

This lab has **eight (8)** checkpoints.

Learning Outcomes

Upon completing this lab, students should be able to

- **trace** Java code that uses an `ArrayList`.
- **clone**, **init**, **config**, and **push** a simple git repository using the Departmental Gitea server at <https://cs-devel.potsdam.edu>
- **use** a predefined **class** as the content in an `ArrayList`, reading data from a file.
- **implement**, **compile**, and **run** a Java application program that uses two different Java **packages**.

Introduction

1. Dr Ladd spent all his time configuring lab computers; he appears to have forgotten how to write Java. When he wrote the following:

```
1 import java.util.ArrayList;
2 import java.util.List;
3
4 public class Dumb {
5     public static void main(String[] args) {
6         List<int> numbers = new ArrayList<int>();
7
8         int sum = 0;
9         for (int i = 0; i < 10; i++) {
10             sum += (i + 1);
11             numbers.add(sum);
12         }
13
14         for (int i = 0; i < numbers.size(); i++) {
15             System.out.println(numbers.get(i));
16         }
17     }
18 }
```

He gets the following error messages

```
Dumb.java:5: error: unexpected type
    List<int> numbers = new ArrayList<int>();
    ^
    required: reference
    found:    int
Dumb.java:5: error: unexpected type
    List<int> numbers = new ArrayList<int>();
                                ^
    required: reference
    found:    int
2 errors
```

Using the official Java documentation, write down how you would fix these errors. (Don't rely on any old StackExchange post.)

Then write down (in your own words) **why** this error occurred.

✓ Show your answers to the lab instructor and expect to have to explain them.

2. What output is generated by the following Java code?

```
import java.util.Scanner;

public class Scan {
    public static void main(String[] args) {
        String longTime = "a long time ago, in a galaxy far, far away...";
        Scanner ltScanner = new Scanner(longTime);
        while (ltScanner.hasNext()) {
            System.out.println "[" + ltScanner.next() + "];"
        }
    }
}
```

✓ Show your sample output to the lab instructor and expect to explain why that is what it is.

3. On the lab machine you are sitting in front of, what is the **version** of Java that you are running?

✓ Show your answer to the lab instructor and expect to explain how you found the version number.

Consider writing a program that reads a collection of annual low temperatures for some number of locations from a text data file. Each record has a year, a temperature in Fahrenheit, and the name of the US city.

When run, the program should

Check if there are too many or too few commandline arguments

Halt with error message if so.

Read file name from args[0]

Create a ClimateHistory object from the named file

Open file, read by line, break each line into three fields

Add a new TempData object to the history ArrayList

Loop

Prompt for city name to standard output.

Read city name from standard input.

Exit if city name is "exit".

Call ClimateHistory.byCity(city) to print all records for city.

Note that TempData.java and ClimateHistory.java will be in the climate package (because they work together to keep climate data) and the main application, ClimateDB.java, will be in the application package.

4. Initialize a **new** git repository in a **new working directory** where you will implement your solution.

git Default Branch

Make sure you have run the following configuration command before you run `git init`. (Should only be needed on a given computer/system once.)

```
git config --global init.defaultBranch main
```

Create a new *working directory*¹ for this lab. The name and location of this folder are up to you.

Use the `git init` command *inside* the working directory to initialize a version control database.

Create a `.gitignore` file in the root of your working tree. This file contains patterns of files that git does *not* track or add. `.gitignore` is added to the repo and lists what kind of files to ignore in version control.

A `.class` file that is created by compiling a `.java` file you wrote does *not* need to be tracked because it can be **generated** by the compiler. So, the contents of this `.gitignore` should be the following two lines:

`.gitignore`

```
# ignore compiled .class files ANYWHERE in the working tree
*.class
```

Use `git add` to *stage* your new `.gitignore` file for git. Then use `git commit` to commit them with a short message on this being the initial commit.

✓ Show the directory contents and `git log` output to the lab instructor. Be prepared to show the contents of `.gitignore` to them as well.

5. Create two directories below the root of your working tree, `climate` and `application`. A **package** in Java matches a directory².

Create the `application/ClimateDB.java` file. Put in a `main` function. In `main`, check the number of arguments and if it is not exactly one, print an error message and call `System.exit()` with a non-zero value (you pick). If you do not exit, print out the argument.

Something like this (where you fix up the comments):

```
if (/*number of args != 1*/) {
    System.err.println(/* ERROR MESSAGE */); // prints to error output
    System.exit(/* non-zero exit code */);
}
System.out.println(/* the single argument */);
```

IMPORTANT: Any Java source file inside a **package** must tell Java that it is inside that package. The **first** line in `application/ClimateDB.java` must be

```
package application;
```

Go up to the *root* of the working tree and compile the code:

¹this *working directory* is the **root** of your *working tree*, the collection of all the directories in your project

²Similarly to how a `class` appears in a source file named for the class within it.

```
javac --class-path . application/ClimateDB.java
```

The name of the *file* to compile requires the name of the *directory* where it lives. But Java needs to know where to start looking for **your** packages. That is what goes after the `--class-path` commandline argument. The single dot means the *current* directory.

After it compiles, the `.class` file(s) are in the directories where the `.java` files are. Thus the line to **run** the program is similar to the above:

```
java --class-path . application.ClimateDB.java
```

The name of *what to run* is now the name of a class inside of a package. So the directory separator, `/` is replaced with Java's name separator, `.`

Use `git add` to add the application folder to version control and then `git commit` to put this working code into the `git` database.

✓ Show your program to the lab instructor. Be prepared to show `git log`, compiling, and/or running the code. When running, make sure it fails when it should.

Put all the files in your working directory for this lab. This is a Java program with multiple `.java` files. How should you compile it?

Compile it and run it.

✓ Show the directory contents, compilation output, and running program output to the lab instructor. Be prepared to explain how it compiles.

6. Look in the `src` folder for this lab. You will see the `ClimateHistory` and `TempData` files.

A `TempData` object contains a year, a low temperature, and the name of a location. `ClimateHistory` contains a list of `TempData` objects. The `ClimateHistory.history(String locationPrefix)` method prints out all history records where the location begins with the given prefix.

`TempData` is *almost* complete. `ClimateHistory` is more of a *shell* with comments that you will replace with working code.

Copy (or move) both Java files to the `climate` package in your working directory. Edit them:

- **both** classes need the appropriate **package** identification at the top of the file.
- replace the comments inside the *read-file-by-line* loop in `ClimateHistory`'s constructor so it *does* what the comments say:
 - read a line into a string
 - wrap a `Scanner` around the line
 - extract the year from the line (an **int**);
 - extract the low temperature from the line (an **int**)
 - extract the rest of the line (the location name) into a string
 - remove the first character from the location (it is a blank space) using `substring`
 - add a new `TempData` for the given record to the history list

Example:

1970 -29 Ann Arbor, MI

becomes a `TempData` with

```

year=1970
tempF=-29
location=Ann Arbor, MI

```

- Complete the method `historyFor` with a loop that goes across the entries in `history` and prints out those with a location that starts with the characters passed in to `locationPrefix`. That is, if `city` is "B", then *Barrow, AK* or *Boston, MA*, and *Brighton, MI* would "match". See documentation for `String.startsWith` to make your life much, much easier.
- In `application.ClimateDB.main`, add a `ClimateHistory` object. You will need to **import** it because it is not in the same package.³ The name of the class is **<package>.<class>**.
- Use **new** to make a new `ClimateHistory`. Compile and get rid of syntax errors. An error about an uncaught exception remains. Why?
Add a **throws** clause to `main` to fix the error.

✓ Show the directory contents, compilation output, and running program output to the lab instructor.

7. Finally, insert the following loop *after* the `ClimateHistory` object is constructed and fill in the middle to have it use `historyFor` in the object.

```

Scanner keyboard = new Scanner(System.in);
System.out.print("City? ");
while (keyboard.hasNextLine()) {
    String city = keyboard.nextLine();
    if (city.equals("exit"))
        break;

    // your code goes here!

    System.out.print("City? ");
}

```

✓ Show the directory contents, compilation output, and running program output to the lab instructor. Use `data/temps.txt` for sample data.

8. Create your turn-in organization on Gitea: `S24-205-<CCID>` (see <http://cs.potsdam.edu/Classes/git-videos/CreatingAGiteaOrganizationForTurnIn.mp4> for step-by-step instruction). Then create a repository for this lab: `S24-205-<CCID>-l001-Packages`, connect the two repositories (local and remote), and push your solution (see <http://cs.potsdam.edu/Classes/git-videos/GitRepoTurnIn.mp4> for details).

✓ Show the repository contents, on the departmental Gitea server, with the *correct* name, in the correct organization, to the lab instructor.

³When you wrote multi-Java file solutions before, all the classes were in the same directory...or, to Java, in the same **package**. You do not need to import classes in the same package.