CSE 15L:
Software Tools and Techniques Laboratory

Winter 2021 - http://ieng6.ucsd.edu/~cs15x

Instructors: Gary Gillespie Keith Muller

Class sessions will be recorded and made available to students asynchronously.

Schedule

Today

1. Finishing Make

2. Tracing the stack

3. Introduction to Shell Scripting - Variables

Make Basics

- A Makefile contains a bunch of triples:

```
target: source_1, … source_N
← Tab → command(s)
```

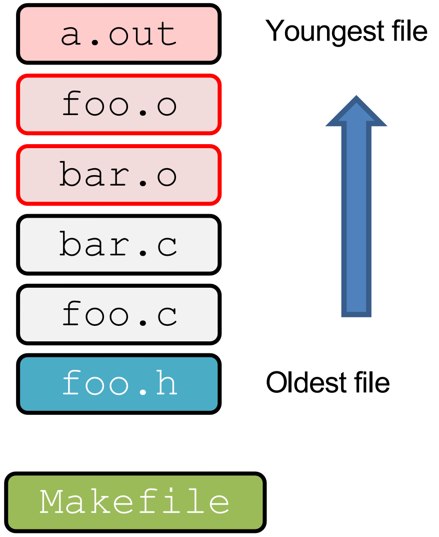
1. **Target compared to sources by file modification time**
2. **When any of source_1 to source_N is either younger (newer file modification time) or does not exist, than the target, then run the command**
3. **Running command must update/create/"touch" the target**
   - Colon after target is *required*
   - Command lines must start with a **TAB**, NOT SPACES
   - Multiple commands for same target are executed *in order*
     - Can split commands over multiple lines by ending lines with '\'

- Example:

```
bar.o: bar.c foo.h bar.h
        gcc -Wall -c bar.c
```

do not forget the tab!!!!

Make looks at mtime the Directory contents



make macros

Example (Makefile contents) :

```
CC = gcc
CFLAGS = -Wall -g
bar.o: bar.c foo.h bar.h
        $(CC) $(CFLAGS) -c bar.c
```

- $(CC) and $(CFLAGS) are example of variables
- Easy to change things (especially in multiple commands)
- Can also specify on the command line (over-ride for OSX for example): (*e.g.* `make foo.o CC=clang CFLAGS=-g`)

```
OBJS = foo.o bar.o baz.o
widget: $(OBJS)
        gcc -Wall $(OBJS)-o widget
clean:
        rm -f $(OBJS) widget
```

default goal (calling just make)

make sure the \ is at the end of the line!:
*** recipe commences before first target

```
OBJECTS = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit: $(OBJECTS)
        cc -o edit $(OBJECTS)
main.o: main.c defs.h
        cc -c main.c
kbd.o: kbd.c defs.h command.h
        cc -c kbd.c
command.o: command.c defs.h command.h
        cc -c command.c
display.o: display.c defs.h buffer.h
        cc -c display.c
insert.o: insert.c defs.h buffer.h
        cc -c insert.c
search.o: search.c defs.h buffer.h
        cc -c search.c
files.o: files.c defs.h buffer.h command.h
        cc -c files.c
utils.o: utils.c defs.h
        cc -c utils.c
.PHONY: clean
clean:
        -rm -f edit $(OBJECTS)
```

```
$ ls -last
total 56
4 drwxr-xr-x  2 kmuller kmuller 4096 Feb  1 09:24 .
4 -rw-r--r--  1 kmuller kmuller  572 Feb  1 09:24 Makefile
4 -rw-r--r--  1 kmuller kmuller   32 Feb  1 09:17 command.c
4 -rw-r--r--  1 kmuller kmuller   13 Feb  1 09:16 buffer.h
4 -rw-r--r--  1 kmuller kmuller   13 Feb  1 09:16 command.h
4 -rw-r--r--  1 kmuller kmuller   12 Feb  1 09:16 defs.h
4 -rw-r--r--  1 kmuller kmuller   31 Feb  1 09:16 utils.c
4 -rw-r--r--  1 kmuller kmuller   31 Feb  1 09:16 files.c
4 -rw-r--r--  1 kmuller kmuller   32 Feb  1 09:15 search.c
4 -rw-r--r--  1 kmuller kmuller   32 Feb  1 09:15 insert.c
4 -rw-r--r--  1 kmuller kmuller   33 Feb  1 09:15 display.c
4 -rw-r--r--  1 kmuller kmuller   29 Feb  1 09:14 kbd.c
4 -rw-r--r--  1 kmuller kmuller   29 Feb  1 09:13 main.c
4 drwxr-xr-x 28 kmuller kmuller 4096 Feb  1 08:47 ..
```

```
OBJECTS = main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
edit: $(OBJECTS)
          cc -o edit $(OBJECTS)
main.o: main.c defs.h
          cc -c main.c
kbd.o: kbd.c defs.h command.h
          cc -c kbd.c
command.o: command.c defs.h command.h
          cc -c command.c
display.o: display.c defs.h buffer.h
          cc -c display.c
insert.o: insert.c defs.h buffer.h
          cc -c insert.c
search.o: search.c defs.h buffer.h
          cc -c search.c
files.o: files.c defs.h buffer.h command.h
          cc -c files.c
utils.o: utils.c defs.h
          cc -c utils.c
.PHONY: clean
clean:
          -rm -f edit $(OBJECTS)
```

```
$ make
cc -c main.c
cc -c kbd.c
cc -c command.c
cc -c display.c
cc -c insert.c
cc -c search.c
cc -c files.c
cc -c utils.c
cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
$
```

```
$ touch buffer.h
$ ls -last
total 100
 4 -rw-r--r--  1 kmuller kmuller   13 Feb  1 09:29 buffer.h
12 -rwxr-xr-x  1 kmuller kmuller 8352 Feb  1 09:27 edit
 4 drwxr-xr-x  2 kmuller kmuller 4096 Feb  1 09:27 .
 4 -rw-r--r--  1 kmuller kmuller  828 Feb  1 09:27 utils.o
 4 -rw-r--r--  1 kmuller kmuller  828 Feb  1 09:27 files.o
 4 -rw-r--r--  1 kmuller kmuller  832 Feb  1 09:27 search.o
 4 -rw-r--r--  1 kmuller kmuller  832 Feb  1 09:27 insert.o
 4 -rw-r--r--  1 kmuller kmuller  832 Feb  1 09:27 display.o
 4 -rw-r--r--  1 kmuller kmuller  832 Feb  1 09:27 command.o
 4 -rw-r--r--  1 kmuller kmuller  824 Feb  1 09:27 kbd.o
 4 -rw-r--r--  1 kmuller kmuller  828 Feb  1 09:27 main.o
 4 -rw-r--r--  1 kmuller kmuller  572 Feb  1 09:24 Makefile
 4 -rw-r--r--  1 kmuller kmuller   32 Feb  1 09:17 command.c
 4 -rw-r--r--  1 kmuller kmuller   13 Feb  1 09:16 command.h
 4 -rw-r--r--  1 kmuller kmuller   12 Feb  1 09:16 defs.h
 4 -rw-r--r--  1 kmuller kmuller   31 Feb  1 09:16 utils.c
 4 -rw-r--r--  1 kmuller kmuller   31 Feb  1 09:16 files.c
 4 -rw-r--r--  1 kmuller kmuller   32 Feb  1 09:15 search.c
 4 -rw-r--r--  1 kmuller kmuller   32 Feb  1 09:15 insert.c
 4 -rw-r--r--  1 kmuller kmuller   33 Feb  1 09:15 display.c
 4 -rw-r--r--  1 kmuller kmuller   29 Feb  1 09:14 kbd.c
 4 -rw-r--r--  1 kmuller kmuller   29 Feb  1 09:13 main.c
 4 drwxr-xr-x 28 kmuller kmuller 4096 Feb  1 08:47 ..
kmuller@keithm-pi4:~/make_example $ make
cc -c display.c
cc -c insert.c
cc -c search.c
cc -c files.c
cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

```
OBJECTS = main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
edit: $(OBJECTS)
           cc -o edit $(OBJECTS)
main.o: main.c defs.h
           cc -c main.c
kbd.o: kbd.c defs.h command.h
           cc -c kbd.c
command.o: command.c defs.h command.h
           cc -c command.c
display.o: display.c defs.h buffer.h
           cc -c display.c
insert.o: insert.c defs.h buffer.h
           cc -c insert.c
search.o: search.c defs.h buffer.h
           cc -c search.c
files.o: files.c defs.h buffer.h command.h
           cc -c files.c
utils.o: utils.c defs.h
           cc -c utils.c
.PHONY: clean
clean:
           -rm -f edit $(OBJECTS)
```

Automatic Variables

When writing Makefiles, you may want to reference targets, dependency names, etc.

Use **make**’s automatic variables!

$@	filename of the target
$%	target member name
$<	first dependency
$^	all dependencies
$?	all dependencies newer than target
$*	basename of the current target

Longer list: https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

Specify/Over-riding Suffix Directives

Suffix rules are a way of defining implicit rules for make

DSTS:

```
DSTS:
        rule
```

'TS' is the suffix of the target file

'DS' is the suffix of the dependency file

'rule' is the rule for building a target

```
.SUFFIXES: .c .o
.c.o:
        cc -c $*.c
```

```
OBJECTS = main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
edit: $(OBJECTS)
          cc -o edit $(OBJECTS)
main.o: main.c defs.h
          cc -c main.c
kbd.o: kbd.c defs.h command.h
          cc -c kbd.c
command.o: command.c defs.h command.h
          cc -c command.c
display.o: display.c defs.h buffer.h
          cc -c display.c
insert.o: insert.c defs.h buffer.h
          cc -c insert.c
search.o: search.c defs.h buffer.h
          cc -c search.c
files.o: files.c defs.h buffer.h command.h
          cc -c files.c
utils.o: utils.c defs.h
          cc -c utils.c
.PHONY: clean
clean:
          -rm –f edit $(OBJECTS)
```

```
OBJECTS = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
.SUFFIXES: .c .o
.c.o:
    cc -c $*.c
edit: $(OBJECTS)
    cc -o edit $(OBJECTS)
$(OBJECTS): defs.h
kbd.o: command.h
command.o: command.h
display.o: buffer.h
insert.o: buffer.h
search.o: buffer.h
files.o: buffer.h command.h
.PHONY: clean
clean:
    $(RM)edit $(OBJECTS)
```

```
$ touch command.h
$ ls -ls
total 96
 4 -rw-r--r-- 1 kmuller kmuller   13 Feb  1 09:29 buffer.h
 4 -rw-r--r-- 1 kmuller kmuller   32 Feb  1 11:29 command.c
 4 -rw-r--r-- 1 kmuller kmuller   13 Feb  1 11:32 command.h
 4 -rw-r--r-- 1 kmuller kmuller  832 Feb  1 11:32 command.o
 4 -rw-r--r-- 1 kmuller kmuller   12 Feb  1 09:16 defs.h
 4 -rw-r--r-- 1 kmuller kmuller   33 Feb  1 09:15 display.c
 4 -rw-r--r-- 1 kmuller kmuller  832 Feb  1 11:32 display.o
12 -rwxr-xr-x 1 kmuller kmuller 8352 Feb  1 11:32 edit
 4 -rw-r--r-- 1 kmuller kmuller   31 Feb  1 09:16 files.c
 4 -rw-r--r-- 1 kmuller kmuller  828 Feb  1 11:32 files.o
 4 -rw-r--r-- 1 kmuller kmuller   32 Feb  1 09:15 insert.c
 4 -rw-r--r-- 1 kmuller kmuller  832 Feb  1 11:32 insert.o
 4 -rw-r--r-- 1 kmuller kmuller   29 Feb  1 09:14 kbd.c
 4 -rw-r--r-- 1 kmuller kmuller  824 Feb  1 11:32 kbd.o
 4 -rw-r--r-- 1 kmuller kmuller   29 Feb  1 09:13 main.c
 4 -rw-r--r-- 1 kmuller kmuller  828 Feb  1 11:32 main.o
 4 -rw-r--r-- 1 kmuller kmuller  353 Feb  1 11:32 Makefile
 4 -rw-r--r-- 1 kmuller kmuller  572 Feb  1 09:24 OMakefile
 4 -rw-r--r-- 1 kmuller kmuller   32 Feb  1 09:15 search.c
 4 -rw-r--r-- 1 kmuller kmuller  832 Feb  1 11:32 search.o
 4 -rw-r--r-- 1 kmuller kmuller   31 Feb  1 09:16 utils.c
 4 -rw-r--r-- 1 kmuller kmuller  828 Feb  1 11:32 utils.o
$ make
cc -c kbd.c
cc -c command.c
cc -c files.c
cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
$ touch defs.h
$ make
cc -c main.c
cc -c kbd.c
cc -c command.c
cc -c display.c
cc -c insert.c
cc -c search.c
cc -c files.c
cc -c utils.c
cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

```
OBJECTS = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
.SUFFIXES: .c .o
.c.o:
    cc -c $*.c
edit: $(OBJECTS)
    cc -o edit $(OBJECTS)
$(OBJECTS): defs.h
kbd.o: command.h
command.o: command.h
display.o: buffer.h
insert.o: buffer.h
search.o: buffer.h
files.o: buffer.h command.h
.PHONY: clean
clean:
    $(RM)edit $(OBJECTS)
```

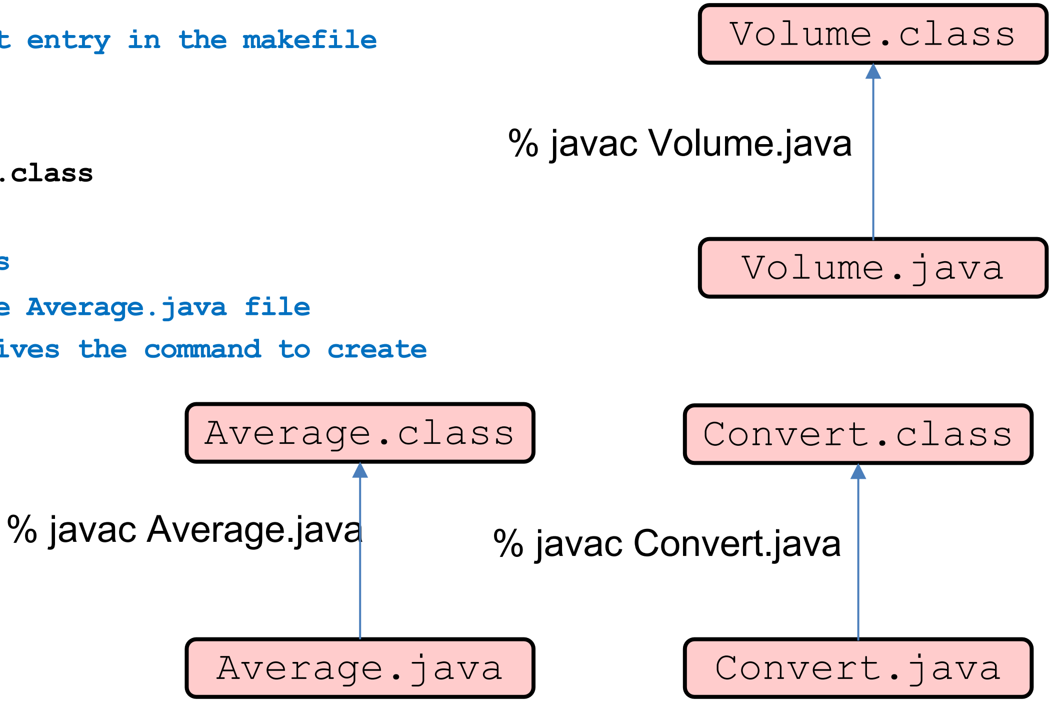
Make in Java Development

- 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

```
# A simple makefile for compiling three java classes
# define a makefile variable for the java compiler
#
JCC = javac
# define a makefile variable for compilation flags
# the -g flag compiles with debugging information
#
JFLAGS = -g
# typing 'make' will invoke the first target entry in the makefile
# (the target default in this case)
#
default: Average.class Convert.class Volume.class
#
# this target entry builds the Average class
# the Average.class file is dependent on the Average.java file
# and the rule associated with this entry gives the command to create
it
#
Average.class: Average.java
           $(JCC) $(JFLAGS) Average.java
Convert.class: Convert.java
          $(JCC) $(JFLAGS) Convert.java
Volume.class: Volume.java
          $(JCC) $(JFLAGS) Volume.java
# To start over from scratch, type 'make clean'.
# Removes all .class files, so that the next make rebuilds them
#
.PHONY: clean
clean:
          $(RM) *.class
```



```
% javac Volume.java
% javac Average.java
% javac Convert.java
```

Suffix Directive For Java

- For every file X.java in your Java project, you could write a rule

```
X.class: X.java
        javac X.java
```

- But if you have a lot of such files, it would be tedious to write a rule for each of them
- By using a **suffix directive**, you can write just one rule that handles all the files at once

Java and Make: Suffix Directives

- In the Makefile, write the suffix directive:

```
    .SUFFIXES: .java .class
```

- And then write the suffix rule:

```
    .java.class:
        javac $<
```

- The **$<** symbol means: the dependency, whatever it is

  (Like any rule, the action line must start with a tab)

  You could also use

```
            javac   $*.java
```

- Now any .class target file will be made from the corresponding .java file

Makefile macros for Java

- Makefiles can contain macros, which act like variables.
- For example, if you have a lot of files in your java project, define a macro like:

```
CLASSES = A.class B.class X.class Y.class
```

- And then write the rule:

```
all: $(CLASSES)
```

- Now with the suffix rule shown before, running **`make all`** will compile all the .java files into .class files

```
# define compiler and compiler flag variables
JFLAGS = -g
JC = javac
# Clear any default targets for building .class files from .java files; we will provide our own target entry to do this in
# this makefile. make has a set of default targets for different suffixes (like .c.o)
# Currently, clearing the default for .java.class is not necessary since make does not have a definition for this
# target, so it doesn't hurt to make sure that we clear any default definitions for these
.SUFFIXES: .java .class
# Here is our target entry for creating .class files from .java files
# This is a target entry that uses the suffix rule syntax:
# DSTS:
#         rule
# 'TS' is the suffix of the target file, 'DS' is the suffix of the dependency file,
# and 'rule' is the rule for building a target.
# '$*' is a built-in macro that gets the basename of the current target
.java.class:
          $(JC) $(JFLAGS) $*.java
# CLASSES is a macro consisting of 4 words (one for each java source file)
CLASSES = Foo.java Blah.java \
          Library.java Main.java
# the default make target entry
default: classes
# This target entry uses Suffix Replacement within a macro:
# $(name:string1=string2)
# In the words in the macro named 'name' replace 'string1' with 'string2’
# Below we are replacing the suffix .java of all words in the macro CLASSES
# with the .class suffix
classes: $(CLASSES:.java=.class)
# RM is a predefined macro in make (RM = rm -f)
clean:
          $(RM) *.class
```

Interpreting the Stack Trace

Errors and Exceptions in Java

- Exceptions are events that occur during execution that disrupt the normal flow of a program
  - **`NullPointerException`**
  - **`ArrayIndexOutOfBoundsException`**
  - **`FileNotFoundException`**
  - **`InterruptedException`**
- **`try-catch`** blocks used to handle errors, commonly placed around code where exceptions expected to occur (for example, file IO, asynchronous waking of thread)

Errors and Exceptions in Java

- Errors are more serious problems that a reasonable application should not try to catch
  - **`StackOverflowError`**
  - **`OutOfMemoryError`**

- Errors and unhandled exceptions will cause your program to terminate!

Stack Trace

By default, errors and unhandled exceptions will result in printing of a stack trace:

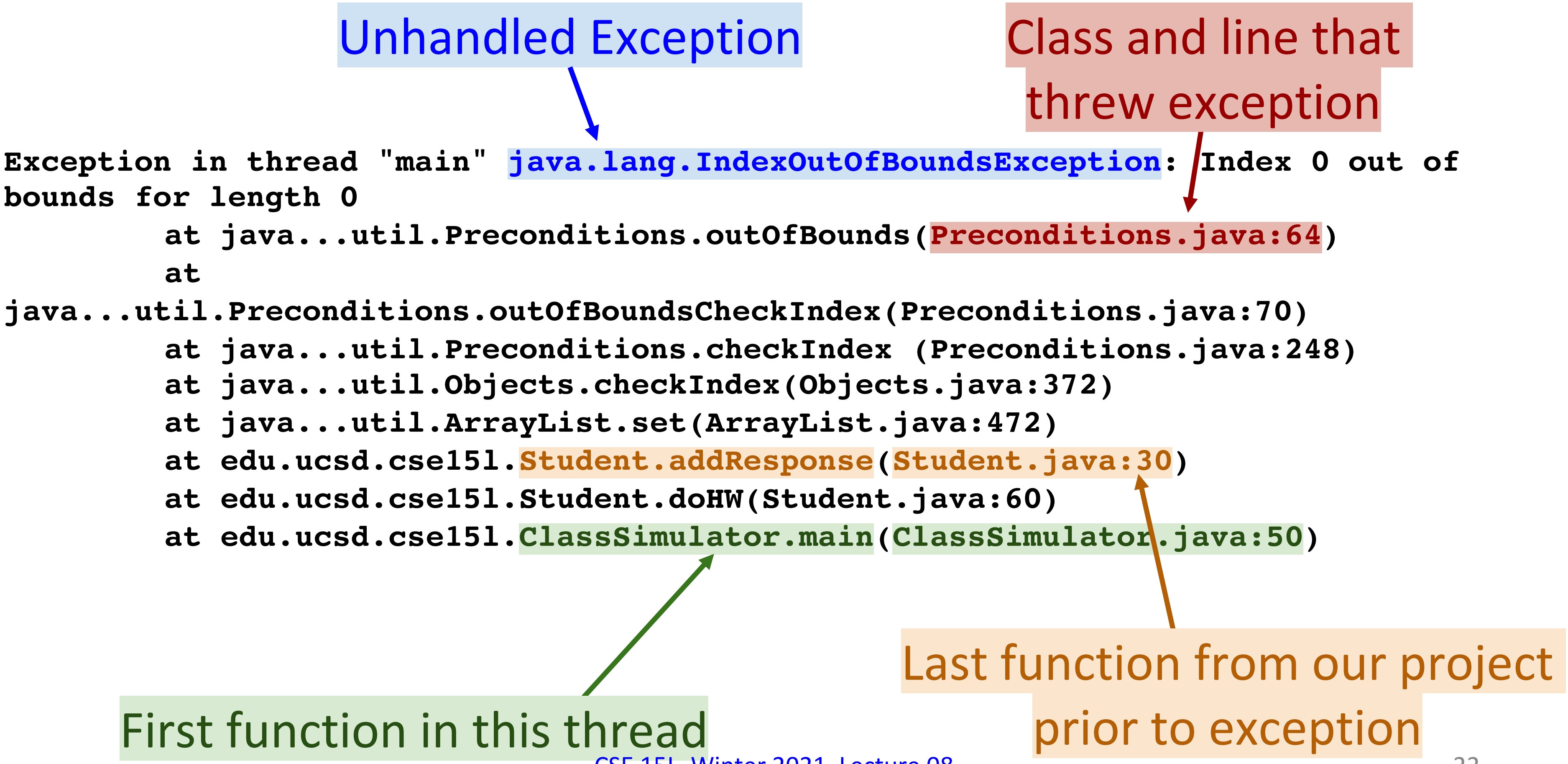
(ClassSimulator.java example available on Piazza > Resources)

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 0 out of
bounds for length 0
        at java...util.Preconditions.outOfBounds(Preconditions.java:64)
        at
java...util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
        at java...util.Preconditions.checkIndex (Preconditions.java:248)
        at java...util.Objects.checkIndex(Objects.java:372)
        at java...util.ArrayList.set(ArrayList.java:472)
        at edu.ucsd.cse15l.Student.addResponse(Student.java:30)
        at edu.ucsd.cse15l.Student.doHW(Student.java:60)
        at edu.ucsd.cse15l.ClassSimulator.main(ClassSimulator.java:50)
```

Note: `java.base/jdk.internal.util` was shortened to `java...util` for display purposes, but the stack trace would list the whole path in a true stack trace!

Stack Trace

Stack trace is full of useful information for debugging!



Unhandled Exception

Class and line that threw exception

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 0 out of
bounds for length 0
        at java...util.Preconditions.outOfBounds(Preconditions.java:64)
        at
java...util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
        at java...util.Preconditions.checkIndex (Preconditions.java:248)
        at java...util.Objects.checkIndex(Objects.java:372)
        at java...util.ArrayList.set(ArrayList.java:472)
        at edu.ucsd.cse15l.Student.addResponse(Student.java:30)
        at edu.ucsd.cse15l.Student.doHW(Student.java:60)
        at edu.ucsd.cse15l.ClassSimulator.main(ClassSimulator.java:50)
```

Last function from our project prior to exception

First function in this thread

How the Stack works

- Each thread creates its own *stack*, which stores local variables and information for function returns
- In Java Virtual Machine, information stored in stack *frames*

```
public class ClassSimulator {
   public static void main(String[] args) {
       Student student = new Student();
       student.doHW(1);
       ...
   }
}

class Student {
   ...
   void doHW(int hw_num) {
       char answer = 'c';
       addResponse(hw_num, answer);
       return;
   }
   ...
}
```

The Stack

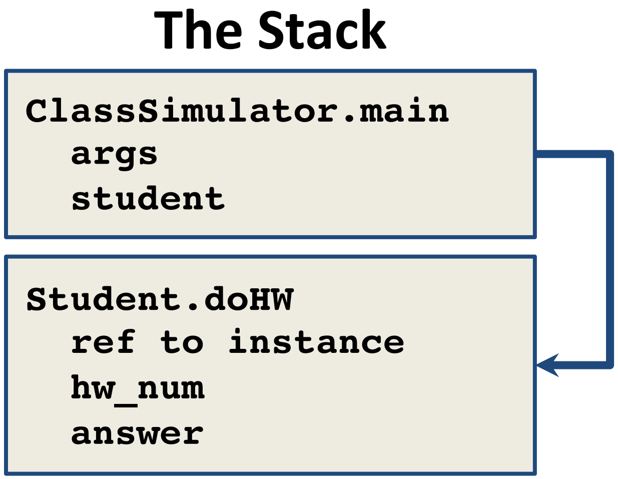
```
ClassSimulator.main
  args
  student
```

How the Stack works

- Each method/function call pushes a new frame onto the stack (adds frame to the end)

```
public class ClassSimulator {
   public static void main(String[] args) {
       Student student = new Student();
       student.doHW(1);
       ...
   }
}

class Student {
   ...
   void doHW(int hw_num) {
       char answer = 'c';
       addResponse(hw_num, answer);
       return;
   }
   ...
}
```

The Stack



```
ClassSimulator.main
  args
  student
```

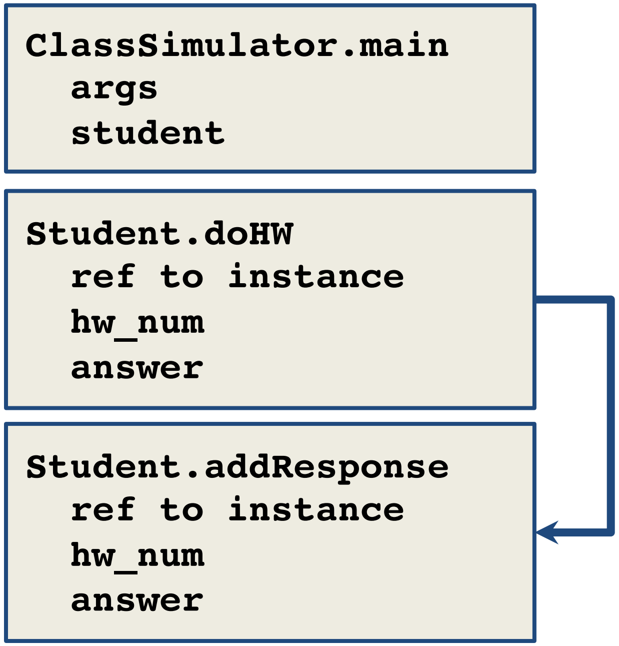
```
Student.doHW
  ref to instance
  hw_num
  answer
```

How the Stack works

- Stack frames continue to add for successive function calls
- Stack has limited size (overrunning causes a **stack overflow**)

```
public class ClassSimulator {
   public static void main(String[] args) {
       Student student = new Student();
       student.doHW(1);
       ...
   }
}

class Student {
   ...
   void doHW(int hw_num) {
       char answer = 'c';
       addResponse(hw_num, answer);
       return;
   }
   ...
}
```

The Stack



```
ClassSimulator.main
  args
  student
```

```
Student.doHW
  ref to instance
  hw_num
  answer
```

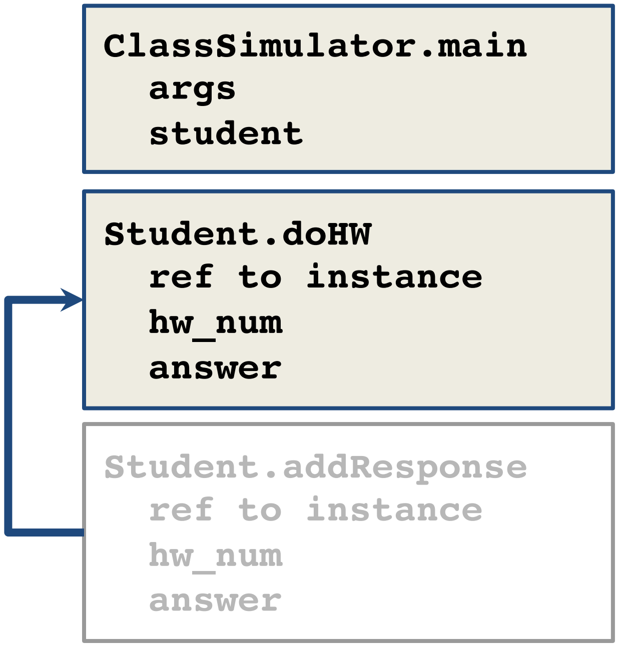
```
Student.addResponse
  ref to instance
  hw_num
  answer
```

How the Stack works

- On return, the program returns to the previous frame
- Program can reuse memory of previous frame for new function calls

```
public class ClassSimulator {
   public static void main(String[] args) {
       Student student = new Student();
       student.doHW(1);
       ...
   }
}

class Student {
   ...
   void doHW(int hw_num) {
       char answer = 'c';
       addResponse(hw_num, answer);
       return;
   }
   ...
}
```

The Stack



```
ClassSimulator.main
  args
  student
```

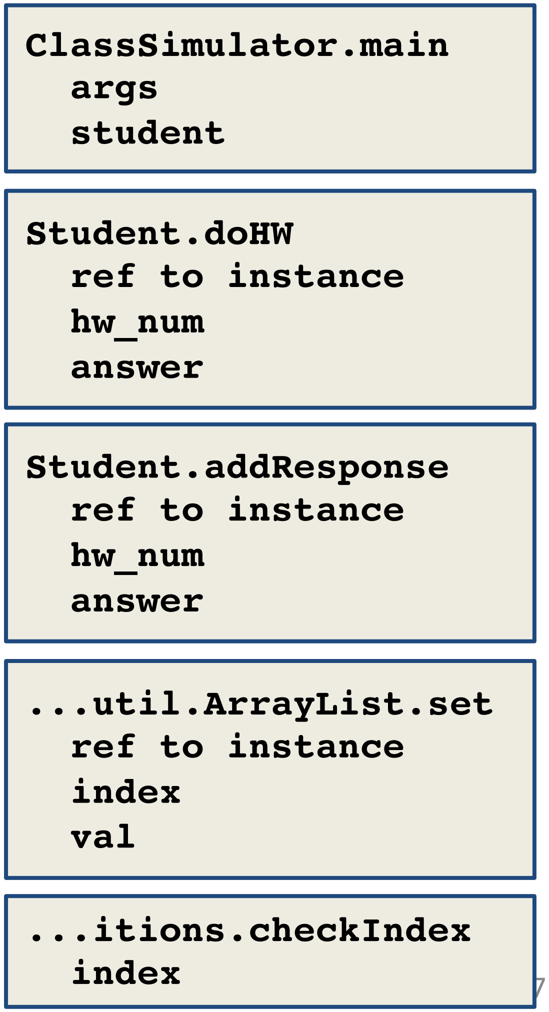
```
Student.doHW
  ref to instance
  hw_num
  answer
```

```
Student.addResponse
  ref to instance
  hw_num
  answer
```

Stack Trace

- In Java, Errors and Unhandled Exceptions result in print of the stack trace
- Stack trace contains class, function, and line information from function
  - Starts at last function called prior to error
  - Proceeds to previous frames until original frame of thread
  - Different stack for each thread!

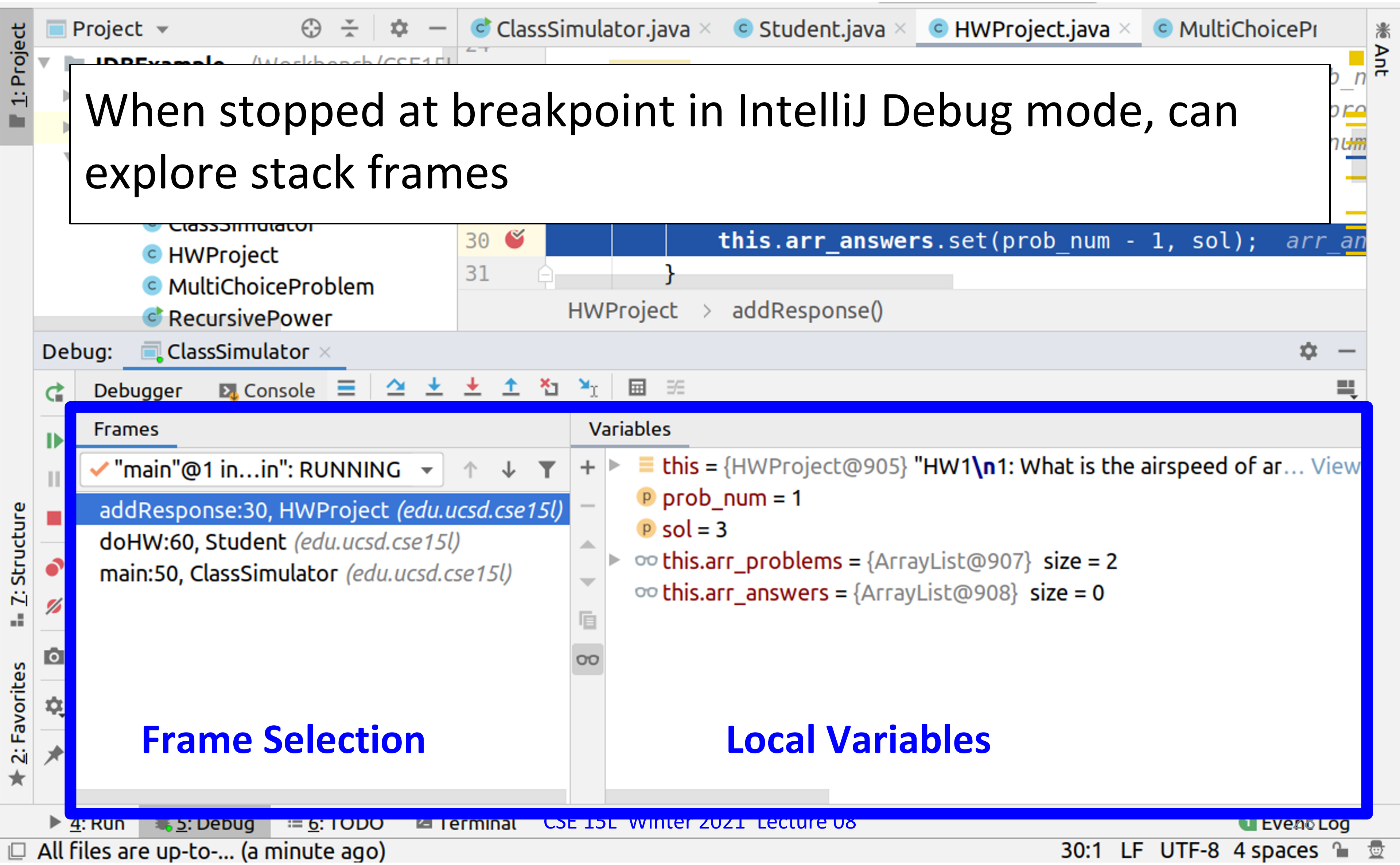
The Stack



```
ClassSimulator.main
  args
  student

Student.doHW
  ref to instance
  hw_num
  answer

Student.addResponse
  ref to instance
  hw_num
  answer

...util.ArrayList.set
  ref to instance
  index
  val

...itions.checkIndex
  index
```

Stack Frames in IntelliJ

When stopped at breakpoint in IntelliJ Debug mode, can explore stack frames



```
30        this.arr_answers.set(prob_num - 1, sol);   arr_an
31    }
```

HWProject › addResponse()

Frames:

✓ "main"@1 in...in": RUNNING

- addResponse:30, HWProject *(edu.ucsd.cse15l)*
- doHW:60, Student *(edu.ucsd.cse15l)*
- main:50, ClassSimulator *(edu.ucsd.cse15l)*

Variables:

- this = {HWProject@905} "HW1\n1: What is the airspeed of ar... View
- prob_num = 1
- sol = 3
- this.arr_problems = {ArrayList@907} size = 2
- this.arr_answers = {ArrayList@908} size = 0

All files are up-to-... (a minute ago)   30:1 LF UTF-8 4 spaces

Frame Selection

Local Variables

When debugging your program, you get the following stack trace print on your console:

```
java.lang.NullPointerException
    at java.util.Objects.requireNonNull(Objects.java:203)
    at java.util.AbstractSet.removeAll(AbstractSet.java:169)
    at edu.ucsd.cse15l.MultiChoiceProblem.<init>(MultiChoiceProblem.java:18)
    at edu.ucsd.cse15l.HWProject.addQuestion(HWProject.java:20)
    at edu.ucsd.cse15l.ClassSimulatorTest.testClass(ClassSimulatorTest.java:38)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
    at java.lang.reflect.Method.invoke(Unknown Source)
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall
    (FrameworkMethod.java:47)
    ...
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.run
    (RemoteTestRunner.java:390)
    at org.eclipse.jdt.internal.junit.runner.RemoteTestRunner.main
    (RemoteTestRunner.java:197)
```

What is the line in what file that started executing from our program before the NullPointerException?

a.) **`Objects.java:203`**

b.) **`MultiChoiceProblem.java:18`**

c.) **`ClassSimulatorTest.java:38`**

d.) **`RemoteTestRunner.java:197`**

Intro to Java Debugger - jdb

- **`jdb`** is a command-line debugger with comparable functionality to **`gdb`**

- Steps to debug **`mypackage.MyProg.main`**:
  1. Compile program using **`javac`** with **`–g`** flag (allows jdb to display local variables on the stack)
  2. Open jdb using **`jdb mypackage.MyProg`** (can include arguments if program requires them)
  3. Command prompt will change to **>**
  4. Program will be stopped before entering **`main`**

Intro to Java Debugger - jdb

!!! CAUTION !!!

Running jdb with **`jdb mypackage.MyProg`** doesn't allow user input from keyboard in the terminal

Steps to debug **`mypackage.MyProg`** with Keyboard Input

1. Configure your program to run in JVM on one terminal and open a port for the debugger

```
java -Xdebug \
-Xrunjdwp:transport=dt_socket,address=<addr>,server=y mypackage.MyProg
```

2. Then, connect the debugger from a different terminal

```
jdb -attach $address
```

3. Viola! When you run **`jdb`** from cs15l account, script is set up to do this for you. Copy the string from one terminal into another and you are good to go!

jdb Script on ieng6

```
$ alias jdb
alias jdb='/home/linux/ieng6/cs15lwi21/public/JDB'

$ cat /home/linux/ieng6/cs15lwi21/public/JDB
#!/usr/bin/bash

target=$1
set `date`
address=80${4#??:??:}
echo IN ANOTHER WINDOW in your current directory '('$PWD')', \
 COPY AND PASTE:
echo "      \jdb -attach $address"
echo "Starting the Java debugger on $target..."
java -Xdebug \
 -Xrunjdwp:transport=dt_socket,address=$address,server=y $target
```

Break Commands: jdb vs gdb

Like **gdb**, set breakpoints when program is paused.

gdb	**jdb**	
`b(reak) <func>`	`stop in <class-name>.<func>`	Place breakpoint at function named **func**
`b(reak) <#>`	`stop at <class-name>:<#>`	Place breakpoint at line **<#>**
`b(reak)`	(none)	Place breakpoint at current line
`delete <bp#>`	`clear <class-name>.<func>`	Remove the specified breakpoint
	`clear <class-name>:<#>`	Remove the specified breakpoint
`l(ist)`	`list`	List lines of source code

Unlike **gdb**, full commands need to be typed into **jdb** !

Move Commands: jdb vs gdb

Navigate using similar movement to other debuggers.

gdb	**jdb**	
r(un)	**run**	Run the program
c(ontinue)	**cont**	Continue until the next breakpoint or termination
s(tep)	**step**	Execute current line, stepping into functions
n(ext)	**next**	Execute current line, stepping over functions
f(inish)	**step up**	Step out of function
q(uit)	**exit** or **quit**	Quits debugger

Looking at Values: jdb vs gdb

We can examine variables in current stack frame and move up the stack!

gdb	jdb	

`info locals`
`info args`
	`locals`	Print local variables
		Print arguments to function in current stack frame

`p(rint) <var>`	`print <var>`	Prints the current value of variable <var>
`p(rint) <expr>`	`print <expr>`	Prints the value of the executed expression <expr>
	`pop`	Go up to previous stack frame (function that called current function)
`frame <#>`		Select frame on stack for inspection
`bt (backtrace)`	`where`	Print a call stack for current thread

Debugging an Exception

As a starting point:

1. Find last executed line from your project code
2. Set a breakpoint at that line
3. Investigate values of local variables
4. Move up stack as necessarily to look at functions that called the problematic function

Shell Variables

Fill in the blanks in the code segment below:

```
#!/bin/bash
# Asks the user to enter two numbers
______ "Enter two numbers"

# Stores users response into variables
# num1 and num2
______ num1 num2
```

a. `cat / input`

b. `read / echo`

c. `echo / input`

d. `echo / read`

e. More than one of the above

Fill in the blanks in the code segment below:

```
#!/bin/bash
# Asks the user to enter two numbers
echo "Enter two numbers"

# Stores users response into variables
# num1 and num2
read num1 num2
```

a. **cat / input**

b. **read / echo**

c. **echo / input**

d. **echo / read**

e. More than one of the above

Shell Variables

- Variables are symbolic names that represent values stored in memory
- Three different types of variables:
  - *Global Variables*

    Environment and config variables, capitalized (e.g., HOME, PATH, USERNAME)
  - *Local Variables*

    Variables defined by you!

    Local variables are only available in current shell (or children shells with **`export – later….`**)
  - *Special Variables*

    Functionality defined by shell.

    Special variables can only be referenced but not assigned (e.g., **`$#`** for number of parameters or **`$?`** for exit status)

A few global (environment) variables

When you login, many global variables are defined and can be freely referenced by your shell scripts!

SHELL	Current shell
DISPLAY	Used by X-Windows to identify the display
HOME	Fully qualified name of your login directory
PATH	Search path for commands
MANPATH	Search path for <man> pages
PS1 & PS2	Primary and Secondary prompt strings
USER	Your login name
TERM	Terminal type
PWD	Current working directory

Referencing Variables

Variable contents are accessed using **$**:

e.g.

```
$ echo $HOME
$ echo $SHELL
```

To see a list of your environment variables:

```
$ printenv   # recommend pipe to more/less
$ set        # recommend pipe to more/less
```

Defining Local Variables

- Like other programming languages, variables can be defined and used in shell scripts.
- Unlike other programming languages, variables in Shell Scripts are not typed. Variables are strings by default!

```
var1=1234       # NOT an integer, it’s a string
var2=$var1+1    # won’t perform arithmetic!
                # var2 is the string '1234+1'
var3=abcdef     # var3 is string
var3='abcdef'   # same as above but much safer.
var3=abc def    # will not work unless 'quoted’
var3='abc def'  # i.e. this will work
```

IMPORTANT NOTE: DO NOT LEAVE SPACES AROUND THE =

Referencing variables with curly braces

- Having defined a variable, its contents can be referenced by the $ symbol. E.g., **`${variable}`** or simply **`$variable`**.
- When ambiguity exists **`$variable`** will not work. Use the rigorous form **`${ }`** to be on the safe side.

```
alpha='Oh, I '
b1=$alphabet
b2=${alpha}bet      # this would not have worked without the { } as
                    # it would try to access a variable named alphabet
```

- Can display a modified version of the string in a single command

`${string%substring}` : Strips shortest match of **`$substring`** from back of **`$string`**

`${string#substring}` : Strips shortest match of **`$substring`** from front of **`$string`**

```
account='cs15lwi21'
echo ${account%??} # this would print cs15lwi21
echo ${account#??} # this would print 15lwi21zz
```

Variable List/Array

- To create lists (array) – parentheses

```
$ colors=(red green blue)
```

- To set a list element – square bracket

```
$ colors[1]=yellow
```

- To view a list element:

```
$ echo ${colors[2]}
```

Note: Variable lists are indexed from 0!

Example: Variable List/Array

```
$ cat arraytest.sh
#!/usr/bin/bash
myarray=(1 2 3)
echo ${myarray[*]}
echo ${myarray[0]}

$ ./arraytest.sh
1 2 3
1
```

Positional Parameters

- When a shell script is invoked with a set of command line parameters/arguments each of these parameters are copied into special variables that can be accessed
  - **`$0`** This variable that contains the name of the script
  - **`$1`**, **`$2`**, … **`$9`** 1st, 2nd, and 9th command line parameter
  - **`$#`** Number of command line parameters
  - **`$$`** process ID of the shell
  - **`$@`** same as **`$*`** but as a list one at a time
  - **`$?`** Return code 'exit code' of the last command
  - **`shift`** command: This shell command shifts the positional parameters by one towards the beginning and drops **`$1`** from the list.
    - After a **`shift`**, **`$2`** becomes **`$1`** and so on.
    - It is a useful command for processing the input parameters one at a time.

Example: Processing Arguments

```
$ cat myinputs.sh
      #!/bin/bash
      echo Total number of arguments: $#
      echo First input: $1
      echo Second input: $2

$ chmod u+x myinputs.sh

$ ./myinputs.sh ONE TWO BUCKLE MY SHOE
   Total number of arguments: 5
   First input: ONE
   Second input: TWO
```

What would be the expected output of **`echo $0`** if added into the shell script?

The "Here" Document (heredoc)

- Sometimes you want to redirect standard input to come from a small section of text
- Rather than create a file with the text, use **<<** to redirect text to come from the command line to **HERE**.
    - Input stops when unique text (**HERE**) is read as a line by itself

```
$ cat << HERE
> This is text that will be redirected
> This text, too
> Redirection stops with HERE
> HERE
This is text that will be redirected
This text, too
Redirection stops with HERE
```

Other forms of heredocs

- To redirect text from the command line, use **<<<**.

```
$ cat <<< This is text from the command line
This is text from the command line
```

- Using **<<-** ignores tab to allow ignoring formatting tabs at beginning of a line in a script

```
$ cat <<- HERE
>         This is text that will be redirected
>         This text, too
>         Redirection stops with HERE
>         HERE
```

Initial tabs won’t be displayed

Next Lecture

1. Test-Driven Development
2. More Shell Scripting