");
  }
}
```

Example Logger



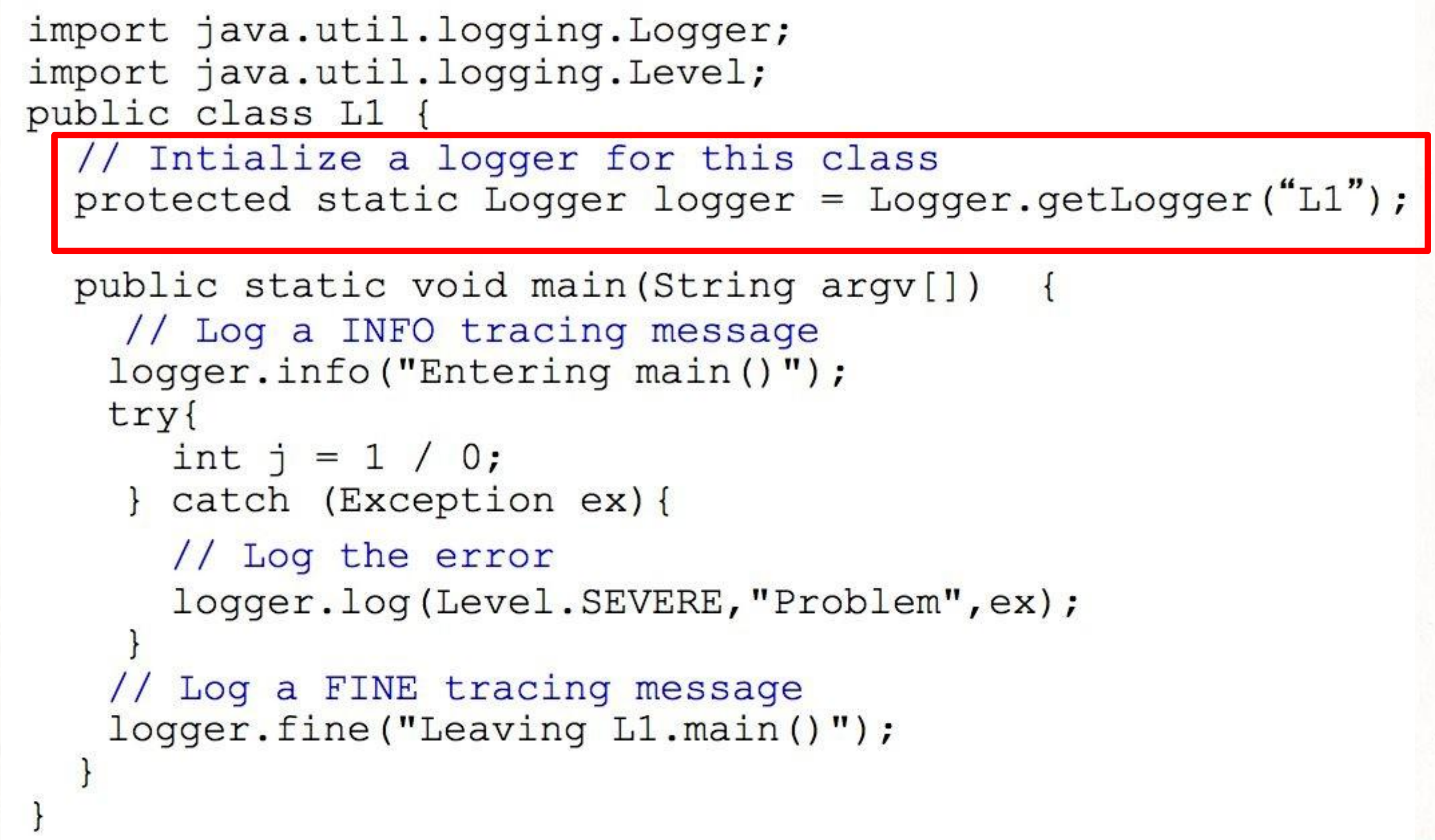
```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
  // Intialize a logger for this class
  protected static Logger logger = Logger.getLogger(“L1”);

  public static void main(String argv[])  {
    // Log a INFO tracing message
   logger.info("Entering main()");
   try{
      int j = 1 / 0;
    } catch (Exception ex){
      // Log the error
      logger.log(Level.SEVERE,"Problem",ex);
    }
   // Log a FINE tracing message
   logger.fine("Leaving L1.main()");
  }
}
```

Example Logger



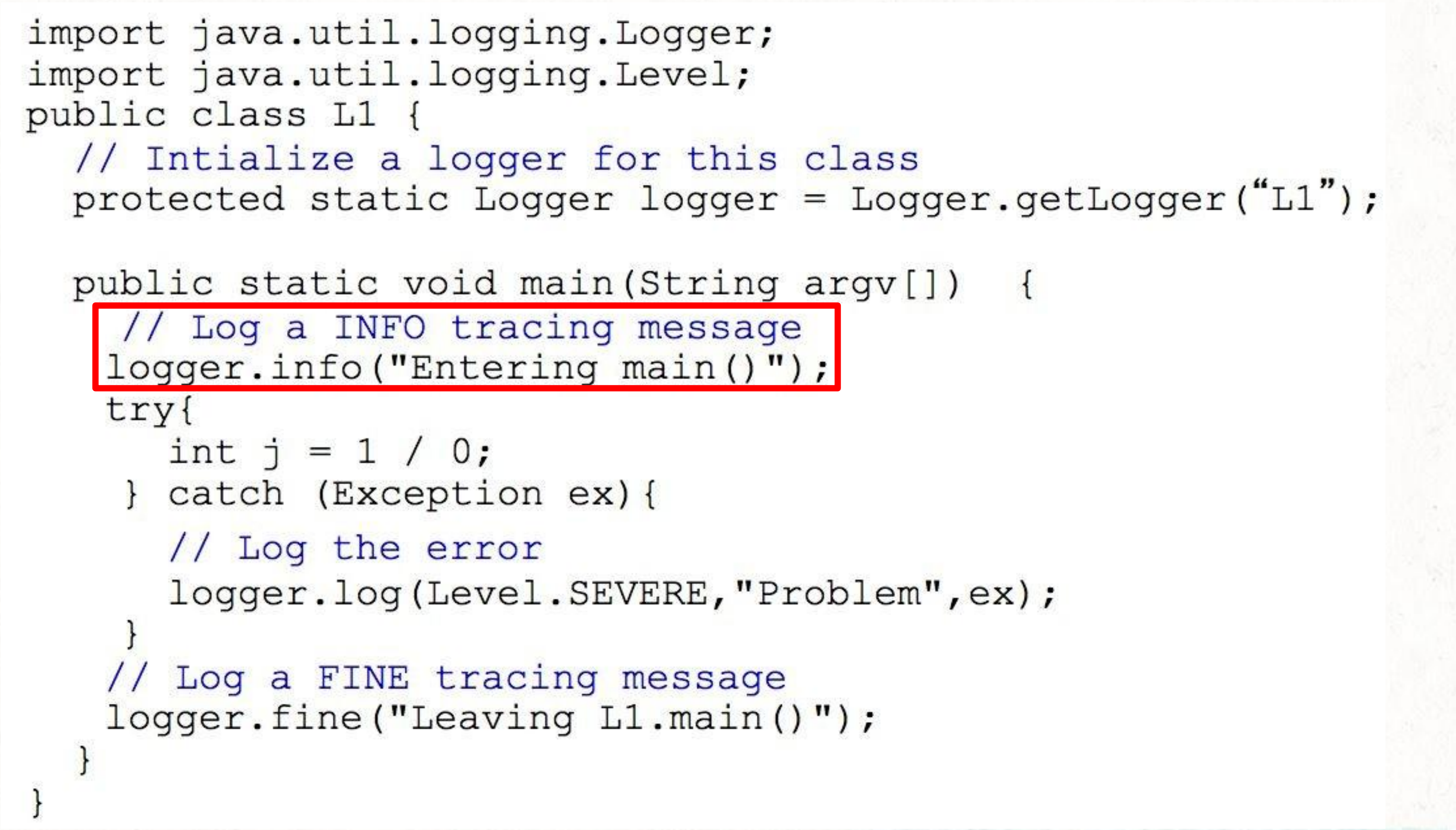
```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
  // Intialize a logger for this class
  protected static Logger logger = Logger.getLogger(“L1”);

  public static void main(String argv[])  {
    // Log a INFO tracing message
   logger.info("Entering main()");
   try{
      int j = 1 / 0;
    } catch (Exception ex){
      // Log the error
      logger.log(Level.SEVERE,"Problem",ex);
    }
   // Log a FINE tracing message
   logger.fine("Leaving L1.main()");
  }
}
```

Example Logger



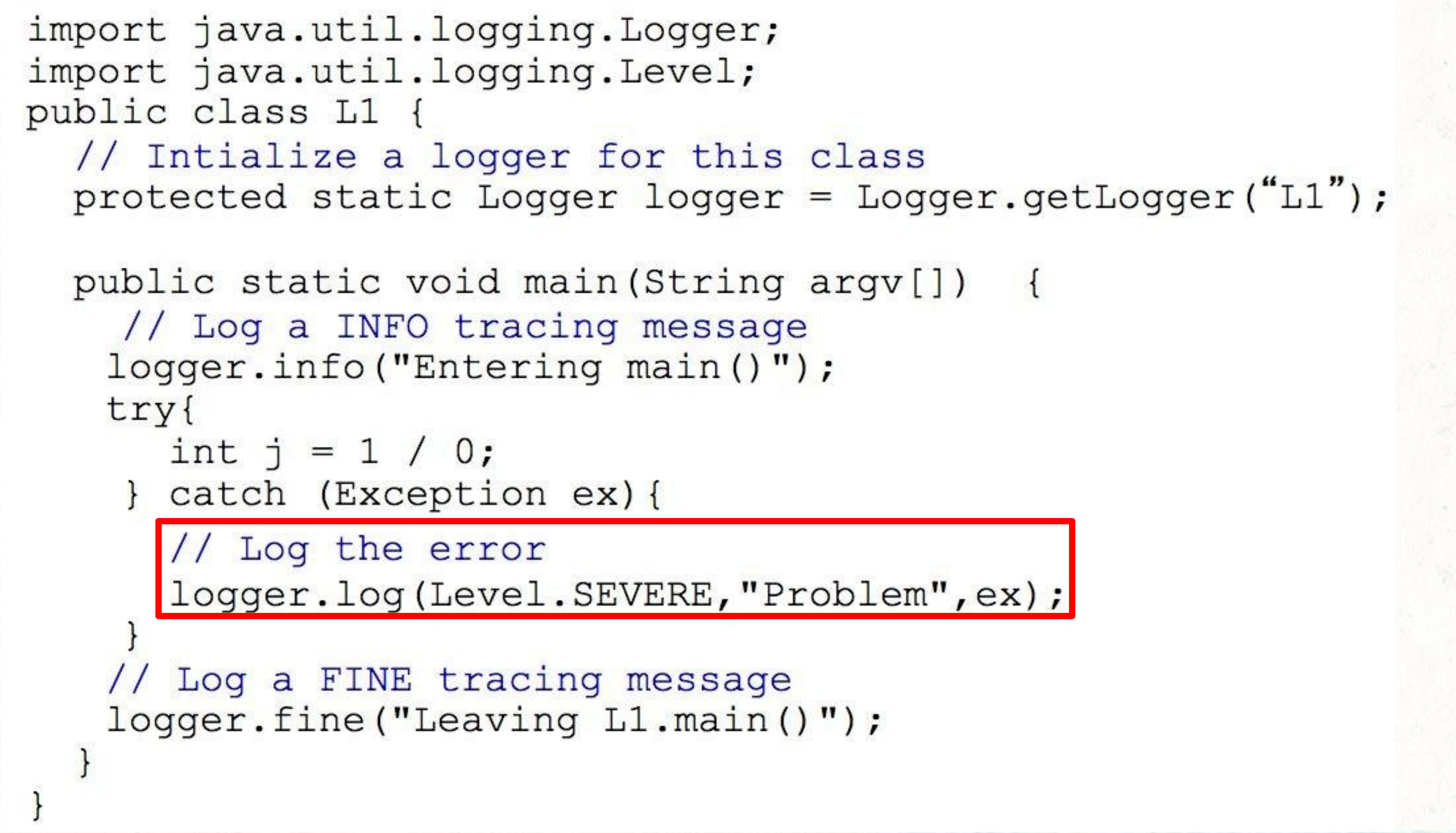
```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
  // Intialize a logger for this class
  protected static Logger logger = Logger.getLogger(“L1”);

  public static void main(String argv[])  {
    // Log a INFO tracing message
   logger.info("Entering main()");
   try{
      int j = 1 / 0;
    } catch (Exception ex){
      // Log the error
      logger.log(Level.SEVERE,"Problem",ex);
    }
   // Log a FINE tracing message
   logger.fine("Leaving L1.main()");
  }
}
```

Example Logger



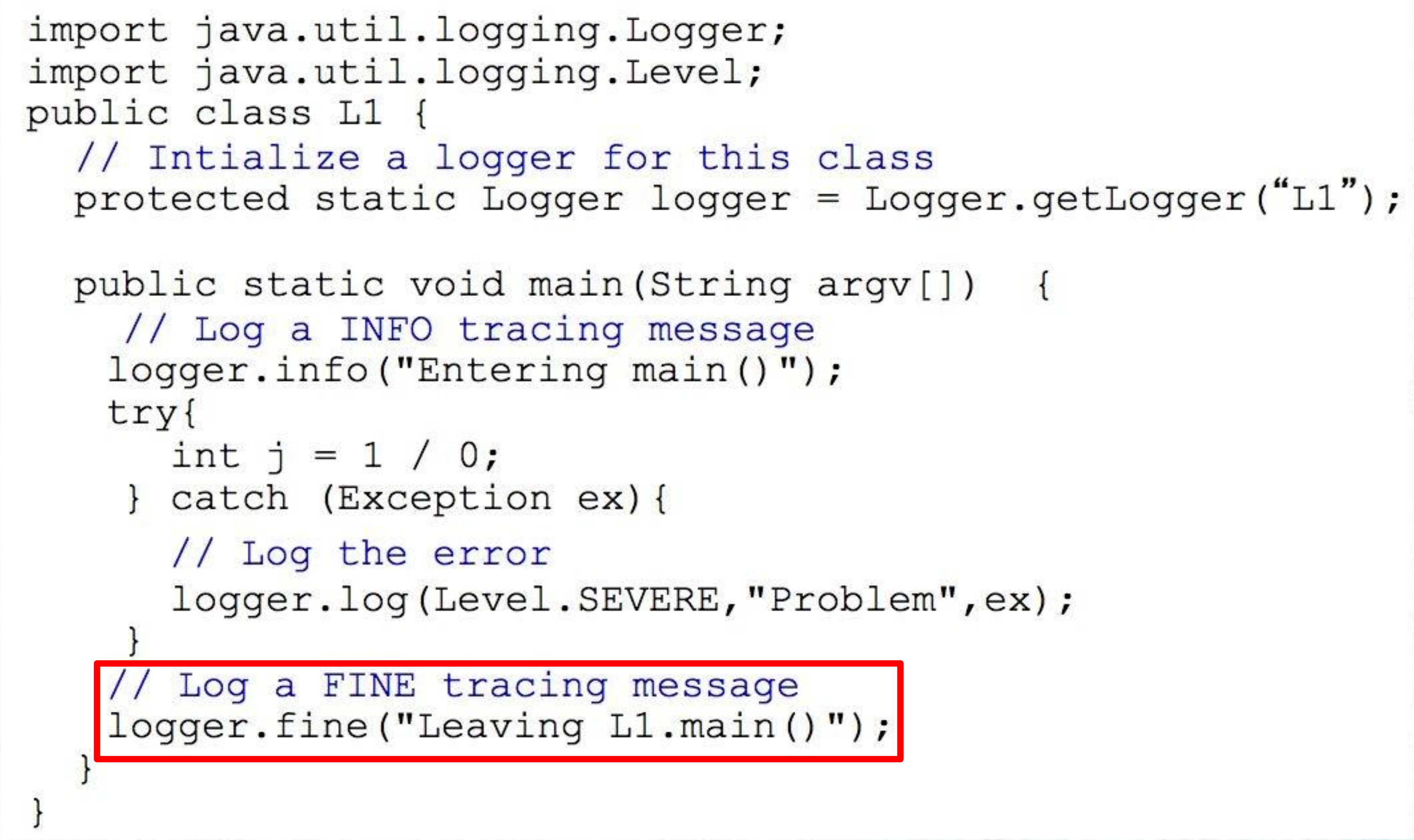
```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
  // Intialize a logger for this class
  protected static Logger logger = Logger.getLogger(“L1”);

  public static void main(String argv[])  {
    // Log a INFO tracing message
   logger.info("Entering main()");
   try{
      int j = 1 / 0;
    } catch (Exception ex){
      // Log the error
      logger.log(Level.SEVERE,"Problem",ex);
    }
   // Log a FINE tracing message
   logger.fine("Leaving L1.main()");
  }
}
```

Example Logger



```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
  // Intialize a logger for this class
  protected static Logger logger = Logger.getLogger(“L1”);

  public static void main(String argv[])  {
    // Log a INFO tracing message
   logger.info("Entering main()");
   try{
      int j = 1 / 0;
    } catch (Exception ex){
      // Log the error
      logger.log(Level.SEVERE,"Problem",ex);
    }
   // Log a FINE tracing message
   logger.fine("Leaving L1.main()");
  }
}
```

Example Logger



```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
  // Intialize a logger for this class
  protected static Logger logger = Logger.getLogger(“L1”);

  public static void main(String argv[])  {
    // Log a INFO tracing message
    logger.info("Entering main()");
    try{
       int j = 1 / 0;
     } catch (Exception ex){
       // Log the error
       logger.log(Level.SEVERE,"Problem",ex);
     }
    // Log a FINE tracing message
    logger.fine("Leaving L1.main()");
  }
}
```

When is a message logged ?

For the message to be logged:

Level of the message ≥ Level of the logger

The Level Class

Level.SEVERE

Level.WARNING

Level.INFO

Level.CONFIG

Level.FINE

Level.FINER

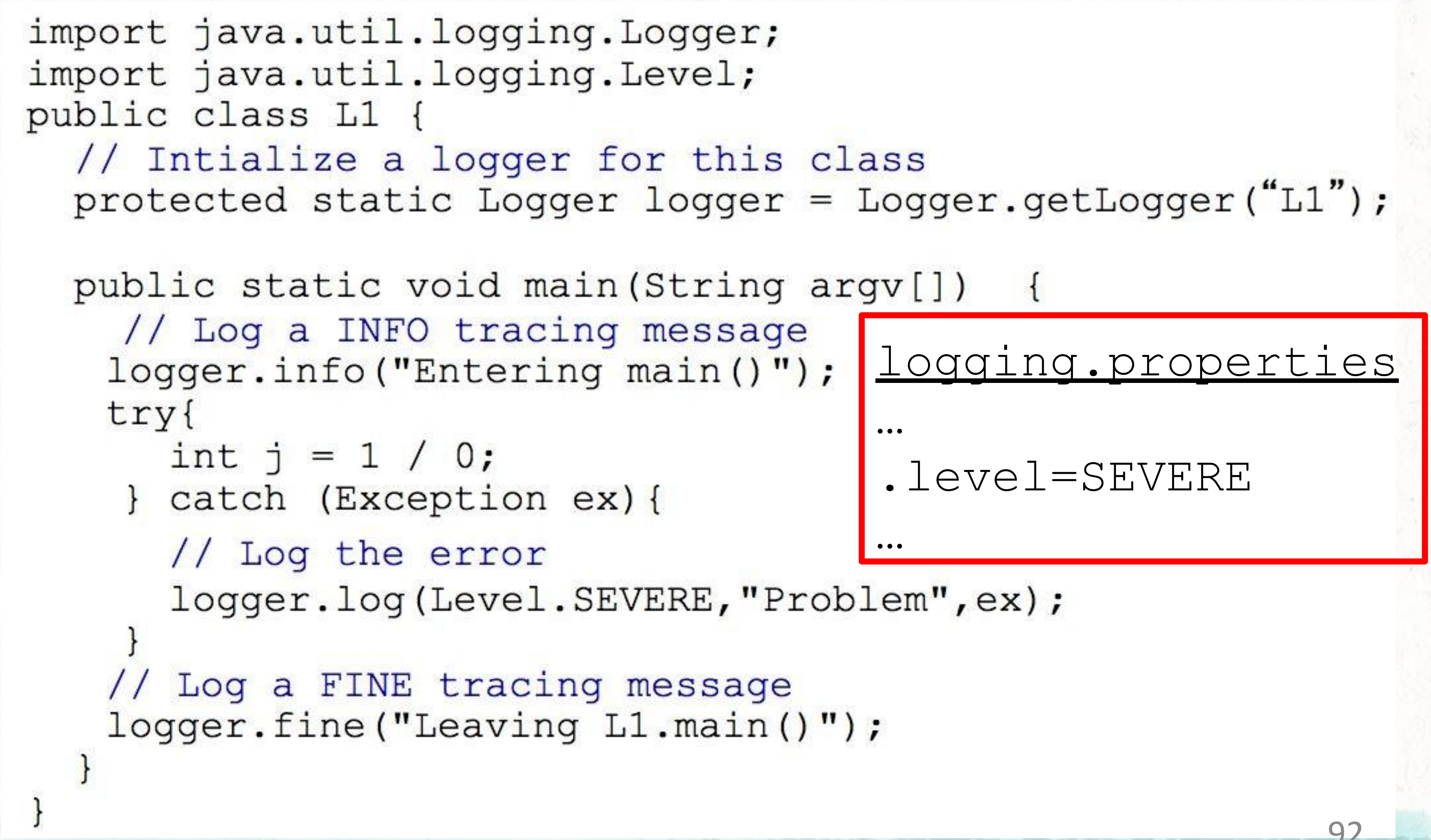
Level.FINEST

Highest Importance

↓

Lowest Importance

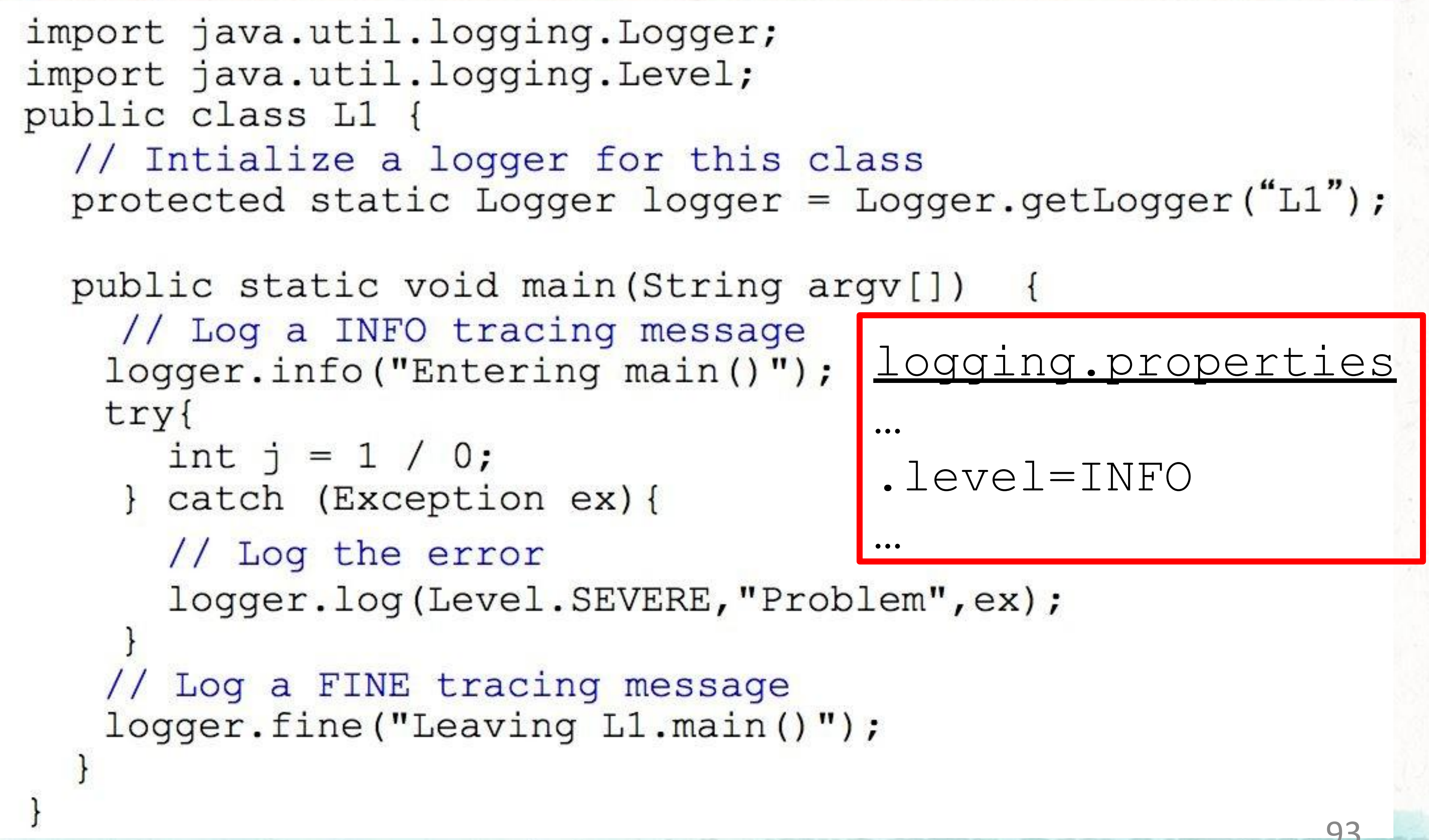
What prints?



```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
   // Intialize a logger for this class
   protected static Logger logger = Logger.getLogger(“L1”);

   public static void main(String argv[])  {
     // Log a INFO tracing message
     logger.info("Entering main()");
     try{
        int j = 1 / 0;
      } catch (Exception ex){
        // Log the error
        logger.log(Level.SEVERE,"Problem",ex);
      }
     // Log a FINE tracing message
     logger.fine("Leaving L1.main()");
   }
}
```

```
logging.properties
…
.level=SEVERE
…
```

What prints?



```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
  // Intialize a logger for this class
  protected static Logger logger = Logger.getLogger(“L1”);

  public static void main(String argv[])  {
    // Log a INFO tracing message
    logger.info("Entering main()");
    try{
       int j = 1 / 0;
     } catch (Exception ex){
       // Log the error
       logger.log(Level.SEVERE,"Problem",ex);
     }
    // Log a FINE tracing message
    logger.fine("Leaving L1.main()");
  }
}
```

logging.properties

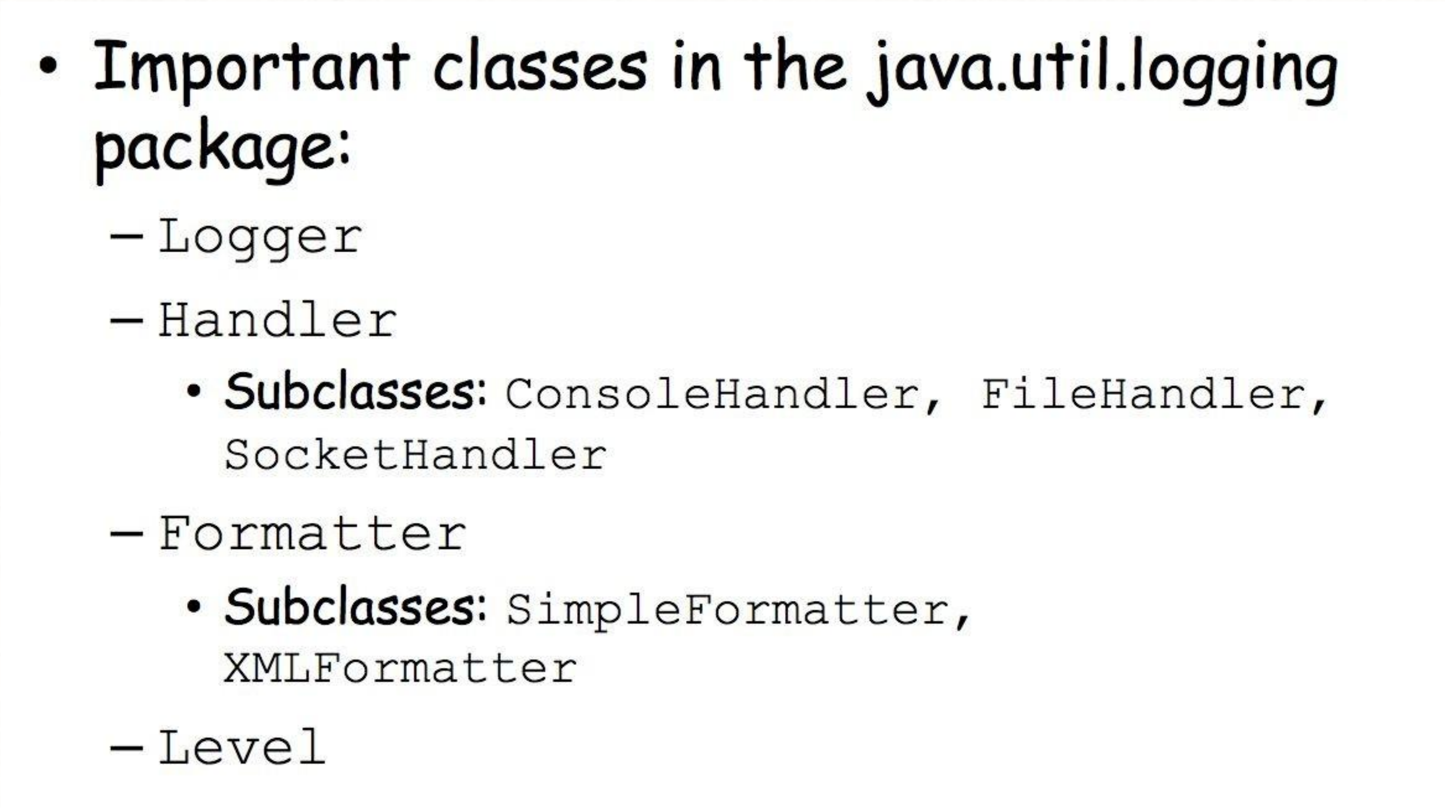
```
…
.level=INFO
…
```

Demo

Logging

- Logging means:
  - automatically recording and diagnostic output from a program
- Logging output can be useful during development and testing, but also in production code:
  - A web server could log IP address of incoming http requests
  - A mail server could log basic info about each email received etc.
- Logging using standard error output, or ordinary file output, can be done
- However it can be better to make use of a logging framework, which provides lots of useful functionality

Java Logging Framework



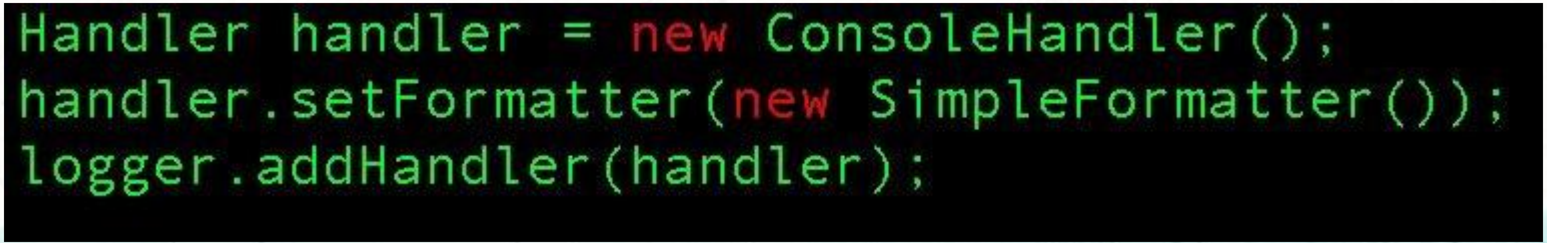
- Important classes in the java.util.logging package:
  - `Logger`
  - `Handler`
    - Subclasses: `ConsoleHandler, FileHandler, SocketHandler`
  - `Formatter`
    - Subclasses: `SimpleFormatter, XMLFormatter`
  - `Level`

Handlers and Formatters

- There are different kinds of handlers (ConsoleHandler, FileHandler, etc.) and two built in formatters (SimpleFormatter and XMLFormatter).

- A Handler is a component that takes care of the actual logging to the outside world.

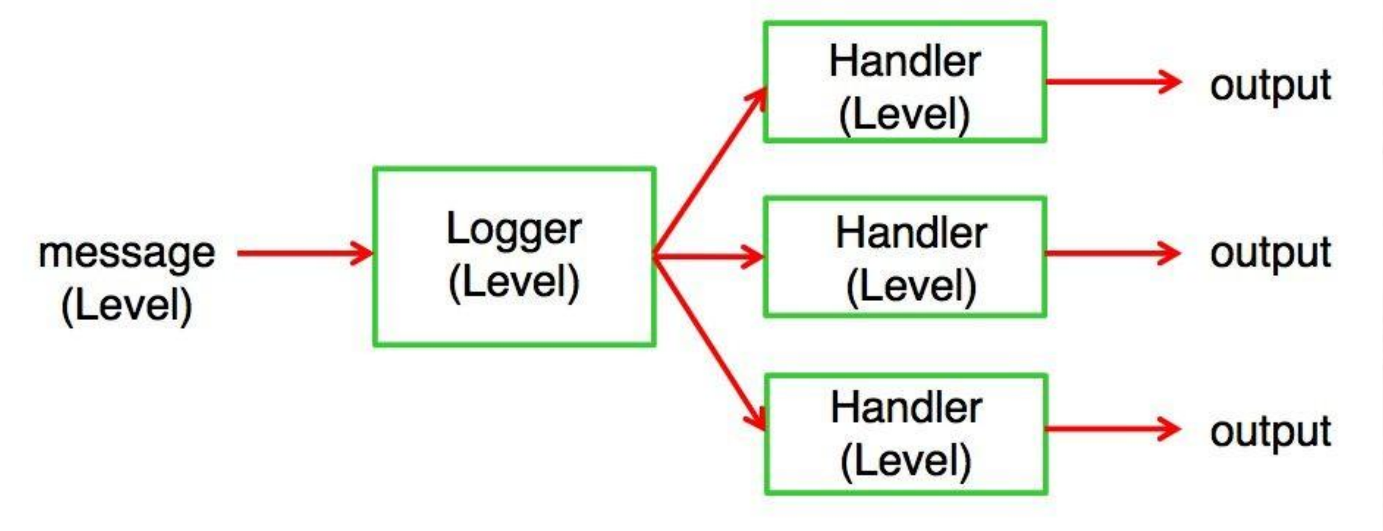
- A Handler uses a Formatter to format the output into a desired form.



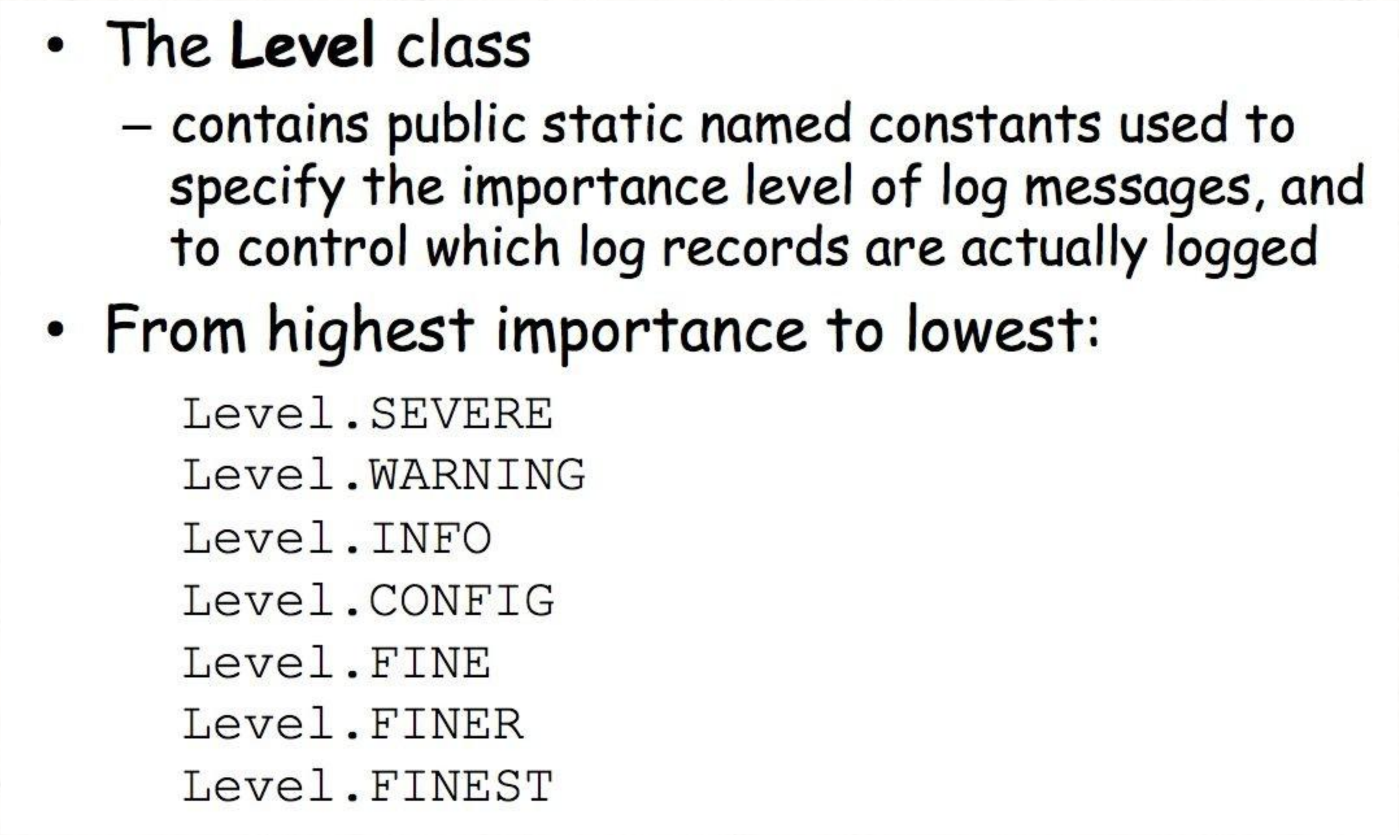
```
Handler handler = new ConsoleHandler();
handler.setFormatter(new SimpleFormatter());
logger.addHandler(handler);
```

Handlers

Each Logger can have several handlers.



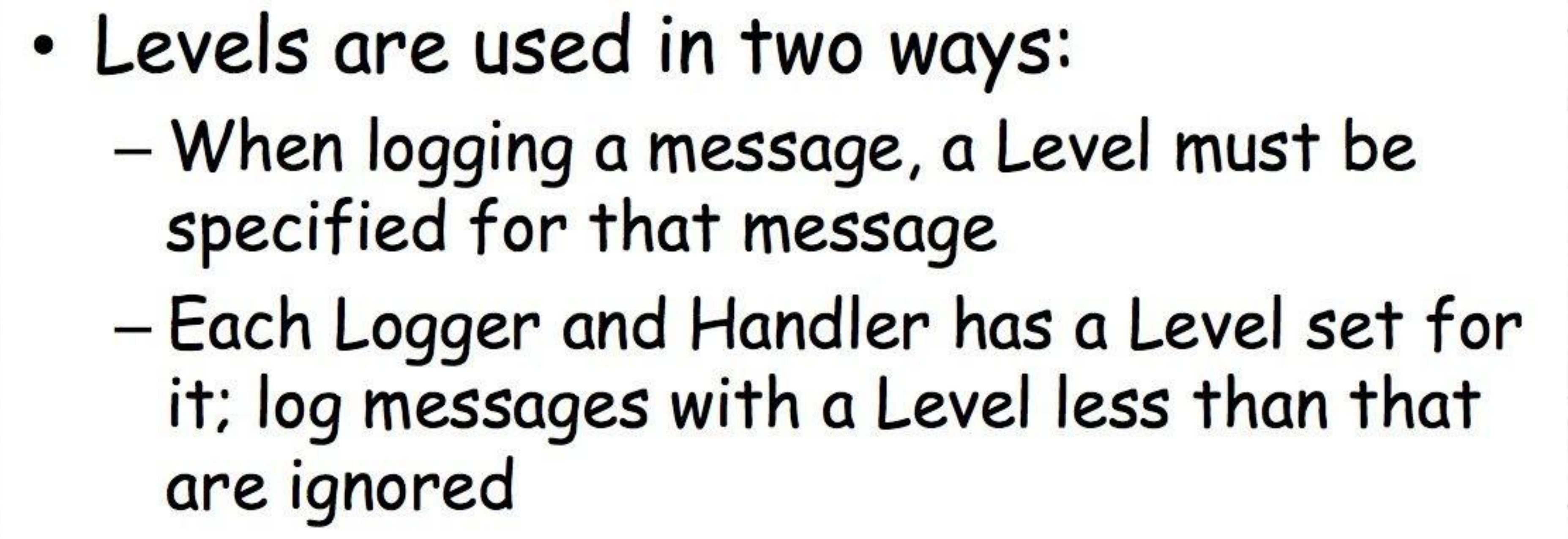
Level



- The **Level** class
  - contains public static named constants used to specify the importance level of log messages, and to control which log records are actually logged
- From highest importance to lowest:

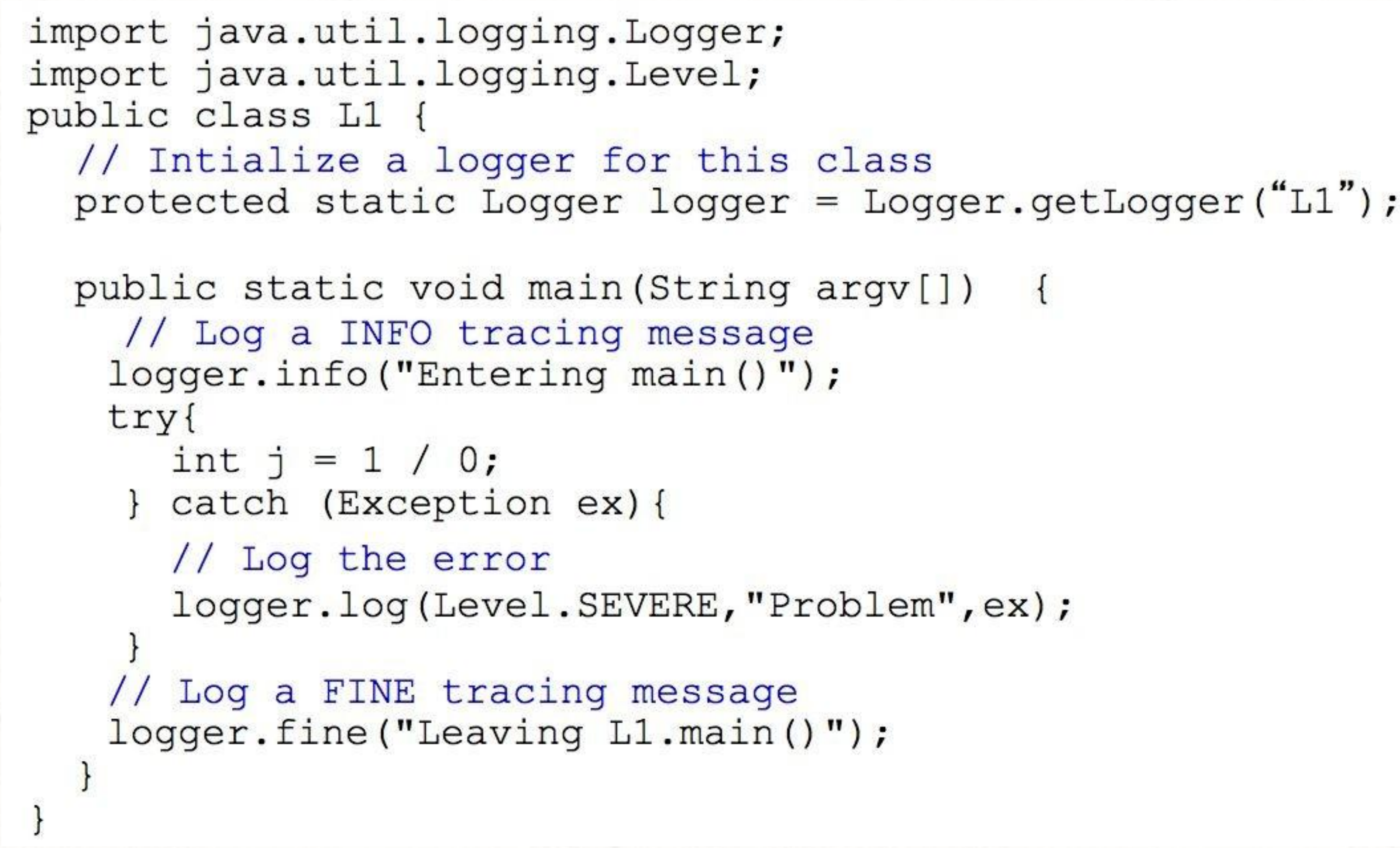
```
Level.SEVERE
Level.WARNING
Level.INFO
Level.CONFIG
Level.FINE
Level.FINER
Level.FINEST
```

Level



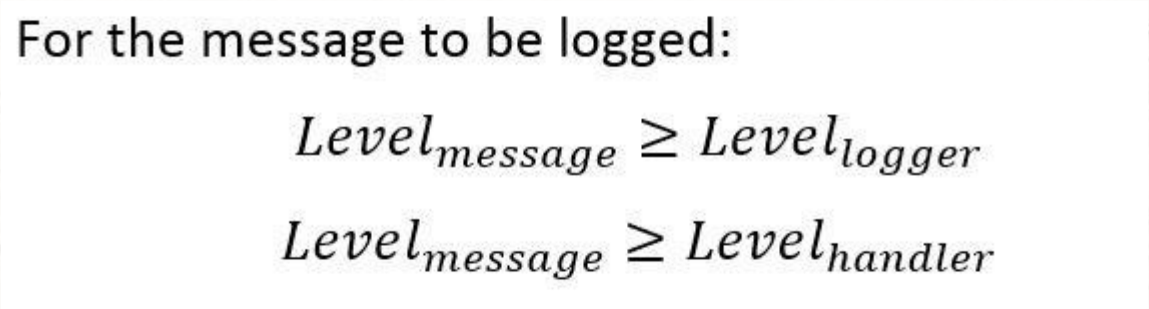
- Levels are used in two ways:
  - When logging a message, a Level must be specified for that message
  - Each Logger and Handler has a Level set for it; log messages with a Level less than that are ignored

Example Logger



```
import java.util.logging.Logger;
import java.util.logging.Level;
public class L1 {
 // Intialize a logger for this class
 protected static Logger logger = Logger.getLogger(“L1”);

 public static void main(String argv[])  {
  // Log a INFO tracing message
  logger.info("Entering main()");
  try{
   int j = 1 / 0;
  } catch (Exception ex){
   // Log the error
   logger.log(Level.SEVERE,"Problem",ex);
  }
  // Log a FINE tracing message
  logger.fine("Leaving L1.main()");
 }
}
```

When is a message logged ?



For the message to be logged:

$$Level_{message} \geq Level_{logger}$$

$$Level_{message} \geq Level_{handler}$$

Properties file

- It can be tedious to constantly change the Handlers, logging levels or formatters in the source files and then recompiling. Instead using a properties file is recommended. Look at your 15L lab for a sample properties file.

-
```
java -Djava.util.logging.config.
file=<propertiesFile> <program>
```

Logging

Is there such a thing as logging too much?
What would happen?

Profiling
(Lab 9)

What?

- Memory
- CPU Usage
- HDD Usage (Swap files etc)
- Network Usage
- Battery
- In short, want to look any resource that your application depends on

What?

Profiling is measuring the time, space, and energy used by a program.

What kinds of things can we measure?

Read More: https://en.wikipedia.org/wiki/Profiling_(computer_programming)

Things we can measure

Any resource that your application depends on:

- Memory
- CPU Usage
- HDD Usage (Swap files, etc.)
  - HDD = Hard drive disk
- Network Usage
- Battery

Why?

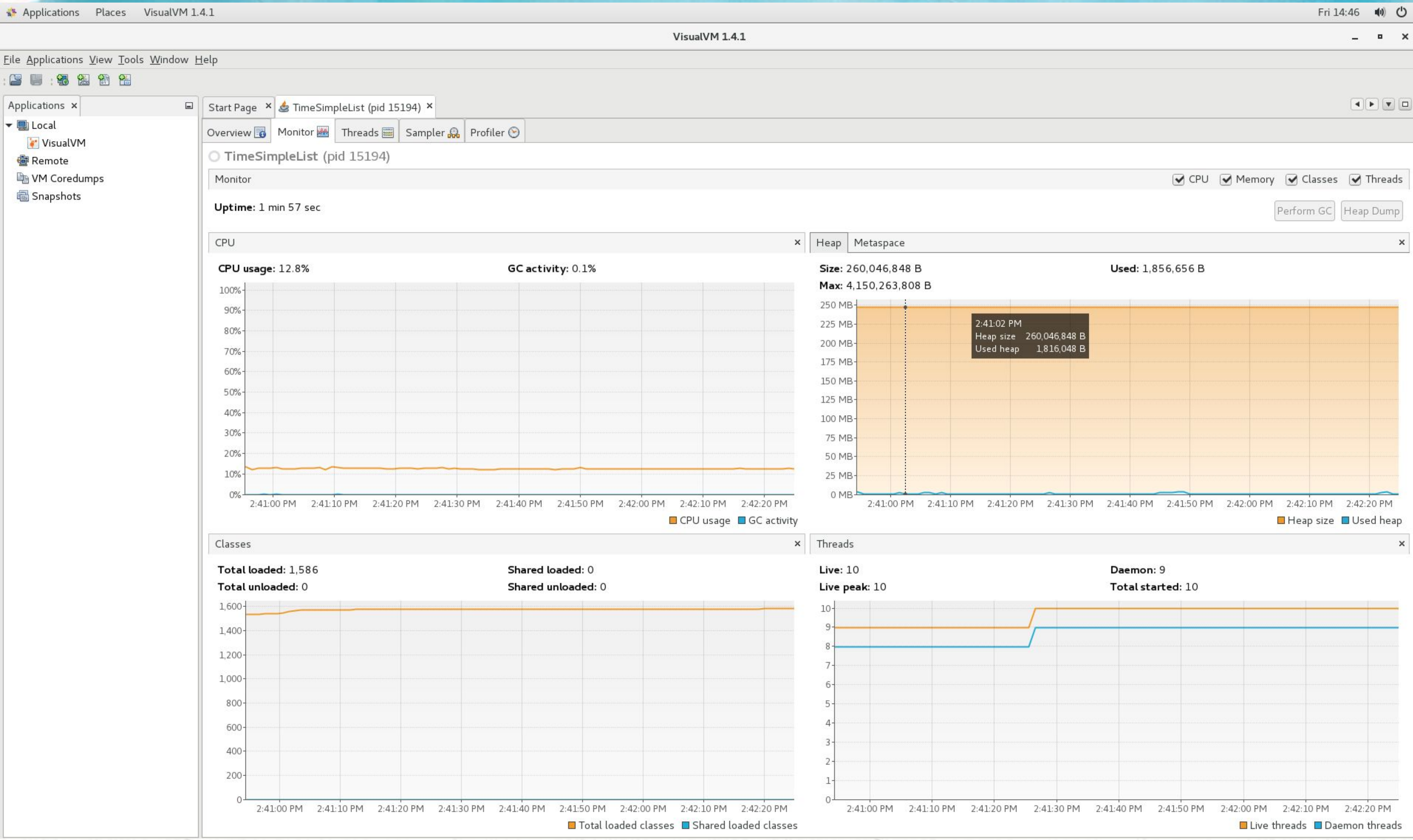
Helps us optimize our programs!

How?

a profiling tool that gives us info about Java applications

Read more: https://en.wikipedia.org/wiki/VisualVM

From Lab 9...



VisualVM 1.4.1

TimeSimpleList (pid 15194)

Uptime: 1 min 57 sec

CPU usage: 12.8%
GC activity: 0.1%

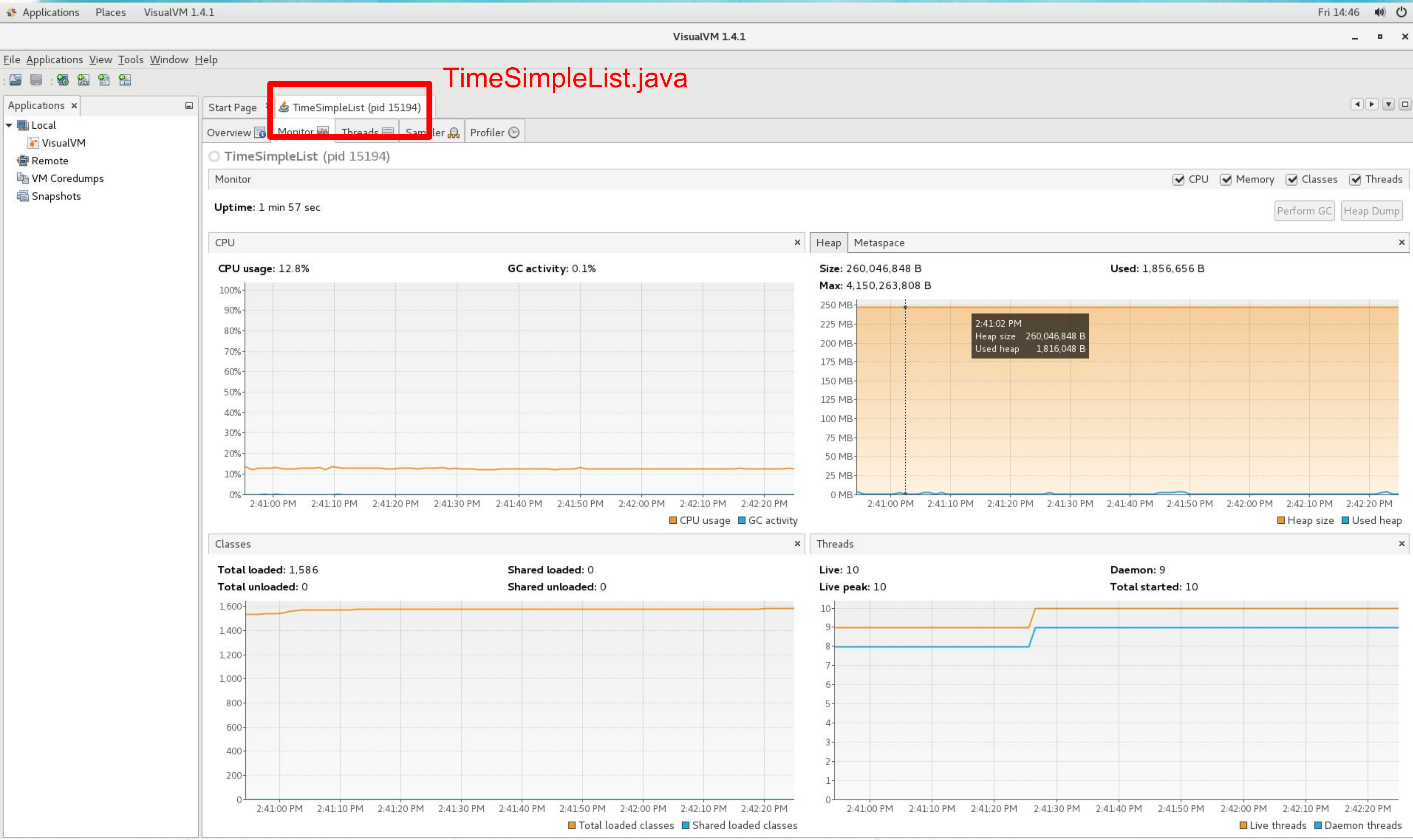
Size: 260,046,848 B
Used: 1,856,656 B
Max: 4,150,263,808 B

2:41:02 PM
Heap size 260,046,848 B
Used heap 1,816,048 B

Total loaded: 1,586
Shared loaded: 0
Total unloaded: 0
Shared unloaded: 0

Live: 10
Daemon: 9
Live peak: 10
Total started: 10

TimeSimpleList.java



TimeSimpleList (pid 15194)

Uptime: 1 min 57 sec

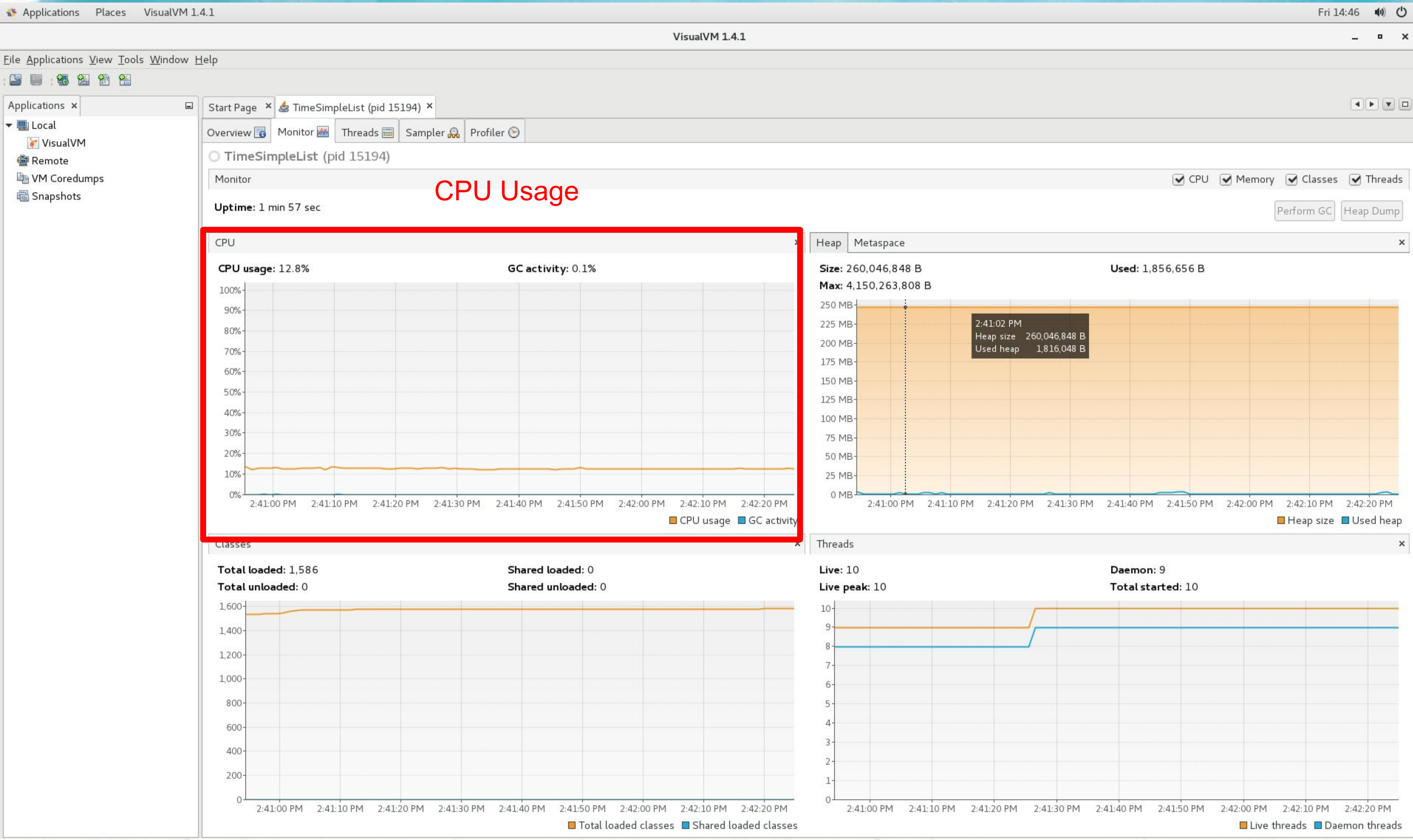
CPU usage: 12.8%
GC activity: 0.1%

Size: 260,046,848 B
Used: 1,856,656 B
Max: 4,150,263,808 B

2:41:02 PM
Heap size 260,046,848 B
Used heap 1,816,048 B

Total loaded: 1,586
Shared loaded: 0
Total unloaded: 0
Shared unloaded: 0

Live: 10
Daemon: 9
Live peak: 10
Total started: 10


Applications
Places
VisualVM 1.4.1
Fri 14:46
VisualVM 1.4.1
File Applications View Tools Window Help
Applications
Local
VisualVM
Remote
VM Coredumps
Snapshots
Start Page
TimeSimpleList (pid 15194)
Overview
Monitor
Threads
Sampler
Profiler
TimeSimpleList (pid 15194)
Monitor
☑ CPU ☑ Memory ☑ Classes ☑ Threads
Perform GC
Heap Dump
CPU
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
2:41:00 PM 2:41:10 PM 2:41:20 PM 2:41:30 PM 2:41:40 PM 2:41:50 PM 2:42:00 PM 2:42:10 PM 2:42:20 PM
🟧 CPU usage 🟦 GC activity
Heap
Metaspace
250 MB
225 MB
200 MB
175 MB
150 MB
125 MB
100 MB
75 MB
50 MB
25 MB
0 MB
2:41:00 PM 2:41:10 PM 2:41:20 PM 2:41:30 PM 2:41:40 PM 2:41:50 PM 2:42:00 PM 2:42:10 PM 2:42:20 PM
🟧 Heap size 🟦 Used heap
Classes
1,600
1,400
1,200
1,000
800
600
400
200
0
2:41:00 PM 2:41:10 PM 2:41:20 PM 2:41:30 PM 2:41:40 PM 2:41:50 PM 2:42:00 PM 2:42:10 PM 2:42:20 PM
🟧 Total loaded classes 🟦 Shared loaded classes
Threads
10
9
8
7
6
5
4
3
2
1
0
2:41:00 PM 2:41:10 PM 2:41:20 PM 2:41:30 PM 2:41:40 PM 2:41:50 PM 2:42:00 PM 2:42:10 PM 2:42:20 PM
🟧 Live threads 🟦 Daemon threads

CPU Usage

Uptime: 1 min 57 sec

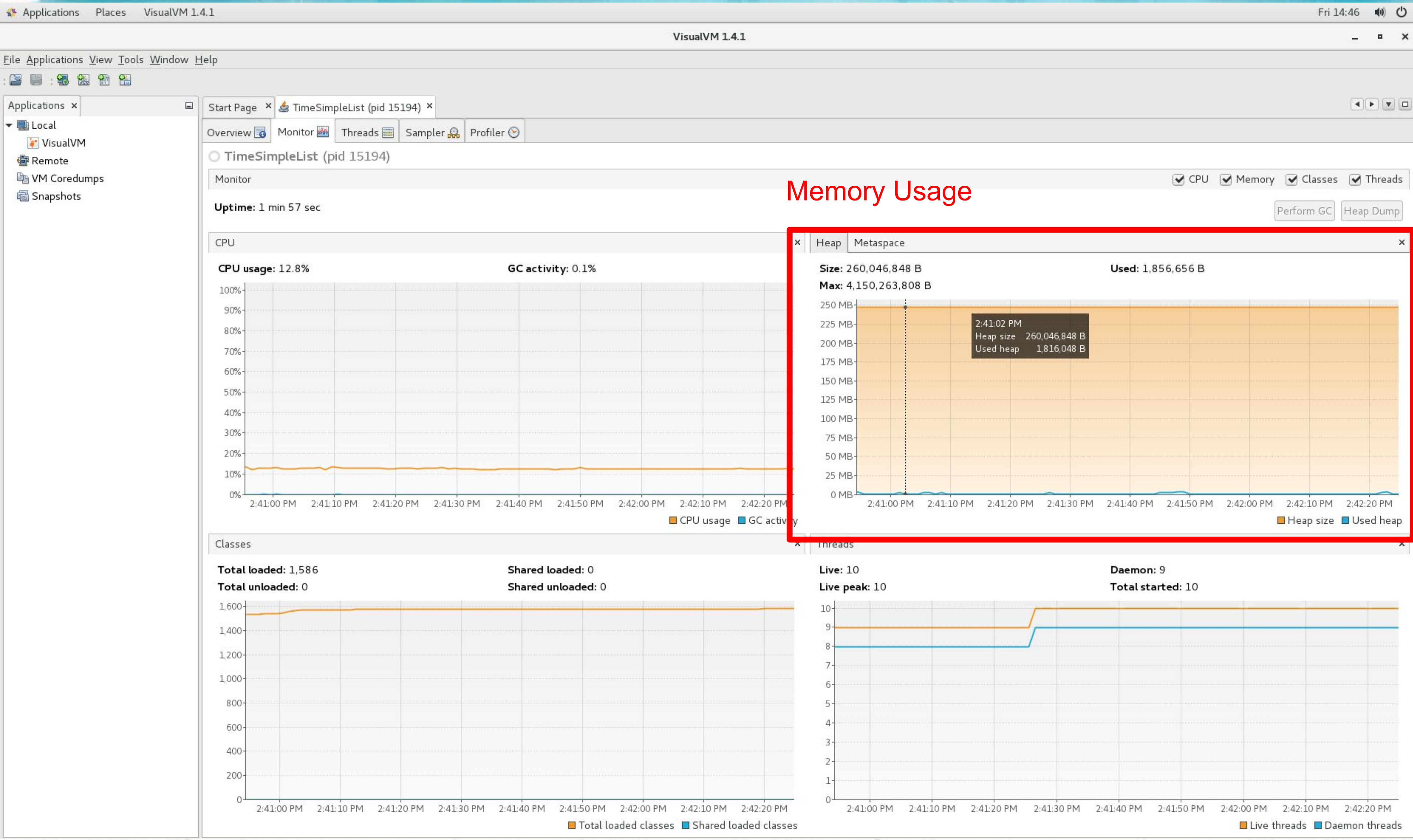
CPU usage: 12.8% **GC activity:** 0.1%

Size: 260,046,848 B **Used:** 1,856,656 B
Max: 4,150,263,808 B

2:41:02 PM
Heap size 260,046,848 B
Used heap 1,816,048 B

Total loaded: 1,586 **Shared loaded:** 0
Total unloaded: 0 **Shared unloaded:** 0

Live: 10 **Daemon:** 9
Live peak: 10 **Total started:** 10


Applications
Places
VisualVM 1.4.1
Fri 14:46
VisualVM 1.4.1
File Applications View Tools Window Help
Applications
Local
VisualVM
Remote
VM Coredumps
Snapshots
Start Page
TimeSimpleList (pid 15194)
Overview
Monitor
Threads
Sampler
Profiler
TimeSimpleList (pid 15194)
Monitor
☑ CPU ☑ Memory ☑ Classes ☑ Threads
Perform GC
Heap Dump
CPU
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
2:41:00 PM 2:41:10 PM 2:41:20 PM 2:41:30 PM 2:41:40 PM 2:41:50 PM 2:42:00 PM 2:42:10 PM 2:42:20 PM
CPU usage
GC activity
Heap
Metaspace
250 MB
225 MB
200 MB
175 MB
150 MB
125 MB
100 MB
75 MB
50 MB
25 MB
0 MB
2:41:00 PM 2:41:10 PM 2:41:20 PM 2:41:30 PM 2:41:40 PM 2:41:50 PM 2:42:00 PM 2:42:10 PM 2:42:20 PM
Heap size
Used heap
Classes
1,600
1,400
1,200
1,000
800
600
400
200
0
2:41:00 PM 2:41:10 PM 2:41:20 PM 2:41:30 PM 2:41:40 PM 2:41:50 PM 2:42:00 PM 2:42:10 PM 2:42:20 PM
Total loaded classes
Shared loaded classes
Threads
10
9
8
7
6
5
4
3
2
1
0
2:41:00 PM 2:41:10 PM 2:41:20 PM 2:41:30 PM 2:41:40 PM 2:41:50 PM 2:42:00 PM 2:42:10 PM 2:42:20 PM
Live threads
Daemon threads

Memory Usage

Uptime: 1 min 57 sec

CPU usage: 12.8% **GC activity:** 0.1%

Size: 260,046,848 B **Used:** 1,856,656 B

Max: 4,150,263,808 B

2:41:02 PM

Heap size 260,046,848 B

Used heap 1,816,048 B

Total loaded: 1,586 **Shared loaded:** 0

Total unloaded: 0 **Shared unloaded:** 0

Live: 10 **Daemon:** 9

Live peak: 10 **Total started:** 10

Practice Problems

Practice problems

What is the difference between the following. Make sure to think about what happens when a file exists/doesn't.

```
$ echo cat
$ cat echo
```

Some Lab 6,7,8,9 Questions

Sample Lab questions.

Write a bash script that prints the numbers from 1 to 10.

```
#!/bin/bash

for i in ___{1..10}______; do

          ______echo $i____   (echo “$i” also works, but must be
double quotes)
done
```

Higher standard deviation values imply more reliable measurements.

True **False**

Allowing client code to access member variables of a class directly is bad programming practice.

True False

You want to insert a large number of integers into a data structure. Which of the following data structures will perform this insert operation at minimum time cost?

Priority Queue **Linked List**

Consider a scenario where users enter their Employee ID (Integer) on a terminal. Employee IDs are entered in random order. You would like to store them in a data structure, so you can retrieve these IDs in increasing order. Which data structure gives you the opportunity to perform this input/output at minimum time cost?

Priority Queue Linked List

Which of the following options for CPU usage profiling with the hprof tool can slow the application significantly?

A. cpu=samples
B. cpu=times
C. cpu=dump
D. cpu=sites

Which Handler object (within the `java.util.logging.Handler` package) will log the message to stderr?

a. FileHandler
b. ConsoleHandler
c. MessageHandler
d. ErrorHandler

In the `Level` class (`java.util.logging.Level`), when specifying the importance level of log messages, which constant has the highest importance out of the following (i.e., Which has the highest importance among the four answers given?):

a. `Level.SEVERE`
b. `Level.FINER`
c. `Level.FINEST`
d. `Level.INFO`

True or **False** (Circle one): The following line in logging properties file will allow printing of log messages to a file with levels lower than or equal to SEVERE.

```
java.util.logging.FileHandler.level=SEVERE
```

If we have the code: `logger.config("The sentence is printed out")` , which level for the ConsoleHandler would display the message?

a. INFO
b. FINE
c. Both A and B
d. None of above

Suggestions

You should go over Lab 6, 7, 8, and 9.

Prioritize!

- Study your midterm
- Study sample practice
- Review lectures
- Review labs

Thank you! Good luck on your finals!