



# **Java Programming: Basics to Advanced Concepts**

Advanced Programming Workshop

**Chris Simber**

Assistant Professor, Computer Science

Rowan College at Burlington County

### **Cataloging Data**

Names: Simber, Chris, author.

Title: Java Programming: Basics to Advanced Concepts  
Advanced Programming Workshop

Subjects: Java (Computer Program Language)

Chris Simber

Assistant Professor of Computer Science

Rowan College at Burlington County

Author contact: [csimber@RCBC.edu](mailto:csimber@RCBC.edu)



**Attribution-NoDerivs**

**CC BY-ND**

This work is licensed under CC BY-ND 4.0. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/4.0/>

# Introduction

This book is intended for use in a programming course in Java for students who are familiar with computer programming in another language such as C++ or Python. It follows the flow of a standard text for a programming language, with a focus on highlighting Java specifics and differences. This means that it moves quickly from variables to multi-file, multi-window advanced programming in a *how to implement things in Java that they already know* perspective. The goal is to provide students with the differences that can be expected when programming in Java, in addition to the capabilities of the language.

The text is designed for instruction in a course in which students develop a semester-long project in Java, but can be used for courses that require multiple programs as well. The classroom format for the semester-long project is a brief lecture followed by a collaborative or team workshop. An example of a comprehensive project is provided in Appendix C and others are available by request.

The examples within the chapters closely follow the coding standards for Java publicized by W3C and set forth in the Java Coding Guidelines created by Joe McManus MGR at Carnegie Mellon University, Software Engineering Institute. An adequate abridgement of programming standards is included in Appendix E. The examples reinforce the material introduced while building on previous material covered. They provide the information necessary to meet each milestone in the accompanying project chronologically. Again, a comprehensive project is in view. For this reason, there are no end-of-chapter reviews, summaries, or questions. However, the chapter exercises are numbered for clarity using a shaded box, and can be assigned to enhance instruction. Beginning in Chapter 5, the examples use

portions of various projects, and there are accompanying slides for instruction.

The Java version in use at the time of this writing is Version 8. The Integrated Development Environment (IDE) selected is Eclipse version 2019-06 which is free to download and use. The Eclipse interface is common in look and feel to most integrated development environments and is used extensively in industry.

Instructions for obtaining and installing Eclipse are provided in Appendix A, including resolving JRE and JDK issues.

Getting started in Eclipse is provided in Appendix B with a sample project start-up program. Links to the Eclipse web site, Java Tutorials, and the Java Coding Guidelines are included in Appendix D which also includes a link to the W3Schools web site Java tutorial.

Acknowledgements:

Rowan College at Burlington County Student Feedback

Revision History:

Draft:	June 2020
Draft (1):	November 2020
First Edition:	January 2020

# Contents

Chapter 1	Java Programming & Process	1
Chapter 2	The Eclipse IDE	9
Chapter 3	Programming in Java	15
Chapter 4	Decisions, Logic, Loops, and Methods	25
Chapter 5	Interface Design and Development	33
Chapter 6	File Handling	49
Chapter 7	Strings, and ArrayLists	63
Chapter 8	Main GUI Design and Components	71
Chapter 9	Main GUI and Data Display	83
Chapter 10	Dates, Time, Sound, and More	101

Appendix A	Installing Eclipse	
Appendix B	Getting Started with Eclipse	
Appendix C	Weather Data Analysis Project	
Appendix D	Helpful Links to Information	
Appendix E	Java Programming Standards	
Appendix F	Multiple Panels and Layout Managers Example	
Appendix G	Index of Programming Examples	

“Five minutes of design time, will save hours of programming” –  
*Chris Simber*

# Chapter 1

## Java Programming & Process

The Java programming language was initiated as a project in 1991 by James Goslin, Mike Sheridan, and Patrick Naughton, and was originally designed for embedded systems. With the introduction of web browsers, and the price reductions and speed increases for computers in the 1990's, Java developed into a general-purpose programming language with the release of version 1.2 (Java 2) in 1998. The current version is Java SE13 released in September 2019. Java is a class-based, object-oriented language that is compiled to bytecode and runs on any virtual machine.

JVM - The Java Virtual Machine (JVM) enables computers to run Java programs. The JVM converts Java bytecode into machine language, manages memory, and is part of the JRE. It allows Java programs to run on any device and operating system.

JDK - The Java Development Tool-kit (JDK) is a development environment for creating Java programs and applets that includes the Java Runtime Environment (JRE), an interpreter, compiler, archiver, and documentation generator. There are a variety of JDK's for different operating systems and environments available.

JRE - The Java Runtime Environment (JRE) is an implementation of the Java Virtual Machine that executes Java programs.

## A second Language

It is recommended that programmers be proficient in a programming language, familiar with others, and not be intimidated by any. Programming trends and employment options warrant a broader knowledge in computing than a single language or development environment provide.

### Programming Trends

- Server side (cloud-like) 1980s
- Stand-alone executables 1980s thru present
  - dramatic increase in available software
  - evolution of interfaces
- Web applications 1990s thru present
- Cloud applications (server side) 1993 thru present
- All of the above Today

The extensive use of Graphical User Interfaces (GUIs) and network and internet utilization including internet interfaces and transactions, have increased the need for multi-language programmers. Maintaining existing programs in various languages is a major area of the computer programming industry as well. For example, FORTRAN has been used in math and science, COBOL for business and finance, C and C++ in many areas, Java in web applications, and so on. At the time of this writing there were approximately 250 programming languages in use. Some of these have been and are used extensively, others not so much. Each has benefits and limitations as well as a following, advocates, and critics.

Given some familiarity with programming (variables, functions/methods, classes and objects, logic, flow of control, algorithm development, etc.), adapting to Java should not be difficult.

As an example, a simple output statement in C++, Python, and Java. Note the similarities.

C++	<code>cout &lt;&lt; "This program computes ..." &lt;&lt; endl;</code>
Python	<code>print ('This program computes ...\n')</code>
Java	<code>System.out.println ("This program computes...");</code>



Here are a few examples of Java code along with comments for explanation.

```
// requesting and obtaining input, and storing it in a double variable
Scanner in = new Scanner (System.in);
System.out.println ("Enter temperature in Fahrenheit");
double tempF = in.nextDouble();

// computation raising windSpeed to the 0.16 power using "Math.pow()"
double var1 = 35.74 + (Math.pow(tempF, 0.16));

// displaying the variable var1 as formatted output in Java
System.out.printf("The value computed is %.2f", var1);
```

There are some similarities in the Java lines above with other languages including data types for variables, the use of braces, two forward slashes for comments, and semicolons as end of line markers. However, keyboard input requires a scanner in Java and output requires some minor statement variations depending on the type and formatting of the output. These and other similarities and differences will be detailed and highlighted throughout the text.

## The Agile Development Process and Design and Development

As with any language, increasing design time shortens development, testing, and debugging time, and design tools typically utilized in other languages can be used with Java: design documentation, pseudo-code, Story Boards, IPO (Input Processing Output) documents, flow charts, UML diagrams, etc. I have often said that "Five minutes of design time can save hours of development and debugging". This statement has proven to be true many times over.

Development cycle and IDE tools can be used as well including: text editors, watch windows, error alerts, breakpoints, comments, and output statements.

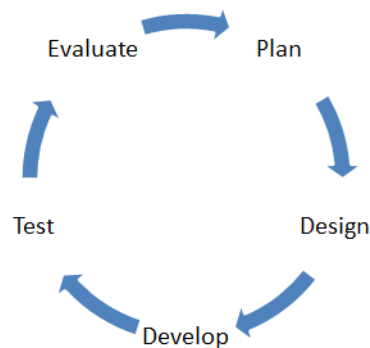
Tools for software teams and software project managers vary in terms of process and method, but are commonly used in industry to plan and measure project progress and provide visibility into the design, schedule status, cost, and quality of the code. The Agile Development Process is a popular method in use today. Agile processes go by various names, but all are iterative and incremental software methodologies that lend themselves to Java program development.

The most popular Agile Methodologies include:

- Scrum – regular meetings, periodic cycles called sprints
- Crystal - Methodology, techniques, and policies
- Dynamic Systems Development Method (DSDM)
- Extreme Programming (XP)
- Lean Development
- Feature-Driven Development (FDD)

The **Scrum** methodology is further explained in terms of **sprints** to align with the milestone tempo of this text for project design and development.

A key component of the Agile Development Process is a sprint. Sprint meetings occur periodically (usually weekly or twice monthly) and include a review and planning event. Tasks completed from the previous sprint plan are reviewed, and completed work may be demonstrated to clients for feedback and approval. The tasks that were not completed from the previous sprint plan are reviewed with a course of action (re-plan). The scope of work that will be completed during the next sprint cycle is planned, and engineers are assigned to the tasks.



### Agile Development Cycle

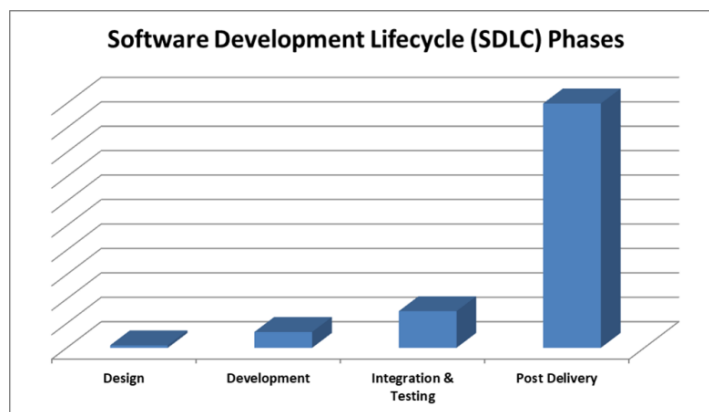
#### Requirements

Prior to the planning and design phase, a complete understanding of what the program is supposed to do is needed. How it will do what it is supposed to do will be determined as the design phase is completed during the software development phase. **Requirements decomposition** is the act of discerning in

detail from the requirements what the program is to accomplish. This process also assists in decomposing the project into manageable “chunks” in terms of schedule and team assignment for development.

## Design

As the requirements are decomposed and documented, the design phase begins, and the break-down of required tasks and logical steps in the program are developed. Design is a very important part of the software development cycle because of the cost escalation of changes and bug fixes further on in the process. This is highlighted in the chart below from the IBM Systems Sciences Institute.



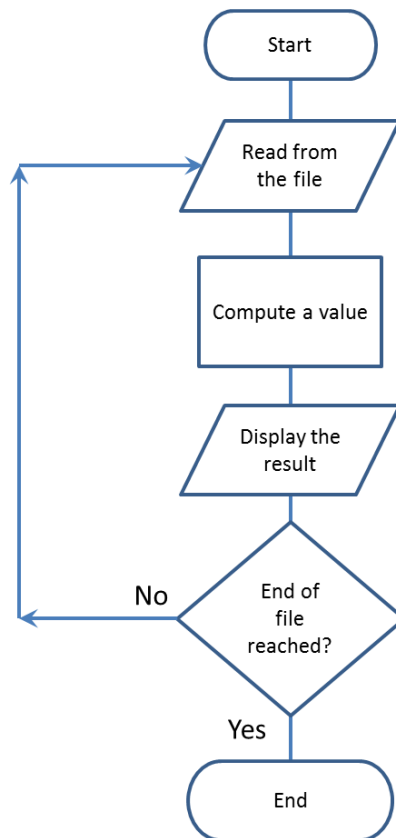
Cost Increase of Fixing Errors by Phase

Software engineering tools that assist in this process include pseudo-code, and flowcharts that graphically show the order of operations. Consider a program to read data from a file, compute a value, and display the results. The pseudo-code for the solution might be:

- Step 1. Start the program
- Step 2. Read data from the file
- Step 3. Compute the value
- Step 4. Display the output
- Step 5. End of file reached?
  - If No, go back to Step 2
  - If Yes, go to Step 6
- Step 6. End the program

The pseudo-code steps above do not include opening and closing the file. They might be considered obvious steps in the process. The level of detail is subjective. Rather than use a document, pseudo-code for portions of the program could be typed into the text editor of the IDE as comments. Later, they can either be replaced with actual code or kept in some form as a comment to the code.

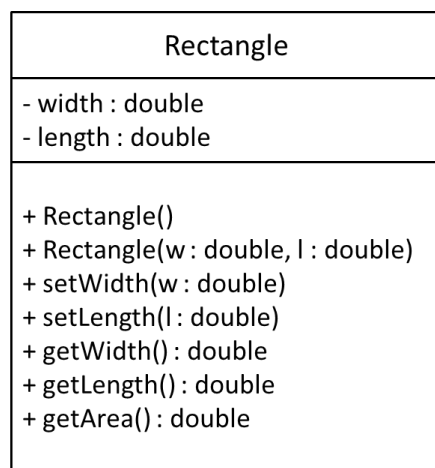
Since we think in pictures and not text, a flowchart provides a faster and clearer depiction of the algorithm and logic. If we are simply ensuring that we have a robust algorithm and haven't missed any steps, then flowcharts can be sketched quickly on a piece of paper and discarded after the program is testing correctly. If the flowchart will be used later, or is part of the deliverable product, then a flowcharting application such as LucidChart® could be used. It is common for larger organizations to divide the design and development tasks into teams or to subcontract the software development portion out-of-house. In these instances, flowcharts are often required to be delivered to the development team or subcontractor together with specific requirements for the code. A flow chart for the example algorithm is shown below.



File Reading Flow Chart

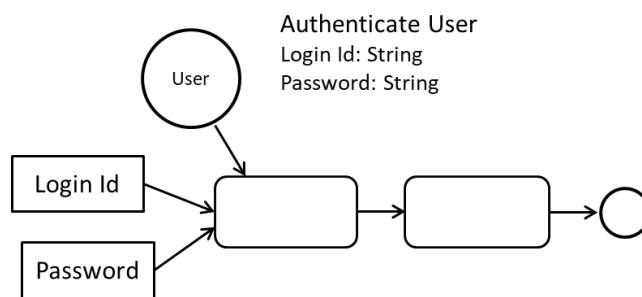
Flowcharts can ensure that steps in the process haven't been overlooked and that there is a complete understanding of the operational flow of the program.

In object oriented programming, Unified Modeling Language (UML) diagrams describe the class and attributes (member variables and methods). The Rectangle example below employs the use of minus signs for private members and plus signs for public. The member variables (top section) or attributes are followed by a colon and the data type. The methods (bottom section) including constructors are followed by parenthesis containing a name, colon and data type. If the method returns a value as is the case with *getWidth()*, *getLength()*, and *getArea()*, the method is followed by a colon and the data type of the return value.



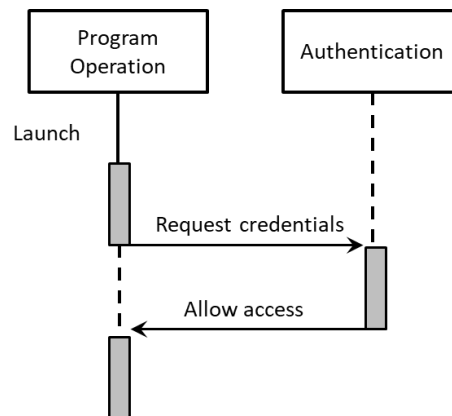
UML Diagram

UML behavior, activity, object activity, or sequence diagrams are used to show the flow of control, data, and transactions. UML Superstructure Specifications provide a standard for object interaction depiction. The diagram below illustrates the authentication of user activity with Login Id and Password.



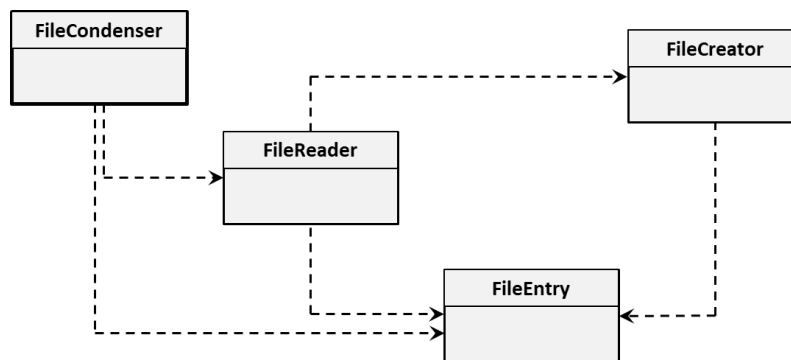
Object Activity Diagram

The Sequence Diagram adds the chronological aspect.



Object Sequence Diagram

Object relationships can be depicted in many ways. The Class Diagram below shows the static view of the program and the relationships. As an example, the FileCondenser class *uses* the FileReader and FileEntry classes.



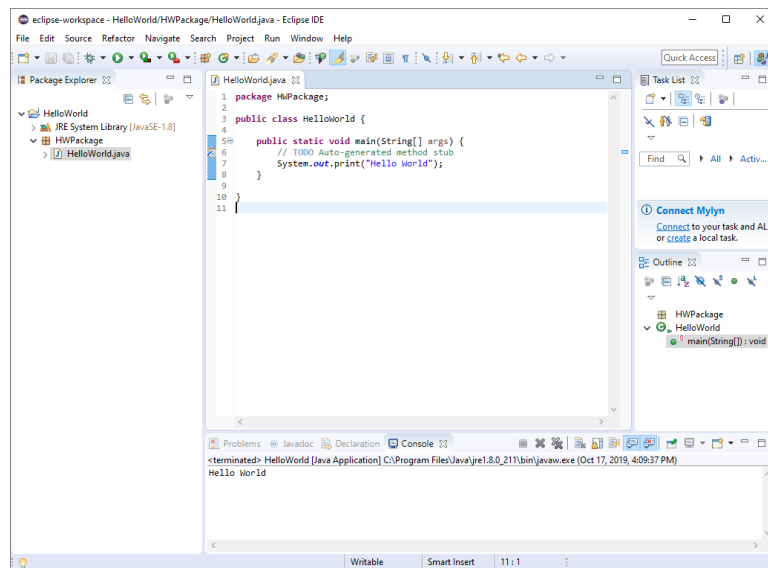
Object Sequence Diagram

Many software engineers use a combination of these tools to design and implement solutions. Pseudo-code may be used for a high-level description of the program or a program area, and a flowchart might be used for more complex sections or compound conditional statements. Either way, the goal is to have a comprehensive understanding of the requirements at every level to ensure that the final product meets the requirements and produces accurate results.

# Chapter 2

## The Eclipse IDE

Obtaining the Eclipse IDE is covered in Appendix A and should be downloaded and installed prior to continuing. The Eclipse Integrated Development Environment (IDE) is free to download and use, and is similar to most IDEs in look-and-feel and capability. It is the most widely used IDE for Java programming, and is suitable for starting out in Java as well as advanced programming and collaboration. It is also used by many companies and provides extensive functionality. The Eclipse version used in this text is 2019-06.



The Eclipse IDE

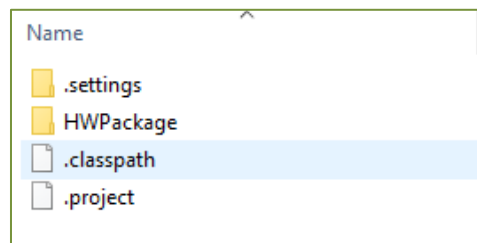
## The Eclipse IDE

Once Eclipse is installed and launched, there are several steps required to write a program. Each program should be created as a project which enables Eclipse to generate supporting files. A project is created using File | New | Java Project. Appendix B walks through the steps required to create a Project, Package, and Class to begin programming and includes screen captures for clarity. The “Quick Start” steps are repeated here for convenience.

### Eclipse – Quick Start

- Launch Eclipse, select the workspace folder from the list or create one
  - Eclipse will start
- Close the Welcome window by clicking on the ‘X’, and the IDE will appear
- Select File | New | Java Project
  - The ‘Create a Java Project’ box will popup
  - Name the project, and the project will appear in the Package Explorer
- With the project name highlighted, add a Package by clicking the ‘New Java Package’ icon, and give it a name.
- With the package name highlighted, add a class by clicking the ‘New Java Class’ icon, and the class creation window will popup.
- Give the class a name the same as the Source/Project name.
  - Check the ‘public static void main(String[ ] args)’ box
- Click on the ‘Finish’ button

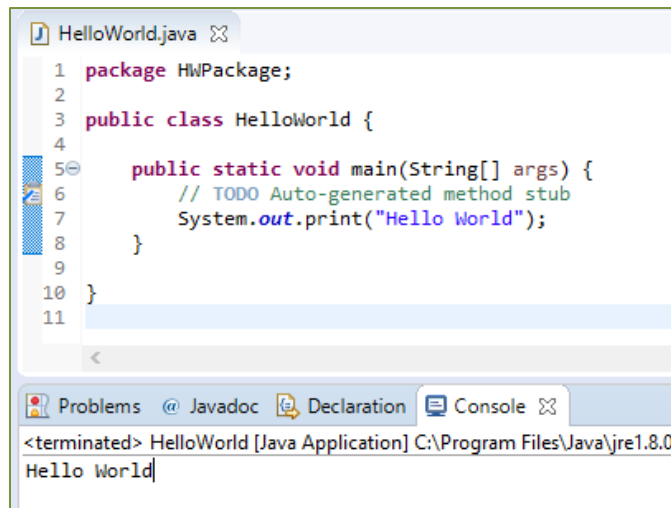
The creation of a project in the Workspace provides an area for the supporting files for the program. The Hello World directory from the screen capture above contains several files and folders as shown here.



Hello World Example Folder



Code is written in the edit window, and standard output is displayed in the Console area in the bottom section of the IDE.



```

HelloWorld.java
1 package HWPackage;
2
3 public class HelloWorld {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         System.out.print("Hello World");
8     }
9
10 }
11

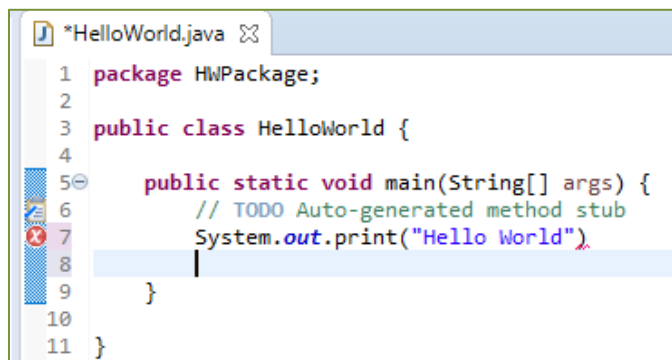
```

Problems Javadoc Declaration Console

<terminated> HelloWorld [Java Application] C:\Program Files\Java\jre1.8.0  
Hello World

Hello World Example Folder

Errors in a Java program in Eclipse are shown in several ways. Below, the semicolon has been removed from line 7. The error is highlighted at the margin with a red circle containing a white “x”, and at the location where the semicolon should be there is a red wavy underline.



```

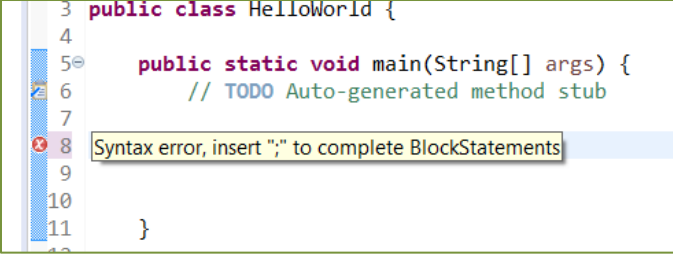
*HelloWorld.java
1 package HWPackage;
2
3 public class HelloWorld {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7         System.out.print("Hello World").
8     }
9
10 }
11 }

```

Hello World Example Edit Window

Hovering over either error indicator with the mouse will display a pop-up message with suggestions for correcting the error. Care should be used when selecting one of the suggestions to ensure that it is the desired solution. Often a list of “Quick Fixes” will be shown that includes a variety of options.

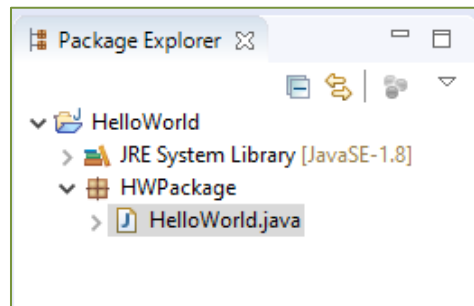
The screen capture below shows the suggestion displayed when the red circle containing the white “x” is hovered over.



```
3 public class HelloWorld {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         Syntax error, insert ";" to complete BlockStatements
9
10
11     }
```

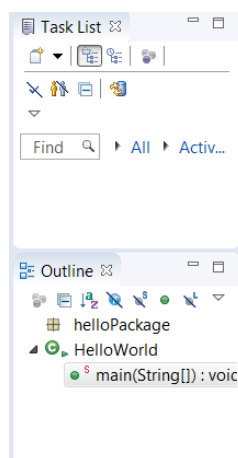
Hello World Example Error Information

The Package Explorer on the left side of the IDE lists projects and their packages, and files associated with each project.



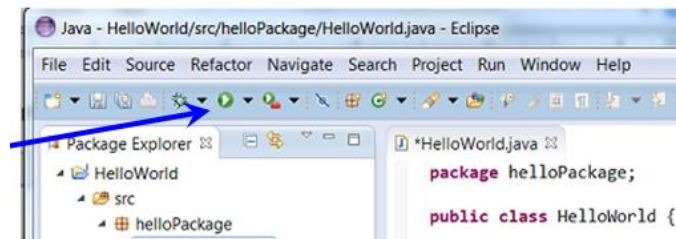
Hello World Example Package Explorer

The right hand side of the IDE contains the Task List and Outline panes.

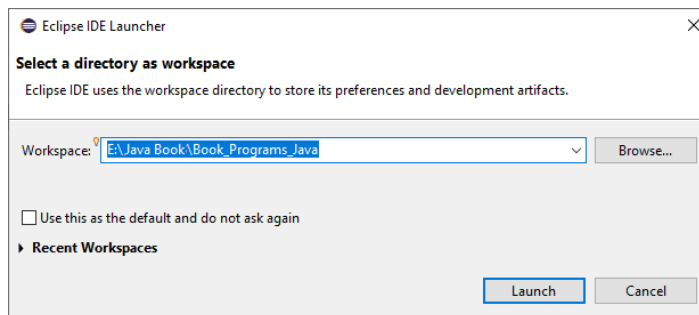


Eclipse Task List and Outline Panes

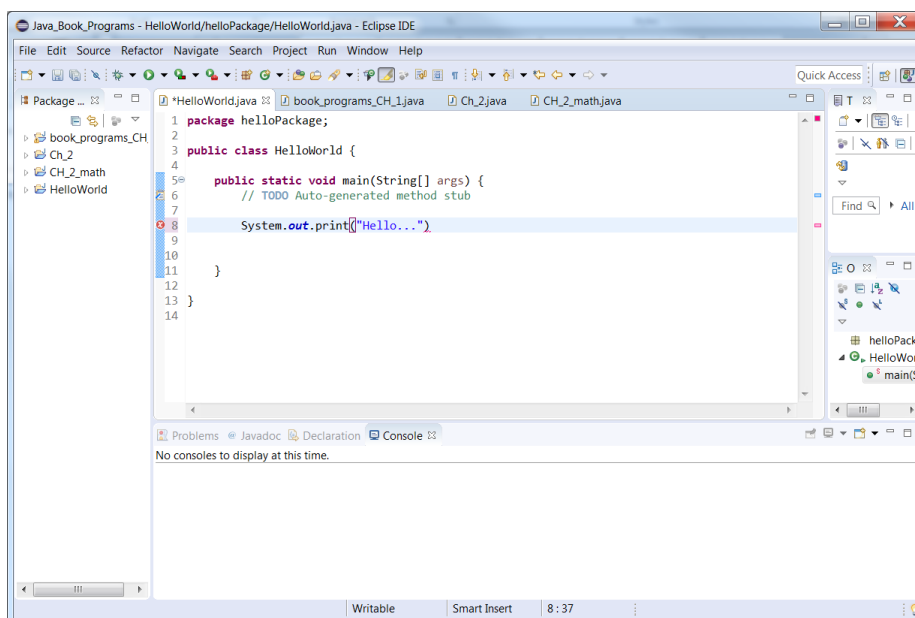
To run a program, click on the green circle with a white triangle inside.



The next time Eclipse is started, the Workspace that was used prior will be the default Workspace to select. When selected, it loads into the IDE along with all of the programs that have been created in that Workspace.



The programs will be listed in the Package Explorer and each file opened will have a tab in the edit window which can be used to select the desired program.



Eclipse Workspace

The screen capture above shows four programs that have been created in the same Workspace. Note the four tabs at the top of the edit window for the programs, and the list of programs in the Package Explorer on the left.

## Exiting Eclipse

To leave Eclipse, just close the program after saving any changes.

To save changes, choose "File" on the menu bar, and "Save" from the drop-down menu or use Control-S.

Workspace information will be saved as the program shuts down.

The link to Eclipse.org (The Eclipse Foundation) and a direct link to tutorials are copied here. These links and additional Java links are provided in Appendix D.

## Eclipse.org

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

## Eclipse User Guide

<https://help.eclipse.org/2019-09/index.jsp>

# Chapter 3

## Programming in Java

### Comments

Single line comments in Java begin with two forward slashes, and are ignored by the compiler. The Eclipse IDE used in this text will color code comments in green font as the default. For multiline comments, a forward slash with an asterisk “/\*” begins the paragraph and an asterisk forward slash “\*/” ends the paragraph. For the Javadoc documentation generator, which creates HTML documents from Java source code, the opening paragraph indicator is a forward slash and two asterisks “/\*\*” and it ends the same as the multiline comment.

### Commenting Code

```
// a single line comment in Java
```

```
/* a multiline  
comment in Java  
*/
```

```
/** A Javadoc comment for the document generator  
*/
```

## Displaying Output

The *print* function used to produce output in Java, requires “*System.out*” to precede it to utilize the Java Utility Class *System*, *out* is an object of the *System* class, and *print* is the Utility method name which sends strings to the console.

The argument passed to the *print* function contains the item or items to display along with any format specifiers. Double quotes are used with string arguments and depending upon the data type of other arguments, either a plus sign or comma is used. Adding a line feed may change the *print* method used (explained later) as does a format specifier as shown in the examples that follow.

### Ex. 3.1 – Displaying Output in Java

```
System.out.print("This is displayed.");           // displays This is displayed.
double num = 123.45;                             // assigns 123.45 to num
System.out.print("Num is " + num);                // displays Num is 123.45
```

The *format* specifier and the *printf* method are used to format output, i.e. *print formatted*. Two arguments are passed to the function: the numeric value or string to be formatted and the format specification which is in quotes and begins with “%”. Format specifiers include “f” (float), “d” (integer), and “s” (string). To set the precision for the output, an integer is placed after the decimal in the specifier.

### Ex. 3.2 – Formatted Output

```
double num = 123/4;                               // Integer divided by integer
System.out.printf("Num is %.3f", num);            // displays Num is 30.000

double num = 123/4.0;                             // Integer divided by double
System.out.printf("Num is %.3f", num);            // displays Num is 30.750
```

When a variable is the only argument, the *format* specifier is still within quotes.

```
System.out.printf("%.2f", num);                   // displays 30.75
```

To provide an amount of spacing to use for an item in the output, the number of spaces is added before the decimal or before the designator if no decimal is used. In this example, “%6d” allocates 6 spaces for the integer variable num.

```
int num = 123;
System.out.printf("Num is %6d", num);    // displays Num is   123
```

When more than one variable is included in the output, the specifiers are included in the string element in the order they are to appear in the output, and the actual variables are included afterward as shown here.

```
System.out.printf("Var1 %4.2f Var2 %4.2f", firstVar, secondVar);
```

Note that the use of *printf* precludes the use of *println*. To display a line feed on its own, *println* with no arguments can be used as shown below, or an escape sequence can be inserted as shown in Ex. 3.4 below.

#### Ex. 3.3 – Line Feed

```
System.out.println("A line feed after");    // adds a line feed after
System.out.println();                      // output just a line feed
```

#### Escape Sequences

Java’s escape sequences include: new line “\n”, tab “\t”, print a double quote “\”, and to print a back slash “\\” two are used. The sequence is surrounded by quotes. When *println* can’t be used because *printf* is being used, a line feed can be inserted as “\n” anywhere a line feed is needed.

#### Ex. 3.4 – Escape Sequences

```
System.out.println("Line feed \n mid-sentence.");    // line feed \n
System.out.println(variable1 + "\n" + variable2.);    // line feed \n
System.out.println("A tab \t mid-sentence.");        // tab \t
System.out.println("\"quotes around.\" ");          // quotes \"
System.out.println("Backslash" + "\\");             // backslash \\
```

```

Line feed
 mid-sentence.
Line feed
mid-sentence.
A tab   mid-sentence.
 "quotes around."
Backslash\

```

Ex. 3.4 Escape Sequence Output

## Ex. 3.4A – Formatted Output revisited

```

String item = "Shrubs";
double cost = 24.69;
int qty = 5;
double subtotal = 123.45;
double tax = 7.41;
double total = 130.86;
System.out.printf("Product %10s \nUnit price: $%.2f \n", item, cost);
System.out.printf("Qty: %13d \nSub Total: $%.2f \n", qty, subtotal);
System.out.printf("Sales Tax:   $%.2f \nTotal:       $%.2f", tax, total);

```

As shown in the lines above, string titles for the values are mixed with the format specifiers for the values, and then after the closing quotes, the variables are listed. The escape sequence “\n” is used for line feeds where needed. The output is shown below.

```

Product      Shrubs
Unit price: $24.69
Qty:                5
Sub Total: $123.45
Sales Tax:   $7.41
Total:       $130.86

```

Ex. 3.4A Program Output

## Variables

In Java, variables are declared by data type, a single equal sign is the assignment operator, and the variable being assigned the value is on the left side of the operator.

*variable = expression or value;*

```
int userAge = 29;           // userAge is assigned 29
```



The variable naming convention most used in Java is uppercasing. A single word variable is all lower case, and a two word variable has the first word in lower case and the first letter of the second word is uppercase. This aligns with W3C (World Wide Web Consortium) as well as other guides and standards for the language. Appendix E contains a short list of Java programming standards.

Java is case-sensitive, and variable names cannot be any of the key words (which will be highlighted by the IDE) and cannot contain spaces. The first character must be a letter or underscore and then letters, digits, or underscores can be used. Software engineering principles and most standards dictate that descriptive variable names be used to add clarity to the code.

The primitive (simple) data types in Java are not objects, and include byte, short, int, long, char, float, double, and Boolean. A wrapper class is used for these.

## Keyboard Input

To obtain keyboard input, Java uses a scanner from the scanner class which requires importing the class `java.util.Scanner`. Import statements are entered between the package and the class in the program as shown in Ex. 3.5 below.

```
import java.util.Scanner;
```

A Scanner object is declared as shown below. The line of code declares a Scanner named "in", assigns "in" a Scanner using "new", and "System.in" is used for obtaining input from the standard system input source (the keyboard).

```
Scanner in = new Scanner(System.in);
```

A Scanner can use various methods available to obtain input.

```
System.out.print("Enter a number: ");  
int myNum = in.nextInt();           // reads and assigns the integer  
System.out.print("The number entered was " + myNum);
```

After the prompt to enter a number in the lines above, the program waits for the Enter key to be pressed and then uses `nextInt()` to read the integer and assigns the input to `myNum`. Note that `nextInt()` is looking for an integer and will fail if an integer was not entered. Methods covered in Chapter 4 will resolve this issue.

The program below combines the statements covered and adds some output. Note that the location of the import statement is between the package and the class. The input for the example below is entered in the Console area where the output appears. A mouse click in the console window gives it the input focus.

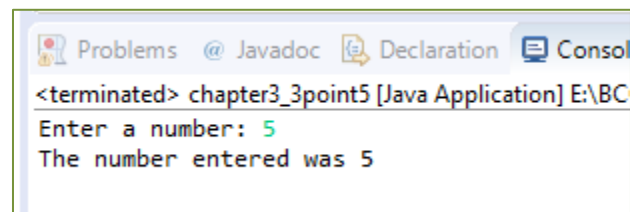
### Ex. 3.5 – Getting Keyboard Input

```

1 package chapter3_5;
2
3 import java.util.Scanner;
4
5 public class chapter3_3point5 {
6
7     public static void main(String[] args) {
8         // TODO Auto-generated method stub
9
10        Scanner in = new Scanner(System.in);
11
12        System.out.print("Enter a number: ");
13        int myNum = in.nextInt();
14        System.out.println("The number entered was " + myNum);
15
16    }
17
18 }

```

Ex. 3.5 IDE Edit Window



```

<terminated> chapter3_3point5 [Java Application] E:\BC
Enter a number: 5
The number entered was 5

```

Ex. 3.5 IDE Output Console

The following methods are used to obtain input for various data types.

```

int myNum = in.nextInt();           // reads an integer
double myDouble = in.nextDouble(); // reads a double
string word = in.next();            // reads up to white space
string line = in.nextLine();        // reads up to a line feed

```

Another way to handle user input of numbers is to read the input as a string and then use *Integer.parseInt()* to convert the data type from a string to an integer.

```
String numString = "32";           // assign "32" as string
int intNum = Integer.parseInt(numString); // covert to integer and store
intNum = intNum + 100;           // used in an equation
System.out.println("int num is :" + intNum); // displays int num is: 132
```

To covert a string to a double, `Double.parseDouble()` can be used.

```
numString = "123.45";           // assign "123.45" as a string
double dblNum = Double.parseDouble(numString); // convert as double
dblNum = dblNum + 100;         // use in an equation
System.out.println("dbl num is :" + dblNum); // displays dbl num is: 223.45
```

The code below fails and throws a `NumberFormatException` (shown below) and does not complete because the parsing attempt fails (exceptions are covered in a later chapter). Note the text in the display below.

```
String badString = "bad";       // assign characters to badString
int badNum = Integer.parseInt(badString); // parsing attempt fails
badNum = badNum + 100;
System.out.println("BAD num is :" + badNum);
```

```
int num is :132
dbl num is :223.45
Exception in thread "main" java.lang.NumberFormatException: For input string: "bad"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    at CH_2Package.Ch_2.main(Ch_2.java:30)
```

### NumberFormatException

## Mathematical Expressions and Operators

Java operators typically align with other languages. Addition uses (+) and subtraction (-), multiplication uses (\*), division is (/), and the modulus operator is (%). One difference is that exponentiation uses the `Math.pow` method from the Math library.

### Exponentiation Example

```
result = Math.pow(2, 3);           // result is assigned 8, (23)
result2 = Math.pow(result, 2);    // result2 is assigned 64, (82)
```

Division results are different for different data types and data type combinations. If one of the values is a floating-point number, the result is a floating-point number. If both numbers are integers as in the first example below, the result is truncated to an integer. The decimal portion is discarded.

```
System.out.println("10 / 3 is: " + 10 / 3);    // displays 10 / 3 is: 3
System.out.println("10 / 3.0 is: " + 10 / 3.0); // displays 10 / 3.0 is: 3.333...
System.out.println("2.5 / 5 is: " + 2.5 / 5);  // displays 2.5 / 5 is: 0.5
```

However, an integer divided by an integer results in a double if the variable it is assigned to is declared as a double.

### Division Examples

```
double result = 10 / 5;           // result is assigned 2.0
double result = 10 / 5.0;        // result is assigned 2.0
double result = 2.5 / 5;         // result is assigned 0.5
```

For rounding numbers, Java has a `Math.round()` method that will round numbers to an integer value even if assigned to a double. To assign the result to an integer, the value must be cast to an integer shown below as (*int*).

```
double result = Math.round(9.4);    // result is assigned 9.0
int result = (int)Math.round(9.6);  // result is assigned 10
```

**Precedence** in Java is (PEMDAS) parenthetical expressions first, followed by exponentiation, then multiplication, division, modulo division, and lastly addition and subtraction. Operators with the same precedence are handled left to right, and precedence can be forced using parenthesis.

**Mixed-type** expressions are promoted to the higher data type in use. In an expression with an integer and float, the integer is temporarily converted to a float, and the expression is promoted with a float as the result of the operation.

The same rule applies to an expression with an int and double. The expression is promoted to double.

## Math Methods

In addition to the *pow()* method, the `java.lang.Math` class contains methods for performing mathematical operations including: *abs(x)*, *acos(x)*, *asin(x)*, *atan(x)*, *cos(x)*, *hypot(x)*, *log(x)*, *sin(x)*, *sqrt(x)*, and *tan(x)* among many others. Each requires placing “Math.” in front of the method. The class also defines a value for pi using `Math.PI`, and conversions for degrees to radians, *Math.toRadians(x)*, and radians to degrees, *Math.toDegrees(x)*. Some examples follow.

```
double var = Math.sin(1);
System.out.printf("Sin(1) in radians: %2.5f", var);
double radius = 2.0;
System.out.printf("\nArea is: %.5f", Math.PI * (Math.pow(r,2)));
```

The output for these lines is:

```
Sin(1) in radians: 0.84147
Area is: 12.56637
```

## Random Numbers

Java includes random number generation within the `Math` class that returns a positive double from 0.0 to 1.0 inclusive.

### Ex. 3.6 – Math Methods – Random Numbers

```
double r = Math.random();           // r is assigned a random number
System.out.println("r is " + r);    // displays r is 0.425314497434798
double ran;
for(int i = 0; i < 3; i++) {
    ran = Math.random();           // ran is assigned a random number
    System.out.print(ran + " ");
}
```

The random number can be manipulated to handle various requirements. A situation needing a random number between 1 and 100 inclusive requires eliminating zero, and adjusting the random number. In Ex. 3.7, the random number is multiplied by 10 and then it is cast to an integer and 1 is added to eliminate zero. In the second example, the number is multiplied by 100 with 1 added to produce a random number between 1 and 100.

#### Ex. 3.7 – Random Number Ranges

```
int num = (int) (Math.random() * 10) + 1;    // number between 1 and 10
System.out.println("num is " + num);        // displays num is 9

int randInt = (int) Math.random() * 100 + 1; // number between 1 and 100
System.out.print("randInt is " + randInt);   // displays randInt is 62
```

The following generates a random number between some minimum and maximum.

```
int rn = (int) Math.random() * ((max - min) + 1) + min);
```

#### Constants

Constants in Java are declared using the *final* key word and all uppercase letters with underscores between words.

```
final double EARTH_RADIUS = 3959.0;
double earth_circumference = 2 * 3.1415 * EARTH_RADIUS;
```

#### Global Variables

The use of global variables is frowned upon in all languages including Java. The language doesn't explicitly have them since every static variable must belong to a class. However, once a variable is in a class, it can be accessed across all class instances.

# Chapter 4

## Decisions, Logic, Loops, and Methods

### If, else, and else if

The Java syntax for the **IF** statement uses parenthesis to surround the conditional statement, and braces to enclose the statements executed when the condition is true. Many Java developers and IDEs place the opening brace on the line with the condition which aligns with most Java standards and is followed in this text.

### Condition Examples

```
if (condition) {  
    statement1;  
    statement2;  
}
```

The **ELSE** condition has no conditional statement.

```
if (condition) {  
    statement(s);  
}  
else {  
    statement(s);  
}
```

For an ELSE IF condition, the format follows the IF format.

```
if (condition_1) {
    statement(s);
}
else if (condition_2) {
    statement(s);
}
else if (condition_3) {
    statement(s);
}
else {
    statement(s);
}
```

**Ex. 4.1** – Conditional Example with  $x = -1$

```
if(x > 1) {
    System.out.println("x is positive.");
}
else if (x < 1) {
    System.out.println("x is negative.");
}
else {
    System.out.println("x is zero.");
}
```

The output of this code would be: x is negative.

## Validating Input

In the last chapter, *nextInt()* and other input methods were introduced with a caution that they will fail if the value is not the expected data type. To ensure



that it is, there is a method *hasNext()* that looks ahead first before the program tries to read and store the value.

Since *nextInt()* is attempting to read an integer, the way to ensure that an integer has been entered is to look-ahead into the input to see if an integer is there to read. The method *hasNextInt()* provides this ability and returns a Boolean based on the next input. In this example, the Scanner was declared as “in”.

```

if(in.hasNextInt()) {                // returns true or false
    int myNum = in.nextInt();        // read the integer
}

```

To test for a double before assigning it to a variable, use *hasNextDouble()*, the *hasNext()* method checks for any item, and *hasNextLine()* tests for a line.

## Boolean Logic and Relational Operators

Boolean Logic and relational operators in Java are similar to other languages, and resolve to either True or False. The operators function as follows:

- > greater than
- < less than
- >= greater than equal to
- <= less that equal to
- = = two equal signs without a space for equivalence
- != not equivalent

Strings cannot be compared using equivalence operators. The member function *string.equals()* is used. Chapter 7 covers strings and their methods.

The logical operators are “&&” for AND, “||” for OR, and “!” for NOT, and the IDE will color code these for clarity. Short circuit evaluation is also used as in other languages; meaning in a logical *and* condition, if the left expression is false, the right expression is not evaluated. In a logical *or* condition, if the left expression is true, the right expression is not evaluated.

### Logic Operator Example

```
value > 0 && value < 20           // logical AND
value < 0 || value > 100        // logical OR
```

Boolean variables are also available in Java as the *boolean* data type which operates as true or false.

```
boolean boolValue = true;        // declares a boolean
```

### Repetition Structures (Loops)

Repetition structures follow the brace and indentation rules associated with conditions. A condition for the loop is enclosed in parenthesis and braces form the block of code executed when the condition is true. Indentation of the statements adds clarity. A **WHILE** loop example follows.

```
while (condition) {
    statement1;
    statement2;
}
```

The Java **FOR** loop follows the standard practice with the initialization, condition, and update on one line.

```
for (int var = 0; var < someValue; var++) {
    statement(s);
}
```

Java SE 5 introduced the *enhanced for loop* (aka for-each loop) which accesses each element and places a copy of the value in a temporary variable for use in the loop. Note that it is a copy and any changes would not affect the actual variable.

```
int [] values = {5, 10, 15, 20, 25};

for (int x : values) {           // each item in values is copied into x
    System.out.print(x);
}
```

Java also provides a **DO WHILE** loop.

```
do {
    statement(s);
}
while (a condition is true);
```

## Methods

Methods (functions) in Java follow the precepts in most other languages. The return data type is included in the method header, and parameters require data types and are received in the order in which they are passed.

The main section of the program in the IDE is itself a method, and is the entry point for the program. This is where execution begins. The header for the main method contains modifiers and definitions as follows:

```
public static void main(String[ ] args)
```

`public` – an access modifier that defines that this method is accessible by any class.

`static` – a key word that defines the method as class related and not instance related. It is not associated with an object.

`void` – does not return a value.

`main` – the name of the method. Main is searched by the JVM as a starting point for an application written in Java.

`String [ ] args` – the parameter to the main method.

The following is a simple method that computes the average of three numbers. The method is `public`, it is `static` (not associated with an instance of an object), it returns a `double`, and it receives three `doubles` as parameters.

```
public static double average (double x, double y, double z) {
    double value = (x + y + z)/3.0;
    return (value);
}
```

A complete program that uses a Scanner to obtain three values from the user, and passes them to the method is shown in Ex. 4.2 below. Recall that creating the program requires creating a project, class, and package. The method is inside the class but outside of the main method. Line numbers are included for explanation.

**Ex. 4.2** – A Simple Method called from main

```

1. public class Average
2. {      // method that computes average
3.     public static double average (double x, double y, double z) {
4.         double value = (x + y + z)/3.0;
5.         return(value);
6.     }
7.
8.     public static void main (String [] args) {
9.         Scanner in = new Scanner(System.in);
10.
11.         System.out.print("Enter three numbers");
12.         double a = in.nextDouble();
13.         double b = in.nextDouble();
14.         double c = in.nextDouble();
15.
16.         double avg = average(a, b, c);      // method call
17.
18.         System.out.print("The average is: " + avg);
19.         in.close();
20.     }
21. }
```

In the program above, line 8 begins the main method where execution begins and line 9 declares a scanner for use in obtaining user input. Lines 11 through 14 prompt for user input and store the values in three variables (a, b, and c). The call to the method on line 16 passes the values of the three arguments (a, b, and c) and assigns the return value of the method to *avg*. The method on line 3 receives the parameters as *x*, *y*, and *z*, computes the average and returns the computed value on line 5. Indentation and braces highlight that the method is inside the project's class, and is separate from main.

```

Enter three numbers: 3
6
10
The average is: 6.333333333333333

```

Ex. 4.2 Output

The method naming convention in Java is the same as the variable naming convention. The first word is all lower case and the first letter of the second word is uppercase. Some languages prefer underscores in function/method names to differentiate them easily from variables. This has not carried over to Java.

Header examples for various methods

```

public static int mySum (int x, int y)           // returns an integer
public static String myString (String s1, String s2) // returns a String
public static boolean myString (int x, int y)    // returns True or False
public static void myOutput (double y)         // void, returns nothing

```

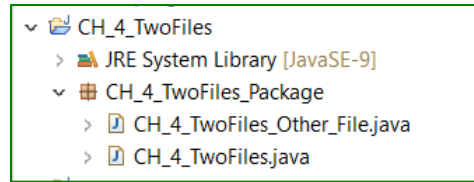
### Methods Cannot Return or Change Multiple Values

Java does not allow pass-by-reference and cannot return multiple values from methods. Methods are used to break a complex problem into smaller, simpler problems and should implement a limited number of operations and return one value or no value at all. Java is an Object Oriented Language (most are) and Objects tend to contain multiple methods, each of which has a single task.

Given that much of Java programming involves graphical user interfaces, components can be used to obtain multiple values from users. These will be covered in later chapters on GUIs. In addition, ArrayLists also covered later, can be changed by methods.

### Methods in Other Files

When a method is in another class file, and that file is part of the project package, the method call includes the class name containing the method (see Appendix B). The following package contains two files with a method call from the first to the second. The package explorer below shows the two files.



Ex. 4.3 Package Explorer

As shown below, the method call from main to `getSquaredVal()` includes the second file's class name followed by the dot operator and name of the method.

Ex. 4.3 – Method in a Second File called from Main

```

package CH_4_TwoFiles_Package;
public class CH_4_TwoFiles {
    public static void main(String[] args) {
        double num = 3;
        double valSquared = CH_4_TwoFiles_Other_File.getSquaredVal(num);
        System.out.println("The squared value is " + valSquared);
    }
}

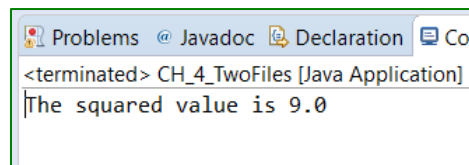
```

// the code located in the other file

```

package CH_4_TwoFiles_Package;
public class CH_4_TwoFiles_Other_File {
    public static double getSquaredVal(double val) {
        double squaredVal = val * val;
        return squaredVal;
    }
}

```



Ex. 4.3 Output

If the method were in another package, the package and the class would be imported.

# Chapter 5

## Interface Design & Development

Graphical User Interfaces are event driven by user input. That is, the user determines the sequence of many of the events; therefore careful design is required to control access to the events. If the click of a button computes a result that requires user input first, the button should either not be enabled until the user has input the required value, or clicking it must produce an error window that alerts the user and takes them back to the entry control. Situations like this need to be considered in the design phase of the interface which adds an engineering consideration for input validation in the program. The value must be entered by the user before allowing computation, and the value entered must be within the correct range of values for the computation to avoid issues such as division by zero.

Consider a program that computes the circumference of a circle based on an input of radius.

1. The radius must be input prior to computation
2. The radius input by the user must be a number
3. The radius input by the user must be a positive number

The graceful handling of incorrect input values is required for a well-engineered solution. In a non-GUI program, a loop might be used that iterates until a correct value is entered. The same concept holds true for a GUI program, but with the added requirement of employing controls to handle the tasks. This situation will

be explored later. Generating the entry or initial GUI is the first step. This requires generating a window and placing some components (controls) on it.

Java is well suited as a language for creating graphical user interfaces. Many Java packages and libraries contain components that are easy to use. The `javax.swing` package and Abstract Window Toolkit (AWT) provide buttons, frames, labels, panels and more to develop user friendly interfaces.

The AWT was Java's first package for creating Graphical User Interfaces (GUIs). It was available in Java 1.0 in 1996, and uses a peer approach, in that each Java control or component has a corresponding component in the windowing system where it is running. Since some windowing systems have different components, only those that are common were included.

The swing components on the other hand are part of Oracle's Java Foundation Classes which provide a user interface for Java programs. It is much more extensive than AWT and matches the look and feel of various platforms.

Some of the Java swing components include:

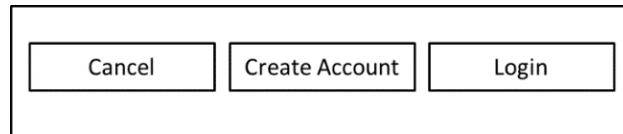
Button	causes an action or event when clicked
Checkbox	On or Off position check boxes
TextArea	display area for text
TextEntry	single line entry component
Frame	rectangular area for graphics
Label	component that displays text
Listbox	user selection list
Menu	list exposed when a menu button is clicked
Panel	rectangular area for frames and components
Radiobutton	select/deselect component

Before selecting components, a preliminary design should be completed either on paper or using an application. This provides a layout for the window and an idea of how it will look and operate prior to writing any code that may need to be changed later. Storyboarding (walking through program operation step-by-step as the user) can also be helpful at this stage. The examples in this chapter build portions of Weather Data Project which is included in Appendix C.



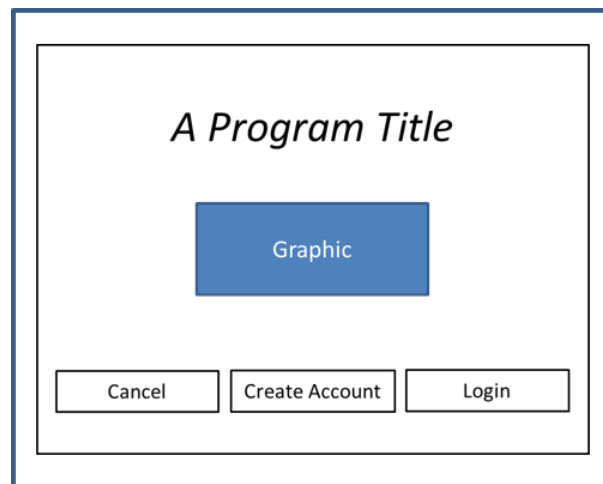
## The Initial Window

The example project requires an initial window with three buttons: Login, Create Account, and Cancel. This is the entry point for the user, and a “first sketch” of the window might simply include a window and the three buttons.



Since the opening or initial window is the first impression of the program for the user, an improved sketch might include a program title and a graphic that reflects the nature of the program.

### Ex. 5.1 – Example Sketch of initial GUI



Initial GUI Sketch

To create an initial GUI with a title, a graphic, and three buttons, a window could be generated (covered later), however Java provides dialog boxes with a variety of features that can be used easily as well. Since the window simply obtains what the user would like to do, a dialog box might be adequate.

In the dialog box statement shown below, the first argument (specification) is the parent window (null in these examples). Note: hovering over the dialog type (*showMessageDialog*) in the IDE displays the specifications which are available for each dialog including *showInputDialog()* which can be used to obtain input.

Error dialog that displays the message, “*Error*”:

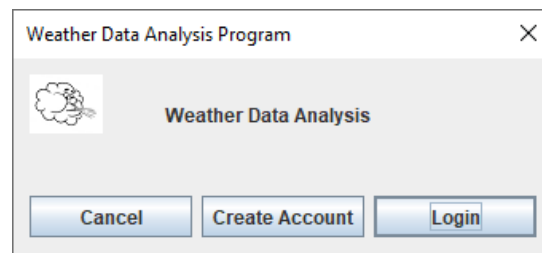
```
JOptionPane.showMessageDialog(null, "Error", "Error",
    JOptionPane.ERROR_MESSAGE);
```

Information dialog with the options yes/no and the message “*Continue?*”.

```
JOptionPane.showConfirmDialog(null, "Would you like to continue?",
    "Continue?", JOptionPane.YES_NO_OPTION);
```

The initial GUI in the example requires a title, a graphic, and three buttons: Create Account, Login, and Cancel. A *showOptionDialog()* solution is shown below.

#### Ex. 5.2 – Initial Window Using a Dialog Box - showOptionDialog



The code to generate this dialog is shown below with line comments for clarity.

```
Object[] options = {"Cancel", "Create Account", "Login"}; // choices for the dialog

int selection = JOptionPane.showOptionDialog(null, // null (parent window)
    "\n Weather Data Analysis\n\n", // text inside dialog
    "Weather Data Analysis Program", // text on the border
    JOptionPane.YES_NO_CANCEL_OPTION, // option type of dialog
    JOptionPane.QUESTION_MESSAGE, // message type
    new ImageIcon("cloudWind.png"), // the image
    options, // the options (button labels)
    options[2]); // initial button focus (login)
```

In the code above, a list of options (the choices) is created to be passed as the seventh argument. The integer “*selection*” is declared to store the return value of the dialog. The arguments to the dialog begin with a “*null*.” This is the entry to the program and no other window is open so there is no parent window. The

next argument is the text inside the dialog followed the text on the border. The next argument is the option type for the dialog followed by the message type. The next argument is the icon chosen to replace the default icon, and then the list of options. This list overrides “Yes”, “No”, and “Cancel”. The final argument is the button that receives the focus as shown by the dotted rectangle inside the “Login” button. Pressing enter will select that button.

To react to the button clicked by the user, the variable “*selection*” that receives the return from the dialog can be used in a conditional statement as shown here.

#### Ex. 5.3 – Dialog Box Button Selection

```

if(selection == 1)                                // Create Account button chosen
{
    System.out.println("Create Account");
    new createAcctWin();                          // create the window
}
else if(selection == 2)                          // Login button chosen
{
    System.out.println("Login");
    new loginWin();                              // create the window
}
else
    System.out.println("Cancel");                // Cancel button chosen

```

In the example, the variable *selection* is used to determine where program control goes next. If the user selects “Create Account”, then a window to obtain the username and password would be created. If the “Login” button is clicked, that window is created.

The next section shows how to create a simple frame as a precursor to a complete interface window.

#### Creating Windows (Frames)

In the example below, a window (frame) is generated using a JFrame which is a container in Java that can hold components.

**Ex. 5.4** – A Simple Window

```
1. // This program creates a simple window.  
2. public static void main (String[ ] args) {  
3.  
4.     JFrame myFrame = new JFrame();  
5.     myFrame.setSize(300,400);  
6.     myFrame. setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
7.     myFrame.setVisible(true);  
8. }
```

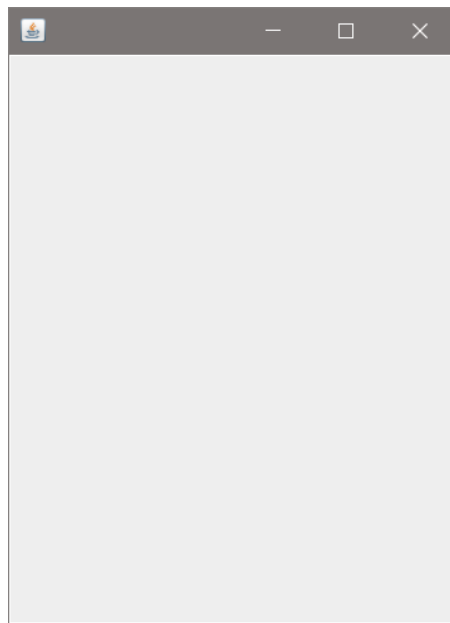
The program above creates the frame shown below.

Line 4 creates an instance of a JFrame and assigns it to myFrame.

Line 5 sets the initial size of the frame.

Line 6 sets the default close operation to *exit* so that if the window is closed the program ends. See the end of chapter 9 for more information and other options.

Line 7 makes the frame visible.



A Simple Window Program

To include components and capabilities, instead of just creating an instance of the JFrame class, inheritance is used and the JFrame class will be *extended*. Inheritance allows the new class to inherit all of the members of the JFrame class

such as *setSize()* and the other methods, and to include additional components and functionality (extending it). A simple example that declares a class for a frame follows.

Ex. 5.5 – A Simple Window class

```

1. public class SimpleWindow extends JFrame {
2.
3.     public SimpleWindow()           // Constructor
4.     {
5.         JFrame myFrame = new JFrame;
6.         myFrame.setSize(300,400);
7.         myFrame.setVisible(true);
8.         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
9.     }
10.

```

The lines above create a class *SimpleWindow* using a *JFrame*. The initial size of the frame is set using *setSize()*, and the close operation ends the program should the user close the window.

To create an instance of this class, main would include a call to the constructor as shown below. An instance of a *SimpleWindow* is created and is assigned to *myWin*.

```

        public static void main (String [ ] args) {

                SimpleWindow myWin = new SimpleWindow();
        }
} // end of class SimpleWindow

```

## Adding Components

Developing the Create Account window and the Login window with text entry components and buttons will require not only creating the window and the components, but positioning them in the window. Positioning them is made easier with the use of a layout manager.

## Generating the Create Account and Login Windows

The Create Account and Login windows will both require prompts and text entry by the user as well as buttons. The layout for the two windows will be similar, but both will be generated as GUIs. To create interfaces, Java includes additional components and layout managers that provide ways of locating and positioning components in windows. Each of the layout managers listed below has benefits and limitations. Others have been introduced in JavaFX which is still maturing.

### Java Layout Managers

- BorderLayout
- BoxLayout
- CardLayout
- FlowLayout
- GridBagLayout
- GridLayout
- GroupLayout
- SpringLayout

The GridBagLayout manager is used in this example to demonstrate the flexibility it provides including the ability to locate components in rows and columns (cells) with constraints and instance variables to tailor positioning. Appendix F demonstrates the use of multiple layout managers.

Some of the instance variables for use with GridBagLayout include:

gridx	positioning in a column
gridy	positioning in a row
gridwidth	specify the number of columns for the grid
gridheight	specify the number of rows for the grid
ipadx, ipady	internal padding for a component
insets	external padding around a component
anchor	positions a component within a cell
weightx, weighty	determines row and column space distribution

The following example uses some of the specifiers listed above. The code below creates a JFrame (window) with a border title, a JPanel with a GridBagLayout to locate components, three JLabels with their output text, two JTextFields that are 10 characters wide to obtain user input, and a JButton to click when the user is finished. The names of the components are in italics for clarity.

**Ex. 5.6** – Create Account GUI

```

1. public class CreateAcct extends JFrame {
2.     JFrame createAcctGUI = new JFrame("Weather Data Program");
3.     JPanel createAcctPanel = new JPanel(new GridBagLayout());
4.     JLabel userNameLabel = new JLabel("Enter a User name: ");
5.     JLabel passWordLabel1 = new JLabel("Create a password...: ");
6.     JLabel passWordLabel2 = new JLabel("upper case letter ...: ");
7.     JTextField textFieldUserName = new JTextField(10);
8.     JTextField textFieldPassword = new JTextField(10);
9.     JButton createAcctButton = new JButton("Create Account");

```

In the continued code below, line 10 begins the constructor which adds and positions components. The initial size of the frame is set on line 11, an instance of constraints for the GridBag is created on line 12 called *con*, and *insets* for spacing are on line 13 and are ordered Top, Left, Bottom, and Right. Lines 15 through 17 set the location for the *UserNameLabel*, and line 18 adds it to the panel.

```

10.     public CreateAcct() { // constructor
11.         createAcctGUI.setSize(500,300); // width then height
12.         GridBagConstraints con = new GridBagConstraints();
13.         con.insets = new Insets(1, 1, 1, 1); // Top, Left, Bottom, Right
14.
15.         con.gridx = 0; // set the column constraint to 0
16.         con.gridy = 1; // set the row constraint to 1
17.         con.anchor = GridBagConstraints.WEST; // Left align
18.         createAcctPanel.add(userNameLabel, con); // add the label
19.
20.         con.gridx = 1; // set the column constraint to 1

```

```

21.         con.anchor = GridBagConstraints.EAST;    // Right align
22.         createAcctPanel.add(textFieldUserName, con); // add entry field
23.         textFieldUserName.setHorizontalAlignment(JTextField.RIGHT);

```

Lines 20 and 21 locate the text entry component. Since the *gridy* specification for this component is also 1, it is not set. In fact once constraints are set, they effect all components after them unless they are changed. Line 22 adds the text field to the panel, and line 23 sets text field alignment to right for user entry.

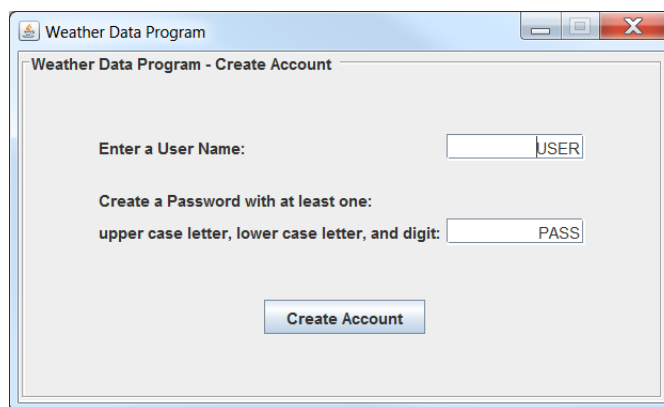
Setting the grid locations of the other components for the window, as well as adding them to the panel would be the same as the lines above except for the actual grid settings. The final lines in the code shown below add the panel to the frame, set resizability to false so that the user cannot stretch the window (which would skew the positions of the components), set the location of the window relative to null since no other window is open, and make the frame visible.

```

createAcctGUI.add(createAcctPanel);    // adds the panel to the GUI
createAcctGUI.setResizable(false);    // precludes resizing
createAcctGUI.setLocationRelativeTo(null); // initial window location - centered
createAcctGUI.setVisible(true);      // make it visible

```

The completed code generates the window below.



Ex. 5.6 Create Account GUI

The password input can be hidden with `passwordField.setEchoChar('*')`; with a `JPasswordField` which allows editing a single line of text while indicating something was typed without showing the original characters.



One addition to the panel that provides some esthetic qualities is a frame border. The Java `BorderFactory` provides beveled, compound, raised borders, and many others. Code for the etched, titled border in Example 5.6 is shown below.

```
createAcctPanel.setBorder(BorderFactory.createTitledBorder(
    BorderFactory.createEtchedBorder(), "Weather Data Program -
    Create Account"));
```

Appendix F includes an example that combines layout managers.

### Action Listeners

Once the user has entered a user name and password, reacting to the button click to create an account requires an event listener object for the event source. In this case an *actionlistener* for the button is used. The listener can be an *ActionListener*, *ButtonListener* or public *Interface ActionListener*. The instructions processed are the *actionPerformed()*. A public interface *ActionListener* example is shown here.

```
public interface ActionListener {

    void actionPerformed(ActionEvent event);

}
```

Event handling classes are in the `java.awt.event` package. The lines below declare a listener class. The program constructs an object of the class and adds it to the button. A *ClickListener* example is shown below.

```
public class ClickListener implements ActionListener {

    void actionPerformed(ActionEvent event) {

        System.out.println("The button was clicked.");

    }

}
```

This code constructs the object and adds it to the button.

```
ActionListener myListener = new ClickListener();
button.addActionListener(myListener);
```

## Subclass ActionListener

In most cases, a `ButtonListener` that is a subclass of the GUI is more appropriate. The Create Account example will use this implementation. The *listener* will get the input from the user and verify the input to create an account. If the password does not meet the criteria, an error should be indicated and the window should remain open to obtain another password.

A listener implementation that is common for components uses an inner class (subclass) implementation. In the example, the code below would be inside the class that implements the Create Account window.

```

1.      class ButtonListener implements ActionListener {
2.
3.          public void actionPerformed(ActionEvent e) {
4.
5.              System.out.println("Button clicked");
6.              String uName = textFieldUserName.getText();
7.              String pWord = textFieldPassword.getText();
8.              // code to validate the input for account creation would go here
9.              createAcctGUI.dispose();
10.         }
11.     };
12.     createAcctButton.addActionListener(new ButtonListener());
13. } // end of constructor
14. }; // end of class

```

In the code above, line 1 declares the class. The reserved word *implements*, declares an interface for the listener. Line 3 is the method that will be executed when the button is clicked. Lines 5 is an output statement for testing, and lines 6 and 7 extract the text entered by the user from the `JTextFields`. Line 8 shows where code or method calls would go to validate the input from the user, and line 9 disposes of the window if the account is validated. Line 11 ends the subclass and line 12 creates an instance of the button listener object. Line 13 is the end of the constructor for the window, and line 14 is the end of the class. Lines 13 and 14 are displayed to highlight the location of the subclass (inside the constructor and class). There may be code below the listener class depending on the design or development.

The following example creates a frame with two buttons, each of which opens another frame with a label. Both listeners are within the *ButtonTest* class.

Ex. 5.7 – Two-button Three-frame Example

```
package buttonTest;

import java.awt.event.*;
import javax.swing.JButton;
import javax.swing.*;

public class ButtonTest {

    public static void main(String[] args) {
        JFrame frame1 = new JFrame();
        frame1.setSize(500, 500);
        frame1.setTitle("Button test");
        JPanel panel = new JPanel();
        JButton button1 = new JButton("First button");
        JButton button2 = new JButton("Second button");
        frame1.add(panel);
        panel.add(button1);
        panel.add(button2);
        frame1.setVisible(true);
        button1.addActionListener(new Action1());
        button2.addActionListener(new Action2());
        frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    } // end of main

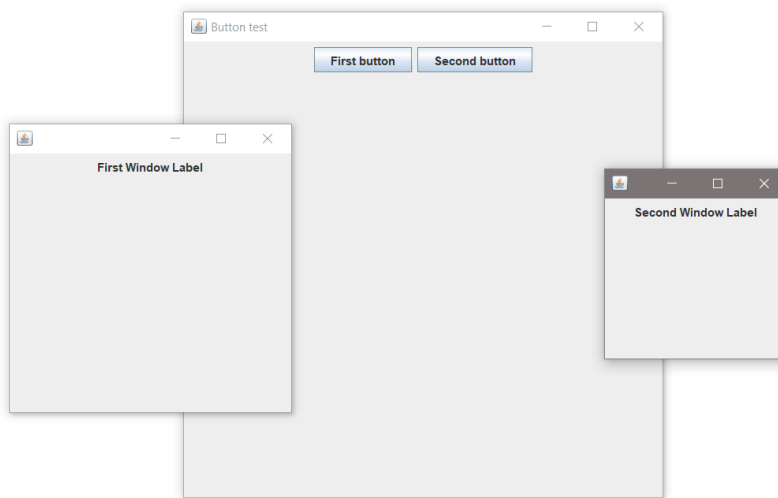
    static class Action1 implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFrame frame2 = new JFrame();           // create another frame
            JPanel panel2 = new JPanel();           // create another panel
            JLabel label2 = new JLabel("First Window Label");
            frame2.setSize(300,300);
            panel2.add(label2);
            frame2.add(panel2);
            frame2.setVisible(true);
        }
    } // end of Action1
}
```

```

static class Action2 implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        JFrame frame3 = new JFrame();           // create another frame
        JPanel panel3 = new JPanel();          // create another panel
        JLabel label3 = new JLabel("Second Window Label");
        frame3.setSize(200,200);
        panel3.add(label3);
        frame3.add(panel3);
        frame3.setVisible(true);
    }
} // end of Action2
}

```

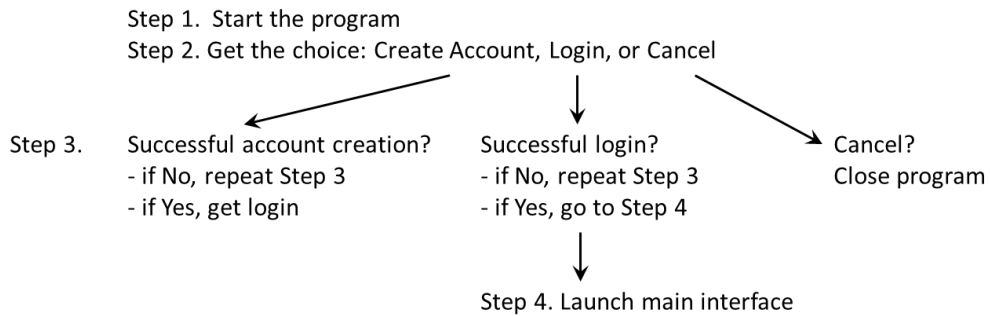
The program in Ex. 5.7 creates a main frame with two buttons that create additional windows which could easily be extended.



### A Word of Caution

Control is transferred to the Create Account window, but main continues. After successful account creation, this window will need to be modified into the Login window, or destroyed and the Login window will need to be created. The main program is not waiting for the window to be modified or destroyed. If the next statement in main creates an instance of a Login window, it will execute while the Create Account window is in view and both windows will be displayed.

Using “*wait()*” is unacceptable due to thread manipulation and a timing loop is inappropriate engineering. It would be easy to implement our way down a rabbit hole and find out that we can’t get back. Thinking through the execution process ahead of development will avoid time consuming errors in implementation.



The solution depends on the design. With a dialog box, the program will wait until the dialog is satisfied. If a separate class is used to generate a window, the previous window could be hidden (set visible to false), or have its’ component disabled. Thinking ahead is an important aspect of design.

### Closing Windows/Programs

The `setDefaultCloseOperation()` method is recommended to end a program when the user closes the window otherwise, the program continues to run.

```
myFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

A close operation can also be written using a window listener to handle other operations before ending the program. This example sets the CONSTANT for the default close operation to do nothing. The window listener stops a running timer and disposes of the frame.

```
myFrame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

myFrame.addWindowListener((WindowListener) new WindowAdapter() {
    @Override
    public void windowClosing(WindowEvent e) {
        myTimer.stop();
        myFrame.dispose();
    }
});
```

The notation, `@Override` used above allows defining specific behaviors for a particular class that override the existing method. It is not required, but is considered a best practice, and lets the compiler know that we are overriding this method.

### Account Creation and Login Requirements

The creation of an account requires validation of the password criteria as well as ensuring that is not already in use. The login operation requires validating the account information. Both of these operations could be accomplished with a database, but for simplicity and explanation the examples will utilize file handling which is covered next.

# Chapter 6

## File Handling

File handling in Java uses the `File` class and two different classes depending upon whether reading from or writing to the file. Reading uses the `Scanner` class much like getting input from the keyboard. Writing uses the `PrintWriter` or `FileWriter` class (see Append an Existing File below).

Reading from a File

```
// create an object of the File class and pass it the name of the file
File inputFile = new File("input.txt");

// create a Scanner object and the file object to the constructor
Scanner in = new Scanner(inputFile);
```

The `Scanner` methods used for keyboard input like `next()`, `nextLine()`, `hasNext()` can also be used to read from the file. Recall that `next()` consumes any leading white-space and reads until it encounters white space. The method `nextLine()` will read including white space until the end of the line.

```
while (in.hasNextLine()) {           // while there are lines to read
    String textVar = in.nextLine(); // read the line into textVar
    System.out.println(textVar);   // display the line
}
```

Closing a file is really closing the stream in Java as shown here. In the chapter on exception handling, try-with-resource will be covered which will close the files automatically.

```
File inputFile = new File ("input.txt");
Scanner in = new Scanner(inputFile);
// code to read from file
in.close();                // close the input stream
```

### Writing to a File

*// create an object of the PrintWriter class and pass it the name of the file.*

```
PrintWriter out = new PrintWriter("output.txt");
```

If the file “output.txt” does not exist, it will be created. If it exists, it will be emptied. The PrintWriter can use the System.out methods like *print()*, *println()*, and *printf()*, and the stream can be closed in the same way as the Scanner.

```
PrintWriter out = new PrintWriter("output.txt");
// code to write to the file
out.close();                // close the output stream
```

### Append an Existing File

When a file is opened by the PrintWriter, any file content is erased. To append to an existing file requires creating an instance of the FileWriter class and then assigning it to the PrintWriter. The first argument passed to the FileWriter constructor is the name of the file in quotes, and the second is the Boolean value true for appending. The FileWriter is then assigned to a PrintWriter as shown below, and the PrintWriter methods can then be used as before for writing.

*/\* create an object of the FileWriter class called fwriter and assign it to a PrintWriter named out. \*/*

```
FileWriter fwriter = new FileWriter("output.txt", true);
PrintWriter out = new PrintWriter(fwriter);
```



## Writing and Reading File Content

### Ex. 6.1 – File Reading and Writing

```
1. // Program creates a file, writes to the file, and reads back the text
2. public static void main(String[] args) {
3.
4.     PrintWriter out = new PrintWriter("data.txt");
5.     out.println("Writing to a file.");
6.     out.close();
7.
8.     File inputFile = new File("data.txt");
9.     Scanner in = new Scanner(inputFile);
10.    String text = in.nextLine();
11.    System.out.print(text);
12.    in.close();
13. }
```

Line 4 creates a `PrintWriter` and assigns it to “data.txt” creating “data.txt”.

Line 5 writes a phrase to the file.

Line 6 closes the output file.

Line 8 creates a `File` object and assigns it to “data.txt”.

Line 9 creates a `Scanner` and assigns it to the file.

Line 10 reads a line from the file and into a string.

Line 11 displays the text that was read.

Line 12 closes the input file.

Lines 8 and 9 from above can be combined into a single statement as shown below.

```
Scanner in = new Scanner(new File ("data.txt"));
```

Example Ex. 6.1 reads the entire line from the file into a string. If reading one word at a time from the file is preferred, the `next()` method would be used which reads until it sees whitespace. A delimiter (data separator) can also be used for

reading and is a Scanner method. As an example, a comma delimited file could be read as shown below. A Scanner named *in2* is declared, and it is assigned the comma delimiter to use. String handling will be covered in a later chapter.

```
Scanner in2 = new Scanner(new File ("dataComma.txt"));
String z = "";           // string declared
in2.useDelimiter(",");   // comma delimiter assigned to Scanner
while(in2.hasNext()) {   // while there are items in the file
    z = in2.next();       // read up to the delimiter
}
```

Example Ex. 6.1 assumed that the file to read (data.txt) existed. If it does not, an exception would be thrown and the program would terminate if the exception is not handled. The IDE will force exception handling in the form of try/catch blocks around areas where an exception could be thrown. This highlights areas in the program requiring error detection and management which is covered in the next section.

## Exceptions

Exception handling is required when a file cannot be created or cannot be opened, or other issues like a data type mismatch. The format for an exception handler in Java is the *try/catch* block. The try block is entered and if a statement raises an exception, the *catch* block (handler) for that exception type is entered, the handler executes, and the program continues. An exception that is not handled will halt execution of the program. There are a variety of exceptions that could be thrown including `NumberFormatException`, `FileNotFoundException`, and `IOException`.

If a statement could throw an exception, the IDE will highlight this and expect handling of the exception. The most straight forward way of handling them is to use a try/catch clause. Catch clauses are exception specific and are a way of handling the errors gracefully and not ending the program.

When a statement in the try clause throws an exception, control is transferred to the catch block matching the exception thrown. No other statements in the try block following the one that threw the exception will execute including closing a

file. Also, if an exception handler for the exception thrown does not exist, the program will terminate.

The following program enhances Ex. 6.1 by adding the try/catch statements and exception handlers.

#### Ex. 6.1A – File Writing and Reading with Exception Handling

```
public static void main(String[] args) {
    try {
        // try block
        PrintWriter out = new PrintWriter("data.txt");
        out.println("Writing to a file.");
    }
    catch (FileNotFoundException e) {
        // catch block
        System.out.print("The output file cannot be opened.");
        e.printStackTrace();
    }

    try {
        // try block
        Scanner in = new Scanner (new File ("data.txt"));
        String text = in.nextLine();
        System.out.print(text);
    }
    catch (FileNotFoundException e) {
        // catch block
        System.out.println("The input file cannot be opened.");
        e.printStackTrace();
    }
}
```

It is customary to use “e” as the exception parameter to receive the exception object, and to print the stack trace during development for additional error information. Below is the stack trace for Ex. 6.1A when the input file is not found.

```
java.io.FileNotFoundException: data.txt (The system cannot find the file specified)
at java.base/java.io.FileInputStream.open0(Native Method)
at java.base/java.io.FileInputStream.open(FileInputStream.java:196)
at java.base/java.io.FileInputStream.<init>(FileInputStream.java:139)
at java.base/java.util.Scanner.<init>(Scanner.java:611)
at Ex_6_1_package.Ex_6_1.main(Ex_6_1.java:13)
```

Error Stack Trace for Ex. 6.1A

To ensure that resources used will be closed if an error occurs, the `PrintWriter` and `Scanner` resources can be declared and instantiated within the `try` clause (after the word “`try`” and prior to the curly brace). This is called a `try-with-resources` statement. The resources will be closed automatically even if an exception is thrown.

Ex. 6.2 – File Reading and Writing using *try-with-resources*

```
public static void main(String[] args) {
    // try with resources statement guarantees the resource will be closed
    try (PrintWriter out = new PrintWriter("data.txt")) {
        out.println("Writing to a file.");
    }
    catch (FileNotFoundException e) {           // catch block
        System.out.print("The output file cannot be opened.");
        e.printStackTrace();
    }

    // try with resources statement guarantees the resource will be closed
    try (Scanner in = new Scanner (new File ("data.txt"))) {
        String text = in.nextLine();
        System.out.print(text);
    }
    catch (FileNotFoundException e) {         // catch block
        System.out.println("The input file cannot be opened.");
        e.printStackTrace();
    }
}
```

### Reading and Writing Numeric Data

The examples so far have been writing text to a file and reading text from a file into a string. Very often numeric data must be handled. Since the `Scanner` is used for reading, `nextInt()` and `nextDouble()` can be used to read integers and doubles, but may run into trouble if what is read is not a numeric value. The use of `hasNextInt()` and `hasNextDouble()` can look ahead to ensure that the expected type is there. To parse Integers and Doubles that have been read as text,

`Integer.parseInt()` and `Double.parseDouble()` will convert the text if possible. Data handling will be covered in more detail in a later chapter.

Example Ex. 6.3 below writes integers to a file with line-feeds, and then a string. The program then reads the integers using `hasNextInt()` which determines if there is an integer to read before reading. It does not read the line feeds or string. (Note: Exception handling is omitted.)

### Ex. 6.3 – Writing Numbers to a File and Then Reading

```

1. // Program that writes numbers and a string to a text file, then reads the
   numbers only for display
2. public static void main(String[] args) {
3.     PrintWriter out = new PrintWriter("data.txt"); // declare a PrintWriter
4.     int x = 4;
5.     while (x < 20) {
6.         out.println(x); // write a set of integers to a file
7.         x = x + 4;
8.     }
9.     out.println("all done"); // write a string to the file
10.    out.close() // close the file
11.
12.    Scanner in = new Scanner(new File("data.txt"));
13.    int num = 0;
14.    while (in.hasNextInt()) { // while there are integers to read
15.        num = in.nextInt(); // read the next integer
16.        System.out.println(num);
17.    }
18. }

```

Lines 14 - 16 in pseudo-code: while there is an integer to read, read it, assign it to "num", and display the value of "num" and a line feed. The String is not read.

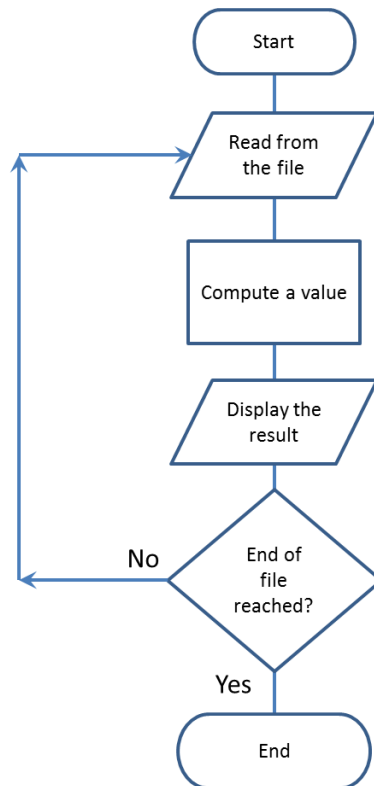
```

4
8
12
16

```

Ex. 6.3 Output Display

The technique used for reading and handling data from a file will be dependent upon the task required. The data can be read one character, item, or line at a time. Loops are typically used for this purpose. The flow chart from Chapter 1 repeated here requires a single computation and display of the result for each item in the file. The loop reading the file stops at the end of the file. It could just as easily end when *hasNextInt()*, *hasNextDouble()*, or *hasNext()* fails depending upon the type being read.



File Reading Flow Chart

## Data Design

For the example project, storing the user names and passwords during account creation, and retrieving them for validation during login requires a data storage design. When designing data files, there are a few considerations including the format, text/binary, delimiters, and any encryption. These items must be well thought out during the design phase due to the effect on development of the current program and potential future expansion (scalability). Many large-scale, data intensive programs require a formal **Data Dictionary** which is a separate

file that contains the data descriptions, format, delimiters (data separators), the ordering of the data, and often, additional information and comments.

### Data Format

Designing the data format is an important task that effects program design and operation, data handling, and the scalability of the data and the program. A data dictionary can provide useful information about file contents and how to extract or parse the data for use in display and analysis. Creating a data dictionary also allows the file to contain only data and flexibility with respect to delimiters.

Data dictionaries are also typically used for databases, and often describe the contents and the relationship between the database elements.

The sample file data dictionary below specifies individual column numbers for the data elements in the data set that follows. Each column (character) may be an individual value or part of a group of characters forming a value.

### Data Dictionary Sample (small portion) from NOAA

```

                                DD/MM/YYYY
                                GENERAL DATA FORMAT
                                ONE HEADER RECORD FOLLOWED BY DATA RECORDS:
                                COLUMN DATA DESCRIPTION
01-05 STATION NUMBER
08-12 RECORDING ENTITY NUMBER
14-25 YEAR-MONTH-DAY-HOUR-MINUTE (GMT)
27-29 DIR = WIND DIRECTION IN COMPASS DEGREES, 990 =
      VARIABLE, REPORTED AS '***' WHEN AIR IS CALM (SPD WILL
      THEN BE 000)
31-37 SPD & GUS = WIND SPEED & GUST IN MILES PER HOUR
39-41 CLG = CLOUD CEILING--LOWEST OPAQUE LAYER WITH 5/8
      OR GREATER COVERAGE, IN HUNDREDS OF FEET, 722 =
      UNLIMITED
43-45 SKC = SKY COVER -- CLR-CLEAR, SCT-SCATTERED-1/8 TO
      4/8, BKN-BROKEN-5/8 TO 7/8, OVC-OVERCAST, OBS-
      OBSCURED, POB-PARTIAL OBSCURATION
47-47 L = LOW CLOUD TYPE, SEE BELOW

```

## Data File Sample

```

USAF  WBAN YR--MODAHRMN DIR SPD GUS CLG SKC L M H  VSB MW MW MW
724074 93780 200601010054 990 6 *** 4 OVC * * * 5.0 * * * * * * * 10 * * * *
724074 93780 200601010154 990 3 *** 2 OVC * * * 3.0 * * * * * * * 10 * * * *
724074 93780 200601010254 300 5 *** 4 OVC * * * 4.0 * * * * * * * 10 * * * *
724074 93780 200601010354 *** 0 *** 4 OVC * * * 5.0 * * * * * * * 10 * * * *
724074 93780 200601010406 *** 0 *** 80 OVC * * * 5.0 * * * * * * * 10 * * * *
724074 93780 200601010454 *** 0 *** 85 OVC * * * 5.0 * * * * * * * 10 * * * *
724074 93780 200601010459 *** * * * * * * * * * * * * * * * * * * * * * * *
724074 93780 200601010554 *** 0 *** 48 OVC * * * 4.0 * * * * * * * 10 * * * *
724074 93780 200601010654 250 5 *** 80 OVC * * * 4.0 * * * * * * * 10 * * * *
724074 93780 200601010754 240 5 *** 75 OVC * * * 4.0 * * * * * * * 10 * * * *
724074 93780 200601010845 220 3 *** 722 SCT * * * 2.0 * * * * * * * 10 * * * *
724074 93780 200601010854 230 5 *** 60 BKN * * * 2.0 * * * * * * * 10 * * * *
724074 93780 200601010908 240 6 *** 60 OVC * * * 4.0 * * * * * * * 10 * * * *
724074 93780 200601010954 *** 0 *** 50 OVC * * * 5.0 * * * * * * * 10 * * * *
724074 93780 200601011054 270 3 *** 55 OVC * * * 8.0 * * * * * * * * * * * *

```

Utilizing data such as this is made possible by examining the data dictionary. Although the data file contains a cryptic header as the first row, an explanation for most of the columns is needed. As an example, the three columns on the right of the sample data are described in the file header row as MW.

The data dictionary indicates that columns 14 thru 25 provide the data and time of the data reading. The excerpt is shown here.

14-25 YEAR-MONTH-DAY-HOUR-MINUTE (GMT)

The last row in the sample file data above for those columns is shown below.

200601011054 (2006 01 01 1054)

The data dictionary makes it clear that this group can be parsed as:

Year 2006, January, 01, and 10:54 Greenwich Mean Time.

## Create Account Operation

For the project user name and password data, there are many possible solutions for file storage and access. Both items could be written to a text file on one line with a space or tab between them (columnar data), two lines could be used, or



even two files adding a security feature of not having them located together. A binary format could be used instead of text, and encryption could be used as well. Regardless of the storage/retrieval algorithm, the operations are the same with some design choices.

During the Create Account operation, the design could require that the user name and password be unique, or that just the password be unique. In the Login operation, both must be validated as a pair and compared with existing accounts. Considering how the data will be used during login provides insight into how it should be handled and stored during the creation operation. Comparing the processes that will utilize the data shows the similarities and the differences for design consideration. In other words, when considering the create account operations it is a good idea to keep in mind how the login operation will work.

<u>Create Account Operation</u>	<u>Login Operation</u>
1. Get user name	Get user name
2. Get password	Get password
3. Verify as <b>unique</b>	Verify as <b>existing pair</b>
4. Reject errors, go to Step 1	Reject errors, go to Step 1

The only difference in operation is the verification process on Step 3. Both of these require string handling which is covered in the next chapter.

### Selecting a File - JFileChooser

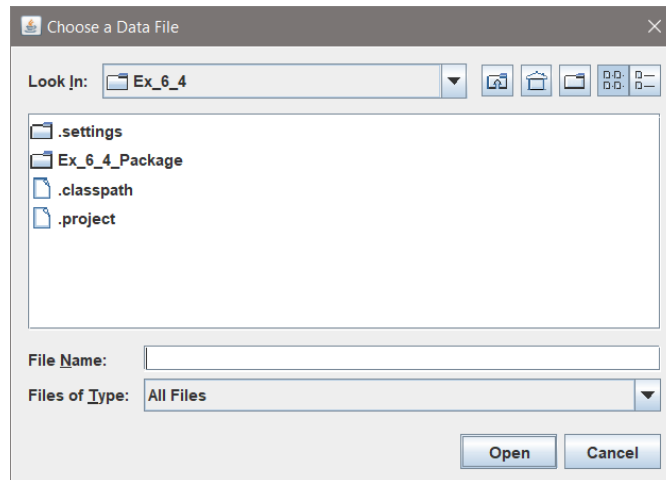
For user file selection, Java provides a graphical interface called the JFileChooser. The following example declares a JFileChooser (line 5 below). It sets the default to the current directory using *setCurrentDirectory()* as shown on line 7 after setting the File *workingDir* on line 3 to the current directory.

Notice that *chooser.showOpenDialog()* method on line 9 returns an integer. This is null if the file was not selected which can be used by the calling method, or returns a value corresponding to a constant as shown on line 12. Line 12 handles the situation when a file is not selected in the try block and null is returned. The *else* handles the case when a file is selected, and line 16 gets the file name and path. Line 17 declares a scanner for reading the file, and line 18 declares a *StringBuilder* called *info* to store the file contents. A line feed is added on line 21 since *nextLine()* removes it, and the *StringBuilder* is converted to a *String* on line

24 and returned to the calling method. The catch block of the try receives the exception and returns null to coincide with line 13 when no file is selected.

**Ex. 6.4** – File Selection Using JFileChooser and Reading using StringBuilder

```
1. public static String fileOpenAndRead() {
2.
3.     File workingDir = new File(System.getProperty("user.dir"));
4.
5.     JFileChooser chooser = new JFileChooser();
6.     chooser.setDialogTitle("Choose a Data File");
7.     chooser.setCurrentDirectory(workingDir);
8.
9.     int status = chooser.showOpenDialog(null);
10.
11.    try {
12.        if(status != JFileChooser.APPROVE_OPTION) {
13.            return null;
14.        }
15.        else {
16.            File file = chooser.getSelectedFile();
17.            Scanner scan = new Scanner(file);
18.            StringBuilder info = new StringBuilder();
19.            while (scan.hasNext()) {
20.                info.append(scan.nextLine());
21.                info.append("\n");
22.            }
23.            scan.close();
24.            return info.toString();
25.        }
26.    } catch (Exception e) {
27.        return null;
28.    }
29. }
```



JFileChooser Window

### Filtering Selectable Files

To filter on a file type (extension), a filter can be declared and assigned to the `JFileChooser`. The following code creates a file chooser and then creates a filter on line 2 which includes only `.jpg` file types. The filter is assigned to the `JFileChooser` on line 4 using `setFilter()`.

#### Ex. 6.5 – File Type Selection using `FileNameExtensionFilter`

```

1. JFileChooser chooser = new JFileChooser();
2. FileNameExtensionFilter f = new FileNameExtensionFilter("JPG", "jpg");
3.
4. chooser.setFilter(f);           // assign the filter
5.
6. int returnVal = chooser.showOpenDialog(parent);
7. if(returnVal == JFileChooser.APPROVE_OPTION {
8.     System.out.println("The file chosen is " +
9.         chooser.getSelectedFile().getName());
10. }
11. else {
12.     System.out.println("No File Was Selected");

```

## Save As using JFileChooser

To save a file, declare a JFileChooser and use the showSaveDialog().

Ex. 6.6 – File “Save AS” using showSaveDialog()

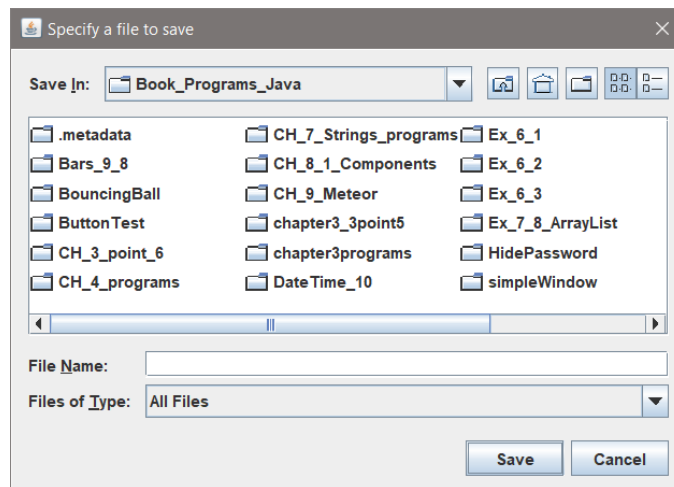
```

JFileChooser fileChooser = new JFileChooser();
fileChooser.setDialogTitle("Specify a file to save");

int userSelection = fileChooser.showSaveDialog(parentFrame);

if (userSelection == JFileChooser.APPROVE_OPTION) {
    File fileToSave = fileChooser.getSelectedFile();
    System.out.println("Save as file: " + fileToSave.getAbsolutePath());
}

```



JFileChooser “Save As” Window

# Chapter 7

## Strings and ArrayLists

Java provides many ways to examine and manipulate strings. The individual characters of a string can be accessed using indexes, and there are substring, concatenation, and trim operations. The character class also provides various tests for digits and case sensitivity which will be introduced later.

Strings are sequential, and the characters can be accessed by index using the *charAt()* method. Index numbering begins at zero, and ends at n-1.

### Ex. 7.1 – Indexing Strings

```
String myString = "something";  
System.out.println(myString.charAt(4));    // displays t
```

The index can also be used to obtain a copy of a single character from a string.

### Ex. 7.2 – Copying a Character from a String

```
String myString = "copy";  
char ch = myString.charAt(2);             // assigns "p" to ch  
System.out.println(ch)                    // displays p
```

If an out of range index is used, a *StringIndexOutOfBoundsException* is thrown.

The '+' operator is used to concatenate strings.

#### Ex. 7.3 – Concatenating Strings

```
String cityString = "New";
cityString = cityString + " York";
System.out.println(cityString);           // displays New York

cityString += " City";                    // "+=" works as well
System.out.println(cityString);          // displays New York City
```

The *length()* method returns the length of a string and can be used as a loop termination condition to avoid an out of bounds error.

#### Ex. 7.4 – The *length()* method with Strings

```
String myString = "first name";
int strLen = myString.length();          // strLen will be assigned 9
int index = 0;
while (index < strLen) {
    System.out.println(myString.charAt(index) + " " + index);
    index += 1;
}
```

```
s 0
o 1
m 2
e 3
t 4
h 5
i 6
n 7
g 8
```

Ex. 7.4 Output

Character testing is handled using the Character class' methods *isDigit()*, *isUpperCase()*, *isLowerCase()*, *isWhiteSpace()*, and *isLetter()*. Each of these returns a boolean. The characters in the string can be accessed as in the examples above, and tested as shown in the example below.

The following program declares a string and assigns it “ABCD12345def”, and enters a while loop to access and evaluate each character in the string.

#### Ex. 7.5 – Character Testing a String

```
String myString = "ABCD12345def";
int index = 0, digit = 0, upper = 0, lower = 0;

while (index < myString.length()) {
    char ch = myString.charAt(index);
    if (Character.isDigit(ch))
        digit++;
    if (Character.isUpperCase(ch))
        upper++;
    if (Character.isLowerCase(ch))
        lower++;
    index++;
}
System.out.println("Digits " + digit);           // displays Digits 5
System.out.println("Uppercase " + upper);       // displays Uppercase 4
System.out.println("Lowercase " + lower);       // displays Lowercase 3
```

#### Substring Method

The *substring()* method returns a portion of a string. The method accepts one or two arguments. When one argument is provided, the method returns a substring beginning at that index and the rest of the string. When two arguments are provided the method returns the portion of the string beginning with the first argument and ending before the second. An example will help to clarify this.

```
String temp = "abcdefg";
String str1 = temp.substring(2);                // 3rd letter to end
String str2 = temp.substring(2, 5);            // 3rd letter to 6th without 6th
System.out.println("str1 is " + str1);         // displays cdefg
System.out.println("str2 is " + str2);         // displays cde
```

## String Modification

The string modification methods include conversion to upper and lower case, as well as a trim method. The Character class has conversion to upper and lower as well.

### Ex. 7.6 – String Manipulation

```
String str1 = "abcdefg";
String upper = str1.toUpperCase();           // convert to uppercase
String str2 = " ALL UPPER ";
String lower = str.toLowerCase();           // convert to lowercase
String noSpaces = lower.trim();              // trim leading/trailing spaces
System.out.println(str1);
System.out.println(upper);
System.out.println("&" + str2 + "&");         // ampersands have been added
System.out.println("&" + lower + "&");       // to show the removal of the
System.out.println("&" + noSpaces + "&");    // spaces
```

```
abcdefg
ABCDEFGG
& ALL UPPER &
& all upper &
&all upper&
```

Ex. 7.6 Output

The *replace()* method returns a copy of a string object with all occurrences of a specified character replaced by another specified character. Notice the case sensitivity in this example.

```
String str1 = "she Sells Sea shells";
String str2 = str1.replace('S', 'T');
System.out.println(str2);           // displays she Tells Tea shells
```

Only the uppercase occurrences of "S" were replaced as specified.



## The StringBuilder Class

The `StringBuilder` class is similar to the `String` class and provides many of the same methods, but the contents of a `StringBuilder` can be changed. The default constructor for the `StringBuilder` accepts no arguments and provides an instance of the object with storage space to hold 16 characters initially. In addition to the `String` methods, the `StringBuilder` class provides `delete()`, `insert()`, `replace()`, and `toString()`, and there are multiple overloaded versions of `append()` available. Example 6.4 utilizes a `StringBuilder`.

## Tokenizing Strings

The process of tokenizing a `String` breaks the string into its components (tokens). The `split()` method can be used for this purpose, and the character separating the tokens is the delimiter. As an example, the following program declares a string as a series of numbers separated by colons (the delimiter). Note: Although its use is discouraged, the `StringTokenizing` class has been retained in the language.

### Ex. 7.7 – String Tokenizing

Breaking a string into individual parts is known as tokenizing. The code below tokenizes the string, stores the individual tokens in an array, and then displays the array using an enhanced for loop (aka range-based loop).

```
String str1 = "12:34:56:78:14";
String [] tokens = str1.split(":");    // tokenize using the colon delimiter
for(String s : tokens)                 // displays all of the tokens
    System.out.print("\t" + s);
```

12	34	56	78	14
----	----	----	----	----

Ex. 7.7 Output

## Arrays

An Array in Java is a fixed size and is declared as shown in below. The elements are accessed using square brackets and the index begins at zero.

```
double [] numbers = new double[20];    // array of 20 doubles
```

An Array can also be initialized when declared as shown below and the size will be large enough to hold the values in the braces.

```
double [] numbers = { 1.2, 2.3, 5.66, 34};    // array of 4 doubles
System.out.print(numbers[2]);                // displays 5.66
```

## ArrayLists

The ArrayList class in Java is similar to an array and allows storing objects (including strings). The ArrayList automatically expands as items are added to it. Items can be removed from an ArrayList as well, and the ArrayList will shrink in size. There are also methods to simplify ArrayList handling including *add()*, *size()*, *remove()*, and *set()*. The indexes of the ArrayList shift to accommodate a removed item or when an item is inserted into the ArrayList.

### Ex. 7.8 – ArrayLists

To declare an ArrayList, angled brackets are used as shown in the example below that declares an ArrayList of Strings.

```
ArrayList<String> myList = new ArrayList<String>();
```

To add to the list, remove from the list, access elements of the list and insert, the following methods are used.

```
myList.add("Betty");                // add three names to the ArrayList
myList.add("James");
myList.add("Devon");
System.out.println("Initial ArrayList");
for(String name : myList)           // display the indexes and names
    System.out.println(name);

myList.remove(0);                  // removes Betty from the ArrayList
myList.add(0, "Allison");          // inserts Allison at index 0
myList.set(1, "Pavin");            // replaces James with Pavin
System.out.println("Modified ArrayList");
for(String name : myList)           // display the indexes and names
    System.out.println(name);
```

```

Initial ArrayList
Betty
James
Devon
Modified ArrayList
Allison
Pavin
Devon

```

Ex. 7.8 Output

## Wrapper Classes

The primitive data types are not used with ArrayLists. Instead a wrapper class is employed. Wrapper class names begin with uppercase letters, and Integer and Character are spelled out. Conversion between primitive types and wrapper classes is automatic (auto-boxing), but the wrapper class must be used when declaring an ArrayList. The Java Wrapper classes are listed below.

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

### Ex. 7.9 – ArrayList of Doubles

The wrapper class “Double” is used in place of “double” within the angled brackets for the ArrayList. All other methods and operation are as shown above for Strings.

```
ArrayList<Double> myList = new ArrayList<Double>();
```

To add to the list, remove from the list, access elements of the list and insert, the following methods are used.

```

myList.add(2.22);           // add three doubles to the ArrayList
myList.add(3.33);
myList.add(3.45);
System.out.println("Initial ArrayList");

for(Double cost : myList)   // display the indexes and values
    System.out.println(cost);

myList.remove(0);         // removes 2.22 from the ArrayList
myList.add(0, 1.23);      // inserts 1.23 at index 0
myList.set(1, 2.34);      // replaces 3.33 with 2.34
System.out.println("Modified ArrayList");
for(double cost : myList)  // display the indexes and names
    System.out.println(cost);

```

```

Initial ArrayList
2.22
3.33
3.45
Modified ArrayList
1.23
2.34
3.45

```

Ex. 7.9 Output

#### Ex. 7.10 – Tab Delimited File into String

```

File inputFile = new File("input.txt");
Scanner in = new Scanner(inputFile);
while (in.hasNext()) {      // while there are items in the file
    String var1 = in.next();  // read until the tab
    String var2 = in.next();  // read until the end of line
    System.out.println("Item 1 " + var1 + " Item 2 " + var2 );
}

```

## Chapter 8

# Main GUI Design and Components

### Back to the Project

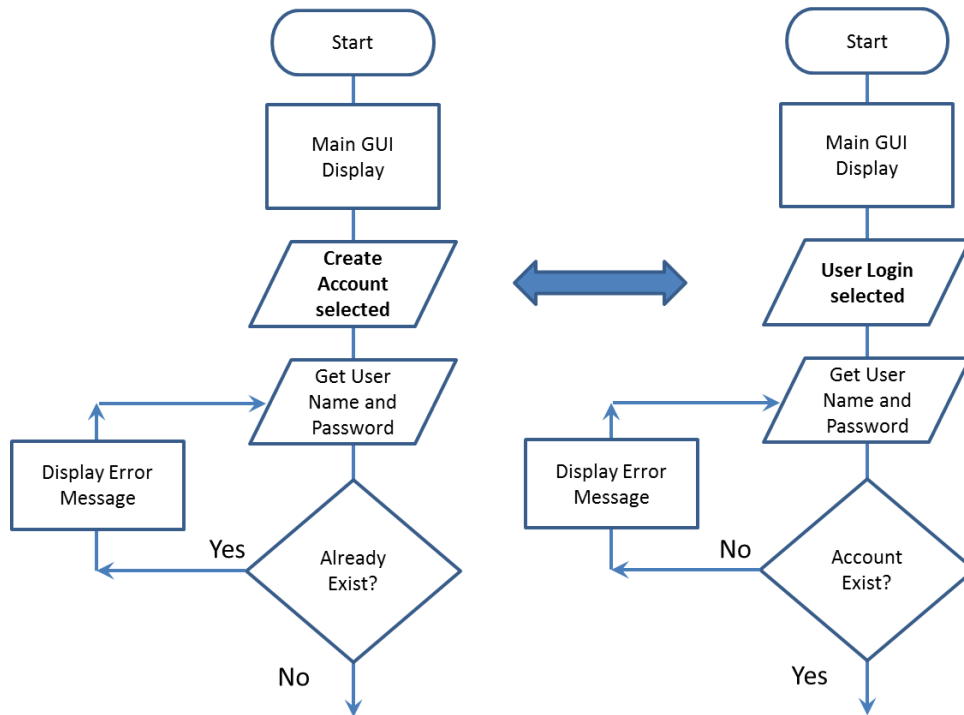
The project requirements for creating a user account and for login are designed and developed using file handling and string manipulation. Consider that when a user logs in, the login information must be verified against current accounts. If the user is creating a new account, it should be completed only if the account does not already exist. Both of these scenarios require a file or files for storing and comparing account information, and string manipulation which was covered previously. They also require another window for obtaining user input, and error handling. After successful login, the main GUI interface is displayed ready for user input. Tackling these requirements all at once would be complex, but they can be divided into segments and completed in pieces more easily.

### Step-wise Refinement and Iterative Enhancement

Breaking down a large problem or task into smaller segments is often referred to as Stepwise Refinement. A large problem or task is decomposed into smaller tasks, and the smaller tasks are then decomposed into even smaller tasks. Once task size and complexity are divided and refined into more manageable portions, design and development begin. Once design of the segments is completed, the smaller segments are developed and the program is built up as the various parts

are completed and added. This process of building and adding software in small segments is referred to as Iterative Enhancement and aligns with the Agile Software Development Process.

With respect to the User Login and Account Creation operations, there are several areas that can be refined, and a flow chart of the operation shows the commonality.



Account Creation and User Login Flow Charts

There are multiple ways of implementing these operations and comparing the operations graphically can be very helpful when deciding on a course forward. This is also true of the main interface.

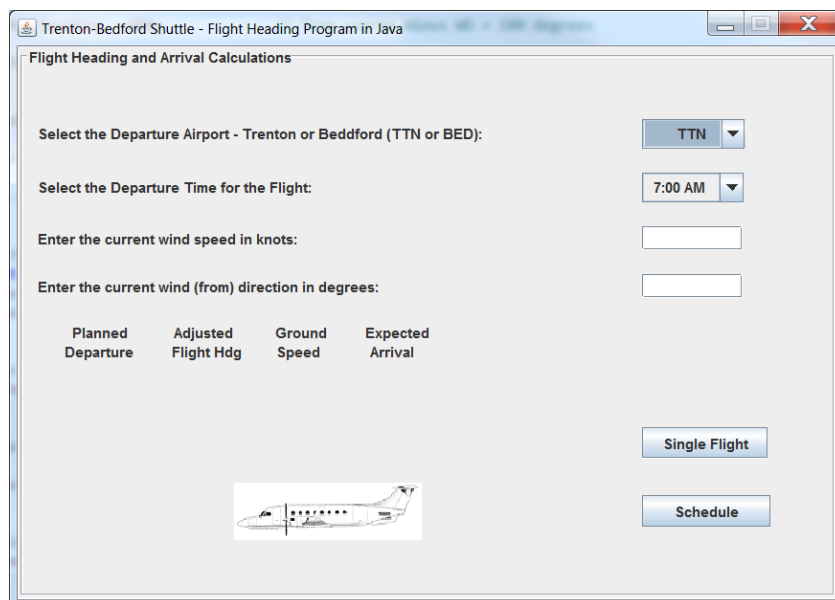
### Main Interface Design

Design is the first step to developing the main interface. The user will interact with the program through this GUI and ease-of-use, and intuitive controls and labels are necessary. The controls to be used on the main GUI depend on what the program does and how the user should interact with it. Are a few buttons

adequate or does the program require a more sophisticated layout? Button groups may suffice, but they allow multiple selections. Radio buttons can be used and implemented to be mutually exclusive. Drop-down menus (option lists) are also mutually exclusive requiring the user to select just one. These considerations during the design phase will save time redesigning or reconfiguring an inadequate interface.

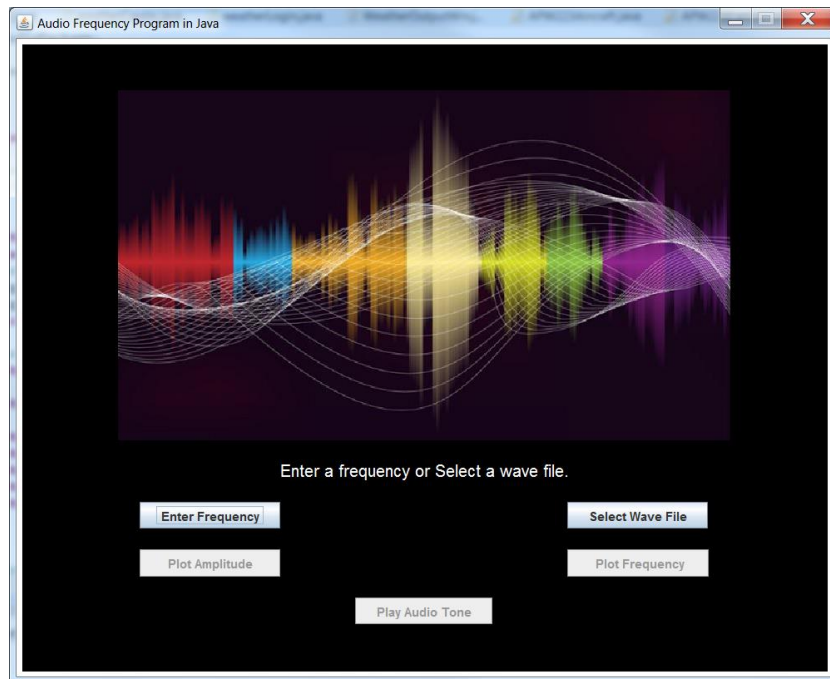
The interface layout should be designed in conjunction with the operational design. Storyboarding, pseudo-code, and flow charts used during design will surface issues that can be corrected early in the process. Software engineers often overlook essential aspects of the interface since they know what the program does, how it does it, and the inputs required. The Agile process typically involves stakeholder reviews and in some cases the client or customer is included. This provides an opportunity for people not familiar with the planned design and operation of the program to offer suggestions for improvement. It also eliminates surprises when the final product is delivered. Examples for a main interface are shown below with various components.

The JComboBoxes (drop-down lists) in this example provide the selections to limit user input. Text boxes and buttons handle the rest.



Depending on the operation of the program, different interface controls may be used for user interaction. In the next example, only buttons are used and are enabled and disabled to prevent errors. In this particular program, a frequency

cannot be played or plotted if one has not been entered or selected, and so those buttons are not yet enabled. A dialog box with an error message could be used when a selection should not be made, but that would require the user to close the message box and continue. Disabling the button is a better solution.



The same design tools that were used for the Create Account and Login windows can be employed including the row and column placement of the components and image(s). The layout options, constraints, and constraint features can be used to set the positions as needed to accommodate the elements of the window and their use. Appendix F includes an example that combines layout managers

## Components

Java provides many components (widgets or controls) for handling user interaction including radio buttons, check boxes, and combo boxes. The components should be located and placed on the frame to accommodate user interaction. The choice for the component should be made during the design and review of how the user will interact with the program. User selection of operations is usually best handled with a component so that there is no erroneous typing by the user and to reduce input validation requirements.



## The Combo Box (drop-down menu)

A drop-down menu or combo-box can provide mutually exclusive selections. When an item in the list is selected, it replaces the read-only text field of the box. A default value can also be indicated. For the Airline example above, the creation of the departure airport combo box and the array of strings to be displayed in the list are created as shown here.

```
String[] dLocs = { " TTN ", " BED " };    // airport designator array
final JComboBox<String> dBox = new JComboBox<String>(dLocs);
```

The box is located on the panel using constraints and a Grid Bag Layout.

```
constraints.gridx = 3;
newPanel.add(dBox, constraints);    // combo box for airport
```

To obtain the user selection, *getSelectedItem()* is used as shown here.

```
// get the combo box selection
String depart = (String) dBox.getSelectedItem();
```



## Radio Buttons

Radio buttons can be mutually exclusive depending on the implementation. The creation is similar to a combo box, and the buttons must be added to a group to create the mutually exclusive relationship. In addition, radio buttons generate an action event when selected which is handled with an action listener. This adds complexity since the user may want to change their mind and select a different button. The *isSelected()* method is used to obtain the input in code. In the example below, two radio buttons are created, added to a group, and then to a panel.

```
JRadioButton rad1 = new JRadioButton("First");
JRadioButton rad2 = new JRadioButton("Second");
```

Next, a group is created for the radio buttons.

```
ButtonGroup radGroup = new ButtonGroup();    // create the button group
```

Then the buttons are added to the group.

```
radGroup.add(rad1);
radGroup.add(rad2);
```

And finally, they are located using constraints or another layout manager and are added to the panel.

```
myPanel.add(rad1);
myPanel.add(rad2);
```

To obtain the user selection in an event handler, the *getSource()* method is used.

```
public void actionPerformed(ActionEvent e) {
    if(e.getSource() == rad1)
        doSomething();
    else if(e.getSource() == rad2)
        doSomething();
}
```

To obtain the user selection within code, each radio button would be tested.

```
if(rad1.isSelected())
    doSomething();
```

## Check Boxes

Check Boxes are a group of elements that can be mutually exclusive or allow the user to select multiple choices. Their creation is similar to radio buttons, and they can be added to a group to create the mutually exclusive relationship. In addition, check boxes generate an action event when selected by the user which can be handled with an action listener, or accessed with the *isSelected()* method.

## Drop-down Menus

A drop-down menu on the border of a window is often used for file handling and program features typically selected by users. Java provides classes for menus including the menu bar, menu, and menu item that are used to create these. The code below creates a file handling menu on a window.

### Ex. 8.1 – Frame Menu

```
public static void main(String[] args) {

    JFrame myFrame = new JFrame();           // create the frame

    JMenuBar mBar = new JMenuBar();         // create the Menu bar
    JMenu fileMenu = new JMenu("File");     // create the Menu

    JMenuItem openItem = new JMenuItem("Open"); // create 4 menu items
    JMenuItem saveItem = new JMenuItem("Save");
    JMenuItem saveAsItem = new JMenuItem("Save As");
    JMenuItem exitItem = new JMenuItem("Exit");

    fileMenu.add(openItem);                 // add the 4 items to the menu
    fileMenu.add(saveItem);
    fileMenu.add(saveAsItem);
    fileMenu.add(exitItem);

    mBar.add(fileMenu);                     // add the menu to the bar

    myFrame.setJMenuBar(mBar);             // add the menu bar to the frame

    myFrame.setSize(300,300);
    myFrame.setVisible(true);
    myFrame. setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
```

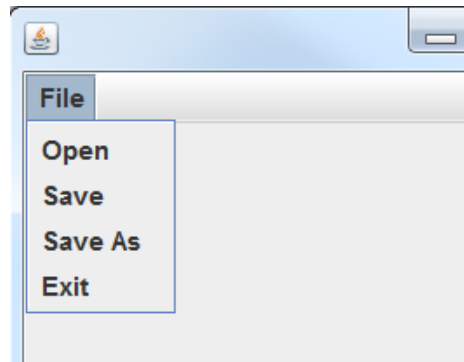
To capture the selection of a menu item, each item would have an `actionListener` for example:

```

class MenuItemListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("SAVE AS WAS SELECTED.");
    }
}

saveAsItem.addActionListener(new MenuItemListener());

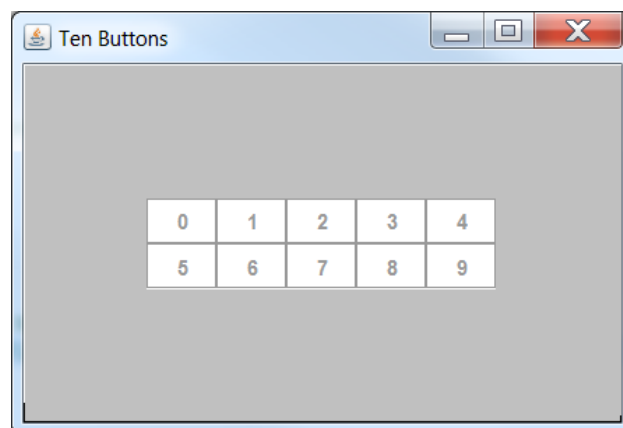
```



Ex. 8.1 Frame Menu

### Multiple Button

When many buttons are needed on a frame, creating, positioning, and writing an action listener for each one would require extensive code. A loop can be used to generate, locate, and add a listener to the buttons reducing the amount of code required. The following example creates ten (10) buttons, locates them, and adds action listeners to them within a loop.



Ten Buttons Example

## Ex. 8.2 – Ten Buttons

```

public static void main(String[] args) {

    JButton[] buttonsArray = new JButton[10];           // Array of buttons
    JFrame mainGUI = new JFrame("Ten Buttons");
    JPanel mainPanel = new JPanel(new GridBagLayout());
    GridBagConstraints constraints = new GridBagConstraints();

    // Array of the names on the buttons
    String [ ] butNames = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"};

    constraints.gridwidth = 1;
    constraints.gridheight = 3;

    int yLoc = 20, xLoc = 2;                          // Starting location on the grid

    // generic listener for the buttons
    ActionListener listener = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            if (e.getSource() instanceof JButton) {
                // get the text on the button
                String text = ((JButton) e.getSource()).getText();
                // change the clicked button color
                ((JComponent) e.getSource()).setBackground(Color.ORANGE);
                // disable the button that was clicked
                ((JComponent) e.getSource()).setEnabled(false);
                System.out.println("Button listener for button: " + text); // test
            }
        }
    };

    // loop that creates and locates the buttons, and adds the listener
    for(int i = 0; i < 10; i++) {
        buttonsArray[i] = new JButton(butNames[i]);
        buttonsArray[i].addActionListener(listener); // add the listener
    }
}

```

```

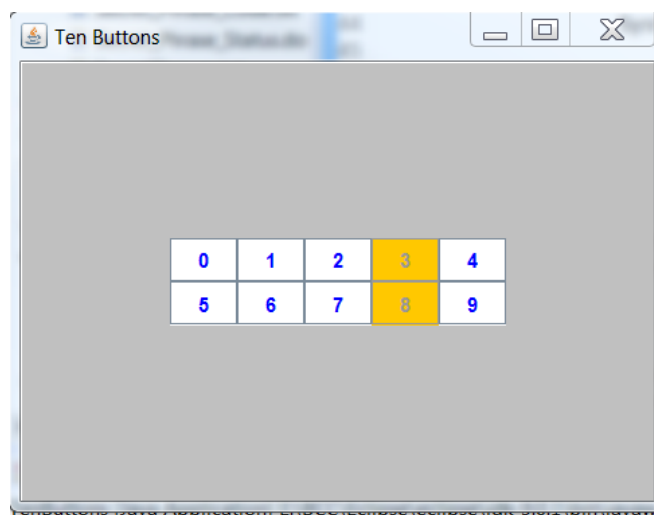
xLoc = i + 2;

buttonsArray[i].setBackground(Color.WHITE);
buttonsArray[i].setForeground(Color.BLUE);

if(i > 4) {
    yLoc = 24;           // go to second row
    xLoc = i - 5 + 2;   // starts at two
}
else {
    xLoc = i + 2;
}
constraints.gridx = xLoc;           // locate the button
constraints.gridy = yLoc;
mainPanel.add(buttonsArray[i], constraints);
}

mainGUI.setSize(840,600);
mainGUI.add(mainPanel);
mainPanel.setBackground(Color.LIGHT_GRAY);
mainGUI.setVisible(true);
} // end of main
}

```



```

Button listener for button: 8
Button listener for button: 3

```

## Main Window Image

Any image can be used as an icon, background, or to enhance the interface, and there are many ways to create one for use in the program. One way is to use Snip to select an image or portion of an image to use, and save it as a file. Then open the image file with MS-Paint or Gimp and use resize to convert the image to the appropriate size using Percentage or Pixels. The image is then added to the directory where the program package is located for use. Positioning the image is handled similar to a control or component. The Audio Frequency program GUI shown earlier has an image that covers most of the main window. The `BufferedImage` class is a subclass of the AWT `Image` class, and has three constructors and methods for getting information about the image. To add an image, create a buffered image and a label to hold it as shown below.

```
BufferedImage myPicture = ImageIO.read(new File("fileName.png"));
JLabel picLabel = new JLabel(new ImageIcon(myPicture));
```

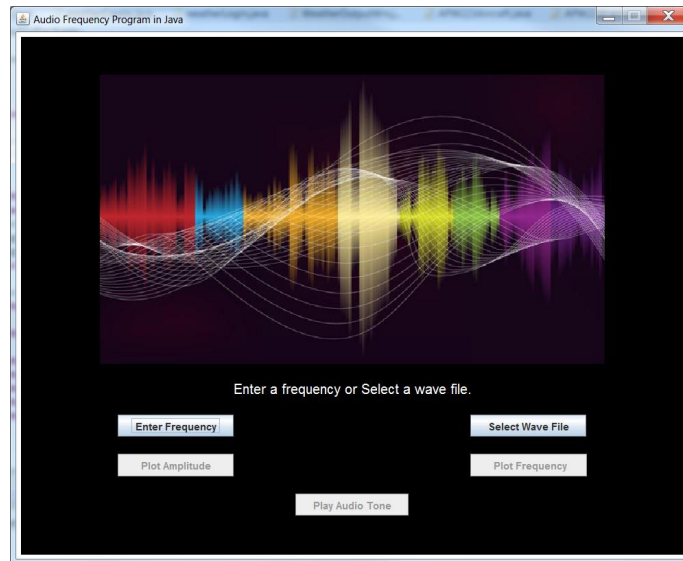
Add the picture label to the panel using constraints for the location.

```
mainPanel.add(picLabel, constraints);
```

The code for the image on the Audio Frequency GUI is shown below.

```
BufferedImage myPicture = ImageIO.read(new File("Sine1.png"));
JLabel picLabel = new JLabel(new ImageIcon(myPicture));
constraints.gridx = 1;
constraints.gridy = 2;
constraints.anchor = GridBagConstraints.CENTER;
constraints.gridwidth = 5;           // set the grid width
newPanel.add(picLabel, constraints);
```

As shown above, the `BufferedImage` is declared and `ImageIO` reads a declared `File`. A `Label` is declared to hold the image and is assigned a new `ImageIcon` which is assigned the picture `File`. The location (`constraints`) is specified and the `Label` is added to the `Panel`. The image on the Audio Frequency GUI is shown again below.



Audio Frequency GUI with Image

### Changing an Image

To change an existing image, use `setIcon()`. The code below accesses an array of images that will change throughout the program using the `setIcon()` method.

```
picLabel.setIcon(new ImageIcon(imageFileArray[imageNum]));
```

Images can also be used in Java with `Graphics2D` along from the `java.awt.Toolkit` in the paint method.

```
class MyCanvas extends JComponent {
    public void paint(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;

        Image img1 = Toolkit.getDefaultToolkit().getImage("yourFile.gif");

        // arguments - image, x, y, ImageObserver
        g2d.drawImage(img1, 10, 10, this);
        g2d.finalize();
    }
}
```



# Chapter 9

## Main GUI and Data Display

Many GUI programs display computed data to users. To provide examples, two situations will be considered; handling output to a display as the user enters data used in a computation, and reading data from a file to display. The examples will display in the main user interface and in a separate window.

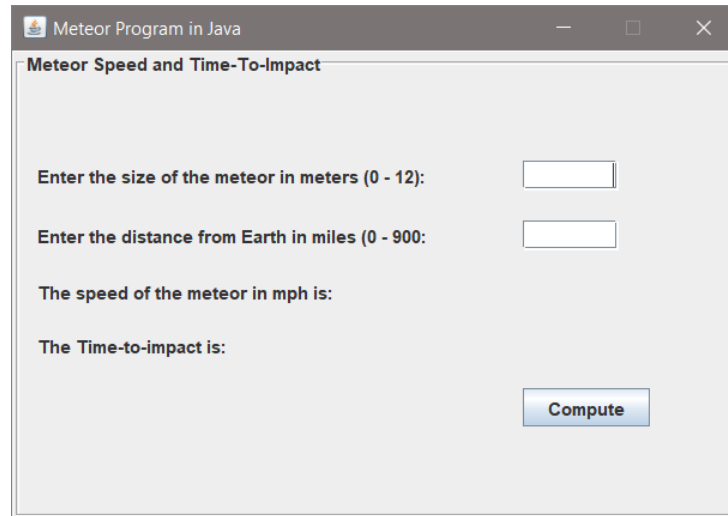
Assume a meteor defense system program that accepts the size of a meteor in meters and the distance from Earth in miles. The program computes the speed of the meteor and the time-to-impact. Entry components on the main window will obtain user input, and a button click will call a compute function to compute the values which will be displayed in the GUI and in columns in a separate output window. For this example, main (shown below) will simply create an instance (object) of the *MeteorWin()* class shown later.

Ex. 9.1 – Meteor Program Example - main

```
package CH_9_Meteor_package;

public class CH_9_Meteor {
    public static void main(String[] args) {
        new MeteorWin();           // create an instance of MeteorWin
    }
}
```

The main GUI (MeteorWin) will have two entry components for user input with description labels, two labels for the output section, and a compute button. The labels for the meteor speed and time-to-impact will be modified and overwritten when the compute button is clicked to show the results.



Meteor Program Example

Ex. 9.2 – Meteor Program Example – MeteorWin()

```
public class MeteorWin extends JFrame {

    private static JFrame mainGUI = new JFrame("Meteor Program in Java");
    private JPanel newPanel = new JPanel(new GridBagLayout());
    private JLabel labelSize = new JLabel("Meteor size in meters (0 - 12): ");
    private JLabel labelDist = new JLabel("Distance from Earth in miles (0-900:");
    private JLabel labelSpeed = new JLabel("The speed of the meteor in mph is: ");
    private JLabel labelTimeToImpact = new JLabel("The Time-to-impact is: ");
    private JTextField textFieldSize = new JTextField(6);
    private JTextField textFieldDistance = new JTextField(6);
    private JButton computeButton = new JButton("Compute");
```

The components are created in the class before the constructor as shown above, and are positioned using a Grid Bag Layout and constraints within the constructor shown below.

## Ex. 9.3 – Meteor Program Example – MeteorWin() Constructor

```

public MeteorWin() {                                     // constructor

    mainGUI.setSize(500,350);

    GridBagConstraints constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.WEST;
    constraints.insets = new Insets(10, 10, 10, 10); // padding for all components
    constraints.weightx = 0.5;

    constraints.gridx = 0;                               // add components to the panel
    constraints.gridy = 0;
    newPanel.add(labelSize, constraints);

    constraints.gridx = 2;
    newPanel.add(textFieldSize, constraints);
    textFieldSize.setHorizontalAlignment(JTextField.RIGHT);

    constraints.gridx = 0;
    constraints.gridy = 1;
    newPanel.add(labelDistance, constraints);

    constraints.gridx = 2;
    newPanel.add(textFieldDistance, constraints);
    textFieldDistance.setHorizontalAlignment(JTextField.RIGHT);

    constraints.gridx = 0;
    constraints.gridy = 3;
    newPanel.add(labelSpeed, constraints);

    constraints.gridx = 0;
    constraints.gridy = 4;
    newPanel.add(labelTimeToImpact, constraints);

    constraints.gridx = 2;
    constraints.gridy = 6;
    constraints.anchor = GridBagConstraints.WEST;
    newPanel.add(computeButton, constraints);

    Font myFont = new Font("Serif", Font.BOLD, 12);    // set the panel font
    newPanel.setFont(myFont);

```

```

newPanel.setBorder(BorderFactory.createTitledBorder( // set the panel border
BorderFactory.createEtchedBorder(), "Meteor Speed and Time-To-Impact"));

mainGUI.add(newPanel); // add the panel to the GUI
mainGUI.setResizable(false); // disallow resizing
mainGUI.setLocationRelativeTo(null); // center the GUI
mainGUI.setVisible(true);

// assign the action listener to the button
computeButton.addActionListener(new ButtonListener());
} // end of constructor

```

### The Button Listener

The work is really accomplished in the action listener once the button is clicked. This includes validating the input, computing the values, and updating the labels with the output. User input is handled first using `getText()` in a “try” block.

#### Ex. 9.4 – Meteor Program Example – ButtonListener

```

private class ButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        double size = 0, distance = 0, speed = 0;
        try {
            String sizeString = textFieldSize.getText(); // get the meteor size
            size = Double.parseDouble(sizeString);

            String distanceString = textFieldDistance.getText();// get the distance
            distance = Double.parseDouble(distanceString);

            // input validation
            if(size <= 0 || size > 12 || distance <= 0 || distance > 900) {
                labelSpeed.setText("Valid data cannot be computed. ");
                labelTimeToImpact.setText("Valid data cannot be computed.");
            } // end of IF
        }
    }
}

```

If the “try” block succeeds, the values are computed and the labels are updated using the `setText()` method. If the “try” block fails because of invalid input, the “catch” block (below) contains a `showMessageDialog()` announcing the error.

## Ex. 9.4 – Meteor Program Example – ButtonListener (continued)

```

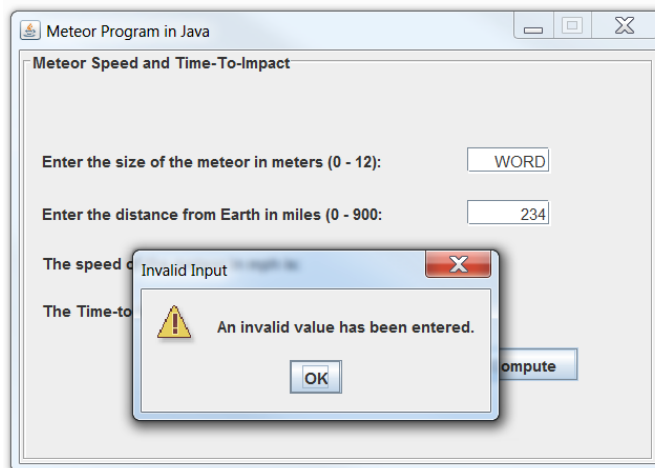
    speed = size * 120.0;
    double TTI = distance/speed;           // compute time-to-impact
    int TTIhours = (int)TTI;               // parse out hours
    int TTImins = (int)((TTI - TTIhours) * 60); // parse out minutes

    labelSpeed.setText("The speed of the meteor is: " +
        String.format("%.1f", speed) + " mph");
    labelTimeToImpact.setText("The time to impact is: " + TTIhours + "
        Hours : " + TTImins + " Minutes");

    // Placeholder to update the output window
}
catch(NumberFormatException ne) {
    JOptionPane.showMessageDialog(null, "An invalid value has been
        entered.", "Invalid Input", JOptionPane.WARNING_MESSAGE);
}
}
}; // end of ButtonListener
} // end of MeteorWin class

```

The exception that could be thrown in this case is a number format exception if a number is not entered by the user as shown below.



Number Format Exception

In the code above, there is a placeholder for calling a function to update the separate output window. This will be covered next.

## Output to a separate window

The example program so far has a main GUI, it accepts and validates user input, and updates the labels on the GUI after computing the values. The separate display output of column data is not yet implemented. The program will first need to create the second window and then update the window as the user enters additional input for computed values. There are a variety of designs for implementing this including: creating the second window when the program starts, having a second button on the main GUI that creates the second window, or creating the second window when the compute button is clicked for the first time.

Depending on the location of the data display window class, this second window could be created in main, within the MeteorWin constructor, or when MeteorWin is created. To ensure that the data display object can be accessed and updated, the example creates the instance in the MeteorWin class.

### Ex. 9.5 – Data Display Window – Create on Program Start

```
public class MeteorWin extends JFrame {
    DataOutputWin ddWin = new DataOutputWin(); // create the data display
    private static JFrame mainGUI = new JFrame("Meteor Program in Java");
```

## Data Display in the Second Window

The data display window will require headers for the columns and a way to add the values as they are computed in column format. Java provides a table and text area solution, or a string could be built through concatenation and formatted with spacing for display. The `StringBuilder` class could also be used. A combination of formatting and the text area are used in this example, and the `DataOutputWin` is declared as a subclass of the `MeteorWin` class. A `JTextArea` is declared as having 40 rows (should be enough) and 4 columns, and a pre-formatted header is assigned when declared. The `String.format()` method creates the column spacing which requires a little trial and error, but once set for the data values works fine. The `append()` method enables adding rows of data, the `setLocation()` method locates the window offset from the main GUI, and the `mainGUIToFront()` ensures that the main GUI is in front of the data display window. There is a placeholder comment in the code for the update function.

## Ex. 9.6 – the Data Display Window

```

public class DataOutputWin {

    private JFrame ddWin = new JFrame("Meteor Data Display");
    private JPanel ddPanel = new JPanel();

    String header1 = String.format("%20s %20s %20s %20s", "Meteor", "Distance",
        "Speed", "Time to");
    String header2 = String.format("%20s %25s %18s %18s", " Size", " from Earth",
        "in MPH", "Impact");

    // text area with 40 rows and 4 columns
    private JTextArea ddTA = new JTextArea(header1, 40, 4);

    DataOutputWin() {                                // constructor
        ddWin.setSize(500,350);
        ddPanel.setBackground(Color.white);

        Font myFont = new Font("Serif", Font.BOLD, 12);    // set the font
        newPanel.setFont(myFont);
        ddTA.setFont(myFont);

        ddPanel.add(ddTA);                            // add the textArea to the panel
        ddWin.add(ddPanel);                          // add the panel to this frame

        ddTA.append("\n" + header2);                // append to the text area

        ddWin.setResizable(false);
        ddWin.setLocation(200,200);
        ddWin.setVisible(true);
        mainGUI.toFront();
    }

    // Placeholder for the update method in 9.7

} // end of DataOutputWin class

} // end of MeteorWin class

```

An update is required each time the “Compute” button is clicked, and the data will need to be formatted to maintain the columns. The placeholder is for a call to a method *updateData()*, and this method is added to the DataOutputWin class.

## Ex. 9.7 – Update Data Method

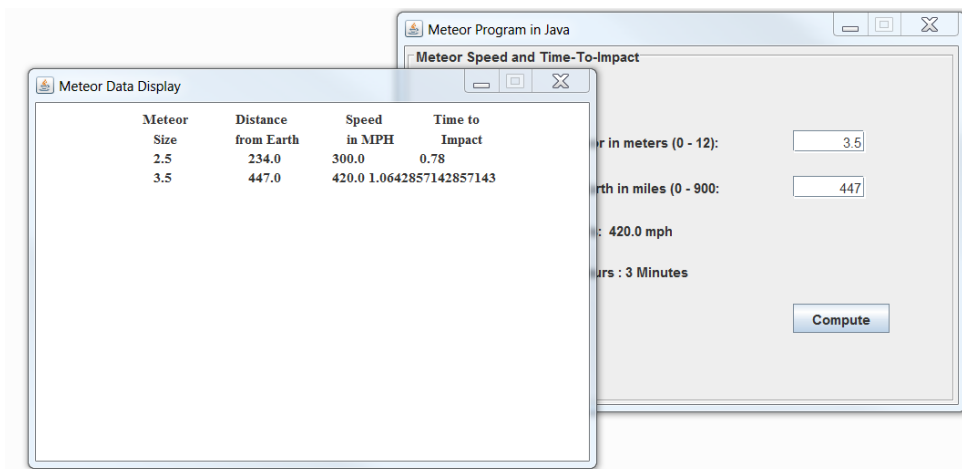
```

public void updateData(double size, double distance, double speed, double TTI) {
    String dataSet = String.format("%20s %25s %18s %18s", size, distance, speed, TTI);
    ddTA.append("\n" + dataSet);           // append to the text area
}

} // end of DataOutputWin class

```

The *updateData()* method receives the values computed when the button was clicked and appends the values to the JTextArea after formatting them for the columns. The time-to-impact value was formatted for the main GUI within the *setText()* method, so the value passed to the method is not formatted correctly as shown in the screen capture below. This can be done prior to passing the value or in the *updateData()* method.



Ex. 9.7 Display Output

The *updateData()* method needs a string and the TTI value is a double that represents hours and hundredths of hours. The same parsing used in the button listener could be employed in the *updateData()* method with some additions like a colon separator for hours and minutes.

The solution below includes handling cases when the minutes portion of time-to-impact is less than 10 and zero would be eliminated from the integer value.



## Ex. 9.7A – Update Data Method Corrected

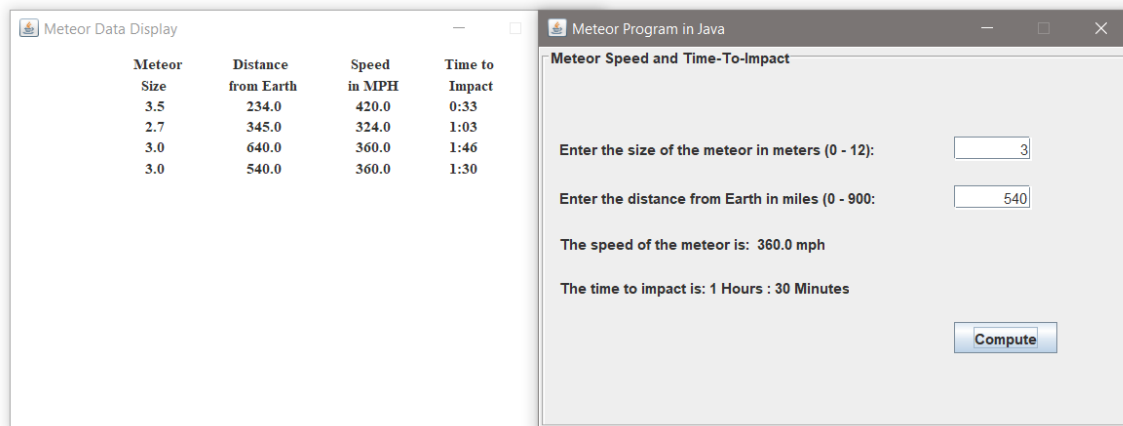
```

public void updateData(double size, double distance, double spd, double TTI) {
    int TTIhours = (int)TTI;
    int TTImins = (int)((TTI - TTIhours) * 60);
    String minTo = "";

    if(TTImins < 10)                // if it is 1 thru 9, the preceding zero would be lost.
        minTo = "0" + (String.valueOf(TTImins));

    String timeTo = TTIhours + ":" + minTo;
    String dataSet = String.format("%10s %25s %23s %18s", size, distance, spd, timeTo);
    ddTA.append("\n" + dataSet);    // append to the text area
}
} // end of DataOutputWin class

```



Ex. 9.7A Display Output

## Scrollbar

To add a scroll bar to a window, a JScrollPane is used and is assigned to the panel. In the program below, an image and BorderLayouts are used. The JScrollPane is added to the frame after assigning it to myPanel and setting the scroll bar specifics.

## Ex. 9.8 – ScrollPane Example

```

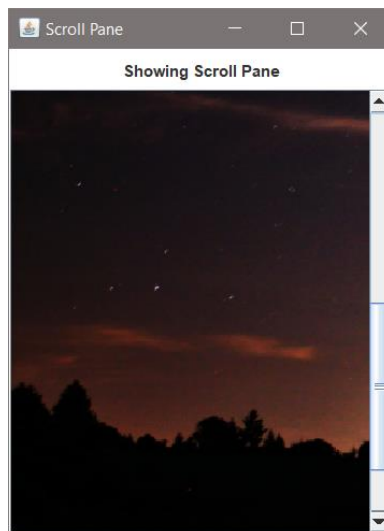
public class ScrollPaneExample extends JFrame {

    JFrame myFrame = new JFrame("Scroll Pane");
    JLabel textLabel = new JLabel("Showing Scroll Pane", JLabel.CENTER);
    ImageIcon image = new ImageIcon("nightsky.jpg");
    JLabel picLabel = new JLabel(image, JLabel.CENTER);
    JPanel myPanel = new JPanel(new BorderLayout());

    public ScrollPaneExample()
    {
        textLabel.setPreferredSize(new Dimension(300,60));
        myFrame.getContentPane().setBackground(Color.WHITE);
        myPanel.add(picLabel, BorderLayout.CENTER);
        myFrame.add(textLabel, BorderLayout.NORTH);
        JScrollPane mySC = new JScrollPane(myPanel,
            JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
            JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
        myFrame.add(mySC, BorderLayout.CENTER);
        myFrame.setSize(300,400);
        myFrame.setVisible(true);
    }
    public static void main(String[] args) {

        new ScrollPaneExample();
    }
}

```



Ex. 9.8 ScrollPane Example Output

## File Data Display

Reading data from a file and displaying it would be handled similar to the update data example above. Assume that there is a data file containing a data set that would be used to compute values and display results in columns. A loop would read from the file and pass the values to the computing method.

## Plotting Data

Data can be plotted (drawn) on a display window, but not on a JFrame object. To draw (including text), a JComponent, JPanel, JTextComponent, or JLabel are used. The paintComponent method does the drawing, and is called when the component is created the first time, and when the window is resized. The paintComponent receives a Graphics object which has the graphics state (color, font, etc.). As an example, the following code will draw some bars in a window.

Ex. 9.9 – Bars on a JComponent

```

package Bars_9_8_package;

public class Bars_9_8 extends JComponent {
    public void paintComponent(Graphics g)
    {
        g.fillRect(0, 10, 200, 10);           // Draw the bars
        g.fillRect(0, 30, 300, 10);
        g.fillRect(0, 50, 100, 10);
    }

    public static void main(String[] args) {
        JFrame frame = new JFrame();
        frame.setSize(400, 200);
        frame.setTitle("A bar chart");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JComponent myChart = new Bars_9_8();
        frame.add(myChart);
        frame.setVisible(true);
    }
}

```

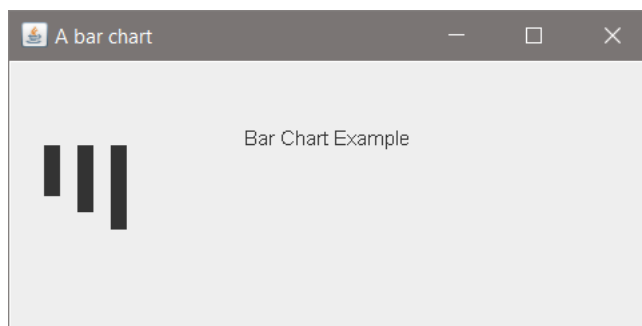


Ex. 9.9 Bar Chart Output

The bars are drawn using *fillRect()* with arguments for starting and endpoints as x-coord, y-coord, and width, and height. When drawing (graphics in general) it is important to remember that x,y coordinates 0,0 are in the top left corner of the output window. The y-coordinate is down from there as a positive number. To illustrate this, the bar chart drawing code from the previous example is modified to draw vertical bars. The starting point for the x-coordinate moves 20 pixels to the right for each new bar. The starting y-coordinate is static at 50 pixels. The width is 10 pixels for each bar, and the endpoint for the y-coordinate increases by 10 pixels for each bar resulting in an increase downward. This example also uses *drawString()* to add text to the window.

Ex. 9.9A – Bars on a JComponent - vertical

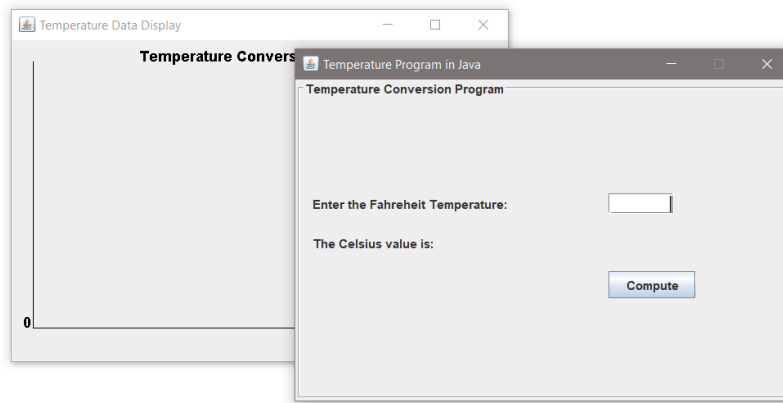
```
public void paintComponent(Graphics g) // vertical
{
    g.fillRect(20, 50, 10, 30); // starting x-coordinate moves 20 pixels right
    g.fillRect(40, 50, 10, 40); // starting y-coordinate is static at 50 pixels
    g.fillRect(60, 50, 10, 50); // width is static at 10 pixels
    g.drawString("Bar Chart Example", 140, 50); // Draw the text
}
```



Ex. 9.9A Bar Chart Output

The previous examples illustrate a single drawing action. To plot as the user enters data will require multiple calls to redraw (repaint) as data is entered. The next example uses a program that computes a Celsius temperature from a Fahrenheit input and plots both values in a second display.

#### Ex. 9.10 – Temperature Conversion and Plot



Ex. 9.10 Temperature Conversion Plot

The user will enter a value and the Celsius temperature will be output on the main GUI while the entered Fahrenheit temperature and computed Celsius temperature will be plotted in the second window. Creating the second window was covered earlier, and that code will be omitted in the example.

The solution requires having the paint component update the frame when data is entered and computed. Since Java repaints the entire panel, earlier data must be preserved and repainted along with new data. An `ArrayList` can be used to store the values, and a loop will cycle through the values for repainting.

Starting with the plot window, this code sets up the frame and `ArrayList` in the class. The `serialVersionUID` is beyond the scope of this text, but has to do with serializing and deserializing of objects. The IDE can add a default value.

```
public class PlotWin extends JComponent {
    private static final long serialVersionUID = 1L;
    private JFrame pWin = new JFrame("Temperature Data Display");
    private JPanel pPanel = new JPanel();
    String Title = "Temperature Conversion Values";
    private ArrayList<Integer> values;
```

Next, the constructor declares the `ArrayList`, sets the size and background for the panel, and then it is added to the frame (`pWin`).

```
PlotWin() {                                     // constructor

    values = new ArrayList<Integer>();

    pWin.setSize(500,350);                     // width, height
    pPanel.setBackground(Color.white);

    // add the panel to this frame
    pWin.add(pPanel);
```

As shown below, a `JComponent` is declared in the `PlotWin` constructor along with the `paintComponent` for drawing. `Graphics` is a helper class that allows drawing things on the panel and `super.paintComponent(g)` erases whatever is currently drawn and prepares the component for drawing. Two fonts are created for the example. One for the title in the plot window and one for the values that are plotted. The panel title is drawn using `drawString()`, and `drawLine()` draws the x and y axis lines.

```
JComponent component = new JComponent() {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // Create a font for the title
        Font titleFont = new Font("Arial", Font.BOLD, 14);
        g.setFont(titleFont);

        // Create a font for the plotted values
        Font FCFont = new Font("Arial", Font.BOLD, 10);

        g.setColor(Color.BLACK);
        // Title in panel
        g.drawString("Temperature Conversion Plot", 125, 20);

        g.drawString("0", 10, 280);           // text for axis
        g.drawLine(20, 280, 400, 280);      // horizontal line
        g.drawLine(20, 280, 20, 20);        // vertical line
```

To plot the values on the panel, the loop uses an offset since 0,0 is the top-left corner of the panel. The loop accesses the `ArrayList` values and since the plot locations require integers, the `ArrayList` was declared as holding `Integers`. They are cast to integers when stored (shown later). The indicator used for the values being plotted is a circle using `fillOval()`, and `drawString()` adds the value above the

plotted circle. The x-coordinate is updated, and the loop continues as shown below. Outside the JComponent, the component is added to *pWin*. Notice the semicolon on the closing brace for the JComponent.

```

int xCoord = 30; // starting x

for(int i = 0; i < values.size(); i = i + 2) {
    g.setFont(FCFont);

    int y = values.get(i);

    g.setColor(Color.BLACK);

    int yCoord = 280 - y;           // move down to 280 and up

    g.fillOval(xCoord, yCoord, 5, 5); // fahrenheit
    g.drawString(String.valueOf(y) + "F", xCoord, yCoord-5);

    y = values.get(i+1);

    yCoord = 280 - y;           // move down to 300 and up
    g.fillOval(xCoord, yCoord, 5, 5); // plot celsius
    g.drawString(String.valueOf(y) + "C", xCoord, yCoord-5);

    xCoord = xCoord + 20;
} // end of for loop

} // end of paint
}; // end of JComponent

pWin.add(component);           // add the JComponent

```

To invoke this process, the ButtonListener is modified to include a call to a method in PlotWin that appends the new data entered and calculated to the ArrayList and then calls the *repaint()* method.

```

pWin.appendData(fahrenheit, celsius); // add to the ArrayList

```

The *append()* method.

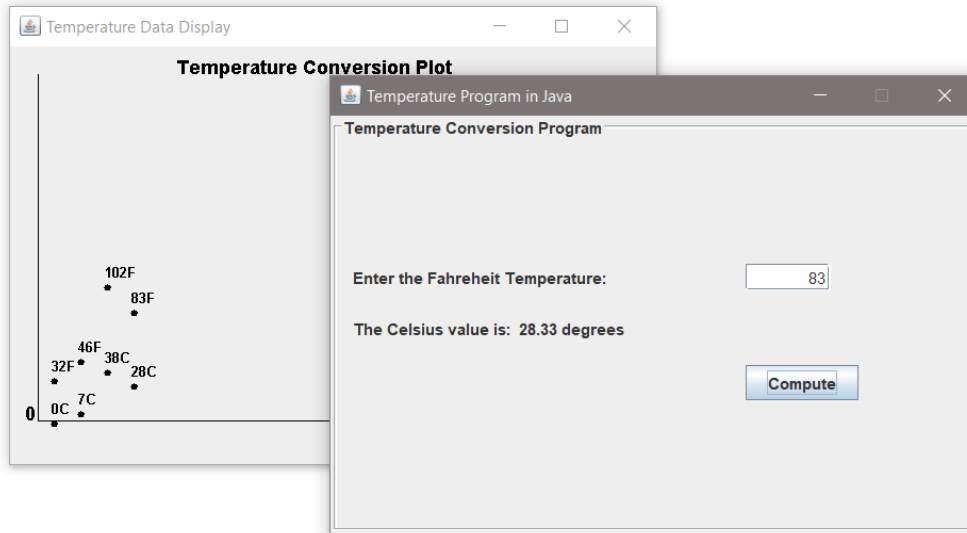
```

// adds new temperatures to the ArrayList, then calls repaint
public void appendData(double F, double C) {

    values.add((int)F);
    values.add((int)C);
    pWin.repaint();
}

```

The result is that each time a new value is entered on the main GUI and the button is clicked, the Celsius temperature is computed, and both values are added to the ArrayList, and the panel is repainted using the ArrayList values.



PlotWin Example Output

## Line Charts

As shown in the section on `paintComponents`, drawing lines requires integer values for a start-*x*, start-*y*, end-*x*, and end-*y* coordinate. In this example, the `JComponent` is added in the constructor for a class with a frame to draw a blue triangle.

```
public class LineChart extends JComponent {

    private static JFrame IWin = new JFrame("Line Drawing");

    public LineChart() { // constructor

        IWin.setSize(350,300); // width, height
        IWin.getContentPane().setBackground(Color.WHITE);
        IWin.setLocation(500, 200);

        JComponent component = new JComponent() {
            public void paintComponent(Graphics g) {
                Font titleFont = new Font("Arial", Font.BOLD, 14);
            }
        };
    }
}
```



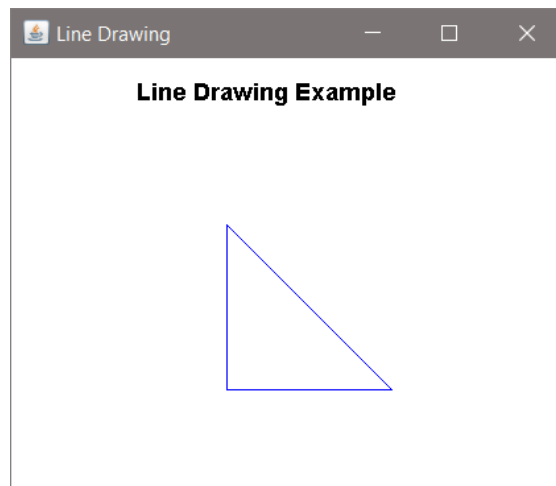
```

        g.setFont(titleFont);
        g.setColor(Color.BLACK);
        g.drawString("Line Drawing Example", 75, 25);
        g.setColor(Color.BLUE);
        g.drawLine(130, 200, 130, 100); // x1, y1, x2, y2
        g.drawLine(130, 100, 230, 200);
        g.drawLine(230, 200, 130, 200);
    } // end of paint
}; // end of JComponent

//Win.add(component);
//Win.setVisible(true);
//Win.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

public static void main(String[] args) {
    new LineChart();
}
}

```

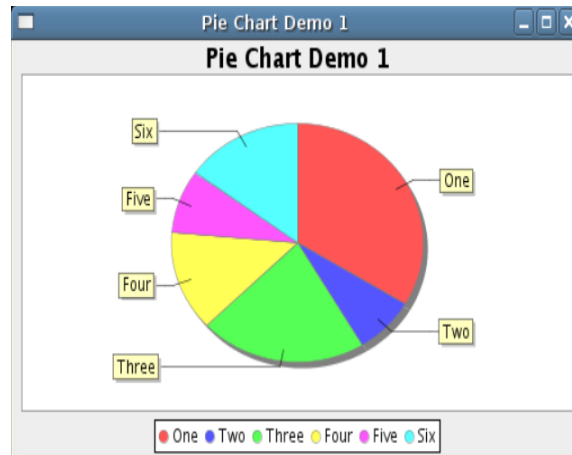


Line Chart Example Output

Drawing line charts from data sets would be a bit more complex since each line in the chart would require start-x, start-y, end-x, end-y specifiers. An algorithm could be developed, but Java charting tools can be used to simplify the task.

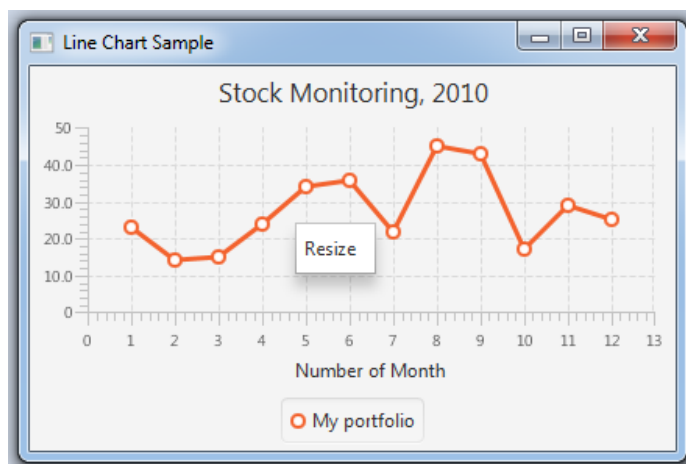
## Charting Tools

There are Java chart tools such as JFreeChart which is free to download (sample from their website below). <http://www.jfree.org/jfreechart/samples.html>



JFreeChart Example

The JavaFX charts and methods in the `javafx.scene.chart` package are also maturing into a comprehensive charting tool with extensive capability. A sample is shown below.



javafx.scene.chart Example

# Chapter 10

## Dates, Time, Sound, and More

### Date and Time

The `java.util` package provides a date and time class for the current date and time. The implementation includes declaring a `Date` object.

```
Date myDate = new Date();  
System.out.println(myDate);
```

The output of this code is: Sun Jan 26 15:29:17 EST 2020

The `java.time` package also provides date and time classes including `LocalTime`, `LocalDateTime`, a `DateTimeFormatter`, and others.

```
import java.time.LocalDateTime;  
import java.time.format.DateTimeFormatter;  
  
DateTimeFormatter df = DateTimeFormatter.ofPattern("MM/dd/yy");  
LocalDateTime now = LocalDateTime.now();  
System.out.println(now);
```

The output of this code is: 01/26/20

The `DateTimeFormatter` provides a variety of formats. Another example follows.

```
DateTimeFormatter df2 =
    DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
LocalDateTime now2 = LocalDateTime.now();
String formattedDate2 = df2.format(now2);
System.out.println(formattedDate2);
```

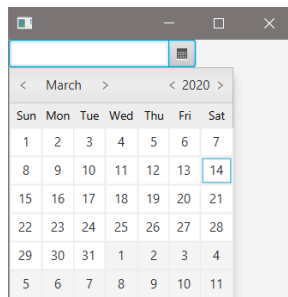
The output of this code is: 26-01-2020 15:58:01

## Calendar

To implement a calendar in Java, there are several available including the `DatePicker` in Javafx. The following program displays a `DatePicker()` calendar that allows the user to enter or select a date.

Ex. 10.1 – Display a User-selectable Calendar

```
public class DatePickerTest extends Application {
    public static void main(String[] args) {
        launch(args);
    }
    public void start(Stage stage) {
        VBox vbox = new VBox(20);
        Scene scene = new Scene(vbox, 400, 400);
        stage.setScene(scene);
        DatePicker dp = new DatePicker();
        vbox.getChildren().add(dp);
        stage.show();
    }
}
```



Ex. 10.1 Calendar Example Output

## Playing Sound

An `AudioInputStream` and a clip resource are used to open a sound file and start play. The code below assigns a wave file to `soundFile`, gets an input stream, a clip resource, opens the file, and plays the sound.

```
File soundFile = new File("myWaveFile.wav");
AudioInputStream audioInputStream =
    AudioSystem.getAudioInputStream(soundFile.getAbsolutePath());
Clip clip = AudioSystem.getClip();
clip.open(audioInputStream);
clip.start();
```

The complete code with the try/catch for exception handling, and thread handling while loops is shown below. Windows 10 issues not yet completely resolved require the while loops. The program assumes that the \*.wav file is located in the java project folder (or jar file), otherwise a full path to the file would be required.

```
try {
    File sndFile = new File("myWaveFile.wav");
    AudioInputStream audioInputStream =
        AudioSystem.getAudioInputStream(sndFile.getAbsolutePath());

    Clip clip = AudioSystem.getClip();
    clip.open(audioInputStream);
    clip.start();

    while (!clip.isRunning())
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    while (clip.isRunning())
        try {
            Thread.sleep(10);
```

```

        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        clip.close();
    }
    catch (UnsupportedAudioFileException e) {
        e.printStackTrace();
    }
}

```

### Sound...Another Way

Another way to play sound is to assign the executable that will play the file and the path to the sound file to be played to a string. The string is then passed to a `Runtime` object which is assigned to a process. This works fine as long as there is a player to select by code. The escapes are for the quotes on both portions of the string.

```

String command = "C:/Program Files (x86)/Windows Media
Player/wmplayer.exe\"C:/myWaveFile.wav\"";

Process p = Runtime.getRuntime().exec(command);

```

### Launching a Browser, Mail, and File Handler

The `Desktop` class allows a Java application to launch associated applications registered on the native desktop to handle a URI (Uniform Resource Identifier) or a file. Supported operations include: launching the user-default browser to show a specified URI; launching the user-default mail client with an optional `mailto` URI; launching a registered application to open, edit or print a specified file.

```

if (Desktop.isDesktopSupported()) {
    Desktop.getDesktop().browse(new URI("http://www.example.com"));
}

```

Available `Desktop` methods include: `browse()` which launches the system default browser, `edit()` which launches the associated editor application and opens the

file, *getDesktop()* which returns the Desktop instance of the current browser content, *isDesktopSupported()* which determines if the current desktop is supported, *mail()* which opens the default mail client and opens a mail window, *open()* which opens a file with the associated application, and *print()* which prints a file in the desktop default printing application using the file's associated application's print command.

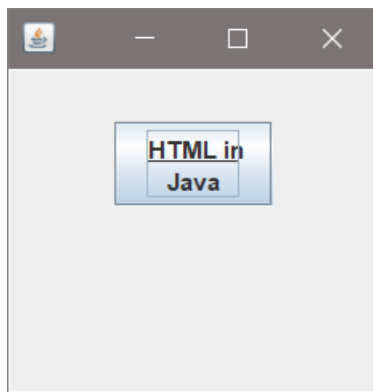
## HTML in Java

To mix fonts or colors within text, or for formatting such as multiple lines, HTML can be used in Java. HTML formatting can be used in all Swing buttons, menu items, labels, tool tips, and tabbed panes, as well as in components such as tables that use labels to render text.

To specify that a component's text has HTML formatting, put the `<html>` tag at the beginning of the text, then use any valid HTML in the remainder. Below is an example of using HTML in a button's text.

```
JButton button1 =
```

```
    new JButton("<html><center><u>HTML in</u><br>Java</center></html>");
```



HTML in Java Example

## Animation

Implementing animation in Java can be accomplished with a sequence of images and the AWT Timer, or with the JavaFX 2.2 Animation class which simplifies some of the operations. The code below shows the declaration of a timer with a 125 millisecond delay between ticks. At each tick of the timer, a `TimerListener()`

reacts to the clicks of the timer by calling the `repaint()` method. The `paintComponent()` calls methods to update the images or whatever is being painted.

```
Timer myTimer = new Timer(125, new TimerListener());
myTimer.start();
```

```
class TimerListener implements ActionListener
{
    public void actionPerformed(ActionEvent ae)
    {
        myPanel.repaint();
    }
};
```

```
JComponent component1 = new JComponent() {
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        // change the image, move x and y coordinates to
        // animate a drawn entity, or resize something
    }
};
```

## Repainting

A cautionary note about repainting: in Java, calling `repaint()` may not result in the component or applet window being repainted. The interpreter will ignore calls to `repaint()` if it can't process them as quickly as they are being called, or if some other task is taking up most of its time.



## Appendix A

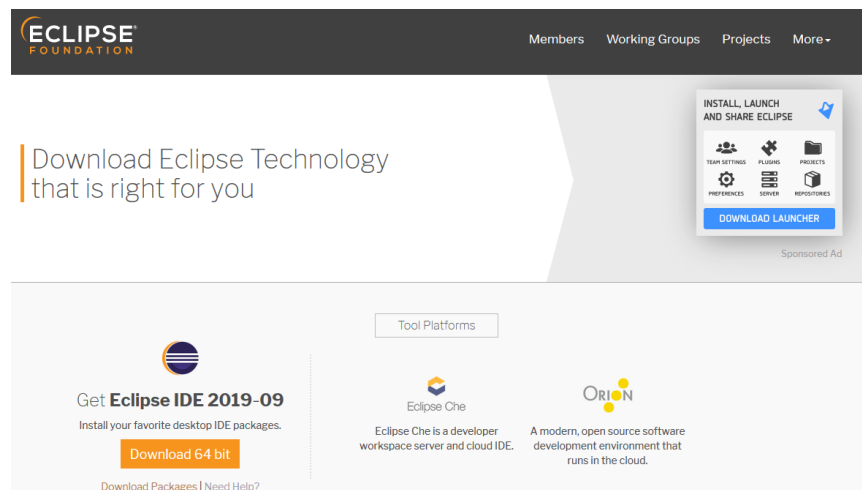
### Obtaining Eclipse

- Eclipse is available from Eclipse.org <https://www.eclipse.org/>
- Eclipse will run on most machines
- The JRE and JDK will be installed with Eclipse
- Eclipse will run fine on a flash drive for portability and access
  - Copying the JRE to the flash drive simplifies running

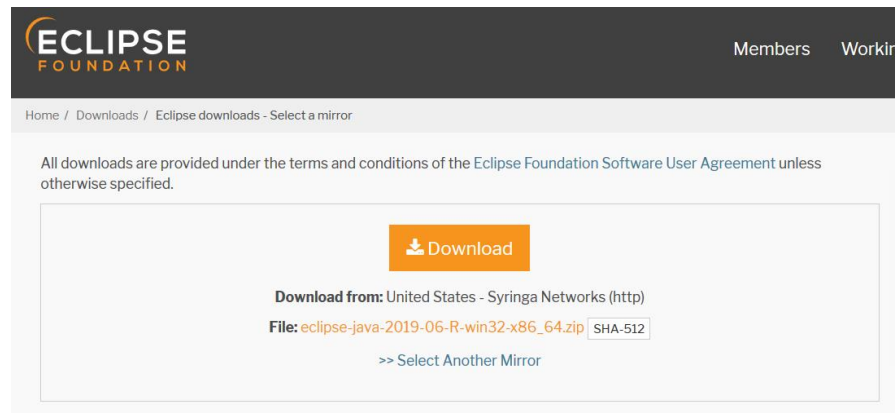
Browse to the Eclipse web site shown here and select “Downloads”.



From the Downloads window shown select the appropriate version for your computer. On most Windows machines, select the “Download 64 bit” button.

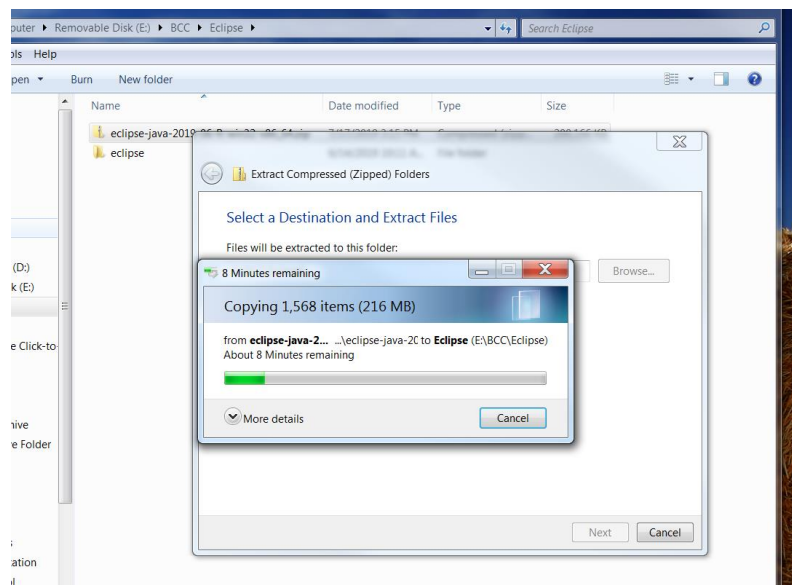


## Appendix A



Download or save the zip file. (Eclipse will run fine on a flash drive and can be installed there if you prefer.)

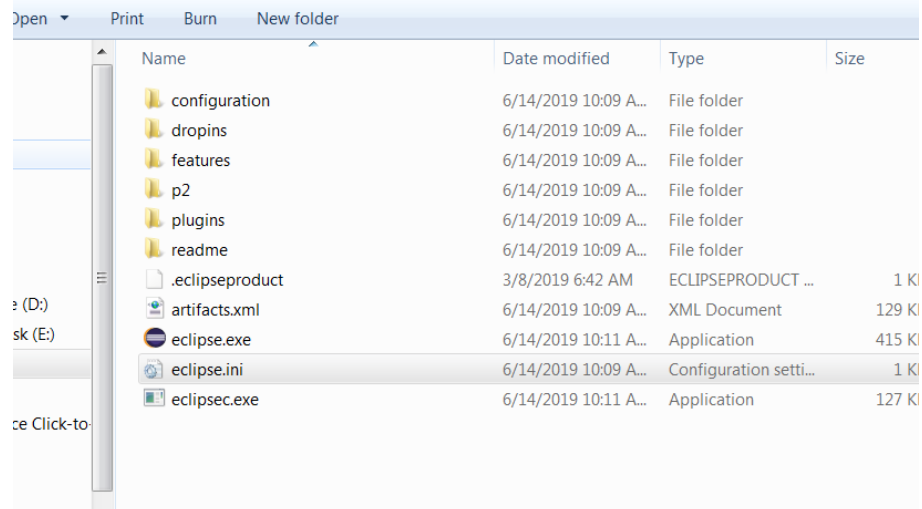
Once the zip file downloads, create a folder called “Eclipse” on your drive or flash drive and place the zip file there, then right mouse click and “extract all” to that folder. This will take a while...



The folders and files shown below are installed with Eclipse. The eclipse.exe file launches the program.

---

## Appendix A



### The Java Development Kit

Installing the JDK in this directory with Eclipse will ensure that Eclipse will always have (find) the JRE as well. The issues below usually have to do with Eclipse not finding supporting files.

### Launch Eclipse

If you launch Eclipse and get **exit code 13** (shown below) or the “A Java Runtime Environment...” error (shown below), Eclipse cannot find the JRE (Java Runtime Environment) or jdk (Java Development Kit).

You may need to add the following code before the line that includes **-vmargs** in the **eclipse.ini** file.

```
-vm
```

```
C:\Program Files\Java\jdk1.7.0_40-64\bin\javaw.exe
```

**Note:** The second line may be different depending upon version of the java jdk installed in your machine, or if you are pointing it to a jdk on your flash drive.

- **A Few important points to remember while configuring eclipse.ini file:**
    1. The Java File's Path must be Relative Path or Absolute Path. It should not just point to the Java Home Folder.
    2. The **-vm** option and its path should be on a separate line.
    3. The **-vm** option should be before **-vmargs**
-

## Appendix A

## Error...exit code=13



## Error...Eclipse cannot find the JRE or JDK. Notice that it looked for it.

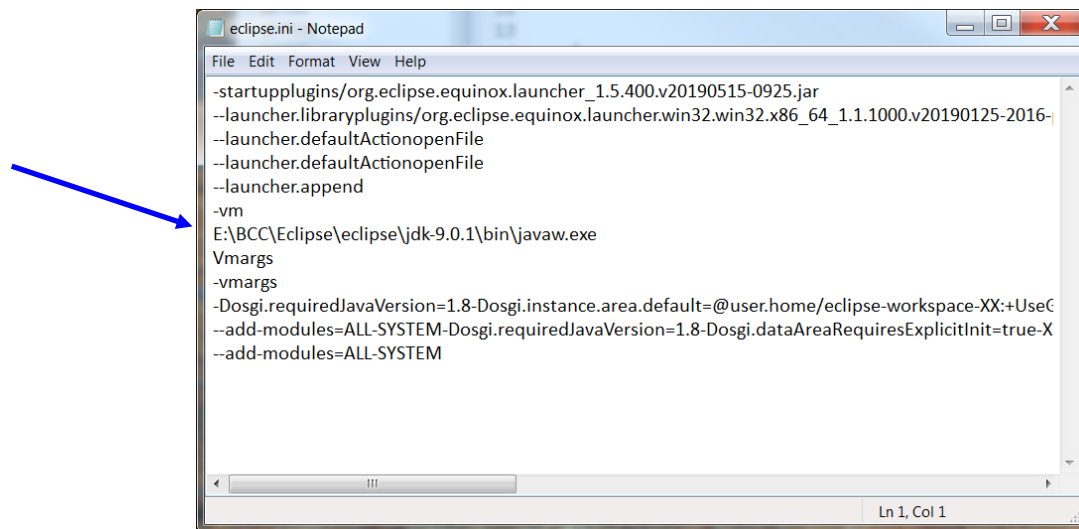


## Appendix A

The eclipse.ini file is an initialization file that tells Eclipse where to find things like the jdk and the location of your last used Workspace. To modify it, use a text editor like Notepad

Be sure that java is installed on your machine. Check the Program Files directory.

- **The jdk (Java Development Kit) is used by the Eclipse.**
- **If you install on a flash drive, it is easier to place a copy of the jdk in the Eclipse folder on the flash drive and point the eclipse.ini file to that folder.**



```
eclipse.ini - Notepad
File Edit Format View Help
-startupplugins/org.eclipse.equinox.launcher_1.5.400.v20190515-0925.jar
--launcher.libraryplugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.1000.v20190125-2016-
--launcher.defaultActionopenFile
--launcher.defaultActionopenFile
--launcher.append
-vm
E:\BCC\Eclipse\eclipse\jdk-9.0.1\bin\javaw.exe
Vmargs
-vmargs
-Dosgi.requiredJavaVersion=1.8-Dosgi.instance.area.default=@user.home/eclipse-workspace-XX:+UseC
--add-modules=ALL-SYSTEM-Dosgi.requiredJavaVersion=1.8-Dosgi.dataAreaRequiresExplicitInit=true-X
--add-modules=ALL-SYSTEM
Ln 1, Col 1
```

- You need to add the path to javaw.exe in the eclipse.ini file and it must be before the line that includes **-vmargs**. Find the jdk on your machine or flash drive and open bin to find javaw.exe. Use the full path for the eclipse.ini file.

*-vm*

*E:\Eclipse\jdk-9.0.1\bin\javaw.exe*

- The second line above will be different depending upon the version of the java jdk installed and the directory (folder) where you placed the jdk.

### Important point to remember:

- When you use the flash drive in a different machine, note the drive letter for the flash drive. In the example above the drive letter is "E", but may be "D" or another letter depending on the machine. If this is the case, open the eclipse.ini
-

## Appendix A

file and change the drive letter...just remember to change it back when you move to another machine.

- **Help Documentation is available:**

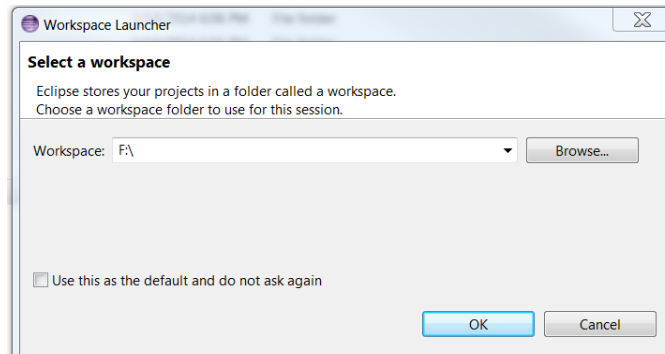
<http://www.eclipse.org/users/>

---

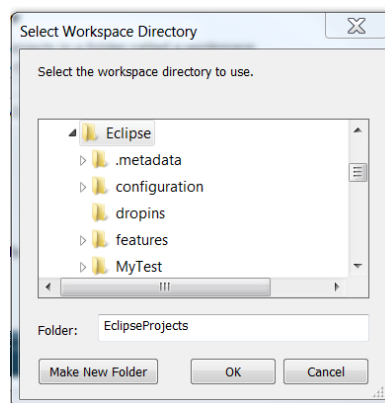
## Appendix B

### Getting Started in Eclipse

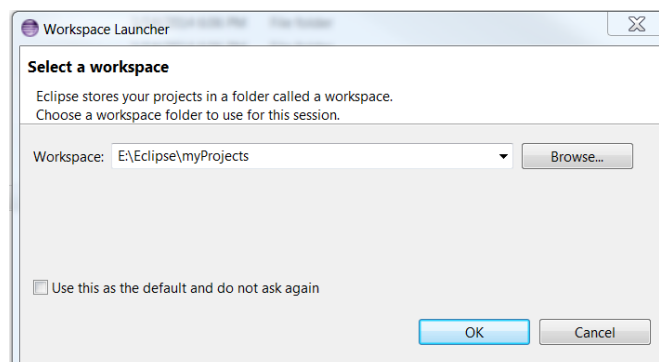
When Eclipse is launched for the first time, a “Workspace” needs to be created. The Workspace organizes programs and projects and adds supporting files. The next time Eclipse starts, the Workspace will be shown as the default in this window.



Choose the “Browse” button, and decide where the program files will be stored. After choosing a directory, select the “Make New Folder” button. Name the folder (something like projects) and select “OK”.

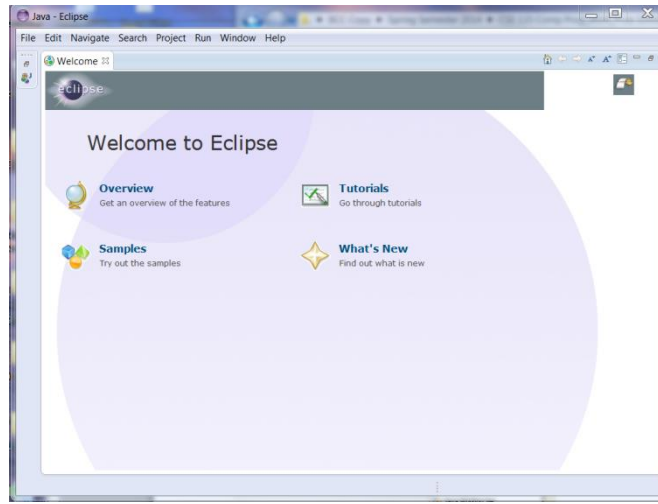


Then, back in the “Workspace Launcher” window, select ‘OK’ again.

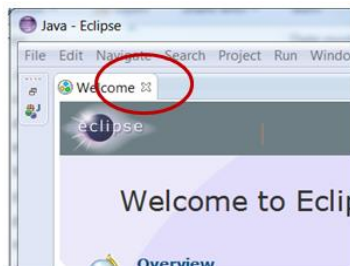


## Appendix B

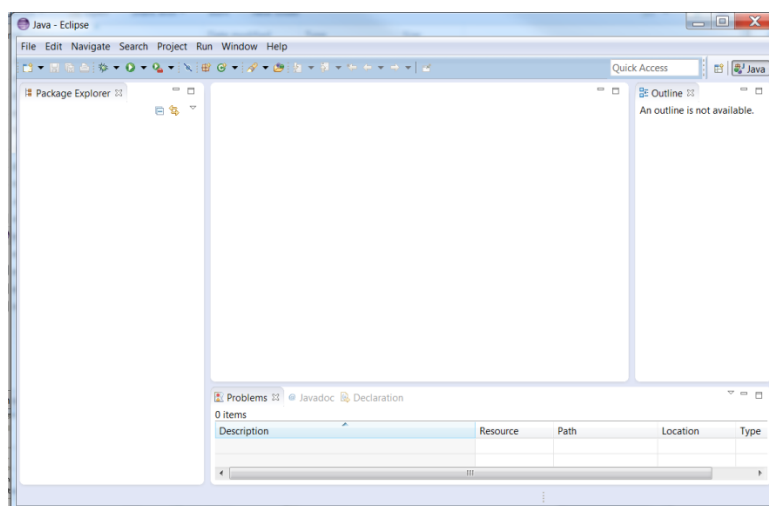
Eclipse will open to the “Welcome” window.



Close the “Welcome” window by clicking the “X” (top left).



The IDE will be displayed.



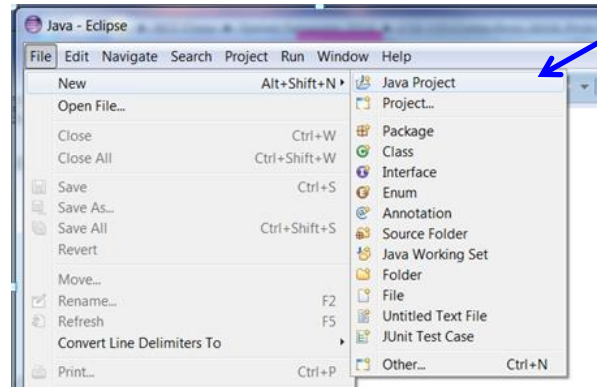


## Appendix B

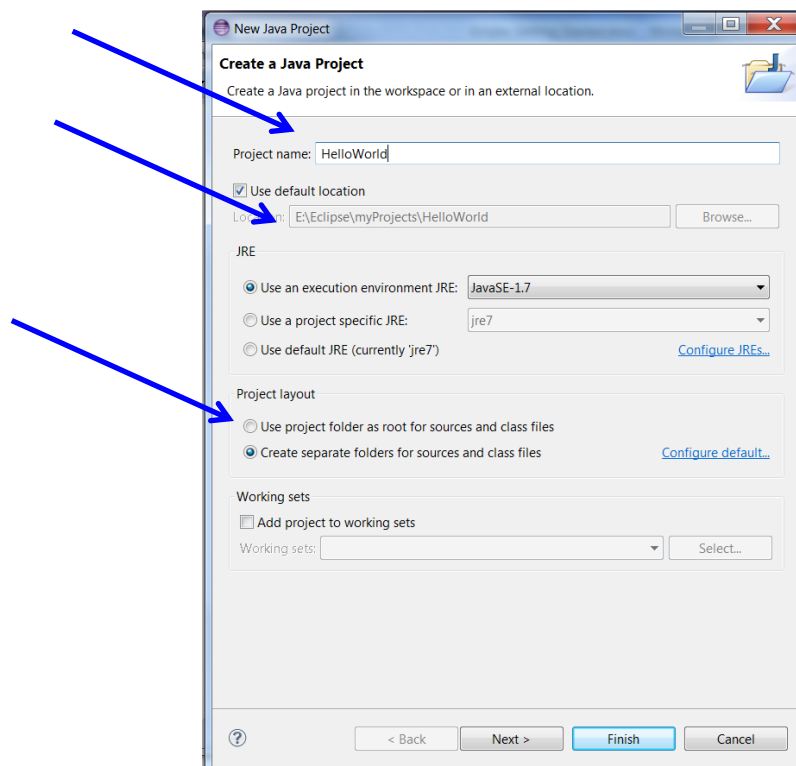
### Creating a Project

Creating a “**Project**” and not a “**File**” is important for creating supporting files and how the Workspace handles them. This will be clearer the next time Eclipse is launched and the Workspace is selected. All of the projects in the Workspace and their supporting file will be loaded automatically.

Select **File** | **New** | **Java Project**

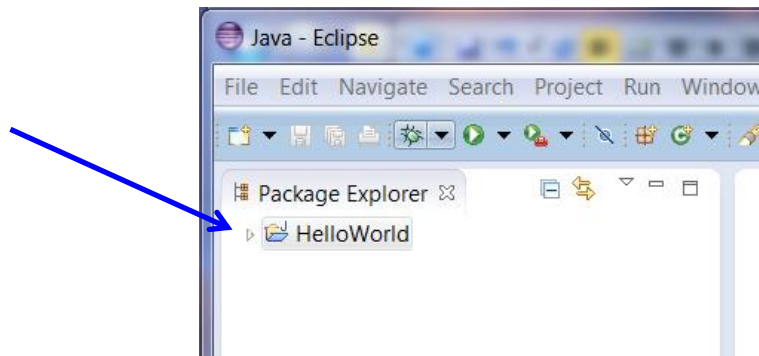


The ‘Create a Java Project’ box will appear. Give the project a name, “HelloWorld” in the example. As it is typed, the location box will add the text. Select the “Use project folder...” radio button. Then click “Finish”.

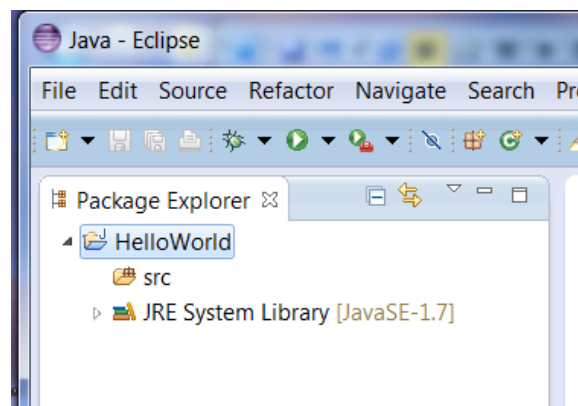


## Appendix B

The project will appear in the Package Explorer on the left side of the IDE. Expand it by clicking the triangle.



The "src" and "JRE System Library" may be shown. Leave the project name highlighted as shown here.

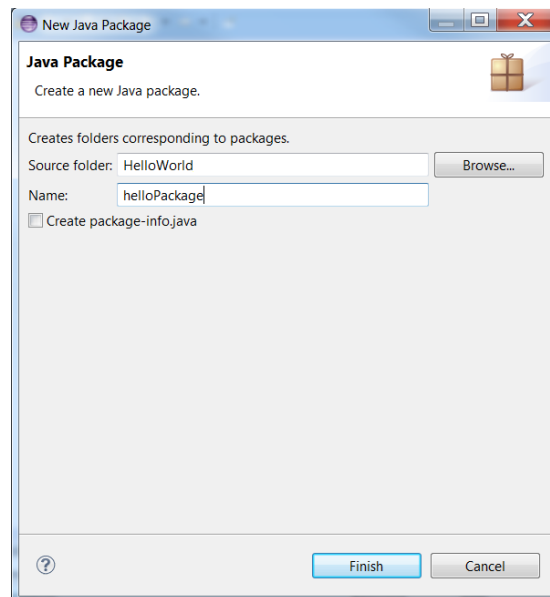


The first step to programming in Java is to create a "package" which will contain the project files. With the **project name still highlighted**, click on the package icon.

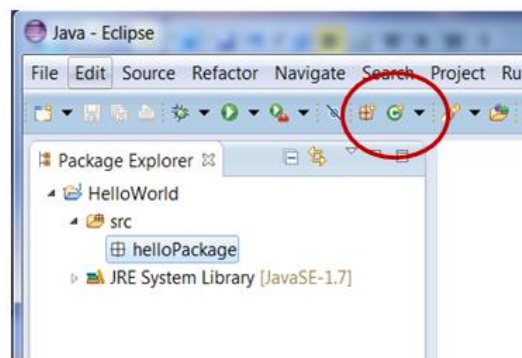


## Appendix B

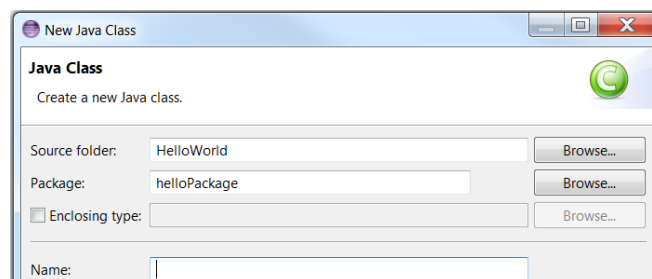
The New Java Package window will appear. Give the package a name that is relevant to the project (helloPackage shown here). Then click “Finish”.



The package will now appear in the project explorer. Be sure that the **package is still highlighted** and select the class icon circled below.



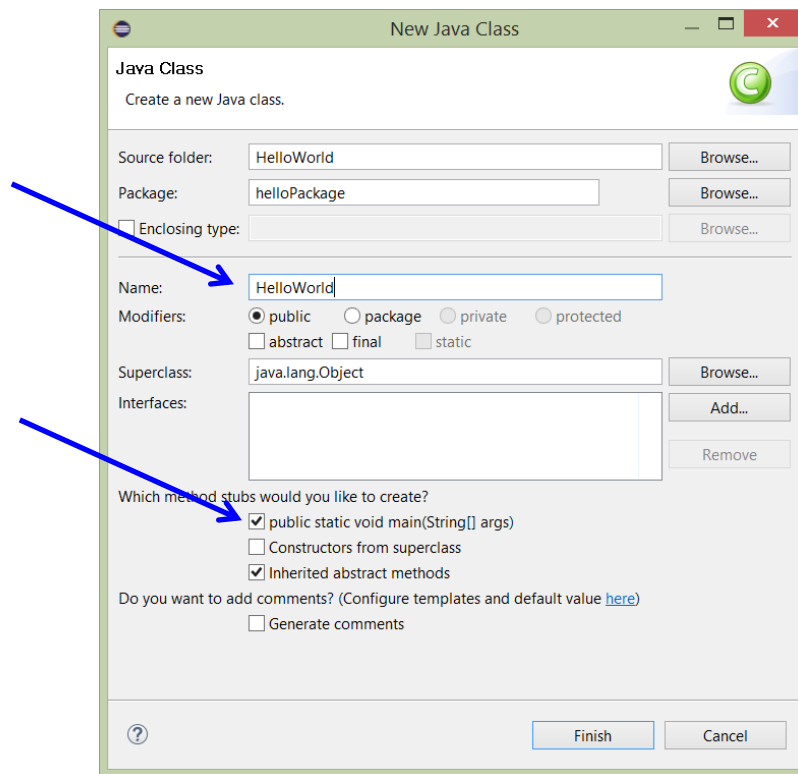
The class creation window will appear.



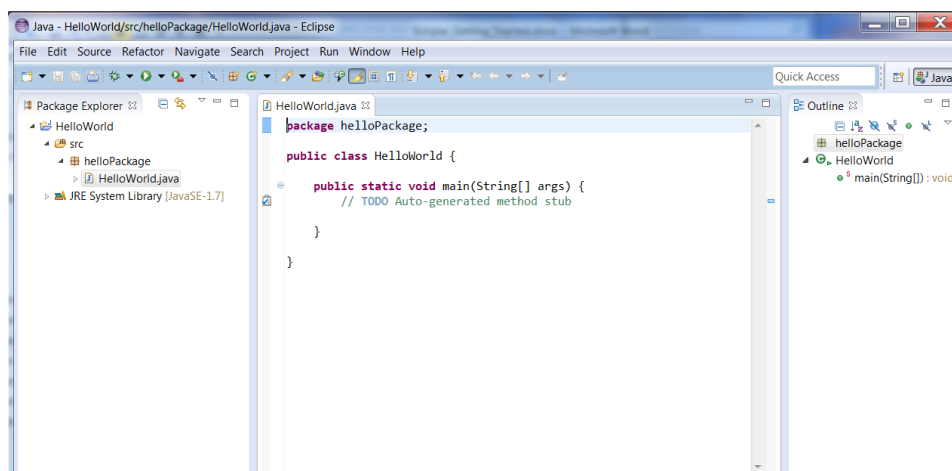
## Appendix B

Give the class the **same** name as the project name that was chosen earlier. In the screen capture below, notice that the “Source Folder” name and the class name are the same.

**Check** the “public static void main(String[] args)” box, and click the “Finish” button.

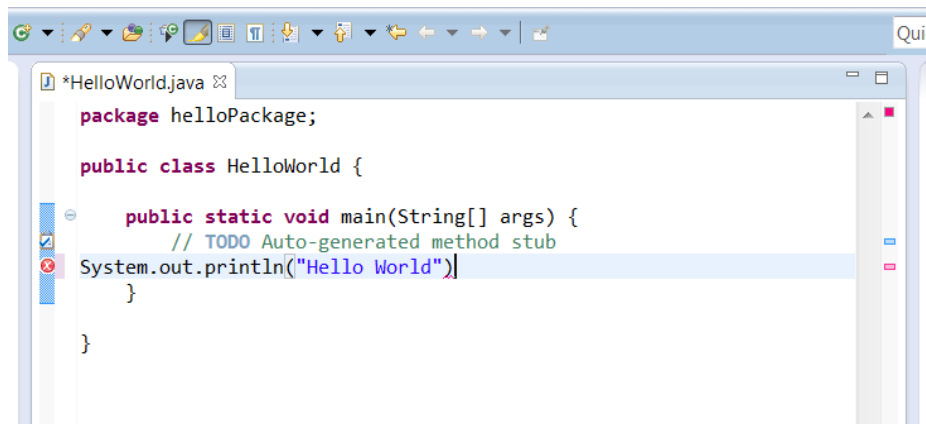


The project is now created with a package and a class, and the main method has been added to the program.



## Appendix B

Add the output line of code shown below. The red circle containing the white “x” at the margin indicates an error on the line (the semicolon at the end of the line is missing).



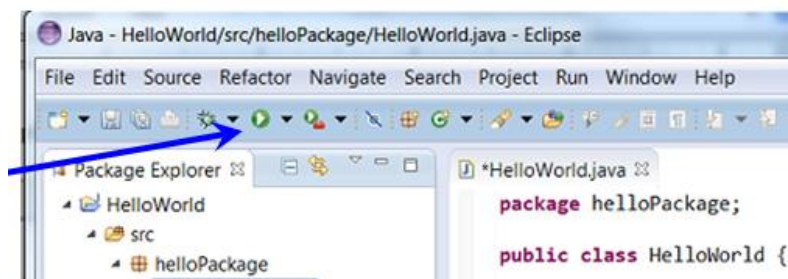
```
*HelloWorld.java
package helloPackage;

public class HelloWorld {

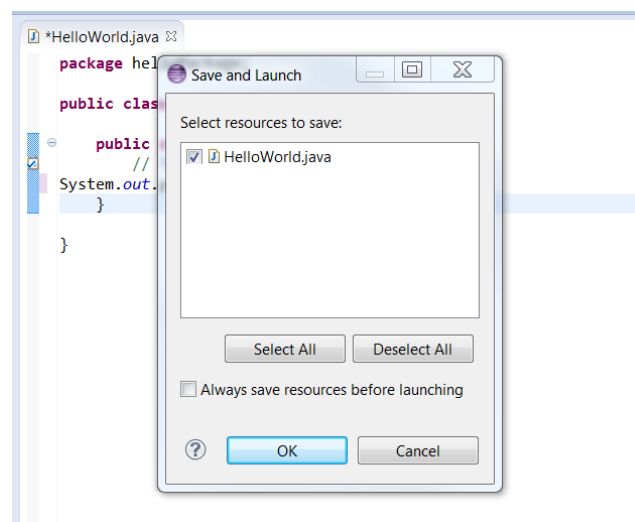
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Hello World")
    }

}
```

After adding the semicolon, run the program by clicking on the green circle with a white triangle inside.

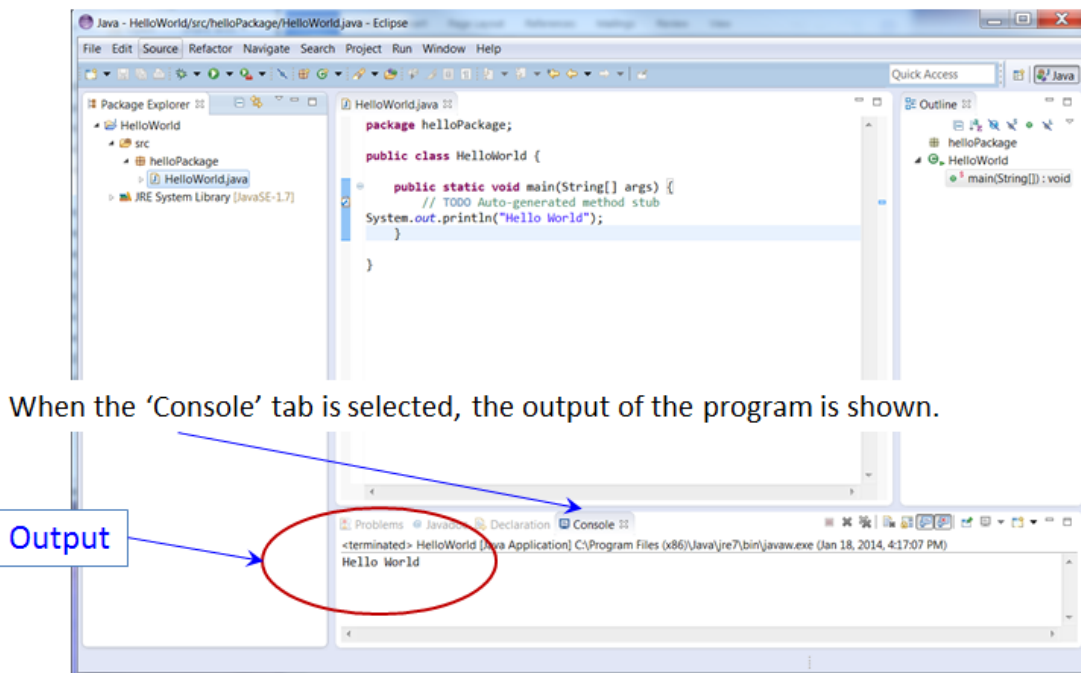


The ‘Save and Launch’ window will be displayed. Eclipse ensures that changes are saved before the program runs.



## Appendix B

HelloWorld.java is already selected so just click the 'OK' button, and the program will run. That's it.



## Eclipse – Quick Start

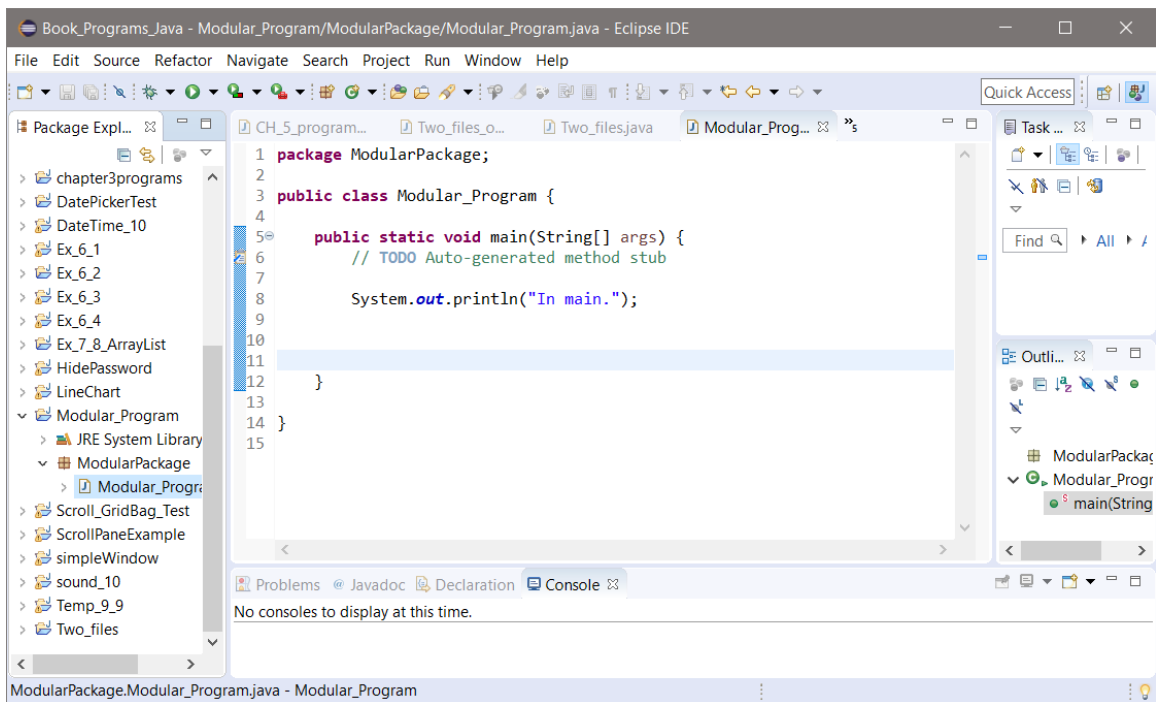
- Launch Eclipse, select the workspace folder from the list, Eclipse will start
- Close the Welcome window by clicking on the 'X', and the IDE will open
- Select **File > New > Java Project**
  - The 'Create a Java Project' box will popup
  - Name the project, and the project will appear in the Package Explorer
- With the project name highlighted, **add a Package** by clicking the 'New Java Package' icon, and give it a name.
- With the package name highlighted, **add a class** by clicking the 'New Java Class' icon, and the class creation window will popup.
- Give the class a name the same as the Source/Project name.
  - **Check the 'public static void main(String[] args)' box**
- Click on the 'Finish' button

## Appendix B

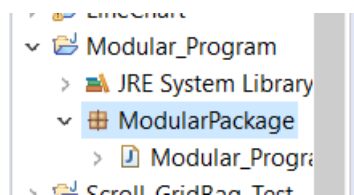
### Creating a Second File – Modular Programming

Multiple files can be used to separate various parts of the program. By creating modules (files), the program is easier to maintain and add functionality, portions can be easily reused, and multiple engineers can work on various parts of a large program at the same time.

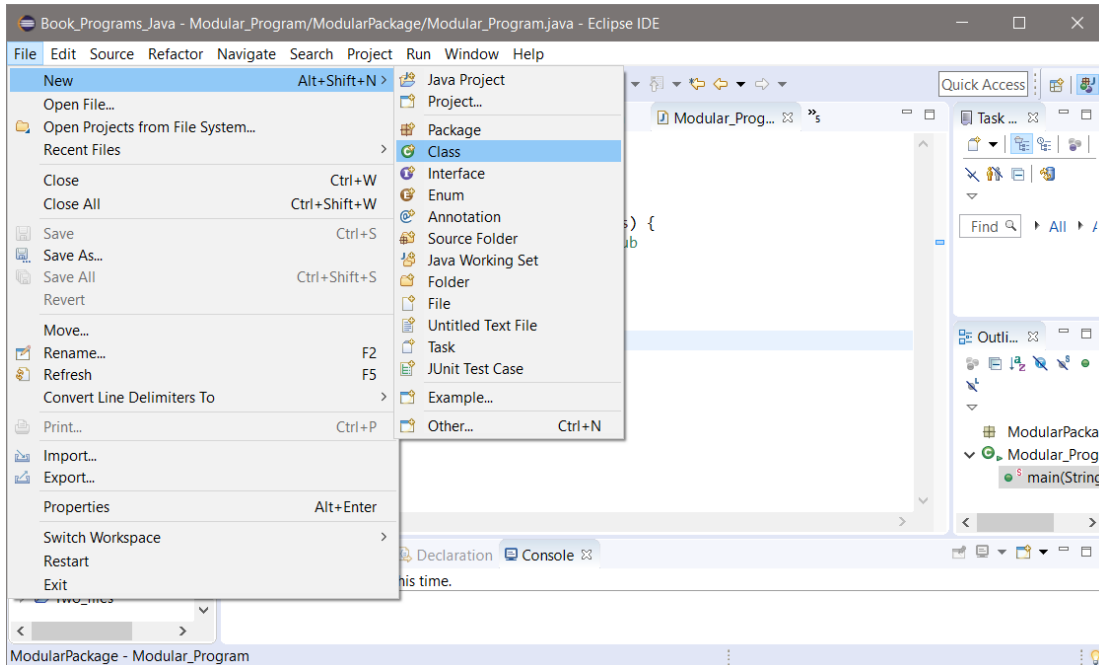
The example creates a method in a second file, which is imported and used in the program. The main program has been created “Modular\_Program” in the “ModularPackage”.



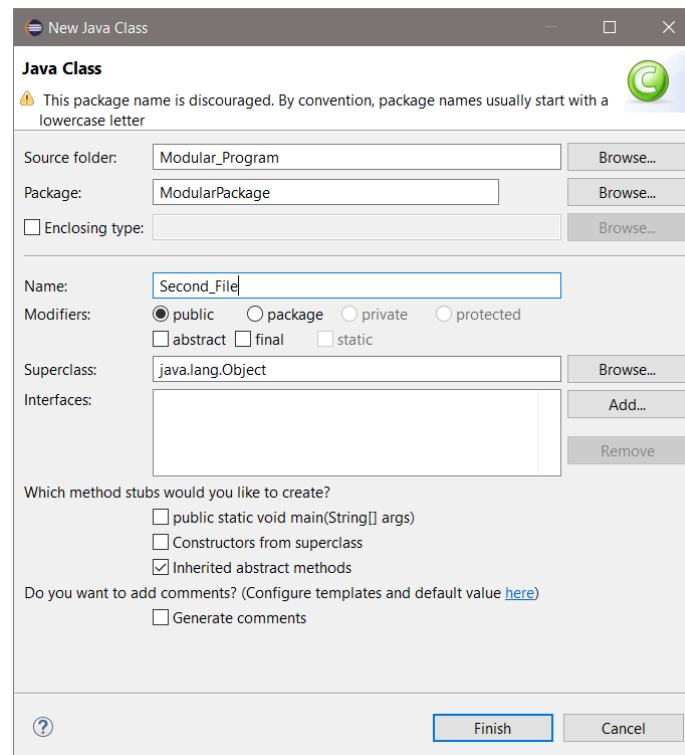
Next, with the package for the program highlighted in the Package Explorer (click on it), select File | New | Class.



## Appendix B



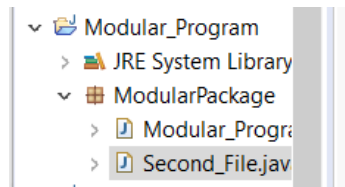
The Java Class window will appear. In the example, the class has been named `Second_File` and the check box for `public static void main(String[] args)` is not checked.



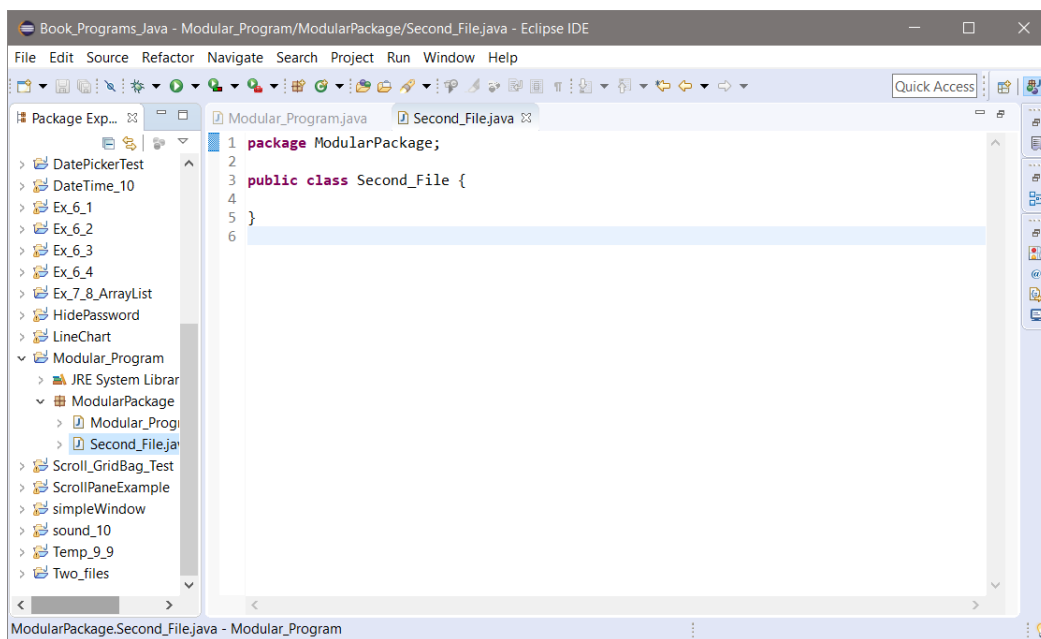


## Appendix B

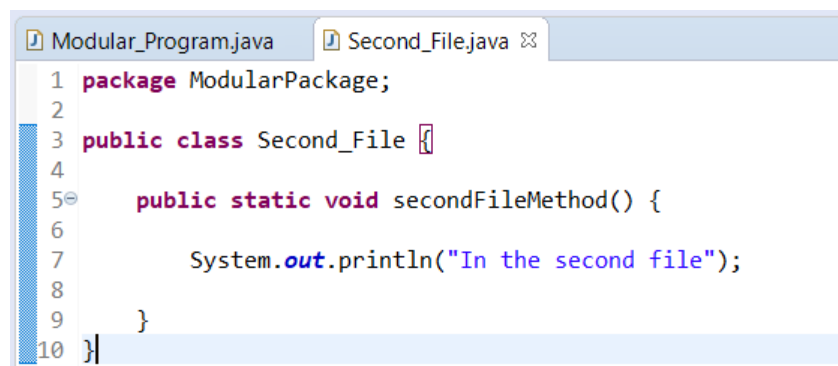
Click the Finish button, and the file is added to the program as shown in the Package Explorer.



There are now two tabs for the program in the IDE.



For the example, a method with an output statement is added to the second file.

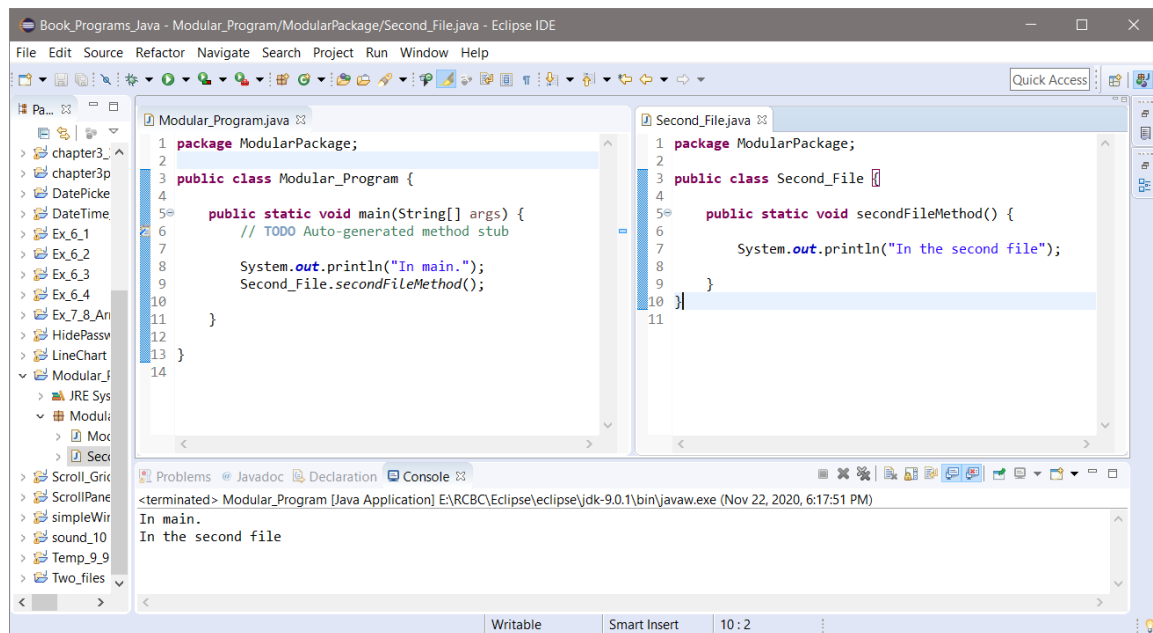


## Appendix B

The method is then called from main using the class name and dot operator.

```
*Modular_Program.java  Second_File.java
1 package ModularPackage;
2
3 public class Modular_Program {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7
8         System.out.println("In main.");
9         Second_File.secondFileMethod();
10
11     }
12
13 }
```

By dragging the Second\_File.java tab to the right of the edit panel, both files are shown at once. The program ran and produced the expected output.



## Appendix C

### **Weather Analysis Program in Java**

Design and create a GUI program in Java for a meteorologist that calculates the wind chill factor and cloud base altitude for inputs of temperature in Fahrenheit, wind speed in mph, and the dew point in Fahrenheit, displays the data in the main interface, and a data display window, and plots the temperature and wind chill values.

When the program begins, a window with three (3) buttons will allow the user to: Create Account, Login, or Cancel. Account creation will be handled in a separate window using a class in a separate module. Accounts require a unique username and a password of at least nine (9) characters, with at least one digit, uppercase and lowercase letter. The program will handle error messages and prompt for a password until a valid password is created. Valid account information will be stored for retrieval for future login. The Login operation will verify the account, and after successful Login, the main interface will be displayed and have data entry and selection controls.

When data is entered from the keyboard and a compute button is clicked, the input will be validated with error handling and dialogs. If data entered from the keyboard is invalid, an error dialog message will be displayed, the output display windows will not be updated, and the program will continue.

When valid data is entered, the computed results will be displayed in the main (GUI) data entry window, and a column aligned and formatted data output window with titles, column headers, units, commas separators for thousands, and decimal places. The computation results will also be plotted in a separate window as noted below.

The user entry controls will accept input right-aligned, and there will be functionality to save the data in the output window and access it with open file functionality. When open file is selected, a file selection window will be displayed. File opening errors will be handled with exceptions and dialog boxes.

*The screen captures in this document are for reference only. The design and interface of your program do not have to mirror the examples in this document in terms of appearance and controls used, but must handle the operations.*

### **Program Milestones**

#### **Milestone #1 – Initial GUI – Create Account/Login/Cancel**

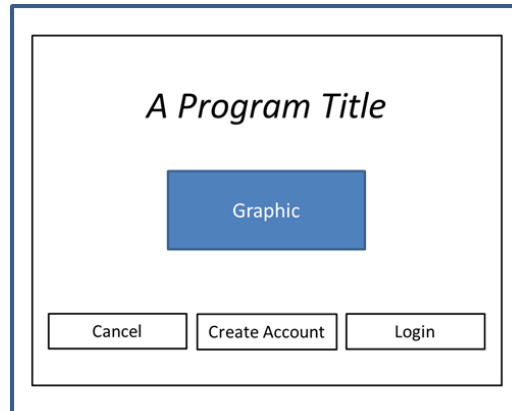
Obtain a copy of Eclipse and complete “Getting Started in Eclipse”. Create your project in Eclipse and the package and class. Design and develop the Create Account/Login/Cancel interface. The buttons should function and create the appropriate windows from classes. Full functionality of

---

## Appendix C

create account and login is not required until the next milestone, but the interfaces should appear when the buttons are clicked.

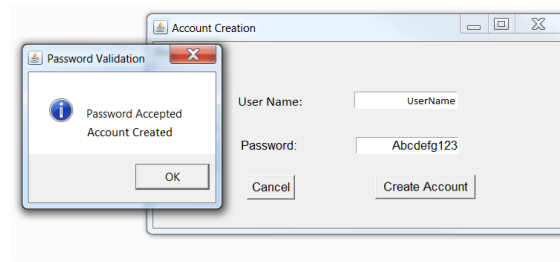
The window should contain at a minimum, “Create Account”, “Login”, and “Cancel” buttons. Graphics enhance the interface for the user. Create the Design Document (Word/pdf) with descriptions, screen captures, and code at the end and submit it as Milestone #1.



### Milestone #2 – Create Account, Login, and Main Interface

Complete the account creation functionality for username and password, validating the password for length (9 characters or more), an uppercase and lowercase letter, and a digit. The program must handle invalid input. The window should contain instructions for the user. Complete the username and password algorithm by saving the data and storing it in a file for retrieval by the login function. (File handling is covered in chapter 6 of the text).

Complete the login operation and validation of the username and password for an account.



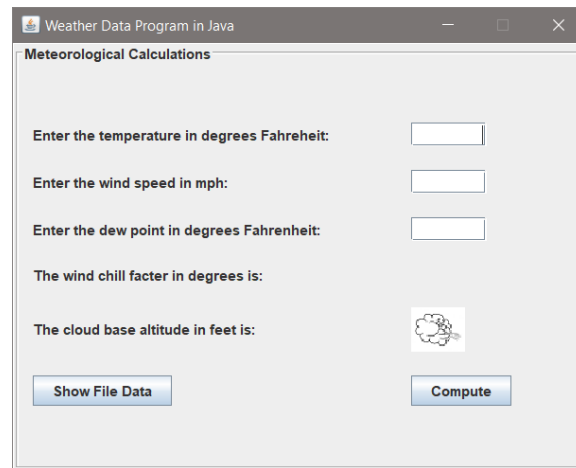
Design the main interface including the components (all functionality is not required at this point). Consider how the program will handle various situations, and how will the user interact with the program. Implement display of the main interface for the project after login occurs, including any data entry controls and buttons appropriately positioned and aligned.

Clean up the appearance of the program including the various displays and window handling. Windows should appear centered on the desktop, and not hide one another incorrectly. Components should be aligned or centered with text explaining their functionality.

---

## Appendix C

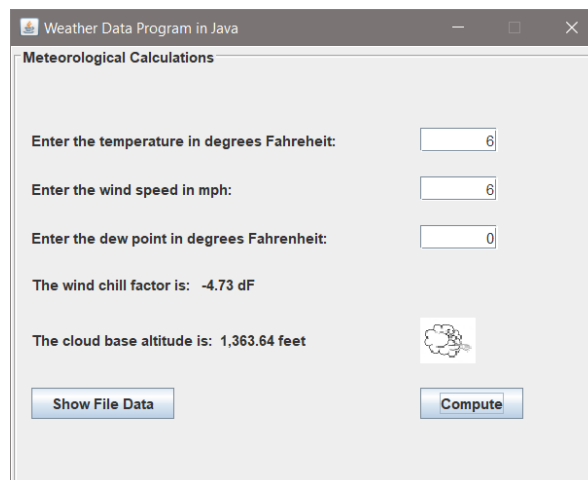
The design and development of the main interface should be nearly complete at this point. Consider how the user will interact with the program and the order of operations.



Update the Design document with screen captures and explanations of operation, and submit the Design Document with the code at the end and submit Milestone #2.

### Milestone #3 – Functioning Keyboard Entry with Input Validation

Complete the main interface and keyboard entry functionality with error handling. Implement the output on the main interface of the computed results from the keyboard entered values. Functionality needed by other areas of the program should be in methods/classes. Error dialogs can be used for bad input or when a wind chill is not valid, and the text in the interface should reflect the issue.



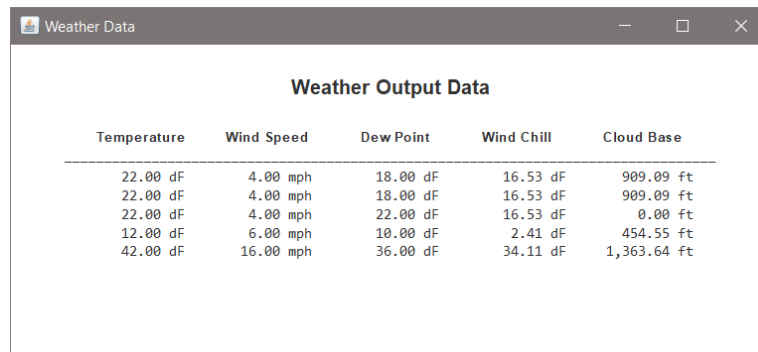
Update the Design document and Present the documentation and running program in class (if face-to-face instruction), and submit the Design Document with screen captures of functionality and code at the end and submit it for Milestone #3.

---

## Appendix C

### Milestone #4 – Output Display for Keyboard Entry

Create the keyboard entry data output display window as a class, which will update each time new values are entered on the main GUI. Formatting of the data is required. The data display window will append the data each time new values are entered. Update the Design document with screen captures and explanations of operation, and submit the Design Document with the code as Milestone #4.



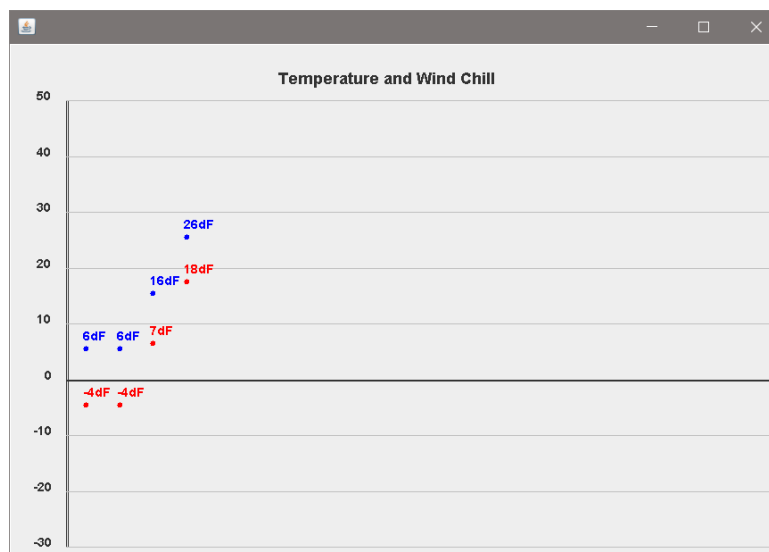
The screenshot shows a window titled "Weather Data" with a table titled "Weather Output Data". The table has five columns: Temperature, Wind Speed, Dew Point, Wind Chill, and Cloud Base. The data is as follows:

Temperature	Wind Speed	Dew Point	Wind Chill	Cloud Base
22.00 dF	4.00 mph	18.00 dF	16.53 dF	909.09 ft
22.00 dF	4.00 mph	18.00 dF	16.53 dF	909.09 ft
22.00 dF	4.00 mph	22.00 dF	16.53 dF	0.00 ft
12.00 dF	6.00 mph	10.00 dF	2.41 dF	454.55 ft
42.00 dF	16.00 mph	36.00 dF	34.11 dF	1,363.64 ft

### Milestone #5 – File Chooser, File Data Output, and Plotting

Implement the “File Save” operation to allow saving an analysis from the output window (a JMenu on the window can be used with a “Save As” dialog). Implement the file open functionality using a JFileChooser to view the saved data and handle exceptions. The data file should have column headers and appear similar to the output display.

Implement the plot functionality for temperature and wind chill when values are entered on the keyboard. A separate window is required, with a different color indicator and text for the two values. Each time a data set is entered, and the compute button is clicked, the window should be updated with the new data. A line, bar, or point chart can be implemented.



## Appendix C

Update the Design document and prepare and present the documentation and program in class (if face-to-face instruction). Prepare the Design Document for final submission and submit the Design Document with screen captures of operations and code at the end as Milestone #5.

### Final Document Submission

The final document should be polished and professional. It should have a cover sheet, centered screen captures of consistent size, with proper use of fonts and spacing. The document should represent a step-by-step journal of design and development of the milestones. The quality should be such that it could be presented to a prospective employer as a work sample.

### Equations

The equation for approximating the **wind chill** factor in North America is:

$$wc = 35.74 + 0.6215 T_a - 35.75V^{0.16} + 0.4275 T_a V^{0.16}$$

Where  $T_a$  is the air temperature in Fahrenheit

$V$  is the wind speed in mph (*consider* pow(windSpeed, 0.16))

Also, wind chill temperature is defined only at or below 10.0° C (50.0° F), and wind speeds above 4.8 kilometers per hour (3.0 mph). The program must check this.

The **cloud base** in feet above ground level is determined by the “temperature spread” which is the difference between the temperature and the dew point, and is calculated:

$$cloudBase = \text{temperature spread} / 4.4 * 1000$$

---

## Appendix D

### Resource Links



Eclipse.org

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

Eclipse User Guide

<https://help.eclipse.org/2019-09/index.jsp>

Java Development Guidelines – Carnegie Mellon University:

<https://wiki.sei.cmu.edu/confluence/display/java/Java+Coding+Guidelines>

Java Downloads and Information:

<https://www.java.com/en/>

JFreeChart charting tool:

<http://www.jfree.org/jfreechart/samples.html>

W3Schools Java Tutorial:

<https://www.w3schools.com/java/default.asp>

---



# Appendix D



## Appendix E

### Java Software Engineering Standards

Software Engineering standards provide a critically consistent way of designing and developing computer based solutions that reduce errors, debugging time, maintenance costs, and ensure a consistency across the organization. Virtually all businesses (including Microsoft, NASA, all Defense Contractors, NOAA, et.al) impose standards similar to those listed here. The following standards including style and techniques shall be utilized when writing programs. Note the emphasis on technique and style in the quotes below.

*“Superior coding techniques and programming practices are hallmarks of a professional programmer.”* – Bob Caron, **Microsoft**

*“The purpose of the process is to develop source code that is traceable, verifiable, consistent, and correctly implements the requirements.”* – **NASA Langley**

#### Variable naming conventions

Variables shall be declared using descriptive names. A single letter or ambiguous abbreviation is unacceptable unless local to a method when no ambiguity is introduced (see below). Uppercasing shall be used and be consistent. Variables should be declared together whenever possible. Declaring a variable when needed increases maintenance time.

##### Unacceptable:

*type p, val, t;*

##### Acceptable:

*type grossPay, salesValue, buttonWidth;*

Inside a method, variables with local scope may be declared using an alias when no ambiguity is created and no clarity is lost.

```
// This method returns the average of three arguments
public static double average(double x, double y, double z) {

    double avg = (x + y +z) / 3.0;
    return avg;
}
```

#### Constants

Constants are declared as static and final. Names for constants are all uppercase letters with underscores between words. Constants must be initialized when declared.

```
public static final double MIN_HEIGHT = 1.0;
```

---

## Appendix E

### Method Naming Conventions

Methods shall have descriptive names that describe what they accomplish. A single letter or ambiguous abbreviation is unacceptable. The uppercasing convention shall be used.

#### Unacceptable

```
cb(double t, double wc);
```

#### Acceptable

```
computeCloudbase(double t, double wc);
```

Variables needed internally by the method shall be declared inside the method (local variables). Temporary variables local to methods shall be used to store complex computed values and as the return variable. Return statements should not contain computations, and are only permitted in class member functions. This significantly enhances the ability to debug and maintain a program. An example is shown below.

```
// Returns the circumference of a circle or
// zero if a negative or zero radius is received.
public static double getCircumference(double r)
{
    double c = 0;

    if (r > 0)
        c = 2 * pi * r;
    return c;
}
```

### White Space, Indentation, and Blank Lines

Spacing between operators and variables, and sections of code enhance the readability and maintainability of the code as well as reducing the number of errors and debugging time. Spaces shall be used between operators, literals, and variables, and between data types and variables.

#### Unacceptable

```
totalPrice=price+price*salesTax;
```

```
for(int i=0;i<str.length();i++)
```

#### Acceptable

```
totalPrice = price + price * salesTax;
```

```
for(int i = 0; i < str.length(); i++)
```

Indentation shall be used to emphasize grouping and align sections of code. Logical sections of code shall be separated with blank lines to add clarity for scalability and maintainability. Closing braces should be on separate lines.

---

## Appendix E

### Unacceptable

```
public static int countWords(String str) {
    int count = 1;
    Boolean justSawOne = false;
    for(int i = 0; i < str.length(); i++) {
        char ch = str.charAt(i);
        if(ch == " && justSawOne == false) {
            justSawOne = true;
            count++;
        } else {
            justSawOne = false; }
        return count;
    }
}
```

### Acceptable:

```
public static int countWords(String str) {

    int count = 1;
    Boolean justSawOne = false;

    for(int l = 0; l < str.length(); i++) {
        char ch = str.charAt(i);

        if(ch == " && justSawOne == false) {
            justSawOne = true;
            count++;
        }
        else {
            justSawOne = false;
        }

    return count;
}
```

### **Commenting Code**

Comments shall be used to explain functions, computations, the reasoning behind design choices, and the use of literals. Comments shall be used inline and tabbed right or above code, but not to explain poorly written code. Comments describing functions shall be above the function and left justified.

---

## Appendix E

### **Classes and interfaces**

The first letter of the name shall be capitalized, and the first letter of each additional word.

```
public class MeteorWin()
```

There can be only one public class in a file, and the name of the file must match the name of the public class. For example, a class declared as “public class MeteorWin()” must be in a source code file named MeteorWin.java.

### **Program Layout/Logic**

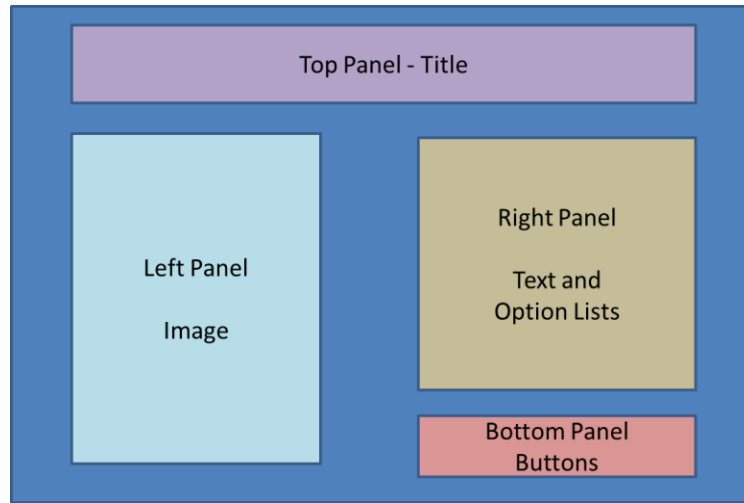
Programs shall be developed in a logical and organized manner. The order of operations shall be easy to determine and follow by anyone viewing the code. Programming should be deliberate and anticipate that another programmer will be reading the code in the future.

---

## Appendix F

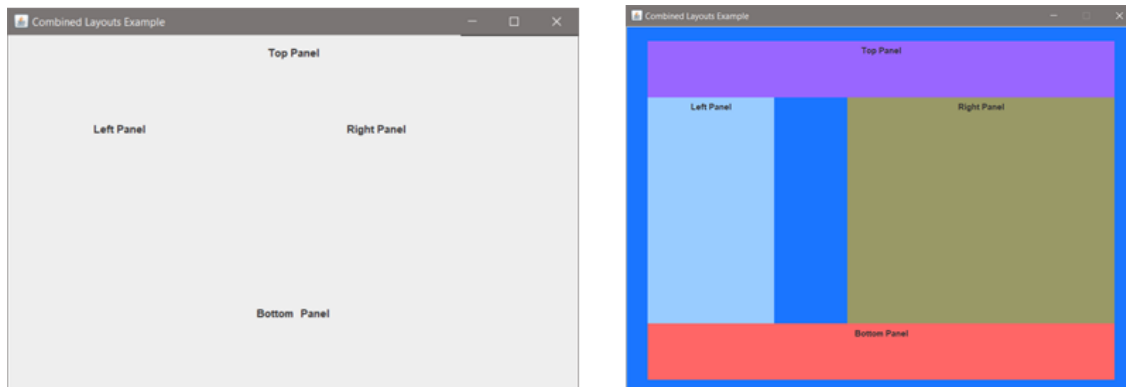
### **Multiple Panels and Layout Managers Example**

Design and creation of an interface requires careful consideration, and a design sketch can be a valuable tool for component placement and development. This example combines multiple panels and layout managers for placement of the component areas and controls. The example (sketch below) has a main frame, a main panel, and four (4) smaller panels for locating various components and positioning. The four panels will be built with their components and then added to the main panel.



Design Illustration

The main interface has five (5) panels as shown, and is implemented by creating a frame and the main panel, and then the four smaller panels which are placed on the main panel. A border layout was used and the background colors were added to the panels on the right to highlight their locations.



Notice that without the color, we cannot see where one panel ends and another begins. Also note the differences in the sizes of the panels. The panels will be sized to accommodate the controls and components that will be located on them as things develop.

---

## Appendix F

The border layout used in this example for the main panel provides North, South, East, and West positioning. The default placement locates them outermost in their quadrants.

The code below declares the main frame, all of the panels (including a main panel), and the labels. The constructor sets the sizes, provides the background colors, and then adds the sub-panels to the main panel which is then added to the frame.

```
public class CombinedLayouts {

    JFrame mainFrame = new JFrame("Combined Layouts Example");

    JPanel mainPanel = new JPanel();           // declare the panels
    JPanel topPanel = new JPanel();
    JPanel leftPanel = new JPanel();
    JPanel rightPanel = new JPanel();
    JPanel bottomPanel = new JPanel();

    JLabel topPanelLabel = new JLabel("Top Panel");           // declare the labels
    JLabel leftPanelLabel = new JLabel("Left Panel");
    JLabel rightPanelLabel = new JLabel("Right Panel");
    JLabel bottomPanelLabel = new JLabel("Bottom Panel");

    public CombinedLayouts() {                       // constructor

        mainFrame.setSize(700, 700);

        // Set the specifics for each panel and add the label
        topPanel.setPreferredSize(new Dimension(700, 80));   // width, height
        topPanel.setBackground(new Color(153,102,255));
        topPanel.add(topPanelLabel);

        leftPanel.setPreferredSize(new Dimension(180, 200));
        leftPanel.setBackground(new Color(153,204,255));
        leftPanel.add(leftPanelLabel);

        rightPanel.setPreferredSize(new Dimension(380, 200));
        rightPanel.setBackground(new Color(153,153,102));
        rightPanel.add(rightPanelLabel);

        bottomPanel.setPreferredSize(new Dimension(700, 80));
        bottomPanel.setBackground(new Color(255,102,102));
        bottomPanel.add(bottomPanelLabel);

        mainPanel.add(topPanel, BorderLayout.NORTH);           // add the panels
        mainPanel.add(leftPanel, BorderLayout.WEST);
        mainPanel.add(rightPanel, BorderLayout.EAST);
        mainPanel.add(bottomPanel, BorderLayout.SOUTH);
    }
}
```

---

## Appendix F

```
// add the main panel to the Frame.
mainFrame.add(mainPanel);

mainFrame.setLocationRelativeTo(null);
mainFrame.setVisible(true);
mainFrame.setDefaultCloseOperation(1);

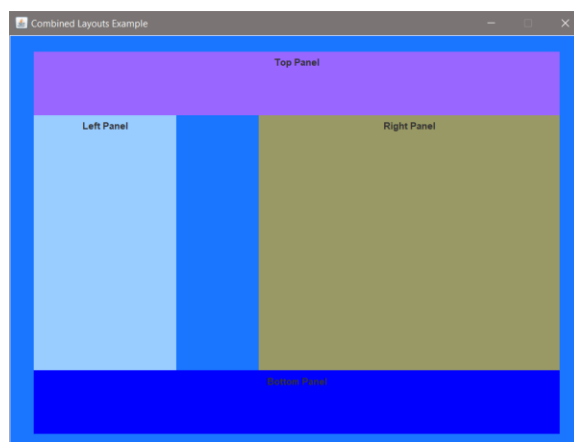
} // end of constructor
```

Creating the individual panels should be done in methods to modularize the program and separate the code. This will be added next. For now, the program runs and produces a preliminary layout of the interface. The code in main to generate an instance of the class is shown here.

```
public static void main(String[] args) {
    CombinedLayouts CL = new CombinedLayouts();
}
```

Main creates an instance of the Frame as “CL” and it can manipulate the Frame and any of its members (attributes) or pass the CL object to a method that can do the same. To show this, the code below changes the color of the bottom panel to blue from main after the object is created.

```
public static void main(String[] args) {
    CombinedLayouts CL = new CombinedLayouts();
    CL.bottomPanel.setBackground(Color.BLUE);
}
```



Next, the individual sections will be built and will be divided up (Step-wise Refinement) by handling the panels separately in methods. This places code that is specific to a panel in a separate area of the project (a method) which is then called by the constructor to “build” the pieces individually before they are added to the main panel.

---



## Appendix F

The top panel simply contains the title text, so that is a good place to start writing methods to build the panels. The default font is used by the program, and should be changed to a larger font and maybe a different style. The existing code for this panel (shown below) in the constructor will be moved to the method, and replaced with a call to the method that will build the panel.

```
topPanel.setPreferredSize(new Dimension(700, 80));
topPanel.setBackground(new Color(153,102,255));
topPanel.add(topPanelLabel);
```

After declaring the method (note the name), and movement of the code from the constructor, the declaration of the label for the top panel can also be moved to the method. The goal is to locate as much code as possible that relates to creating this panel in the method.

```
public void populateTopPanel(JPanel topPanel) {

    topPanel.setPreferredSize(new Dimension(700, 80));
    topPanel.setBackground(new Color(153,102,255));
    JLabel topPanelLabel = new JLabel("Top Panel");

    topPanel.add(topPanelLabel);
}
```

A call to the method now replaces the code that was in the constructor.

```
public CombinedLayouts() { // constructor

    mainFrame.setSize(700, 700);

    populateTopPanel(topPanel);
```

After the label is created, a customized font can be assigned to it as shown here.

```
public void populateTopPanel(JPanel topPanel) {

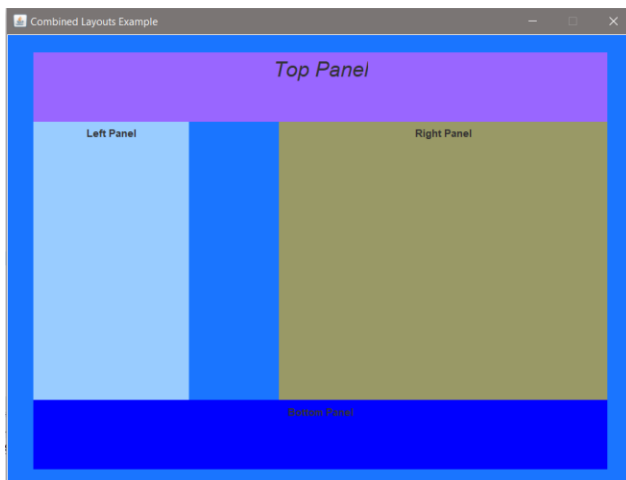
    topPanel.setPreferredSize(new Dimension(700, 80));
    topPanel.setBackground(new Color(153,102,255));
    JLabel topPanelLabel = new JLabel("Top Panel");
    topPanelLabel.setFont(new Font("Arial", Font.ITALIC, 24));

    topPanel.add(topPanelLabel);
}
```

An italic Arial font is tried with a guess at the size. The result is shown below and although the text is centered horizontally, it is not centered vertically.

---

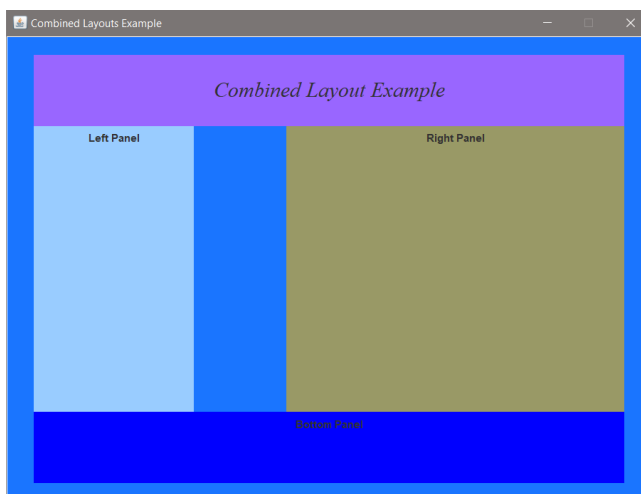
## Appendix F



There are several options for centering the label vertically. Since there is only one component, adding an empty border allows setting pixel spacing around the label. In the code below, the font has been changed and an empty border with top spacing has been added.

```
public void populateTopPanel(JPanel topPanel) {
    topPanel.setPreferredSize(new Dimension(700, 80));
    topPanel.setBackground(new Color(153,102,255));
    JLabel topPanelLabel = new JLabel("Combined Layout Example");
    topPanelLabel.setFont(new Font("Times New Roman", Font.ITALIC, 24));
    topPanelLabel.setBorder(new EmptyBorder(20, 0, 0, 0)); // top, left, bottom, right
    topPanel.add(topPanelLabel);
}
```

The details of the panel and the components are all within the method keeping them out of the constructor, and with the exceptions of the color background, the top panel is now complete.

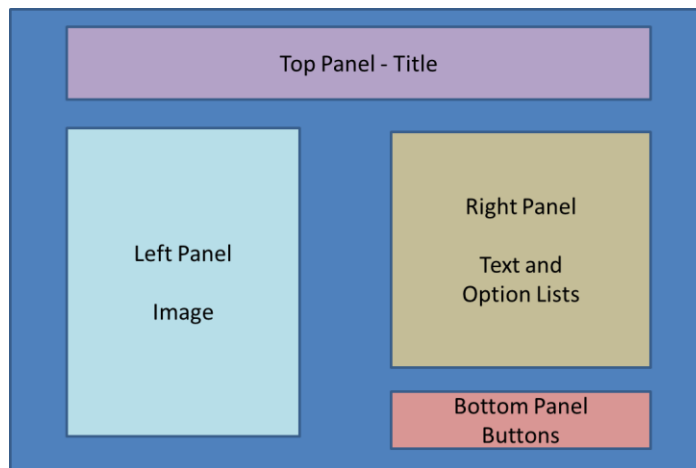


Note that at each step, running the program ensures that any errors introduced are corrected immediately. Frequent testing can save hours of debugging and fixing minor errors.

## Appendix F

The default layout for a JPanel is Flow Layout, and that was used on the top panel. A flow layout simply allows components to flow left to right, then down and left to right in the order they are added. For more complex panels, other layouts provide greater flexibility. Recall that the main panel uses a Border Layout with North, South, East, and West quadrants, and each of the smaller panels is positioned in one of those quadrants when added to the main panel.

The left panel will be implemented next.



The method for the left panel will be set up the same way as the top panel and the code will be moved out of the constructor as before including the label declaration.

```
public void populateLeftPanel(JPanel leftPanel) {
    leftPanel.setPreferredSize(new Dimension(180, 200)); // width, height
    leftPanel.setBackground(new Color(153,204,255));
    JLabel leftPanelLabel = new JLabel("Left Panel");

    leftPanel.add(leftPanelLabel);
}
```

The call to the method is added to the constructor after the top panel.

```
public CombinedLayouts() { // constructor
    mainFrame.setSize(700, 700);

    populateTopPanel(topPanel);
    populateLeftPanel(leftPanel);
}
```

The left panel requires two labels and an image and will use a Grid Bag Layout to position them. This layout allows row and column placement using “constraints”. First the layout is assigned,

---

## Appendix F

and constraints are declared. The insets put padding around the components and the weight for x establishes the definitive columns. The anchors place the components within the cell of the grid. When the labels are added to the panel, the second argument is the constraints.

```
public void populateLeftPanel(JPanel leftPanel) {

    leftPanel.setPreferredSize(new Dimension(180, 200));    // width, height
    leftPanel.setBackground(new Color(153,204,255));

    leftPanel.setLayout(new GridBagLayout());
    GridBagConstraints con = new GridBagConstraints();
    con.insets = new Insets(10,10,10,10);    // top, left, bottom, right
    con.weightx = 0.5;

    JLabel leftPanelLabel1 = new JLabel("LP Label 1");
    con.gridx = 1;
    con.gridy = 1;
    con.anchor = GridBagConstraints.EAST;
    leftPanel.add(leftPanelLabel1,con);

    JLabel leftPanelLabel2 = new JLabel("LP Label 2");
    con.gridx = 2;
    con.gridy = 1;
    con.anchor = GridBagConstraints.WEST;
    leftPanel.add(leftPanelLabel2,con);
}
```

Next the image for the left panel requires file handling and is placed a in a try block, and positioning is accomplished with constraints and anchoring.

```
BufferedImage myPicture;

try {
    myPicture = ImageIO.read(new File("CombinedLayout.png"));
    JLabel picLabel = new JLabel(new ImageIcon(myPicture));
    con.gridx = 0;
    con.gridy = 4;
    con.gridwidth = 4;
    con.anchor = GridBagConstraints.CENTER;

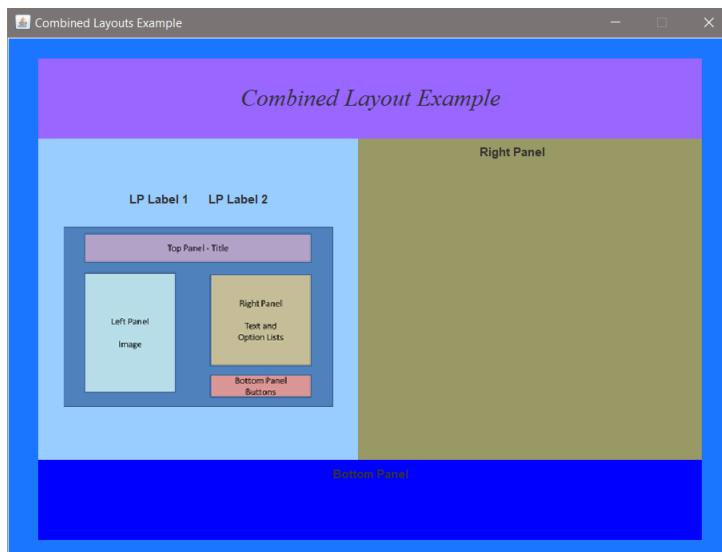
    leftPanel.add(picLabel, con);

} catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

The insets used on the left panel space the components apart and they are fixed should the window be resized. The results for the left panel code are shown here. The image is a screen capture of the design sketch.

---

## Appendix F



Again testing (running the program) is accomplished at each step to ensure things are working and to determine where additional tweaking is needed. In addition, the sizes of the smaller panels have not been changed and no changes have been made to the main panel. After the individual panels are built and their components placed appropriately, then the overall program interface will be worked. Any changes made to the main panel now would probably need to be changed again once everything is finished.

The next panel (right panel) will be done the same way with a method that builds the panel and is called from the constructor of the class after the left panel.

```
public CombinedLayouts() { // constructor

    mainFrame.setSize(700, 700);
    mainPanel.setLayout(new BorderLayout());

    populateTopPanel(topPanel);
    populateLeftPanel(leftPanel);
    populateRightPanel(rightPanel);
}
```

The choice of what layout to use for this or any other panel is subjective, and programmers tend to use the layouts that they are more familiar with. The right panel has text and option lists. This could be done top-down and a flow layout would work, or a grid. Left alignment would be appealing (a design choice), and a flow layout would work with some tweaking.

Since the flow layout simply places the components on the panel left to right in the order that they are added, the number of items or components that fit in a row is dependent upon the size of the component and the width of the panel. As an example, in the code below the components were just created and added to see what happens....where they end up.

## Appendix F

```

public void populateRightPanel(JPanel rightPanel) {

    rightPanel.setPreferredSize(new Dimension(380, 200));
    rightPanel.setBackground(new Color(153,153,102));
    rightPanel.setLayout(new FlowLayout());

    JLabel rightPanelLabel1 = new JLabel("Right Panel first option list.");

    String[] firstOptions = { "Choice 1 ", "Choice 2 ", "Choice 3 ", "Choice 4 " };
    final JComboBox<String> firstBox = new JComboBox<String>(firstOptions);

    JLabel rightPanelLabel2 = new JLabel("Right Panel second option list.");

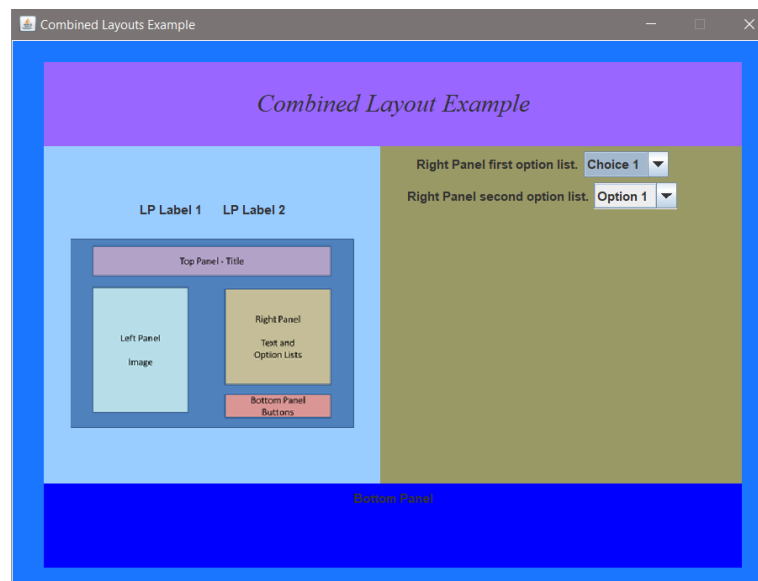
    String[] secondOptions = { "Option 1 ", "Option 2 ", "Option 3 ", "Option 4 " };
    final JComboBox<String> secondBox = new JComboBox<String>(secondOptions);

    rightPanel.add(rightPanelLabel1);
    rightPanel.add(firstBox);
    rightPanel.add(rightPanelLabel2);
    rightPanel.add(secondBox);

}

```

As shown below, the components are centered horizontally and each row contains as many components as it can fit. The design calls for a label, option list, label, option list configuration. Tailoring the layout to the sketch comes next.



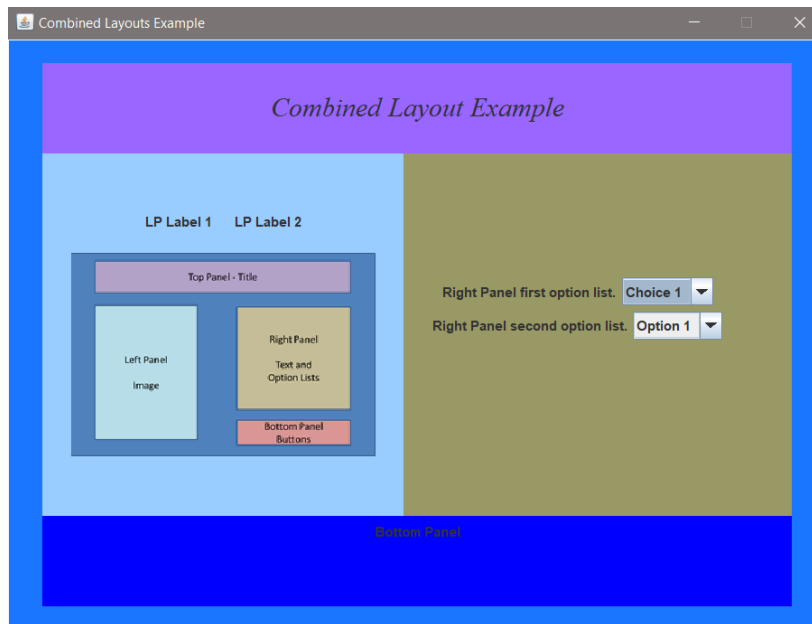
Java provides rigid areas and struts that can be used as spacers to position components. A rigid area can be a custom size and added to the panel to “push” other components around. It is invisible and acts as a spacer. A strut is similar but has no height dimension.

## Appendix F

In the code below, a rigid area has been declared and added to the right panel. The order in which components are added to a flow layout determines their positioning. The rigid area is added first, and the result is that it pushes all of the components down.

The first dimension, which is the width of the rigid area, is almost the width of the panel to ensure that nothing fits on that row. The height determines the number of pixels that it will fill vertically. The resulting display is shown below.

```
rightPanel.add(Box.createRigidArea(new Dimension(360,100))); // width, height
rightPanel.add(rightPanelLabel1);
rightPanel.add(firstBox);
rightPanel.add(rightPanelLabel2);
rightPanel.add(secondBox);
```

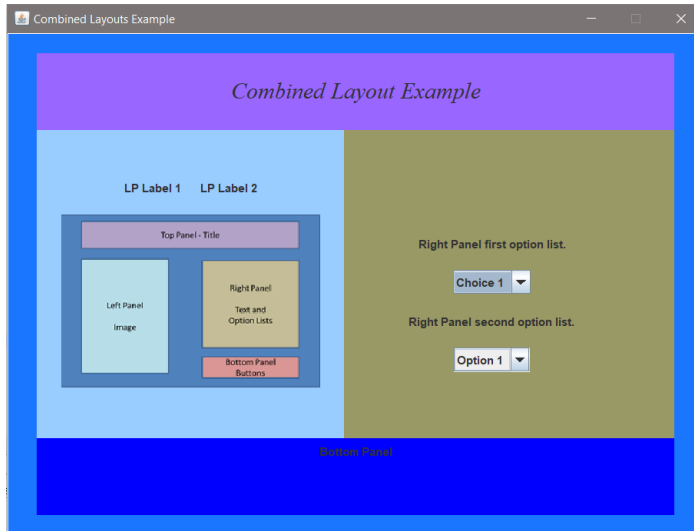


Additional rigid areas could be created and added between each of the components to force them to the next row, and they would be centered by default.

```
rightPanel.add(Box.createRigidArea(new Dimension(360,100))); // width, height
rightPanel.add(rightPanelLabel1);
rightPanel.add(Box.createRigidArea(new Dimension(360,10))); // width, height
rightPanel.add(firstBox);
rightPanel.add(Box.createRigidArea(new Dimension(360,10))); // width, height
rightPanel.add(rightPanelLabel2);
rightPanel.add(Box.createRigidArea(new Dimension(360,10))); // width, height
rightPanel.add(secondBox);
```

The result of the added rigid areas is shown below

## Appendix F

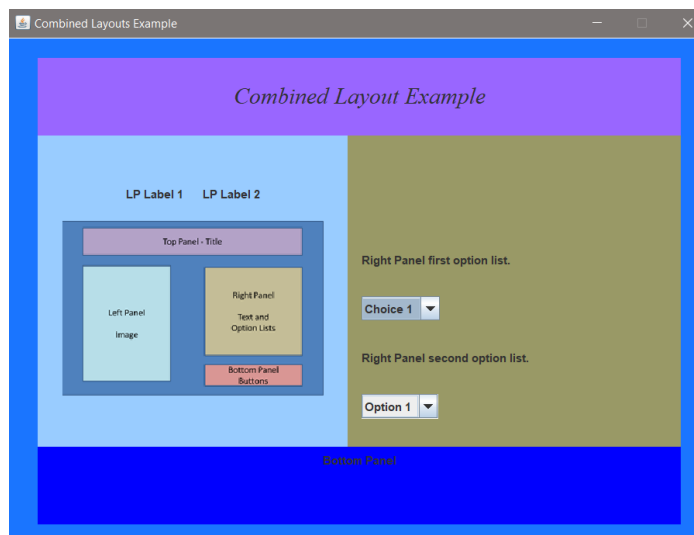


The horizontal alignment of the components is centered as a default. A grid layout would allow them to be aligned by row columns using constraints and allow placement within each column (EAST, WEST). However, the Flow Layout constructor can accept arguments for alignment that apply to all of the components on the panel. The choices are left, right, center, leading, and trailing, and two arguments for horizontal and vertical gaps between components.

As an example, the code has been modified to include the alignment, horizontal gap, and vertical gap arguments when the layout is assigned to the panel. This eliminates the need for a few of the rigid areas.

```
rightPanel.setLayout(new FlowLayout(FlowLayout.LEADING, 50,10)); // alignment, hGap, vGap
```

The result is left alignment of the components and vertical spacing as shown here. Some adjustment in vertical positioning is needed, but the panel is complete.





## Appendix F

The final panel (bottom panel) requires a label and two buttons and currently is not sized or positioned in line with the design sketch. The buttons and labels must be moved to the far right side of the panel and aligned. A rigid area could be created to push the bottom panel components to the right, but it would fill the row height. Two rigid areas (one on each row) would work easily, and allow a flow layout to be used. A method also needs to be created to populate the bottom panel.

To better highlight the example, the color of the panel has been changed back. Recall that an earlier example showed how to access the panel from main and change it to blue.

Another method for the bottom panel is created and added to the constructor.

```
public CombinedLayouts() { // constructor
    mainFrame.setSize(700, 700);
    mainPanel.setLayout(new BorderLayout());

    populateTopPanel(topPanel);
    populateLeftPanel(leftPanel);
    populateRightPanel(rightPanel);
    populateBottomPanel(bottomPanel);
}
```

The components for the panel are now moved to the method, and each time something is moved the program is run to ensure that none of the changes introduces an issue.

```
public void populateBottomPanel(JPanel bottomPanel) {
    bottomPanel.setPreferredSize(new Dimension(380, 140));
    bottomPanel.setBackground(new Color(255,102,102));
    FlowLayout layout = new FlowLayout(FlowLayout.CENTER, 50,10); // alignment, hGap, vGap;
    bottomPanel.setLayout(layout);

    JLabel bottomPanelLabel1 = new JLabel("Bottom ");
    JLabel bottomPanelLabel2 = new JLabel("Panel");

    bottomPanel.add(bottomPanelLabel1);
    bottomPanel.add(bottomPanelLabel2);

    button1.setPreferredSize(new Dimension (100,30)); // width, height
    button1.setText("Button 1");
    bottomPanel.add(button1);

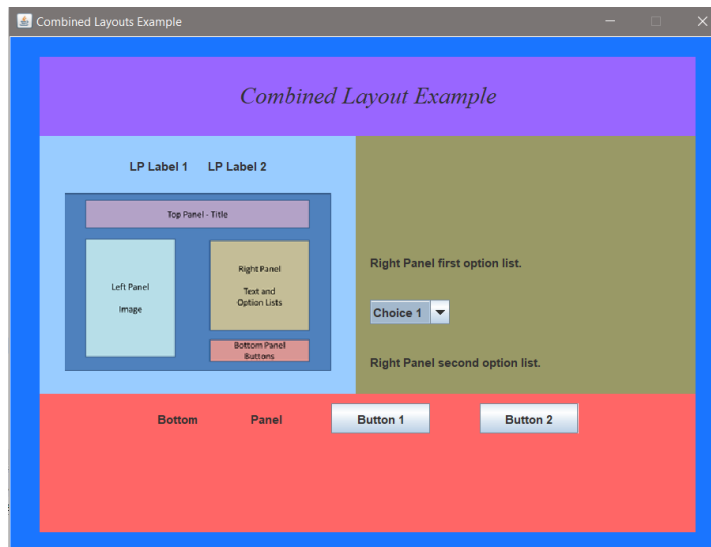
    button2.setPreferredSize(new Dimension (100,30)); // width, height
    button2.setText("Button 2");
    bottomPanel.add(button2);
} // end of populateBottomPanel
```

Notice in the code above that the button sizes can be set. This helps with consistency when the text on one button is shorter than the other, since the buttons will automatically size to fit the text. Also note that the flow layout (at least for now) uses centering, and the hGap and vGap are

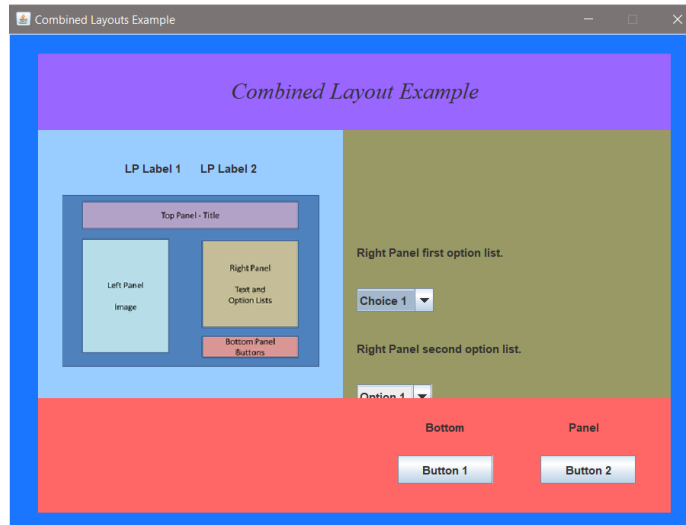
---

## Appendix F

guesses. The running program now produces the window below. Note that the right panel has been partially obscured and the tight panel rigid area will need to be adjusted.



Adding two rigid areas to the bottom panel can push the bottom panel components to the right. If it seems more logical to add one to the main panel, doing that would cause an issue since the bottom panel fills the SOUTH quadrant of the main panel. It would override the rigid area.



The code to set the labels and buttons in place on the bottom panel is shown below. The button declarations have also been moved into the method from the class and Label1 has been given a set size to push Label2 to the right for spacing.

There will be a few more changes for positioning, but the panel is complete.

## Appendix F

```

public void populateBottomPanel(JPanel bottomPanel) {

    bottomPanel.setPreferredSize(new Dimension(700, 120));
    bottomPanel.setBackground(new Color(255,102,102));
    FlowLayout layout = new FlowLayout(FlowLayout.CENTER, 50,20); // alignment, hGap, vGap;
    bottomPanel.setLayout(layout);

    JButton button1 = new JButton();
    JButton button2 = new JButton();

    JLabel bottomPanelLabel1 = new JLabel("Bottom ");
    bottomPanelLabel1.setPreferredSize(new Dimension(100,10));

    JLabel bottomPanelLabel2 = new JLabel("Panel");

    Component rig1 = Box.createRigidArea(new Dimension(280,20));
    bottomPanel.add(rig1);

    bottomPanel.add(bottomPanelLabel1);
    bottomPanel.add(bottomPanelLabel2);

    Component rig2 = Box.createRigidArea(new Dimension(290,20));
    bottomPanel.add(rig2);

    button1.setPreferredSize(new Dimension (100,30)); // width, height
    button1.setText("Button 1");
    bottomPanel.add(button1);

    button2.setPreferredSize(new Dimension (100,30)); // width, height
    button2.setText("Button 2");
    bottomPanel.add(button2);

} // end of populateBottomPanel

```

Next the final sizing and positioning will be accomplished and trial and error can become tedious. Listing all of the dimensions for the panels can make things a bit easier and much faster. The current dimensions are as follows:

<u>Panel</u>	<u>width</u>	<u>height</u>	
Main	700	700	defaults to the frame size
Top	700	80	
Left	300	200	
Right	380	200	
Bottom	700	120	

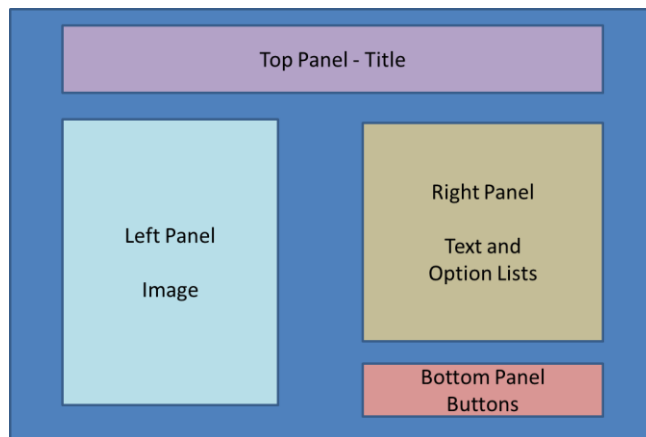
The height of the main panel is 700 pixels, and the combined heights for the left and right sides are 600 pixels, so the height for the main frame is changed to 600.

The width of the right panel is 380 compared to 300 for the left panel, and when the colors are turned off, the text and option lists will be a bit too far to the left compared to the design sketch.

---

## Appendix F

Comparing the design sketch while making minor changes to various dimensions, makes things much easier and saves time.



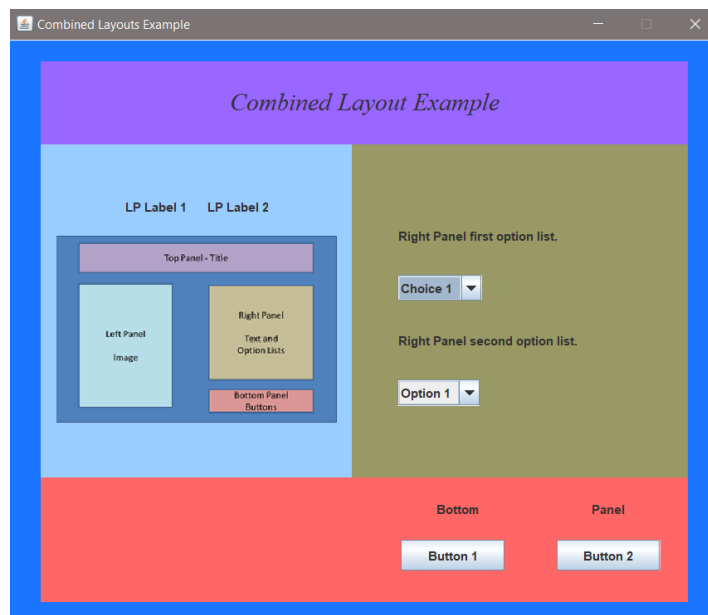
The first change made was to shorten the height of the main frame, and the left and right sides by 40 pixels. That forced an adjustment to the right panel spacing. The height of the topmost rigid area of the right panel was easily adjusted to 60 to realign things,

```
rightPanel.add(Box.createRigidArea(new Dimension(360,60))); // width, height
```

In addition, the flow layout hGap value was changed to 100 to move everything to the right.

```
rightPanel.setLayout(new FlowLayout(FlowLayout.LEADING, 100,10)); // alignment, hGap, vGap
```

The results are more in line with the sketch.

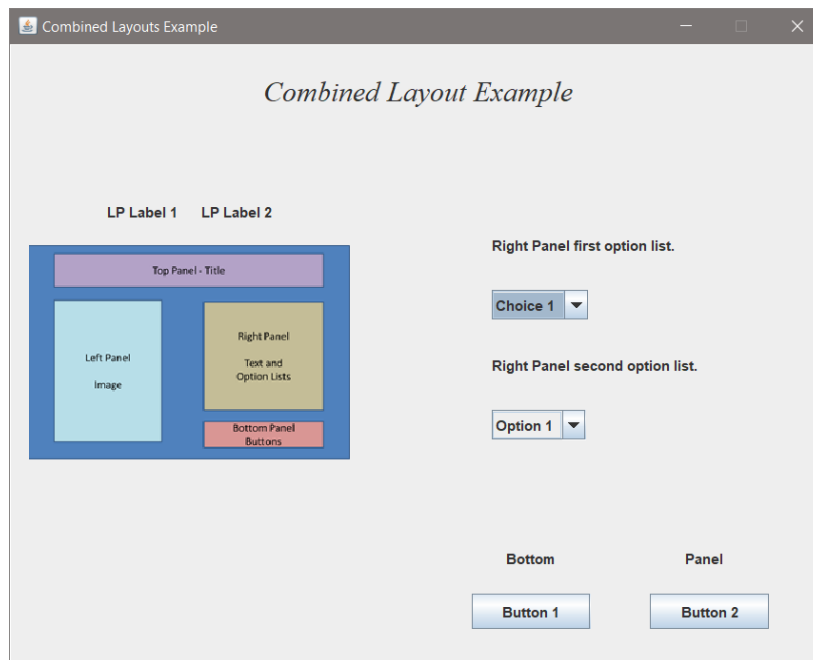


## Appendix F

To ensure that resizing the window by the user does not skew the panel locations, the main frame method `setResizable()` is used and set to `false`.

```
mainFrame.add(mainPanel);
mainFrame.setResizable(false);
mainFrame.setLocationRelativeTo(null);
mainFrame.setVisible(true);
mainFrame.setDefaultCloseOperation(1);
```

The methods that set the background colors for the panels are now commented out (not removed since they may be needed later), and the program is run again.

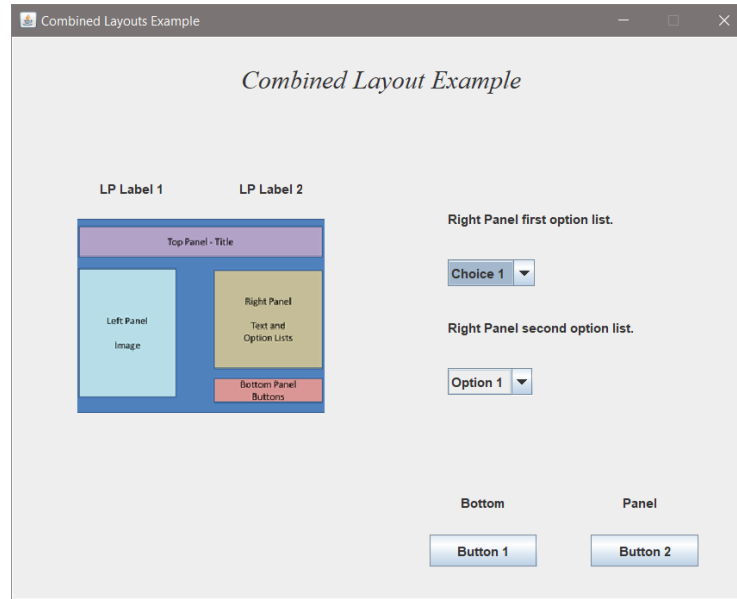


The last tweak would be to move the image and labels on the left panel toward the center. This can be done by adding an empty border to the main panel or changing the insets that were used on the left panel itself which is much easier. The insets “left” argument has been increased.

```
leftPanel.setLayout(new GridBagLayout());
GridBagConstraints con = new GridBagConstraints();
con.insets = new Insets(10,60,10,10); // top, left, bottom, right
con.weightx = 0.5;
```

Again trial and error is used to get the right number of pixels for the left inset to position the components appropriately. The results are shown below.

## Appendix F



The interface is now complete as far as adding the components and positioning them. Separating the interface areas into individual panels allows for the use of multiple layouts and modularizing the program with methods organizing the development.

The class code after modularization is included below.

```
public class CombinedLayouts {

    JFrame mainFrame = new JFrame("Combined Layouts Example");
    JPanel mainPanel = new JPanel();
    JPanel topPanel = new JPanel();
    JPanel leftPanel = new JPanel();
    JPanel rightPanel = new JPanel();
    JPanel bottomPanel = new JPanel();

    public CombinedLayouts() { // constructor

        mainFrame.setSize(700, 600); // width, height
        mainPanel.setLayout(new BorderLayout());
        mainPanel.setBackground(new Color(26,117,255));
        mainPanel.setBorder(new EmptyBorder(20,30,20,30));

        populateTopPanel(topPanel);
        populateLeftPanel(leftPanel);
        populateRightPanel(rightPanel);
        populateBottomPanel(bottomPanel);

        // Add panels to the main panel
        mainPanel.add(topPanel, BorderLayout.NORTH);
        mainPanel.add(leftPanel, BorderLayout.WEST);
        mainPanel.add(rightPanel, BorderLayout.EAST);
        mainPanel.add(bottomPanel, BorderLayout.SOUTH);

        mainFrame.add(mainPanel);
        mainFrame.setResizable(false);
        mainFrame.setLocationRelativeTo(null);
        mainFrame.setVisible(true);
        mainFrame.setDefaultCloseOperation(1);

    } // end of constructor
```

## Appendix F



## Appendix G

**Index of Programming Examples**

<b>Example</b>	<b>Page</b>
Ex. 3.1 – Displaying Output in Java	16
Ex. 3.2 – Formatted Output	16
Ex. 3.3 – Line Feed	17
Ex. 3.4 – Escape Sequences	17
Ex. 3.4A – Formatted Output revisited	18
Ex. 3.5 – Getting Keyboard Input	20
Ex. 3.6 – Math Methods – Random Numbers	23
Ex. 3.7 – Random Number Ranges	24
Ex. 4.1 – Conditional Example with $x = -1$	26
Ex. 4.2 – A Simple Method called from main	30
Ex. 4.3 – Method in a Second File called from Main	32
Ex. 5.1 – Example Sketch of initial GUI	35
Ex. 5.2 – Initial Window Using a Dialog Box – <code>showOptionDialog</code>	36
Ex. 5.3 – Dialog Box Button Selection	37
Ex. 5.4 – A Simple Window	38
Ex. 5.5 – A Simple Window class	39
Ex. 5.6 – Create Account GUI	41
Ex. 5.7 – Two-button Three-frame Example	45
Ex. 6.1 – File Reading and Writing	51
Ex. 6.1A – File Writing and Reading with Exception Handling	53
Ex. 6.2 – File Reading and Writing using <i>try-with-resources</i>	54
Ex. 6.3 – Writing Numbers to a File and Then Reading	55
Ex. 6.4 – File Selection Using <code>JFileChooser</code> and Reading using <code>StringBuilder</code>	60
Ex. 6.5 – File Type Selection using <code>FileNameExtensionFilter</code>	61

---



## Appendix G

Ex. 6.6 – File “Save AS” using showSaveDialog()	62
Ex. 7.1 – Indexing Strings	63
Ex. 7.2 – Copying a Character from a String	63
Ex. 7.3 – Concatenating Strings	64
Ex. 7.4 – The <i>length()</i> method with Strings	64
Ex. 7.5 – Character Testing a String	65
Ex. 7.6 – String Manipulation	66
Ex. 7.7 – String Tokenizing	67
Ex. 7.8 – ArrayLists	68
Ex. 7.9 – ArrayList of Doubles	69
Ex. 7.10 – Tab Delimited File into String	70
Ex. 8.1 – Frame Menu	77
Ex. 8.2 – Ten Buttons	79
Ex. 9.1 – Meteor Program Example – main	83
Ex. 9.2 – Meteor Program Example – MeteorWin()	84
Ex. 9.3 – Meteor Program Example – MeteorWin() Constructor	85
Ex. 9.4 – Meteor Program Example – ButtonListener	86
Ex. 9.4 – Meteor Program Example – ButtonListener (continued)	87
Ex. 9.5 – Data Display Window – Create on Program Start	88
Ex. 9.6 – the Data Display Window	89
Ex. 9.7 – Update Data Method	90
Ex. 9.7A – Update Data Method Corrected	91
Ex. 9.8 – ScrollPane Example	92
Ex. 9.9 – Bars on a JComponent	93
Ex. 9.9A – Bars on a JComponent – vertical	94
Ex. 9.10 – Temperature Conversion and Plot	95
Ex. 10.1 – Display a User-selectable Calendar	102

---

## Index

<b>A</b>		Assignment operator	18
<i>acos(x)</i>	23	<i>atan(x)</i>	23
Abstract Window Toolkit	34	AudioInputStream	104
ActionEvent	43	AudioSystem	103
<i>actionPerformed()</i>	43	<i>getClip()</i>	103
ActionListener, class	43	AWT Abstract Window Toolkit	34
<i>add()</i> , method		axis labels	96
ArrayList	67		
ButtonGroup	76	<b>B</b>	
JFrame	37	Backslash, displaying “\”	17
JMenu	77	bar chart	93
JMenuBar	77	bool data type	19
JPanel	76	Boolean type	28
<i>addActionListener()</i>	43	expressions	23
<i>addWindowListener()</i>	47	logic	27
Agile Development	3	return values	31
Agile Methodologies	4	border	
Addition (+) operator		text, dialog	36
defined	21	title, JFrame	41
concatenation	64	JPanel	43
alignment, output	18		
anchor	41	BorderFactory	43
and operator &&	27	BorderLayout	40
Animation	105	BoxLayout	40
appending data, files	50	<i>browse()</i>	104
<i>append()</i>	89	Browser, launch	104
Arguments, passing	29	BufferedImage	81
ArrayList	67	Button	
declaring	68	<i>addActionListener()</i>	43
<i>add()</i>	68	create	45
<i>append()</i>	97	labels	36
<i>remove()</i>	68	listener	86
<i>set()</i>	68	options	45
<i>asin(x)</i>	23	radio	75

---

## Index

<i>setBackground()</i>	80	Clip, audio	103
<i>setForeground()</i>	80	<i>close()</i>	103
<i>setPreferredSize()</i>	92	<i>open()</i>	103
text	45	<i>start()</i>	103
ButtonGroup	76	<i>isRunning()</i>	103
<i>add()</i>	77	Closing programs	47
ButtonListener, class	44	Columnar data	18
Byte	19	Combo Box	75
		Comments	15
		Concatenation	64
		Conditional statements	25
		Constants	24
		Copying	
		characters	63
		<i>createEtchedBorder()</i>	43
		<i>createTitledBorder()</i>	43
		<b>D</b>	
		data	
		appending to files	50
		file design	56
		reading from files	49
		to text area	89
		writing to files	50
		Data dictionary	56
		data types	19
		bool, int, double, float	19
		Date	101
		DatePicker()	102
		DateTimeFormatter	101
		Decision structures	25
		delimiter	52
		Desktop	105
		Design	5
<b>C</b>			
Calendar	102		
calling methods	29		
CardLayout	40		
Cast	22		
case-sensitive	19		
Centering windows	86		
Char data type	19		
characters	63		
comparing	64		
copying	63		
escape	17		
finding in strings	63		
indexing	63		
newline	17		
tab	17		
Charts			
bar	93		
flow	6		
line	98		
Tools	100		
checkboxbutton component	76		
Classes, example	39		
Class Name, method call	32		
ClickListener, class	43		

---

# Index

Development		textField	44
Agile	3	Equal sign, assignment	18
cycle	3	Equivalence operator	27
methodologies	4	Errors	11
process	3	cost by phase	5
Dialog boxes		dialogs	36
error	36	OutOfBounds	61
File, Save As	61	StackTrace	53
Information box	36	Escape sequences	17
<i>JFileChooser()</i>	59	Event listener	43
<i>showConfirmDialog()</i>	36	Exceptions	52
<i>showInputDialog()</i>	35	FileNotFoundException	53
<i>showMessageDialog()</i>	36	InterruptedException	103
<i>showOpitonDialog()</i>	36	NumberFormatException	21
YES_NO_CANCEL	36	UnsupportedAudioFile	104
Dimension	92	<i>exec()</i>	104
display output	16	exponentiation	21
<i>dispose()</i>	44	example	22
Division	21		
do-while loop	29	<b>F</b>	
double	19	File	49
<i>Double.parseDouble()</i>	21	appending	50
<i>drawImage()</i>	82	<i>close()</i>	50
<i>drawLine()</i>	96	opening	49
<i>drawString()</i>	93	read	49
Drop-down menus	73	read numeric data	54
		writing numeric data	54
<b>E</b>		writing text	50
Eclipse, IDE	9	File selection dialog	59
else clause	25	File “Save As” dialog	60
enhanced for loop	28	FileNameExtensionFilter()	61
Email, launching	105	FileWriter class	50
Entry components		<i>fillOval()</i>	96
TextEntry	34	<i>fillRect()</i>	93

---

## Index

final, key word	24	Graphics object	93
float data type	19	Graphics2D	82
floating point division	22	GridBagLayout	41
Flowchart	6	GridBagConstraints	41
FlowLayout	40	GridLayout	40
font		gridheight	40
create	85	gridwidth	40
<i>setEchoChar()</i>	42	gridx	40
<i>setFont()</i>	85	gridy	40
for loop	28	GridLayout	40
for-each loop	67	GroupLayout	40
<i>format()</i> , String	87	GUI	
formatted output	16	design	35
format specifier	16	dialog	36
Frame component	38	example	41
		positioning components	41
		programming	38
		sketch	35
<b>G</b>			
<i>get(i)</i> , ArrayList	97		
<i>getAbsolutePath()</i>	103	<b>H</b>	
<i>getAbsolutePath()</i>	62	<i>hasNext()</i>	27
<i>getAudioInputStream()</i>	103	<i>hasNextDouble()</i>	27
<i>getClip()</i>	103	<i>hasNextInt()</i>	27
<i>getContentPane()</i>	98	<i>hasNextLine()</i>	49
<i>getDaysInMonth()</i>	102	height, window	41
<i>getDefaultToolkit()</i>	82	Hello World	11
<i>getDesktop()</i>	104	HTML	105
<i>getImage()</i>	82	<i>hypot(x)</i>	23
<i>getName()</i>	61		
<i>getProperty()</i>	60	<b>I</b>	
<i>getRunTime()</i>	104	if-else	25
<i>getSelectedFile()</i>	60	images	81
<i>getSelectedItem()</i>	75	ImageIcon()	81
<i>getSource()</i>	76		
<i>getText()</i>	44		

---

## Index

<i>imageIO.read()</i>	81	java.awt.event	43
Immutable	67	java.awt.Toolkit	82
import		java.lang.Math	23
class	19	java.time	99
example	20	java.util.Scanner	19
package	19	javafx.scene.chart	100
statements	19	javax.swing	34
wildcard	45	JButton	41
indentation	28	JComboBox	73
indexes		JComponent	93
characters	61	JDK	1
ArrayLists	68	JFileChooser	59
strings	63	JFrame	41
Information dialog box	36	JFreeChart	100
Insets	40	JLabel	41
instance	38	JOptionPane	36
integer	19	JPanel	41
<i>Integer.parseInt()</i>	21	JPasswordField	42
Interface Design	33	JScrollPane	91
InterruptedException	103	JTextField	41
ipadx, ipady	40	JRE	1
IPO document	3	Justification, output	18
<i>isDesktopSupported()</i>	104	JVM	1
<i>isdigit()</i>	64		
<i>isLetter()</i>	64	<b>K</b>	
<i>isLowerCase()</i>	64	keyboard input	19
<i>isUpperCase()</i>	64	key words	19
<i>isWhiteSpace()</i>	64		
Iterative Enhancement	69		
		<b>L</b>	
<b>J</b>		Label component	34
Java	1	example	41
Java Foundation Classes	34	Layouts	40
JavaFX	100	multiple	Appendix F

---



## Index

Option Lists	73	print formatted	16
or operator	27	<i>println()</i>	16
Output	2	PrintWriter	49
displaying	16	Process	105
file	50	Program design	3
formatting	16	Pseudocode	5
window	88	public	29
Override	47		
<b>P</b>			
Package, Java	10	<b>Q</b>	
Package Explorer	12	Quick-Start (Eclipse)	10
Package, import	19	quotes, displaying \"	17
Package, import methods	32		
<i>paintComponent()</i>	93	<b>R</b>	
Panel	34	<i>radians()</i>	23
<i>add()</i>	41	Radio buttons	75
example	41	groups	76
<i>repaint()</i>	97	random numbers	23
<i>setBackground()</i>	96	ranges	24
<i>setBorder()</i>	43	Read, file	49
<i>setVisible()</i>	38	Read, keyboard input	19
Panel, Multiple	Appendix F	Relational operators	27
Parameter	30	<i>remove()</i> , ArrayLists	67
Passing arguments	29	<i>repaint()</i>	95
PEMDAS	22	<i>replace()</i> , strings	66
pi variable	23	Requirements	4
Pie Chart	100	return statements	29
plotting	93	right justification	18
Precedence	22	rounding	22
Primitive data types	19	Runtime	104
Programming Trends	2		
print function	16	<b>S</b>	
<i>printf()</i>	16	“Save As” dialog	61

---



## Index

Scanner	19	Simple data types	19
Scene	102	<i>sin(x)</i>	23
Scroll bars	91	Sprint	4
Scrum	4	Software Development Process	72
Sequence Diagram	7	Sound	103
serialVersionUID	95	SpringLayout	40
<i>set()</i> , ArrayList	67	<i>sqrt()</i>	23
<i>setBackground()</i>	79	Stage	102
<i>setBorder()</i>	43	Standards	Appendix E
<i>setColor()</i>	96	static	29
<i>setCurrentDirectory()</i>	59	static class	45
<i>setDefaultCloseOperation()</i>	47	Stepwise Refinement	71
<i>setDialogTitle()</i>	60	Storyboarding	34
<i>setEchoChar()</i>	42	String	63
<i>setEnabled()</i>	79	Array of	80
<i>setFilter()</i>	61	concatenate	64
<i>setFont()</i>	85	conversion	21
<i>setForeground()</i>	80	comparing	27
<i>setHorizontalAlignment()</i>	42	modification	66
<i>setIcon()</i>	82	<i>StringBuilder()</i>	67
<i>setJMenuBar()</i>	77	<i>charAt()</i>	63
<i>setLocation()</i>	89	<i>equals()</i>	27
<i>setLocationRelativeTo()</i>	42	<i>format()</i>	88
<i>setResizable()</i>	42	<i>length()</i>	64
<i>setSize()</i>	38	<i>replace()</i>	66
<i>setText()</i>	86	<i>substring()</i>	65
<i>setTitle()</i>	39	<i>toLowerCase()</i>	66
<i>setVisible()</i>	38	<i>toUpperCase()</i>	66
Short data type	19	<i>split()</i>	67
<i>showConfirmDialog()</i>	36	<i>trim()</i>	66
<i>showMessageDialog()</i>	36	<i>valueOf()</i>	97
<i>showOpenDialog()</i>	60	StringBuilder()	67
<i>showOptionDialog()</i>	36	Subclass	44
<i>showSaveDialog()</i>	61	Substring	65

---

## Index

Subtraction (-) operator	21	<i>useDelimiter()</i>	52
<i>super.paintComponent()</i>	97	User data entry	41
swing components	34	User interface	33
<b>T</b>			
tab <code>\t</code>	17	<b>V</b>	
<i>tan(x)</i>	23	Validating Input	26
Text files	55	Variables	18
Thread.sleep	103	Naming conventions	19
Time	99	Types	19
Timer	103	VBox	102
<i>start()</i>	106	<b>W</b>	
<i>stop()</i>	106	W3C	19
TimerListener	106	W3Schools, link	Appendix D
<i>toFront()</i>	89	<i>wait()</i>	46
<i>toLower()</i>	66	Web browser, launch	104
<i>toUpper()</i>	66	<i>weightx</i> , <i>weighty</i>	40
Toolkit	82	While loop	28
Tokenizing	67	width, window	41
Traceback	53	Wildcard import	45
<i>trim()</i> , strings	66	Window	
Truncation	22	border title	41
try/catch	52	See <i>set</i> methods above	
try-with-resources	54	chart	99
Types, data	19	Create Account	40
primitive	19	frame	37
<b>U</b>			
UML Unified Modeling Language	7	image	81
UML Diagram	7	interface	72
UML Superstructure	7	JFileChooser	60
UnsupportedAudioFileException	104	Login	40
URI, Uniform Resource Identifier	104	menu	77
		parent	35
		WindowAdapter()	47

---

## Index

WindowClosing()	47
WindowEvent	47
WindowListener	47
Workspace, Eclipse	10
World Wide Web Consortium	19

### X

x axis, line	97
xCoord	97

### Y

y axis, line	96
yCoord	97

---