**Homework 1**

**Programming Languages Principles and Implementation**

**Name:**

**Homework 1 will permit you to review Java.**

**Instructions**

* This homework assignment is to be done alone or in a group of 2 students (not 3).
* Your written solution must be submitted in a separate document under Assignment 1 in https://classes.pace.edu/. Do not change the order and do not remove text. This part will be submitted through Classes.
* You will create a GitHub account as CS361

**Par 1 – Setting up Visual Studio Code 0. Pts**

Visual Studio Code for CS50 Online version: https://cs50.dev/

* Install [Visual Studio Code](https://code.visualstudio.com/docs/languages/java) and extensions for C and Java example such as code runner
* <https://code.visualstudio.com/docs/java/java-tutorial>
* [JAVA](https://code.visualstudio.com/docs/java/java-gui#java-articles)
  + [Getting Started](https://code.visualstudio.com/docs/java/java-tutorial)
  + [Navigate and Edit](https://code.visualstudio.com/docs/java/java-editing)
  + [Refactoring](https://code.visualstudio.com/docs/java/java-refactoring)
  + [Formatting and Linting](https://code.visualstudio.com/docs/java/java-linting)
  + [Project Management](https://code.visualstudio.com/docs/java/java-project)
  + [Build Tools](https://code.visualstudio.com/docs/java/java-build)
  + [Run and Debug](https://code.visualstudio.com/docs/java/java-debugging)
  + [Testing](https://code.visualstudio.com/docs/java/java-testing)
  + [Spring Boot](https://code.visualstudio.com/docs/java/java-spring-boot)
  + [Application Servers](https://code.visualstudio.com/docs/java/java-tomcat-jetty)
  + [Deploy Java Apps](https://code.visualstudio.com/docs/java/java-on-azure)
  + [GUI Applications](https://code.visualstudio.com/docs/java/java-gui)
  + [Extensions](https://code.visualstudio.com/docs/java/extensions)
* From VS Code terminal type : ***cd desktop***
* Create these folder and files :
* Make a directory/folder in desctop: CS361Test.
* ***mkdir CS361Test***
* ***cd CS361Test***
* By typing ***code .*** you will directly open the newly created folder CS361Test
* Create a file Echo.java and save it in CS361Test located in Desktop.
* Create a test file Welcome.java and save in the folder CS361Test located in Desktop
* Create a test file CommandLineArguments.java in folder CS361Test located in Desktop

[Example 1 Hyperlink#3 Echo.java](https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html)

public class Echo {

public static void main (String[] args) {

for (String s: args) {

System.out.println(s);

}

}

}

lleshmiraj@Lleshs-Air ~ % ***cd desktop***  
lleshmiraj@Lleshs-Air desktop % ***cd CS361Test***  
lleshmiraj@Lleshs-Air CS361Test % ***javac Echo.java***  
lleshmiraj@Lleshs-Air CS361Test% **java Echo Drink Hot Java**

**Drink**  
**Hot**  
**Java**

**Example 2 Welcome.java**

// Name of the file :: Welcome.java

// To compile :: javac Welcome.java

// To execute :: java Welcome xxxx, where xxxx is the argument

// Command line arguments are arguments that are specified when executing

// the program from the command line. When using Eclipse, they are called

// Program arguments. Think of these arguments as parameters to the main

// method that are given from the command line OR as parameters to the

// program itself.

// prints a command\_line argument within an enclosing String

public class Welcome

{

public static void main(String[] args)

{

System.out.println("Hello, " + ***args[0]*** + ". Welcome to Java!!!");

}

}  
  
  
lleshmiraj@Lleshs-Air ~ % cd desktop  
lleshmiraj@Lleshs-Air desktop % cd CS361Test  
lleshmiraj@Lleshs-Air CS361Test % javac Welcome.java  
lleshmiraj@Lleshs-Air CS361Test % java Welcome xxxx, where xxxx is the argument  
Hello, xxxx,.  Welcome to Java!!!  
lleshmiraj@Lleshs-Air CS361Test % java Welcome xxxx, where xxxx is the argument  
Hello, xxxx,.  Welcome to Java!!!  
  
 **Example 3 Welcome.java**  
// ***prints argument at index 1***

public class Welcome

{

public static void main(String[] args)

{

System.out.println("Hello, " + ***args[1]*** + ". Welcome to Java!!!");

}

}  
  
lleshmiraj@Lleshs-Air CS361Test % javac Welcome.java  
lleshmiraj@Lleshs-Air CS361Test % java Welcome xxxx, where xxxx is the argument  
Hello, where.  Welcome to Java!!!  
lleshmiraj@Lleshs-Air CS361Test %   
  
**Example 4 CommandLineArguments.java**

public class CommandLineArguments {

public static void main(String[] args) {

System.out.println("Number of Command Line Argument = "+args.length);

for(int i = 0; i< args.length; i++) {

System.out.println(String.format("Command Line Argument %d is %s", i, args[i]));

}

}

}

lleshmiraj@Lleshs-Air ~ % cd desktop/CS361Test  
lleshmiraj@Lleshs-Air CS361Test % javac CommandLineArguments.java  
lleshmiraj@Lleshs-Air CS361Test % java CommandLineArguments "A" "B" "C"                             
Number of Command Line Argument = 3  
Command Line Argument 0 is A  
Command Line Argument 1 is B  
Command Line Argument 2 is C

\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_  
\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_

1. The purpose of this assignment is to familiarize you with some of the

   tools you will be using for the assignments this semester.  You may already know this stuff, in which case feel free to just skim it.  If not, now's the time to learn!

  2.  Try out the "man" command.  At the terminal prompt, type

             man passwd

      You will see the name of the passwd command, the list of possible.

      command line options, and a description of what the command and the

      options do.You will find that almost every command has more

      options than you realized, most of which no one ever uses (try man

      cat!).Also try the following commands (don't bother reading all

      the details about every option - you won't remember them anyway).

             man less

             man ls

             man od

      You can find out more about the man command (which stands for

      "manual") by typing

             man man

  3. Install visual studio code editor and set it up to run C, and Java Programs. Write a "hello, world" program in C++.  You can use the following

      code.

             #include <iostream>

             using namespace std;

             int main()

             // this is a comment

             /\* this is also

                  a comment \*/

             {

                   int i;

                   cout << "Hello, world.\n";

                   i = 3;

                   cout << "i = " << i << endl;

                   return 0;

             }

      Use an editor (vim or emacs) to type this into a file named "hello.cc".

      For those not familiar with C++, the #include command tells the

      compiler to read declarations from another file.  The pointy

      brackets around the name tell the compiler to look in a library of

      "standard" declaration files.

  4.  Compile the "hello, world" program using the command

             g++ -v hello.cc

      The -v option (for verbose) shows the name and arguments of each

      program the compiler executes.

  5.  You should now have an executable file "a.out" which you can run by

      simply typing

             ./a.out

  6.  Read the man page for the compiler and find the command-line switches

      that will cause it to save the various intermediate files you saw in

      step 4 above.  Take a look at these files, using 'less' to look at

      text, and 'nm', 'od', or 'objdump' to look at object (machine

      language) files.  You can find out if a file is text or data by

      using the 'file' command with the filename as the argument.

  7.  Type the following lines into a file named "Makefile":

             hello: hello.cc hello.h

                     echo first message

                     @echo another message

                     g++ -o hello hello.cc

      Be warned that the indented lines have to start with a tab character,

      not a series of spaces.  Create a file called "hello.h" with the

      following line in it:

             const int VAL = 3;

      Finally, edit hello.cc and change it so that it reads as follows:

             #include <iostream>

             #include "hello.h"

             using namespace std;

             int main()

             // this is a comment

             /\* this is also

                  a comment \*/

             {

                   int i;

                   cout << "Hello, world.\n";

                   i = VAL;

                   cout << "i = " << i << endl;

                   return 0;

             }

      Type "make" and notice the output.  You should now have a file called

      "hello" which you can run. Type "make" again and notice that it does

      not recompile hello (because you haven't changed it).

  8.  To see an example of separately compiled source files, type the

      following into a file called "main.cc":

             #include <iostream>

             #include "dumb.h"

             using namespace std;

             int main()

             {

                   cout << "Hello, world.\n";

                   dumb\_func(10);

                   return 0;

             }

      and type the following lines into a file called "dumb.cc".

             #include <iostream>

             #include "dumb.h"

             using namespace std;

             void dumb\_func(int n)

             {

                   cout << "n = " << n << "\n";

             }

      The header file "dumb.h" should have this line in it:

             void dumb\_func(int n);

      Finally, change your Makefile to read as follows:

             .cc.o:

                   g++ -c $\*.cc

             hello: main.o dumb.o

                   g++ -o hello main.o dumb.o

             main.o: main.cc dumb.h

             dumb.o: dumb.cc dumb.h

      Now type "make" and observe what happens.  The strange looking new

      rule at the top of the file tells make how to turn an arbitrary .cc

      file into a .o file.  A warning: make has some built-in rules.  If

      you forget to provide the .cc.o rule, make will use its own, which

      may not call g++ the way you expect.

      Change each of the files main.cc, dumb.cc, and dumb.h and run make

      after each change.  Notice that a change to a source file does not

      force recompilation of unrelated source files.

   9. Use gdb to run one of the above programs or one of your own.  You

      should change your Makefile so that it compiles your program with

      the "-g" option, which facilitates debugging.  You can force make

      to use this option automatically by including it in the .cc.o

      rule:

             .cc.o:

                     g++ -c -g $\*.cc

      To use gdb type "gdb filename" where filename is the name of the

      executable file.  If you have not used gdb before, type "help" at the prompt. Experiment with setting breakpoints and single-stepping, and

      use the commands "list", "display", "where", and "print".

  10. Tag files contain the source file locations of all function,

      macro, and type definitions used in a programming project.  Most

      editors, including vim and emacs, are able to jump directly to the

      source code location of a function, macro,or type definition, as

      specified in the project's tag file.

      To create tag files, edit Makefile and add the following makefile

      target:

             tags:

                     etags \*.cc \*.h

                     ctags \*.cc \*.h

      Now run "make tags" to create tag files for both vim and emacs.

      To use the tags, open hello.cc in your favorite editor, and type

      the appopriate command:

             vim:

                     :ta dumb\_func

             emacs:

                     Meta-.

                     dumb\_func

      This should take you to the definition of function dumb\_func.

      To return to where you came from, type ^t (control-t) in vim, or

      Meta-\* in emacs.

  11. Version control (VC) systems are a crucial tool for keeping track

      of the history of changes made to files.  Such systems maintain

      only the diffs between versions, minimizing space requirements.

      They allow you to recover if you accidentally delete a file, or to

      "back out" to a previous version if you decide you don't like your

      recent changes.  They also facilitate logging of notes about your

      changes.  More advanced features mediate concurrent changes by

      multiple members of a team, either via locking or via intelligent

      merging.

      Much of the systems world has converged around the git VC system.

      Unfortunately, git is very complicated.  Other, simpler

      alternatives include rcs (very old, very simple), cvs (built on

      top of rcs, and still pretty old), subversion (svn), bazaar (bzr),

      mercurial (hg), and perforce (p4).

      Documentation for all of these can be found online.

\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_

**Part 2: Setting up GitHub. 0 pts**

Create a GitHub account (if you do not have one). <https://github.com>.

Create a repository called cs361.

During the course, you will submit your work in GitHub. It will need to be organized. It is up to you but I must not have problems to find your work!

Please open the following link to the management spreadsheet.

<https://docs.google.com/spreadsheets/d/1btUrB613mbIwsXhvzM9f12FZhKFhwWIjD1PFF7iWlMs/edit?usp=sharing>

1. Click the tab GitHub and make sure that I typed your name and email correctly.
2. Post your GitHub cs361 link in the management spreadsheet under GitHub tab
3. Click the Older languages tab and pick a team for your project.
4. Click the Newer languages tab and pick a team for your project.
5. Click the Parser/Scanner tab to pick a team for developing your project.

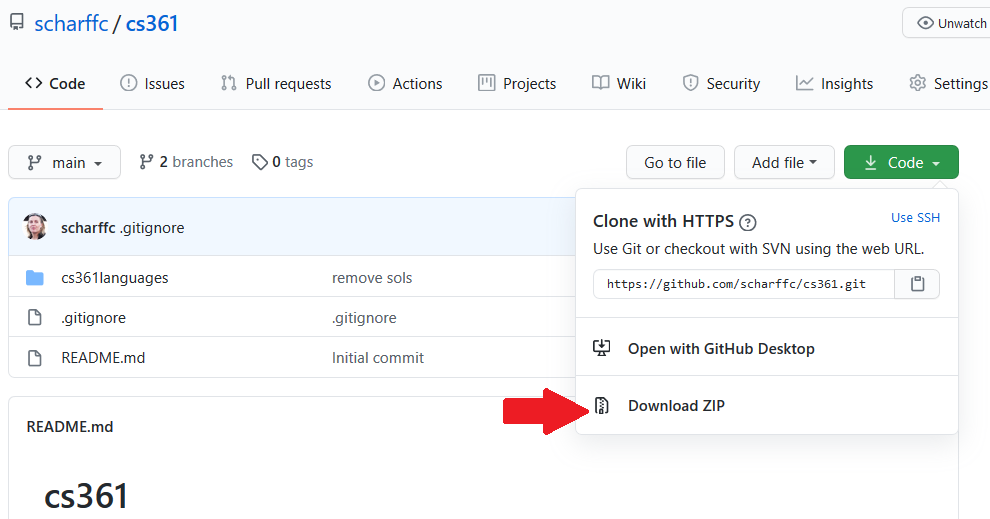
**Part 3: Setting up Eclipse IDE. 0 pts**

Download Eclipse for Java Developers (We do not need the Java EE version). Download the latest version or update / upgrade your current version.

<https://www.eclipse.org/downloads/packages/>

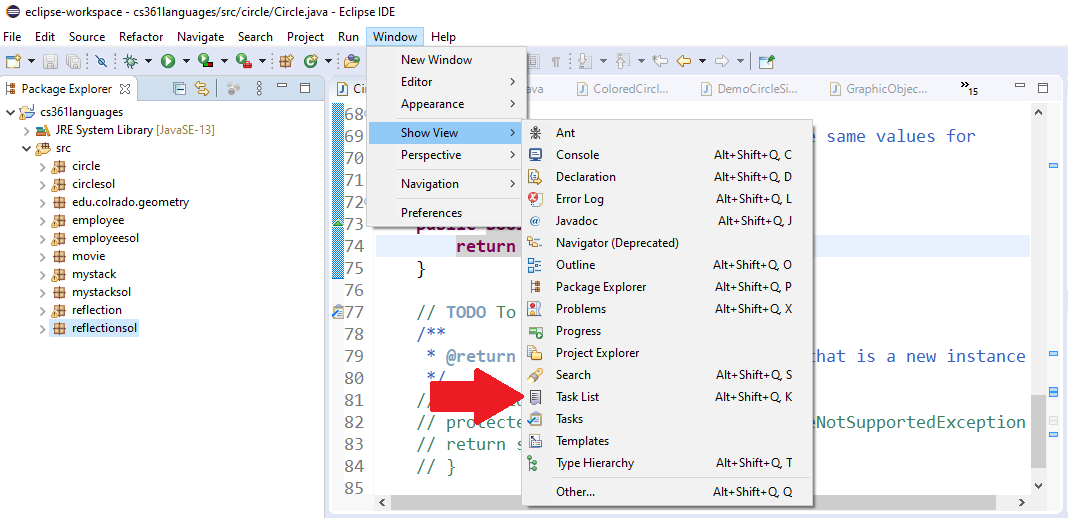
Download the code at <https://github.com/scharffc/cs361>. You can download a zip file.

If you are more comfortable with another IDE it is fine but I may sometimes ask you to have EXACTLY the same organization of files as me for automated grading.

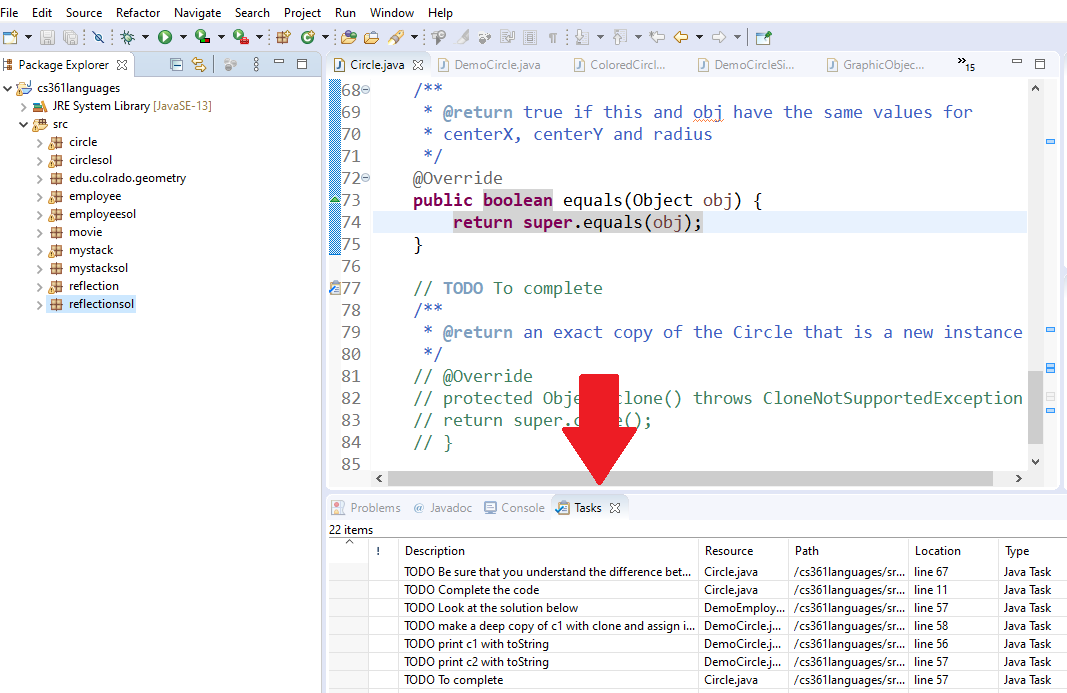


Open the file in Eclipse. The code in cs361 languages was written with Eclipse.

In Eclipse, add the View Tasks.



You should now see 22 TODO tasks.



**Exercise 1: Do the 22 TODO tasks. - 66 pts**

Do not remove the TODO such that I can go through them when correcting your work.

Code is provided to you. In real life, you often have to go through code written by someone else. You will have to read all the code in each package to do the required work.

The code corresponds to the lecture notes. There are explanations in the PDF.

22 TODO tasks may sound a lot but some of them are short!

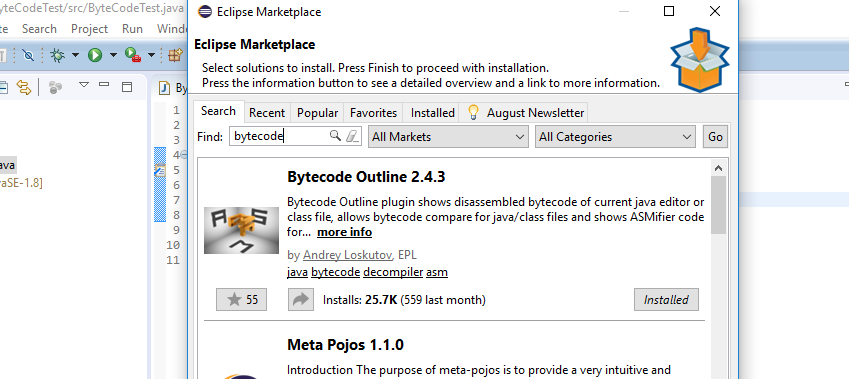
* Circle package
  + This package will make you review polymorphism and crucial Object methods: toString, clone and equals.
* Employee package
  + This package will make you review polymorphism.
* Stack package
  + This package will make you review generics.
* Reflection package
  + This package will make you practice with Reflection.

**Exercise 6: Java Eclipse Bytecode – 27 pts**

In the past, I was using an Eclipse plugin such that we look at Java bytecode. The plugin does not work for latest version of Eclipse.

The infos about the plugin are here:

<http://andrei.gmxhome.de/bytecode/index.html>



You may want to install it but it is not required…

You will find below the Java bytecode generated for the following Java code below.

**public** **static** **int** sum\_for(**int** n) {

**int** i = 0, sum = 0;

**for** (i = 0; i <= n; i++) {

sum += i;

}

**return** sum;

}

1 to 39 are the line numbers that you can use to explain the code.

// access flags 0x9 1

public static sum\_for(I)I 2

L0 3

LINENUMBER 4 L0 4

ICONST\_0 5

ISTORE 1 6

L1 7

ICONST\_0 8

ISTORE 2 9

L2 10

LINENUMBER 5 L2 11

ICONST\_0 12

ISTORE 1 13

GOTO L3 14

L4 15

LINENUMBER 6 L4 16

FRAME APPEND [I I] 17

ILOAD 2 18

ILOAD 1 19

IADD 20

ISTORE 2 21

L5 22

LINENUMBER 5 L5 23

IINC 1 1 24

L3 25

FRAME SAME 26

ILOAD 1 27

ILOAD 0 28

IF\_ICMPLE L4 29

L6 30

LINENUMBER 8 L6 31

ILOAD 2 32

IRETURN 33

L7 34

LOCALVARIABLE n I L0 L7 0 35

LOCALVARIABLE i I L1 L7 1 36

LOCALVARIABLE sum I L2 L7 2 37

MAXSTACK = 2 38

MAXLOCALS = 3 39

Use the references to explain the following bytecode commands:

* ILOAD
* IINC
* GOTO

Explain how this bytecode corresponds to the Java code. In particular, explain how assignments, loops etc. are implemented.

**References**

* The JVM <https://docs.oracle.com/javase/specs/jvms/se8/html> (Java 8 SE) (2020)
* Java Bytecode Basics <http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html> (1996)
* <http://www.beyondjava.net/blog/java-programmers-guide-java-byte-code/> (2015)

**Exercise 7: Java Visual Studio Code Bytecode – 7 pts**

//1) Copy this code into Visual Studio Code editor

public class Test2 {

     public static int sum\_for(int n) {

                int i = 0, sum = 0;

                for (i = 0; i <= n; i++) {

                      sum += i;

                 }

      return sum;

     }

     public static void main(String[] args) {

             System.out.println("Number: " + sum\_for(3) + "\n");

          }

}

//2) Save files as Test2.java

//3) Use javac Test2.java to Compile: and create Test2.class

***javac Test2.java***

***java Test2***

//View bytecode in Test2.class by using javap

***javap -c Test2.class***

// 4) See Bytecode below

Compiled from "Test2.java"  
public class Test2 {  
  public Test2();  
    Code:  
       0: aload\_0  
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V  
       4: return

  public static int sum\_for(int);  
    Code:  
       0: iconst\_0  
       1: istore\_1  
       2: iconst\_0  
       3: istore\_2  
       4: iconst\_0  
       5: istore\_1  
       6: iload\_1  
       7: iload\_0  
       8: if\_icmpgt     21  
      11: iload\_2  
      12: iload\_1  
      13: iadd  
      14: istore\_2  
      15: iinc          1, 1  
      18: goto          6  
      21: iload\_2  
      22: ireturn

  public static void main(java.lang.String[]);  
    Code:  
       0: getstatic     #7                  // Field java/lang/System.out:Ljava/io/PrintStream;  
       3: iconst\_3  
       4: invokestatic  #13                 // Method sum\_for:(I)I  
       7: invokedynamic #19,  0             // InvokeDynamic #0:makeConcatWithConstants:(I)Ljava/lang/String;  
      12: invokevirtual #23                 // Method java/io/PrintStream.println:(Ljava/lang/String;)V  
      15: return  
}

Use the references to explain the following bytecode commands:

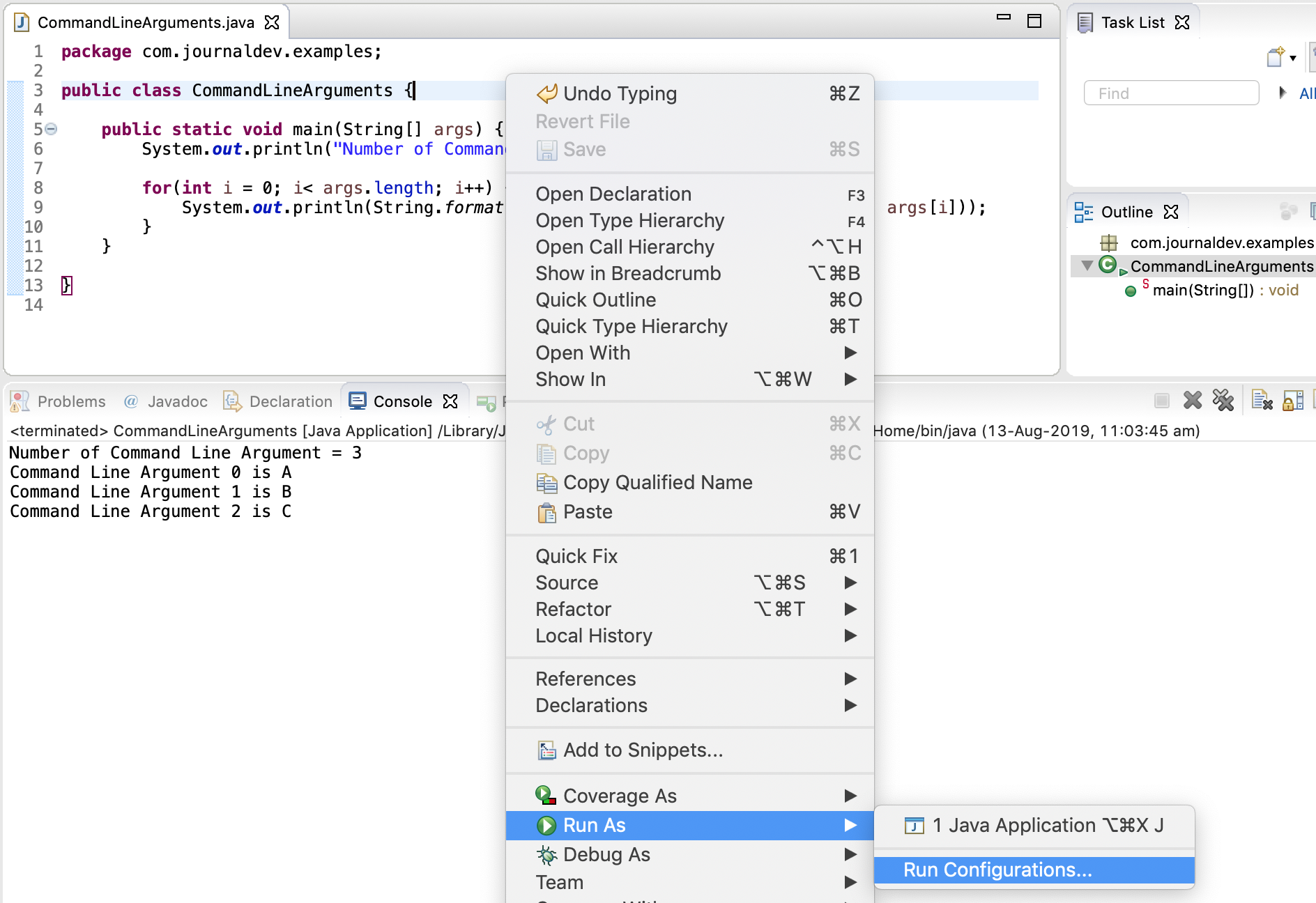
* ILOAD
* IINC
* GOTO

Explain how this bytecode corresponds to the Java code. In particular, explain how assignments, loops etc. are implemented.

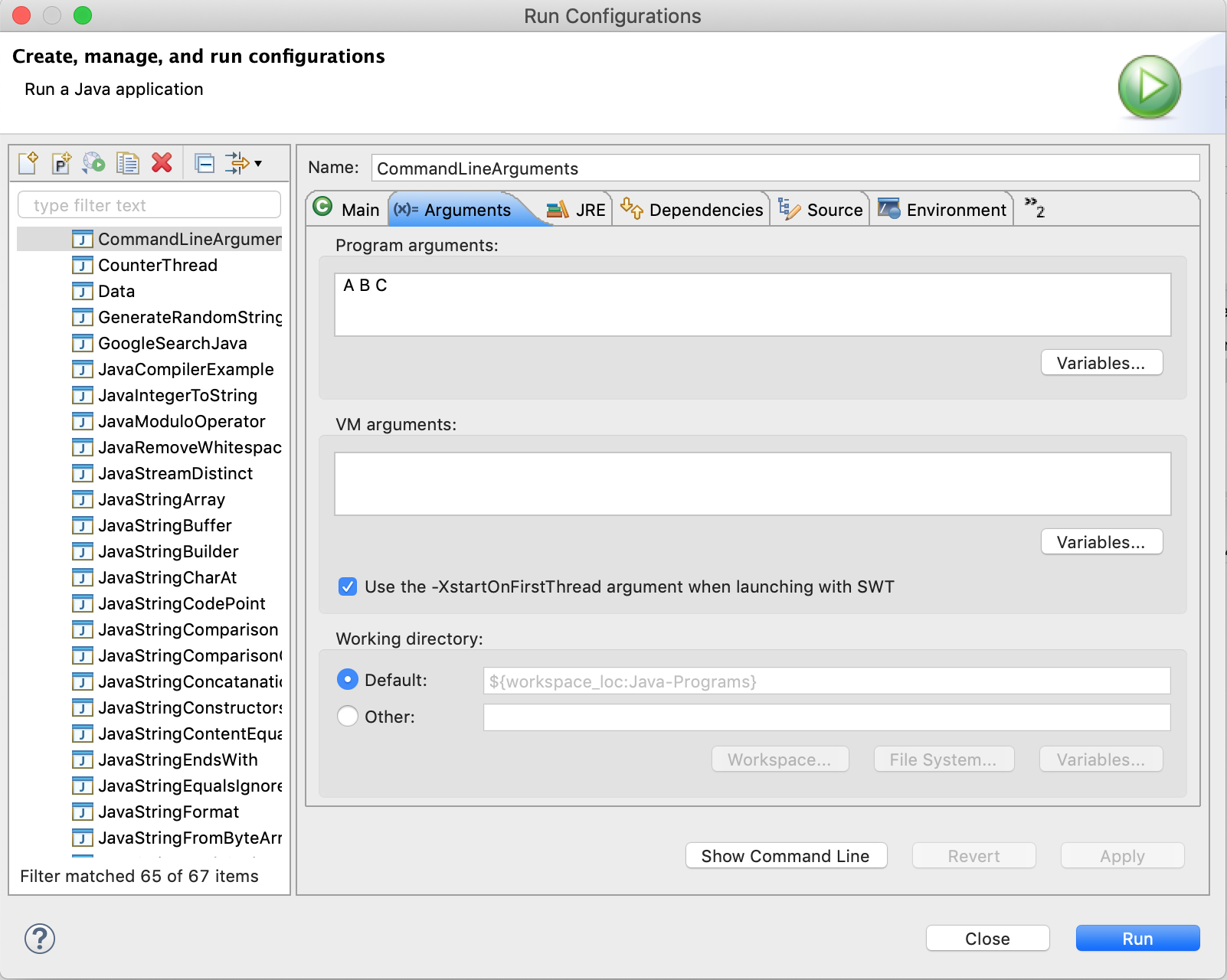
**References**

* The JVM <https://docs.oracle.com/javase/specs/jvms/se8/html> (Java 8 SE) (2020)
* Java Bytecode Basics <http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html> (1996)
* <http://www.beyondjava.net/blog/java-programmers-guide-java-byte-code/>  (2015)
* **How to Pass Command Line Arguments in Eclipse -0 pts**
* We can also pass command-line arguments to a program in Eclipse using Run Configurations.

#### Step 1: Open the Class Run Configurations Settings

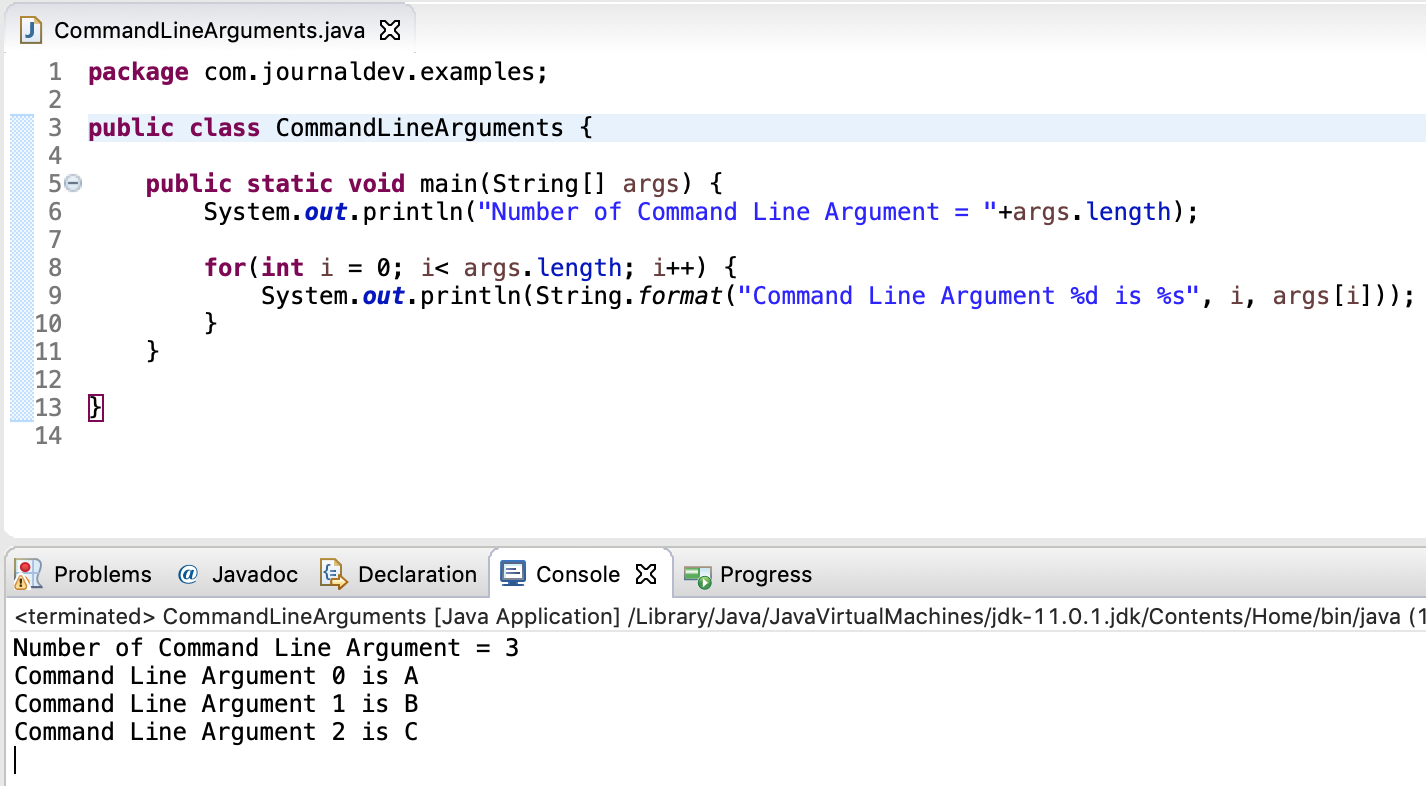
* From the class editor, right click and chose “Run As” -> “Run Configurations…”.
* Eclipse Run Configurations

#### Step 2: Specify the Program Arguments in the Arguments Tab

* In the pop up window, click on the Arguments tab. Then provide the command line arguments value in the “Program Arguments” text box.
* Eclipse Command Line Arguments

#### Step 3: Click on the Run button

* When you will click on the Run button, the run configurations will be saved and the program will execute with the specified command-line arguments.



Eclipse Command Line Arguments Example

* If you run the class again, the saved run configuration will be used. So if you want to override the command-line arguments or remove them, you will have to open the run configurations window and make necessary changes.

## [Conclusion](https://www.digitalocean.com/community/tutorials/command-line-arguments-in-java#conclusion)

* The command-line arguments are used to provide values that are essential to run the program. For example, we can specify the database credentials to be used by the program. We can specify the configuration file location from where the program should pick the required values. Reference: [Command-Line Arguments Oracle Docs](https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html)