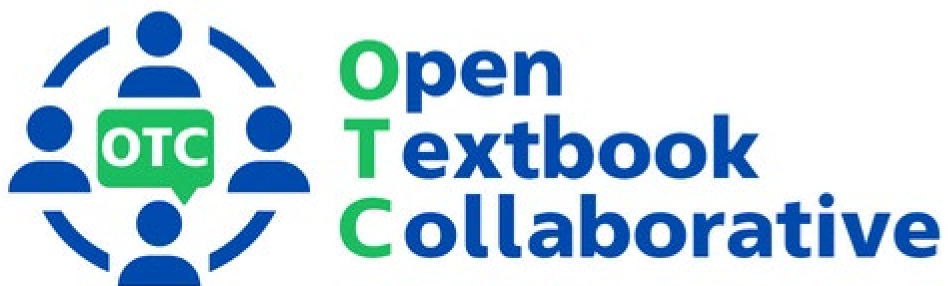
Introduction to Computer Programming with Java

Chris Simber

Computer Programming in Java: Starting Out in Eclipse

Contributing Authors

Chris Simber, Rowan College at Burlington County



Original Publication Year 2022

Computer Programming in Java: Starting Out in Eclipse by Chris Simber is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License, except where otherwise noted.

To learn more about the Open Textbook Collaborative, visit https://middlesexcc.libguides.com/OTCProject

Under this license, any user of this textbook or the textbook contents herein must provide proper attribution as follows:

If you redistribute this textbook in a digital or print format (including but not limited to PDF and HTML), then you must retain this attribution statement on your licensing page.

If you redistribute part of this textbook, then you must include citation information including the link to the original document and original license on your licensing page.

If you use this textbook as a bibliographic reference, please include the link to this work http://opennj.net/AA00001549 in your citation.

For questions regarding this licensing, please contact library@middlesexcc.edu

Funding Statement

This material was funded by the Fund for the Improvement of Postsecondary Education (FIPSE) of the U.S. Department of Education for the Open Textbooks Pilot grant awarded to Middlesex College (Edison, NJ) for the Open Textbook Collaborative.

Open Textbook Collaborative

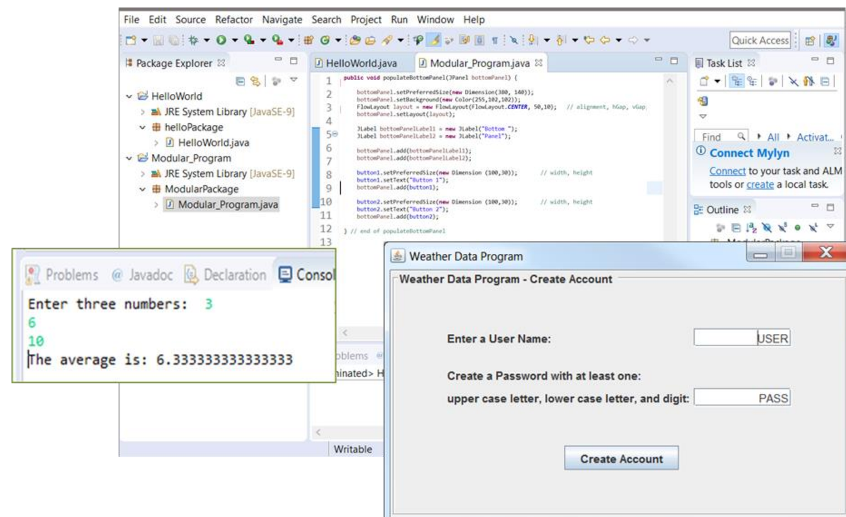
The Open Textbook Collaborative. (OTC) project is a statewide project managed by Middlesex College along with assistance from Brookdale Community College, Ocean County College, Passaic County Community College, and Rowan University.

The project engages a consortium of New Jersey community colleges and Rowan University to develop open educational resources (OER) in career and technical education STEM courses.

The courses align to career pathways in New Jersey's growth industries including health services, technology, energy, and global manufacturing and supply chain management as identified by the New Jersey Council of Community Colleges.

Computer Programming in Java

Starting Out with Eclipse



Chris Simber

Assistant Professor, Computer Science

Rowan College at Burlington County

This material was funded by the Fund for the Improvement of Postsecondary Education (FIPSE) of the U.S. Department of Education for the Open Textbooks Pilot grant awarded to Middlesex College for the Open Textbook Collaborative.

Cataloging Data

Names: Simber, Chris, author.

Title: Computer Programming in Java

Starting out with Eclipse

Subjects: Java (Computer Programming Language)

Chris Simber

Assistant Professor of Computer Science

Rowan College at Burlington County

Author contact: csimber@RCBC.edu

Open Educational Resources Team and Contributors:

Steven Chudnick, Project Coordinator
Alison Cole, Librarian, Felician University
Joshua Gaul, Educational Technology Manager, Edge
Robert Hilliker, Curriculum Council Manager
Marilyn Ochoa, Director, Library Services, Middlesex College
Laura Wingler, Instructional Designer, Ocean County College

Computer Programming in Java: Starting out with Eclipse by Christopher Simber is licensed under a Creative Commons Attribution – NonCommercial-ShareAlike 4.0 International License, except where otherwise noted. Images may not be distributed individually.
All screenshots are included on the basis of Fair Use.

Creative Commons — Attribution-NonCommercial-ShareAlike 4.0 International — CC BY-NC-SA 4.0

Under the following terms: Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

creativecommons.org

Introduction

This book is intended for use in an introductory course in programming using the Java programming language, and includes content for students who are not familiar with programming. The book introduces general computer information and basic computer operations, as well as software engineering principles and processes used in industry to implement computer-based solutions. Algorithm development and problem-solving techniques are introduced as well.

The Eclipse integrated development environment (IDE) is utilized and instructions to obtain, install, and "Getting Started in Eclipse" are provided in the appendices. The goal is to provide students with an overview of computers, software engineering tools and techniques, and to introduce them to computer programming in Java quickly for hands-on instruction.

The examples and exercises reinforce the material being introduced while building on previous material covered, and follow the programming standards for Java publicized by the World Wide Web Consortium (W3C) and set forth in the Java Coding Guidelines created by Joe McManus MGR at Carnegie Mellon University, Software Engineering Institute. Appendix F provides an adequate abridgement of programming standards. The in-chapter exercises are numbered for clarity using a shaded box, and can be used for in-class assignment purposes in addition to the end-of-chapter assignments.

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

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

Getting started in Eclipse is provided in Appendix C with a sample start-up program. Links to the Eclipse web site, Java Tutorials, and the Java Coding Guidelines are included in Appendix E which also includes a link to the Java Tutorial at the W3Schools website.

There are end-of-chapter assignments, and an answer key, exams with answer keys, and accompanying lecture slides are available.

Contents

Chapter 1	Computers and Programming	1
Chapter 2	Java Language Basics	19
Chapter 3	Decision Structures and Boolean Logic	53
Chapter 4	Loops and Repetition Structures	81
Chapter 5	Methods, Modules, and Basic Graphics	111
Chapter 6	Arrays and ArrayLists	139
Chapter 7	File Operations and Exceptions	167
Chapter 8	Classes and Objects	191
Chapter 9	Inheritance and Interfaces	227
Chapter 10	Graphical User Interfaces	253
Chapter 11	GUI Programs	289

Appendix A ASCII Character Set (partial list)

Appendix B Obtaining and Installing Eclipse

Appendix C Getting Started with Eclipse

Appendix D Modular Programming

Appendix E Helpful Links to Information

Appendix F Java Programming Guidelines and Standards

Appendix G Multiple Panels and Layout Managers Example

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

Chapter 1

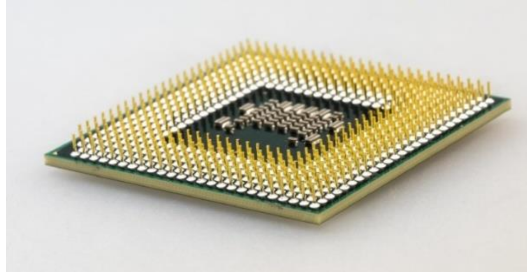
Computers and Programming

Computers are simply data processing devices. They are machines that receive input, process it in some way, and produce output. Computer programs are made up of a series of statements that tell a computer what to do and in what order to do it. These programs provide a sequence of small steps for the computer to execute and produce the desired result. Millions or even billions of these small steps are being executed by the computer as we run programs. This may seem odd to the casual computer user, but these small steps combine to provide what appear to be seamless operations as we interact with the computer. Computer programs are referred to as ***software*** and they are needed to make the computer useful. People who design, write, and test software are commonly referred to as ***software engineers***, software developers, or computer programmers. To better understand the computing process and programming in general, a familiarity with the parts that make up a computer is necessary.

The Central Processing Unit (CPU)

Any part of the computer that we can physically touch (including inside the casing) is referred to as ***hardware***. One important piece of hardware is the Central Processing Unit ***CPU*** which is the "Brains" of the computer. The CPU performs millions of basic instructions per second and controls computer operations. The CPU has two parts. The Arithmetic Logic Unit (ALU) handles

basic arithmetic and comparisons of data such as less than, greater than, equivalent, and not equivalent. The Control Unit retrieves and decodes program instructions and coordinates activities within the computer.



Brown and Green Computer Processor Pixabay is licensed under CC0

Figure 1.1 - Central Processing Unit (CPU)

The CPU (shown upside down above) has pins that plug into a socket located on a circuit board called the motherboard.

Main Memory

RAM stands for Random Access Memory and is often referred to as main memory. RAM is a series of memory chips on a circuit board installed in the computer on the motherboard along with the CPU. The memory in these chips contains a series of memory addresses that enable the computer to store and locate information. These memory addresses are ***volatile*** and when the computer is turned off, RAM does not retain information. It is erased. When a program is launched, the program is copied into RAM for the CPU to use.

RAM by Headup222 is licensed under Pixabay License

Figure 1.2 - Laptop Main Memory (RAM)

The CPU can access instructions and data from RAM very quickly, and RAM is located close to the CPU on the motherboard. The RAM circuit card for a laptop

computer is shown above. The large rectangles are the memory chips, the notches are an aid for inserting the RAM into position on the motherboard, and the gold edge makes the connections for data access. Since RAM is erased when the computer is turned off, ***secondary storage*** is used to retain information.

Read Only Memory or ROM contains the startup instructions for the computer including the BIOS or Basic Input Output System. It is ***non-volatile*** and retains information between sessions.

Secondary Storage

Secondary storage devices are ***non-volatile*** and the information stored is not erased when the power is off. Secondary storage devices include the hard drive inside the computer, external drives connected to the computer, and flash drives. The hard drive inside the computer may be a disk drive which houses a rotating disk and data access arm (shown below), or a solid-state drive which has no moving parts and operates faster than a traditional disk drive.

Laptop-hard-drive-exposed by Evan Amos is licensed under CC BY-SA 3.0

Figure 1.3 - Hard Drive (cover removed)

External drives are typically solid-state and connect to the computer through a cable plugged into a USB (Universal Serial Bus) connector, or plug directly into the USB port of the computer as in the case of flash drives. These drives use flash memory, and do not have a disk drive.

Input and Output Devices

Input devices are anything that provides input or data for a computer. Common input devices are the keyboard, mouse, microphone, and camera.

Output devices include anything that accepts computer output such as monitors, speakers, and printers.

Since data files located on storage devices can be used for reading data into a computer or writing output from a computer, they could be considered both input and output devices.

Software

Computers are machines that follow an input, processing, output sequence of operations and need to be told what to do and in what order to do it. This is accomplished through sets of instructions called ***software***. There are essentially two types of software: ***system software*** (Operating Systems), and ***application programs*** (all other software).

The operating system (OS) provides an interface for us to use computers more easily. Originally a command line interface was used, followed by menu interfaces, and the Graphical User Interface (GUI) replaced that and has been used since. The operating system provides the Graphical User Interface that is now commonly used to interact with the computer, and acts like an orchestra conductor by controlling computer hardware, managing devices connected to the computer, and interacting with programs that are running. Operating systems commonly in use today are Windows, macOS, and Linux.

Application software includes the computer programs that we commonly use to accomplish work such as word processors, spreadsheet programs, collaboration tools, and audio/video editing tools, as well as gaming software.

The Language of Computers

The language of computers is a ***binary*** language consisting of ones and zeros. This is the beauty and simplicity of the computer. At the basic level, everything is a 1 or a 0, is either On or Off, Yes or No, True or False. The ***bit***, or binary digit

used in computing represents a logical state that can be 0 or 1. It is the smallest information representation in a computer. A ***Byte*** is 8 bits consisting of a combination of 0s and 1s, and in computers, each letter, number, and special character consists of a binary representation. For example, a lower case "f" is represented in binary as 01100110 in accordance with the ***ASCII*** standard (pronounced askee). ASCII stands for the American Standard Code for Information Interchange which was developed as a character encoding standard.

The ASCII table consists of binary representations for the 26 uppercase and 26 lowercase letters, and the 9 digits, as well as special characters, punctuation marks, and other symbols and keyboard keys. Appendix A provides a partial list of the binary and decimal representations for the upper and lowercase letters, digits, and punctuation. A portion is shown here with the decimal equivalent.

Decimal	Binary	ASCII
64	0100 0000	@
65	0100 0001	A
66	0100 0010	B
67	0100 0011	C
68	0100 0100	D

ASCII Table (excerpt)

The ***Unicode*** standard incorporates the 256 item ASCII standard and expands to include the binary representations for symbols and characters for most of the world's languages. Unicode 13.0 contains representations for 143,859 characters.

Instructions for the computer must be in its' language including numbers that are stored or used in computations. Numeric ***integers*** (whole numbers) are represented in computers using the positions of the bits and powers of 2 starting from right and working left.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Binary Number Bit Representations

As an example, the number 90 would be represented in binary as 01011010. Each bit is either 0 or 1 and is multiplied by the power of 2 at its position. The results are then added together.

0	1	0	1	1	0	1	0
0×2^7 +	1×2^6 +	0×2^5 +	1×2^4 +	1×2^3 +	0×2^2 +	1×2^1 +	0×2^0
(0×128) +	(1×64) +	(0×32) +	(1×16) +	(1×8) +	(0×4) +	(0×2) +	(0×1)
0 +	64 +	0 +	16 +	8 +	0 +	2 +	0

= 90

Binary Number Conversion

The limit to the numbers that can be stored using 8 bits is 255 with all 8 bits being 1's. To store larger numbers, two Bytes would be used with a combination of sixteen 1's and 0's. That would allow storing numbers as large as 65535. To store larger numbers, more Bytes could be used. To store negative numbers and floating-point numbers (numbers with a decimal or fractional part), other numbering schemes are used such as the two's complement and floating-point notation.

Images (which are made up of pixels) are stored by converting each pixel to a numeric value which is then stored in binary. Sound is stored using samples that are converted to the nearest numeric value.

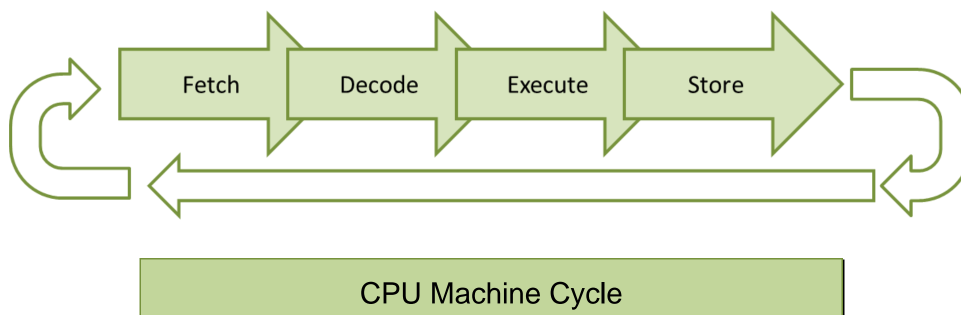
Programming Languages

Although the computers' language, referred to as machine language, is a binary language, it would be tedious for us to write instructions for computers in Binary. Machine language is one of the two languages commonly referred to as ***low-level languages*** for computers, the other being Assembly language. Assembly language consists of very basic instructions like move a value into memory, add another value to that one, and store the result in memory. An assembler is used to convert Assembly language programs into executable machine code. Both of these low-level languages mirror the operations of the CPU in simplicity and basic operations.

The CPU goes through what is called a ***machine cycle*** in which it performs the same series of simple steps: fetch, decode, execute, and store. The instructions executed during a machine cycle are very simple, but billions of instructions can be processed per second.

CPU Machine Cycle

1. fetch the required piece of data or instruction from memory
2. decode the instruction
3. execute the instruction
4. store the result of the instruction



CPU Machine Cycle

The processing power of a CPU is dependent upon the number of instructions that the CPU can execute per second, and is measured in hertz (cycles-per-second). This is often referred to as the CPU clock-speed. This does not refer to a wall clock, but the computer's clock or the internal timing of the computer. A one-gigahertz (1 GHz) CPU can execute one billion cycles (instructions) per second, and the clock-speed would be one-gigahertz.

Writing software (programming) in a low-level language is possible, but ***high-level languages*** provide a much easier way. As more software was written, high-level languages were introduced to make programming easier and more efficient. In high-level languages, multiple instructions are combined into a single statement. There are hundreds of high-level languages (700+) that have been developed. Today there are approximately 250 high-level programming languages in use by programmers. Some of these are used extensively, others not so much. Each language was created for a purpose and has benefits and limitations as well as a following, proponents, and detractors. The following is a short list of some popular high-level languages and their intended uses.

BASIC	Beginners All-purpose Symbolic Instruction Code
C, C++	powerful general-purpose programming
COBOL	Common Business-Oriented Language - business programs
FORTRAN	FORmula TRANslator for math and science
Pascal	teaching programming
Java	applications running over the internet
JavaScript	Web site operations (not related to Java)
PHP	web server applications and dynamic web pages
Python	general-purpose applications and data handling

Popular High-level Programming Languages

Writing software in any of these languages is much easier than the low-level languages of Machine and Assembly, but the computer is still only interested in machine language. To translate programs written in a high-level language to the machine language for the computer, compilers and interpreters are used.

A ***compiler*** translates the high-level language into a separate machine language program. Software engineers refer to this as compiling or "building" the program. A "Build" is a compiled version of the software and the program can then be run whenever needed because it is a stand-alone executable program. Java uses a compiler.

An ***interpreter*** on the other hand, reads, translates, and executes the program one line at a time. As an instruction in the program is read by the interpreter, it converts the instruction into machine language and then executes that one instruction. The Python programming language is an interpretive language and uses an interpreter to execute the instructions.

The instructions (programs) written by programmers are referred to as ***source code***, which is written using a text editor in an ***Integrated Development Environment (IDE)***. IDE's are software applications that include integrated tools for software developers to write, execute, and test software. There are many IDEs available, but they are very similar in the way that they look, are used, and operate.

Developing Software

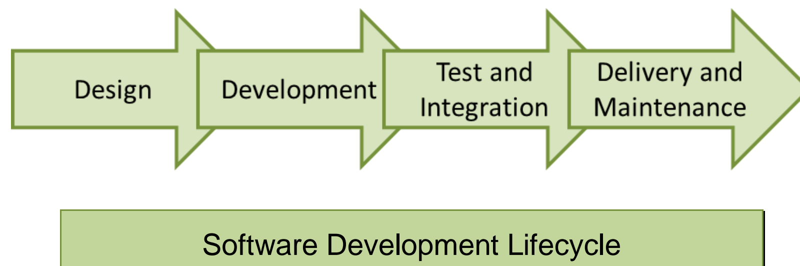
There are specific phases in the process of developing software that provide for the development of accurate, maintainable, and scalable software, that meets the project or program requirements. These phases include design, development, test and integration, and delivery and maintenance. But before any work can begin, a complete understanding of what the program is supposed to do is required. This is derived from the project or program requirements.

Requirements

The requirements for a computer program detail what the program is supposed perform. How it will do what it is supposed to do will be determined as the design phase is completed during the software development phase (SDP). ***Requirements Decomposition*** is the act of discovering in detail from the requirements what the program is required to accomplish. As requirements are reviewed, additional information may be needed and questions may arise. It is important to determine the specifics before moving forward. This process also assists in decomposing the project into manageable "chunks" in terms of the schedule and team assignments for development. Once the requirements are thoroughly understood, the software development lifecycle begins.

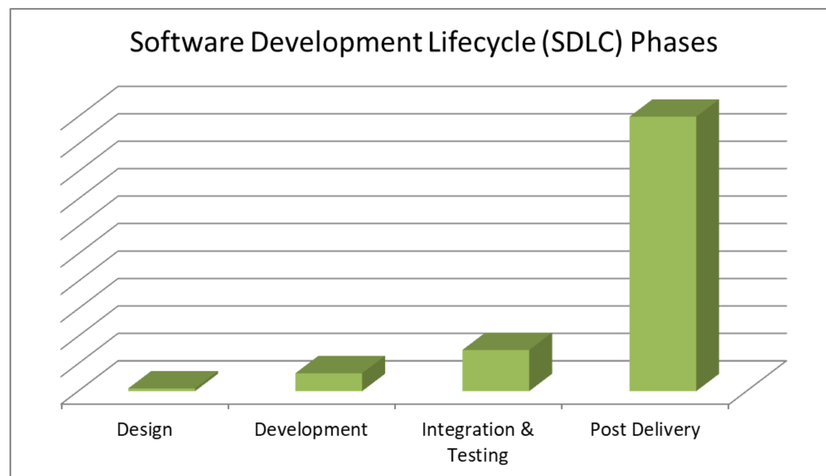
Software Development Life Cycle (SDLC)

The Software Development Life Cycle includes the steps necessary to design, develop, deliver, and maintain the computer program. The phases follow one another and are often accomplished by different teams members with collaboration as questions and issues arise. As an example, a software developer may meet with a design engineer to clarify information in the design, or a Test Team member may contact a software developer regarding test results.



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 life cycle due to the increased costs of making changes or fixing errors later in the process (errors in code are referred to as bugs). The sooner an issue is resolved, the less rework and testing of the code are needed. This is highlighted in the chart below from the IBM Systems Sciences Institute.



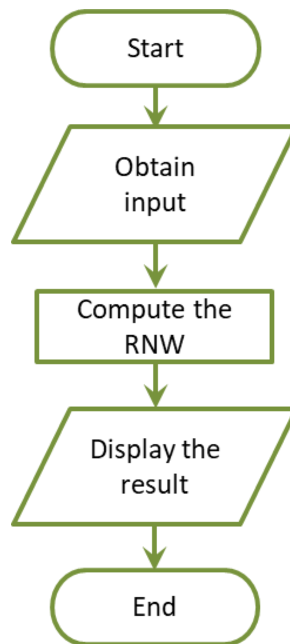
Cost of Fixing Errors by Phase

Software engineering tools that assist in the design (and development stage as well) include ***pseudocode*** (sort of code). Pseudocode is a short-hand version of the order of operations for a program. Consider a requirement for a program that obtains age and salary information from a user, computes Recommended Net Worth, and displays the result. The pseudocode for the solution might be:

```
Step 1    Start the program
Step 2    Obtain the age and salary information
Step 3    Compute the RNW (age x salary divided / 10)
Step 4    Display the output
Step 5    End the program
```

Pseudocode

A ***flowchart*** often provides a clearer representation of the ***algorithm*** (logical steps to the solution). Various geometric shapes are used to indicate different operations (shapes may vary depending on industry). The order of operations is typically top down, and lines with arrows are used to indicate the order or flow of control. 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. They can also be used to assist programmers when developing a complex part of a program.



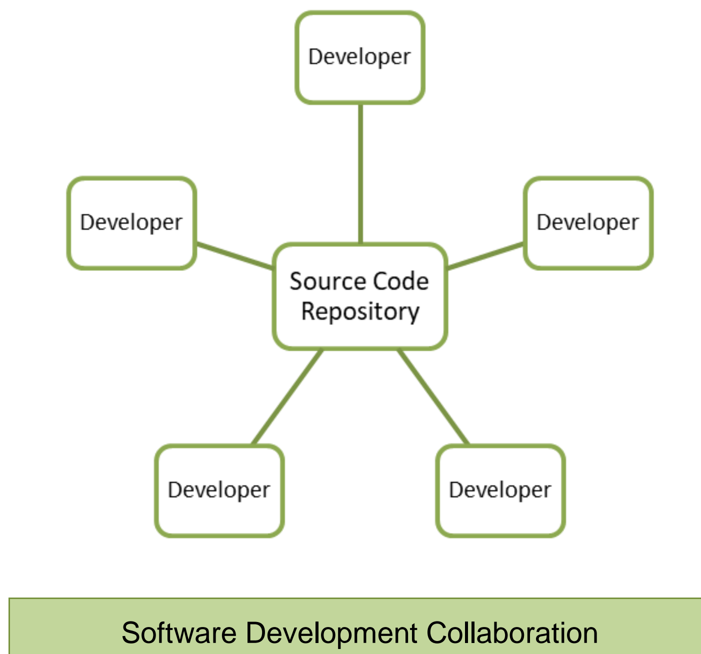
Example Flowchart

A flowchart can be a simple sketch or developed using a flowcharting application such as LucidChart® or SmartDraw®. Flowcharts are often required to be delivered to a development team or subcontractor together with specific requirements for the code, and are often required in customer documentation.

Many software engineers use a combination of tools. Pseudocode 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. 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 is as error-free as possible.

Development

Once a design is complete (or nearly complete since some aspects of the solution may not be knowable during design), the development phase begins. The development phase includes writing the code that will be executed to produce the desired result and meet the requirements. Most often, the development of a program is divided among multiple programmers and requires collaboration and regular discussion to ensure a cohesive solution. To manage software development projects and enable multiple people to work on the same program at the same time, a ***Configuration Management System (CMS)*** is used with a source code repository that stores and maintains all of the program files.



Software Development Collaboration

Programmers access this repository to obtain a copy of a file containing the source code to add functionality or make modifications. The code is written in the copy of the file, and this changed file is tested with the other files in the source code repository. After testing, the modified file is placed into the repository and is used by all of the other programmers in place of the original file. The original file is retained by the configuration management tool as a version control mechanism.

If a new file needs to be created, it is created in the configuration management tool and added to the source code repository. CMS tools facilitate collaborative

development, and version control of the files and the overall project. Many industries and customers require their use.

Many configuration management systems have integrated suites that include: scheduling and tracking, task assignment, defect reporting, and issue tracking systems. In addition, tools for software teams and software project managers are commonly used in industry to plan and measure project progress, and to provide visibility into the design, schedule status, cost, and quality of the code.

Software Development Processes

For the software development phase, the ***Agile Development Process*** is a popular method in use today. Agile processes go by various names (a few listed below), but all are iterative and incremental software methodologies. This process of developing portions of the software and adding them to the overall project incrementally is commonly referred to as ***Iterative Enhancement*.**

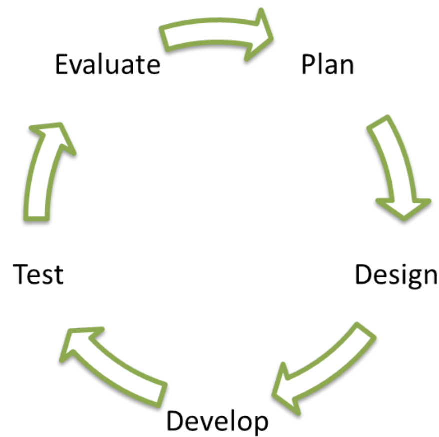
Common Agile Methodologies

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

Agile Software Development Methodologies

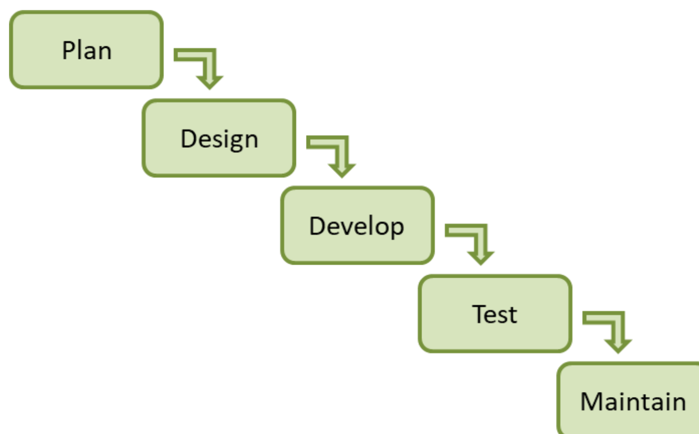
A key component of the Agile Development Process is the ***sprint*** (the development period between status meetings). Sprint status meetings (scrums) are review and planning events that occur regularly. Tasks completed from the previous sprint plan are reviewed, and completed work is demonstrated to stakeholders for feedback and approval. The tasks that were not completed from the previous sprint plan are reviewed with a course of action (re-plan) for the next development cycle. The scope of new work that will be completed during the next sprint cycle is planned, and engineers are assigned to the tasks.

The phases in the Agile Development Process include: Plan, Design, Develop, Test, and Evaluate, and are repeated during each sprint. Once delivered, the project would be in the maintenance phase.



Agile Development Process Phases

Another software development methodology is the ***Waterfall model*** which uses similar phases, but they are sequential, and are non-repeating. Each phase depends on the completion of the previous phase, although there may be some overlap. The Waterfall model was used extensively in the past, and is still common in some industries today. The maintenance phase follows the delivery of the software and includes updates.



Waterfall Development Process Phases

Test and Integration

As development is completed, the next phase in the software development life cycle is test and integration. In the initial test phase, the programmer runs the program to ensure that there are no errors in the code, and that it performs correctly (meets the requirements). In large organizations, a test team or test engineer will also run the program and report any errors found to the developer for correction. On large-scale programs, a formal Test and Integration Team would be responsible for this phase and would run a variety of tests including: Unit Tests on the modules (portions of code being added or modified), and an Integration Test which verifies that the parts of the program work well together when the new code is integrated into the overall project. Adding the new or modified code into the program may introduce new errors which must be corrected. Regression testing compares new test results with previous results and ensures that the program functions correctly.

Types of Errors

The three types of errors that are looked for during the test phase are syntax errors, logic errors, and runtime errors.

Syntax errors have to do with violating language specific rules like indentation and punctuation and are found by the compiler or interpreter and the code will not execute. The programmer must correct these and most IDEs will highlight them as an aid in development.

Logic errors are errors in the algorithm or the way that the algorithm was written by the programmer. For example, if the requirement is that the program multiply a number by two only if it is greater than ten, and the programmer writes the code so that a number is multiplied by two if it is less than ten, that would be a logic error. The program compiles and runs, but it produces incorrect results.

Runtime errors are logic errors that cause the program to stop executing. An example would be a part of the program attempting to divide a number by zero. The IDE will provide a Traceback of the sequence causing the error. Runtime errors can be avoided by thoroughly designing and testing the algorithm.

Delivery and Maintenance

The final phase of the software development life cycle is the delivery and maintenance phase. In this phase, the program is delivered to the client or customer and a period of maintaining the program begins. Maintenance of a program would include updates that fix errors or security issues found after initial delivery, or upgrades that provide additional functionality or capability. Updates to software programs are commonplace today.

Ergonomics

The set-up or arrangement of the computer and furniture to minimize the risk of injury or discomfort is a field of engineering called ergonomics. It includes the study of the physical effects of repetitive motion and working in a stationary position for an extended period. As more people spent their days working at computers, a variety of health issues surfaced. Some guidelines include:

- Monitor position – with respect to eye level (dry eyes)
  - Eyes should be looking slightly downward
- Adjustable chair – arm posture (tennis elbow)
  - Elbows should be at 90-degree angles
- Proper posture – back posture (lumbar issues)
  - Lumbar support and back straight
- Taking periodic breaks – eye strain, posture, repetitive motion
  - 20-20-20 rule says every 20 minutes look 20 feet way for 20 seconds
  - Standing or walking away for few minutes
- Adequate lighting – eye strain
  - Dark areas and dark backgrounds cause eye strain

Ergonomic Guidelines

Chapter 1 Review Questions

1. Computers are simply _________ __________ devices.
2. The physical parts of the computer are referred to as _____________.
3. The CPU is considered the ___________ of the computer.
4. The CPU performs basic ____________ and controls computer _________.
5. Main memory (RAM) is ________ and is erased when a computer is turned off.
6. __________ Storage device memory is non-volatile and is retained when the power is turned off.
7. A computer keyboard, mouse, and camera are examples of _________ devices.
8. Computer monitors, speakers, and printers are examples of __________devices.
9. Sets of programmed instructions for a computer are referred to as __________.
10. _____________ and _____________ are the two basic types of software.
11. The language of computers is a _________ language.
12. The smallest information representation in computing is a _____ or binary digit.
13. A binary digit can have a logical state of _____ or _____.
14. A Byte is a combination of ________ bits that are either one or zero.
15. The number represented by 0110 1001 is _______.
16. The binary representation of the number 255 is ___________.
17. The names of the two low-level languages are __________ and __________.
18. A Machine cycle consists of ________, ________, ________, and ________.
19. A 2 GHz (gigahertz) processor can execute __________ instructions per second.
20. High-level languages make programming a computer _____ and more _______.
21. Java is a _______ -level language.
22. A (n) ___________ translates a high-level language into a separate machine language program.
23. A (n) ___________ reads, translates, and executes a program one line at a time.
24. The four steps in the Software Development Life Cycle are ________, ____________, _______________, and _________________.
25. The costs associated with fixing errors in code are __________ when caught early in the process.

26. A written shorthand version of the steps to complete a task in a computer program is called _______________.
27. The act of discerning in detail from the requirements what the program is to accomplish is called ___________________ ____________________.
28. A set of logical steps taken to complete a task is called a(n) _____________.
29. Plan, design, develop, test, and evaluate are the five steps in the ___________ development process.
30. The three types of programming errors are _________, __________ and ____________ errors.

Chapter 1 Short Answer Exercises

31. Explain the major difference between main memory and secondary storage.
32. List at least three (3) input devices.
33. List at least three (3) output devices.
34. List the two (2) types of software.
35. Write the word Java using the binary representations of the letters.
36. Write the binary representation for the number 176.
37. List the two low-level languages.
38. What is the purpose of a source code repository?
39. List the five phases of the Agile Development cycle.
40. Explain the difference between logic and syntax errors.

Chapter 1 Programming Exercises

41. Write the pseudocode for the steps required to determine the total price for some number of items entered by the user priced at $9.00 each with a 7% sales tax.
42. Draw a flowchart of the steps in Programming Exercise 1 above.
43. Ensure that you have access to a copy of Eclipse (ref. Appendix B).

Chapter 2

Java Language Basics

The Java programming language was initiated as a project in 1991 by James Gosling, Mike Sheridan, and Patrick Naughton, and was originally designed for embedded systems. With the introduction of web browsers, and the reduction in prices 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.

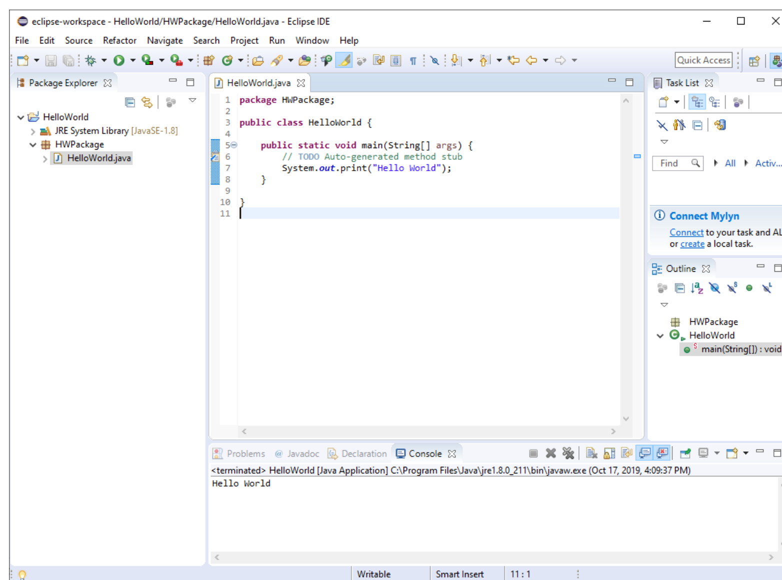
JVM - The Java Virtual Machine (JVM) enables computers to run Java programs. The JVM converts Java bytecode into machine language, manages memory, and is part of the JRE. It allows Java programs to run on most devices and operating systems.

JDK - The Java Development Tool-kit (JDK) is a development environment for creating Java programs and applets that includes the 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.

The Eclipse IDE

The Eclipse IDE provides all of the Java development tools necessary to develop programs in Java. It is the most widely used IDE for Java programming, is used by many companies, and is suitable for starting out in Java as well as advanced programming and collaboration. It is free to download and use, and is similar to most IDEs in look-and-feel and capability. The Eclipse version used in this text is 2019-06. Obtaining a copy is covered in Appendix B and Getting started in Eclipse is covered in Appendix C and should be completed before continuing.



The Eclipse IDE

Parts of a Java Program

The "Hello World" program from the appendix is repeated below with line numbers for explanations of the parts. On line 1 is the package or project name. The package in Java is used to group related classes and files. On line 3 is the class name for the program. Every Java program has at least one class. The word public is an access specifier indicating that the class is publicly accessible for use. Other access specifiers will be covered later.

```
1 package helloPackage;
2
3 public class HelloWorld {
```

The brace following the class name on line 3 begins a block of code for the class. The closing brace for the class is at the margin on line 10 and aligns vertically with the word public preceding the class. Aligning braces with the block of code that they close is important for readability and the IDE will automatically align and indent them. Line 5 is the header for the main method for the program. This is where execution of the program begins when it runs. Note the brace at the end of the line which begins another block of code. The closing brace is on line 8 and is aligned with the indented header for the main method (the word public).

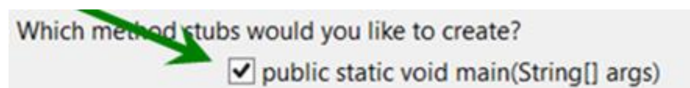
```
 1 package helloPackage;
 2
 3 public class HelloWorld {
 4
 5     public static void main(String[] args) {
 6         // TODO Auto-generated method stub
 7         System.out.println("Hello World!");
 8     }
 9
10 }
11
```

The parts of the main method header are:

- public – specifying that it is accessible outside the class
- static – designation that it is a class method and not associated with any object
- void – indicating that it has no return value
- (String[ ] args) – a provision for command line arguments

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

Line 6 is a comment added by Eclipse indicating that the main method was automatically created as a result of the checkbox when the class was created.



Line 7 is an output statement executed by the program. This line ends with a semicolon which is the end-of-line marker in Java. To the compiler, a semicolon indicates the end to a statement just as a closing brace indicates the end of a block of code.

Syntax and Grammar

Each programming language has some characteristics and rules that must be followed when writing programs in that language. Two of these are the syntax and grammar of the language.

The ***syntax*** of a language refers to the rules for properly combining symbols, operators, and punctuation, as well as the proper use of operators.

The ***grammar*** of a programming language determines the structure of the sentences containing the symbols, operators, and punctuation that make up the instructions for the computer.

Another characteristic of programming languages is the use of ***keywords*** or reserved words. Keywords are reserved by the language for a specific use and cannot be used for another purpose. Eclipse will display them in a color font to highlight them as shown below (package, public and class).

```
1 package helloPackage;
2
3 public class HelloWorld {
```

The following is a list of some of the Java keywords.

boolean	catch	char	class	double
else	extends	final	finally	for
if	implements	import	int	new
package	private	public	return	static
String	throw	try	void	while

Java Keywords (partial list)

Note: True, false, and null are literals and are reserved in Java as well.

Comments

In addition to the tools mentioned in Chapter 1 for designing and developing software, comments within the code can be helpful and are often required.

Comments in programs are lines of code that are not executed, and are ignored by the compiler. They are provided for human readers, and are used to clarify values or sections, or explain complex operations. This is important because most software is maintained, updated, and expanded. Code is written once, but is read many times, and the person who wrote the code may not be the person making the modifications, or the person who wrote the code may not remember why a section was written a certain way or why a specific value was used. Adding comments to code while it is being written can save hours of reading through the lines later when the code is being changed.

Comments can also be used as a development tool. Pseudocode can be written in the edit window as a comment to act as a place-holder or reminder that will be replaced later by actual code.

Single line comments in Java begin with two forward slashes. The Eclipse IDE used in this text will color code comments in green font as the default. For multi-line 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 multi-line comment.

```
// a single line comment in Java

/* a multiline
  comment in Java
*/

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

Java Comment Types

Variables

Variables are elements in programs that are used to allocate memory and store information that the program will use. They are called variables because what is stored in them can vary as the program runs. A variable is declared and named

by the programmer to allocate memory for use by the program. The computer remembers the memory address of where it is stored, and the programmer refers to it in the program by the name that was used to declare it.

In the Hello world program, a literal string (sequence of characters) was passed to the output statement. A variable could also be used as shown in Ex. 2.1 below. In the example, a String variable is declared on line 7 and is assigned the value "initially". This is referred to as initializing the variable. The equal sign is the ***assignment operator*** in Java and is used to assign a value to a variable. The computer allocates memory for a String named myWord and stores the value "initially" there. On line 8, the variable is passed to the output statement. The variable is then assigned "currently" on line 10 which replaces the value that was previously stored and line 11 displays it again. The Eclipse console output is shown below the program. The use of *println()* instead of print on lines 8 and 11 causes the output to be on separate lines.

Ex. 2.1 – String Variable Declaration, Initialization, and Modification

```
5    public static void main(String[] args) {
6
7        String myWord = "initially";
8        System.out.println(myWord);
9
10       myWord = "currently";
11       System.out.println(myWord);
12
13   }
```

```
<terminated> Ex_2dot [Java Application]
initially
currently
```

Variable Declaration and Utilization

Technical Notes

Notice in the example that when the variable myWord is assigned a new value, it is not preceded by the word String. String is the data type (covered next) of the variable which is only used when a variable is being declared.

Data Types

The example in Ex. 2.1 declared a String variable and assigned it a value that was a literal string (a series of characters within double quotes). When a variable is used to store a number, a different data type is declared. The data types in Java include int for integers (whole numbers), and float and double for floating-point numbers (numbers with a decimal). The data type tells the computer how much memory to reserve for the variable. To store an integer, or float, 4 bytes are allocated. To store a double, 8 bytes are allocated to accommodate more precise numbers (the double data type is recommended for fractional numbers). The numeric ranges for integers and doubles in Java are sufficient for most program requirements and are used for whole and fractional numbers in this text.

Table 2.1 below lists some of the basic data types used in Java.

Type	Size	Description
byte	1 byte	whole numbers from -128 to 127
short	2 bytes	whole numbers from -32,768 to 32,767
int	4 bytes	whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4 bytes	single precision floating point numbers (6 decimal places) from 1.40129846432481707e-45 to 3.40282346638528860e+38
double	8 bytes	double precision floating point numbers (15 decimal places) from 4.94065645841246544e-324 to 1.79769313486231570e+308
boolean	1 bit	Logical values - true, false
char	2 bytes	single character/letter or ASCII value

Table 2.1 - Java Primitive (simple) Data Types

The next example declares two integer variables and uses them in three different output statements to highlight some additional considerations when working with numbers in Java. Each of the output statements uses *println()* which adds a line feed after the output is displayed. The first output statement adds the two

values of the variables, but the second does not. The difference is the occurrence of the literal string that precedes the expression. This causes Java to interpret the plus sign as adding additional textual output and not numbers. Note that num1 could not be mathematically added to the literal string. The value stored in num2 could be added to num1, but the individual values are displayed. The third output statement forces the addition of the values using parenthesis.

Ex. 2.2 – Numeric Variable Declaration, Initialization, and Output

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

    int num1 = 23;
    int num2 = 17;

    System.out.println(num1 + num2);

    System.out.println("num1 plus num2 equals " + num1 + num2);

    System.out.println("num1 plus num2 equals " + (num1 + num2));

}
```

```
<terminated> Ex_2dot2 [Java Application] E:\
40
num1 plus num2 equals 2317
num1 plus num2 equals 40
```

Program Output

Variable Names

The variable naming convention most used in Java is called 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 in uppercase. This aligns with W3C (World Wide Web Consortium) as well as other Guides and Standards for the language. When naming variables, there are a few rules that need to be followed:

- none of the Java key words can be used as a variable name
- there cannot be any spaces in the name
- the first character must be a letter (or an underscore)
- uppercase and lowercase letters are distinct

In addition, the name of a variable should describe the data that it stores. Software Engineering Principles and Programming Standards require descriptive variable names to enhance readability. A longer name is usually better. Using variable names like `var` or `pd` are ambiguous and make removing errors (debugging) and maintaining the code more difficult. If a comment is needed to describe a variable, then the name of the variable is inadequate.

Table 2.2 below lists some examples of good and bad variable names.

Name	Comment
x	does not describe the data being stored
5star	cannot begin with a digit
myNum	Ambiguous
how?many	can only contain letters, digits, or underscores
windSpeed ✓	proper naming
interest ✓	proper naming
grossPay ✓	proper naming

Table 2.2 - Variable Naming

Error Notifications

Java is case sensitive and common programming errors include case errors and misspelling a previously declared variable name. These types of errors will be highlighted in most IDEs as they are introduced so they can be corrected immediately. If errors exist in the code and an attempt is made to run the program, it will cause a compiler error and will not run. In the error example below, the variable `number` is declared with all lowercase letters, but an uppercase letter is used in the output statement. The IDE underlines the variable name with red and places an error indicator at the margin on the line containing the error. Hovering over the indicator provides a description of the error.

```
6
7         int number = 3;
8  Number cannot be resolved to a variable nt(Number);
9
```

When an attempt is made to run the program, a compiler error is output to the console area of Eclipse with a description and line number containing the error.

```
 5⊖    public static void main(String[] args) {
 6
 7        int number = 3;
 8        System.out.print(Number);
 9
10    }
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
	Number cannot be resolved to a variable

	at ErrorPackage.Ch2Error.main(Ch2Error.java:8)
```

Error Example

Variable Assignments

When a variable is declared, it is typically assigned an initial value. This is referred to as initialization. A single equal sign is the assignment operator, and the variable being assigned is on the left side of the operator. The right side of the assignment operator can be a value or an expression. In this statement, a variable userAge is declared as an integer, and is assigned the integer value 29.

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

As shown previously, a variable can be changed while the program is running. The new value assigned to a variable overwrites the old value in memory where the variable is stored.

Ex. 2.3 – Changing the Value Stored by a Variable

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

    int temp = 3;
    System.out.println(temp);

    temp = temp + 2;
    System.out.println(temp);

}
```

In Ex. 2.3, a variable is declared and initialized. The variable is then changed using the old value in the expression to assign a new value. Note that a variable can be on both sides of the assignment operator. The right side is evaluated first by the computer and the result is assigned to the left side.

Constant Variables (Named Constants)

A ***constant*** is a variable that cannot be changed by the program, must be initialized when declared, and cannot be assigned a new value. There are several situations when this is preferred. One is to eliminate the use of unidentified numbers in programs, and another is to ensure that a specific value is used throughout the program.

An unidentified number in programming is referred to as a ***magic number***. They are literal numbers in a program without an obvious meaning. When a program is being modified and an expression uses a magic number, it may be difficult to determine what the number means even by the original programmer. As an example, the following line appears in a program that declares a double as `diameter`, but the meaning of 3963.2 is unknown. Since the equation results in a diameter, it appears to be a radius. It isn't clear.

```
double diameter = 2 * 3963.2;    // What is 3963.2 ?
```

By using a named constant in the code, the meaning is clear. The constant is declared and initialized and then used in place of the literal number wherever it is needed in the program. Constants are declared using the ***final*** keyword followed by the data type, name, and initialization. The naming convention for constants is all uppercase letters with underscores between words.

```
final double EARTH_RADIUS = 3963.2;

double diameter = 2 * EARTH_RADIUS;
```

Named constants are also used to ensure that the same value is used throughout the program and by all programmers. If multiple programmers are working on a program that calls for them to use the radius of the earth in various equations, they can use a named constant to ensure that the same value is used. The earth is not a sphere and there are multiple values for its radius.

Named constants also prevent typographical errors when the same value is being used multiple times. In addition, when a new value is needed for the constant, the change is made in a single place in the code. As an example, a scientist overseeing a program using EARTH_RADIUS may decide that the equatorial radius being used in the program should be changed to the pole radius of 3950.0. It will only need to be changed to the new value in one place in the code. This eliminates the possibility of typographical errors or missing an occurrence when updating all of the equations that use the value.

Global Variables

A global variable is a variable that is declared outside all methods including the main method. This makes them accessible to all parts of the program. Most programming standards do not permit their use except when they are global constants because they could be changed arbitrarily by any part of the program. This makes debugging more difficult. Since a constant cannot be changed, a global constant provides for using a consistent value across all of the code in a program. Java doesn't explicitly have global variables since every variable must belong to a class, but once declared, they can be accessed using the class name.

Mathematical Operators and Expressions

The operators for mathematical expressions in Java include: addition (+), subtraction (-), multiplication (*), division (/), and the modulus operator (%).

Operator	Description	Example	Result
+	addition	2 + 3	5
-	subtraction	72 - 12	60
*	Multiplication	4 * 6	24
/	Division	7 / 2	3.5
%	remainder (mod)	17 % 4	1

Table 2.3 - Arithmetic Operators

The mathematical operators combine with variables and expressions in programs to perform operations. The lines of code in Ex. 2.4 declare three integers and use them in various equations. Recall that the computer evaluates the right-hand side of the assignment operator, and then assigns the result to the left-hand side.

Ex. 2.4 – Mathematical Operators

```
int first = 3;
int second = 7;

int third = first + second; // third is assigned 10

first = third / 2;          // first is assigned 5

second = first * second;    // second is assigned 35
```

The results from division in Java are different for different data types and data type combinations. If one of the values is a floating-point number (number with a decimal), 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 and the decimal portion is discarded. One way to remember this is "int divided by int is an int".

```
System.out.println(10 / 3);     // displays 3

System.out.println(10 / 3.0);   // displays 3.333...

System.out.println(2.5 / 5);    // displays 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.

```
double result = 10 / 5;     // result is assigned 2.0
```

The modulus operator produces the remainder after division (sometimes referred to as modulo divide). The operand on the left of the operator is divided by the operand on the right and the result is the remainder after division.

```
int num1 = 10 % 3;      // num1 is assigned 1

int num2 = 20 % 7;      // num2 is assigned 6

int num3 = 1792 % 10;   // num3 is assigned 2
```

Precedence in Java is parenthetical expressions first, followed by 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. The use of parenthesis is often preferred even when they align with precedence. This enhances the readability of the expression and helps to eliminate errors. The equations in the table below are the same, but the results are quite different.

Expression	Result
4 * 2 + 15 / 5 – 2	9
4 * (2 + 15) / 5 – 2	11.6
4 * 2 + 15 / (5 – 2)	13
4 * (2 + 15) / (5 – 2)	22.6666…
(4 * 2) + (15 / 5) – 2)	9

Table 2.4 - Precedence and Parentheses

Mixed-type expressions are promoted to the higher data type in use. In an expression with an integer and a double, the integer is temporarily converted to a double, and the expression is promoted resulting in a double. The same rule applies to an expression with an integer and float.

Technical Notes

When using floating-point (fractional) numbers, the *double* data type is recommended over *float* for variables since it is more precise as the results below illustrate.

Results for 1.0 / 3.0 using float

```
&lt;terminated&gt; Ex_2dot4 [Java Application]
0.3333333432674408
```

Results for 1.0 / 3.0 using double

```
&lt;terminated&gt; Ex_2dot4 [Java Application]
0.3333333333333333
```

Programming Algebraic Expressions

When converting mathematical expressions into Java code, the translation may require adding operators and parentheses to ensure the correct result. As an example, the expression 3xy in algebra would produce a syntax error. The multiplication operator must be inserted as in 3 * x * y. When an expression contains fractions, precedence requires careful consideration to ensure that operations occur in the correct order. With extremely complex equations, breaking the expression into parts may be the best course of action.

Conversion examples:

$3x$	`3 * x`
$5xy$	`5 * x * y`
$x = \frac{3y}{2a}$	`x = 3 * y / 2 * a`
$x = \frac{y+5}{z-3}$	`x = (y + 5) / (z - 3)`
$x = \frac{n(n-1)y}{2+n}$	`x = (n * (n - 1) * y) / (2 + n)`

Converting Algebraic Expressions

Math Methods

The Java Math library contains constants like PI and methods for exponentiation, rounding numbers, and other common operations. To use these operations, the method is preceded by the library name "Math" as shown below. The functions are passed ***arguments*** which are values passed to methods and functions for their use. As an example, for rounding numbers, Java has a **Math.*round()*** function. The line of code below passes the number 9.4 as an argument to the round function. The function executes and the result is assigned to the variable.

```
double rounded = Math.round(9.4);  // rounded is assigned 9.0
```

For operations with exponents, the **Math.*pow()*** function is used. In this case, two arguments are required. The first argument is the number to be raised, and the second is the exponent.

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

In addition to the *round()* and *pow()* functions, the java.lang.Math library contains functions for performing other 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 others. Below is an example that uses the square root function.

```
double root = Math.sqrt(4);
```

Converting Data Types

Converting from one data type to another is referred to as ***casting***. The round method in the example below returns a double, which is then cast to an integer so that it can be assigned to the integer variable roundInt. The data type that the value is being cast to is in parenthesis.

```
int roundInt = (int)Math.round(9.4);  // roundInt is assigned 9
```

Obtaining Keyboard Input

To obtain input from the keyboard requires a Scanner. To use a Scanner, the class java.util.Scanner must be imported. Import statements are located between the project package and the class, and provide access to Java libraries.

Ex. 2.5 – Keyboard Input

```
package ex_2dot5Package;

import java.util.Scanner;       // import the Scanner class

public class Ex_2dot5 {

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

The line of code below declares a Scanner named in and assigns it a Scanner using "new" and "System.in" which is the standard system input source (the keyboard).

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

When a program needs to obtain keyboard input, a ***prompt*** is used to describe the input being requested. Once the requested data is entered, the user will press the *Enter* key. To obtain the input, a variation of the ***next()*** method is used. In the example below, the prompt requests a number and ***nextInt()*** is used which reads an integer. Note that *nextInt()* is preceded by the name of the scanner and the dot operator to access the method.

When the program runs, the prompt is displayed and the program waits. Once a value is entered in the console area of Eclipse and the Enter key is pressed, *nextInt*() will obtain the value and it will be assigned to the variable. After the output statement, the Scanner is closed (a recommended practice).

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

    int input;

    Scanner in = new Scanner(System.in);

    System.out.print("Enter a number ");     // prompt

    input = in.nextInt();                   // read the input

    System.out.print("The number entered is " + input);

    in.close();     // release the scanner
```

```
<terminated> Ex_2dot5 [Java Application]
Enter a number 25
The number entered is 25
```

Obtaining Keyboard Input

The Scanner has methods to obtain input for various data types. During the design phase of the program, different approaches and methods should be considered depending upon how the data will be used. Examples using the different versions of the method are shown below.

```
int num1 = in.nextInt();            // reads an integer

double number = in.nextDouble();    // reads a double

String word = in.next();            // reads up to white space

String line = in.nextLine();        // reads a line of text
```

Scanner Methods for Obtaining Input

Design Consideration

The *next()* method will read input until whitespace is encountered. Whitespace can be a space, tab, or line feed. Consider a program that requires the user to enter the name of a city. If the city name entered is Denver, then *next()* will read the full name. If the name of the city is New Brunswick, then *next()* would only read the word New. In this situation, using ***nextLine()*** would ensure that the entire city name would be read.

Formatting Output

The ***print()*** function that was used in previous examples to display output has two other versions. One version, ***println()*** adds a line feed after the output is displayed.

Ex. 2.6 – Line Feed Output

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

    System.out.println(10);
    System.out.println(100);
    System.out.println(1000);

}
```

```
<terminated> Ex_2dot6 [Java Application]
10
100
1000
```

Program Output

The other version for output is ***printf()*** which is used when formatting output. A ***format specifier*** is used to indicate the formatting to be applied. Arguments are passed to the function: the format specification(s) in quotes, and the value(s) or string(s) to be formatted. The specifier begins with "%", is followed by the formatting, and ends with the type: "f" (float), "d" (integer), and "s" (string). An integer is placed after a decimal in the specifier for the number of decimal places.

```
System.out.printf("%.2f", 25.6);    // displays 25.60
```

In Ex. 2.7 below, three different decimal specifiers are used. Note that using *printf()* eliminates the ability to use *println()*, so line feeds are added as separate output statements in the example.

Ex. 2.7 – Formatted Output

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

    double number = 123.45678;

    System.out.printf("%.2f", number);
    System.out.println();

    System.out.printf("%.3f", number);
    System.out.println();

    System.out.printf("%.4f", number);

}
```

```
<terminated> Ex_2dot7 [Java Application]
123.46
123.457
123.4568
```

Program Output

Technical Notes

Previously the "+" operator was used in output statements that combined text with numeric values. With format specifiers, a comma separates the specifier and the variable as shown in Ex 2.7.

A decimal place specifier that is fewer than the number to be formatted will cause rounding as shown in the example.

When a literal String is part of the output, the format specifier can be located within the same quotes as shown here.

```
System.out.printf("The number is %.2f", number);
```

When more than one variable is included in the output, the specifiers are included in the string portion in the order in which they are to be used in the output. The actual variables are included afterward as shown here. Notice that the numbers in the output were rounded due to specifying two decimal places.

```
double var1 = 123.456;

double var2 = 7.898;

System.out.printf("First %4.2f Second %4.2f", var1, var2);
```

```
<terminated> Ex_2dot6 [Java Application]
First 123.46 Second 7.90
```

To add commas for large numbers, a comma is included in the specifier after the percent sign. Other formatting techniques and methods will be covered later in the text.

```
double num1 = 123456.78;

System.out.printf("%,.2f", num1);
```

```
<terminated> Ex_2dot8 [Java Application]
123,456.78
```

This line of code adds the dollar sign to the output.

```
System.out.printf("$%,.2f", num1);
```

```
<terminated> Ex_2dot8 [Java Application]
$123,456.78
```

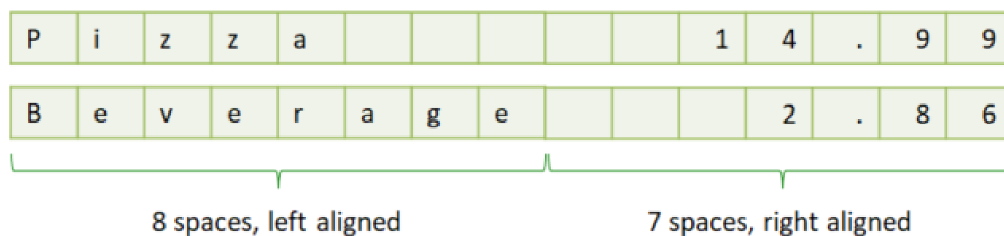
Numeric data is often output in columns and right-aligned. To designate a specific amount of space to use for output, the number of spaces is added to the format specifier before the decimal or designator if no decimal is used. A negative sign before the spacing value is used for left alignment of the output. In Ex 2.8, the descriptions are left-aligned and the prices are right-aligned.

Ex. 2.8 – Specifying Alignment and Output Spacing

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

    double price1 = 14.99;
    double price2 = 2.86;

    System.out.printf("%-8s", "Pizza");
    System.out.printf("%7.2f", price1);
    System.out.println();
    System.out.printf("%-8s", "Beverage");
    System.out.printf("%7.2f", price2);
}
```

```
<terminated> Ex_2dot8 [Java Application]
Pizza     14.99
Beverage   2.86
```

Program Output



In Ex. 2.8, the word "Beverage" used all of the allocated space. If a String or value does not fit in the specified space, the item will still be displayed entirely but without any spaces.

Escape Sequences

To insert certain characters and formatting within literal strings, escape characters are used. The ***escape sequences*** include: new line "\n", tab "\t",

double quote \”, and back slash “\\”. The sequence is surrounded by quotes unless it is within a literal string. 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.

Escape	Result
\n	output a line feed
\t	output a tab
\’	output a single quote
\”	output a double quote
\\	output a backslash

Table 2.5 - Escape Characters

Ex. 2.9 – Escape Sequences

```
System.out.println("Line feed \n mid-sentence.");

System.out.println("A tab \t mid-sentence.");

System.out.println(" \"quotes around.\" ");

System.out.println("A backslash \\");
```

```
<terminated> Ex_2dot9 [Java Application] E
Line feed
 mid-sentence.
A tab 	mid-sentence.
 "quotes around."
A backslash \
```

Strings

A ***string*** is a sequence of characters. When a literal string is written in the code, it is surrounded by double quotes and is referred to as a ***string literal***. A String variable is the String data type in Java and can contain a large amount of text

(2,147,483,647 characters). A character is also a data type that can store a single character. It is declared using ***char*** and is assigned using single quotes. A String can contain a single character, but is always surrounded by double quotes.

```
String name = "Rashid";
String initial = "R";
char letter = 'A';
```

Characters are stored using their ASCII numeric values, so the assignment statement above would actually store 65 as the value for "A" (see Appendix A). The computer knows that the data type is *char* and handles the translation.

The characters contained in a String are in positions called ***indexes*** beginning with index zero. In the String below, the "e" is at index 3 although it is the 4th letter in the word.

f	r	i	e	n	d	l	y
0	1	2	3	4	5	6	7

To obtain a character from a String the ***charAt()*** method is used and is passed the index of the character to obtain. Note that the method is preceded by the String variable's name and the dot operator (or dot notation). In the code below, the character "e" would be assigned to the *char* variable letter.

```
String word = "friendly";
char letter = word.charAt(3);
```

To obtain a portion of a String, the ***substring()*** method is used. When two arguments are passed to *substring()*, the first argument is the index of the starting point of the substring to obtain, and the second argument is one index beyond the portion to obtain. As an example, the following statements assign "friend" to the String variable portion.

```
String word = "friendly";
String portion = word.substring(0,6);
```

When one argument is passed to *substring()*, the argument is the index of the starting point of the substring to obtain, and the ending point is the end of the String. The following statements assign "car" to the String variable ending.

```
String phrase = "A blue car";

String ending = phrase.substring(7);
```

Strings also have a method for obtaining their length. The ***length()*** method returns the number of characters in a String including any spaces. In the code below, 8 would be assigned to the variable len. Remember that the indexes for the characters are 0 through 7, one less than the length.

```
String word = "friendly";

int len = word.length();
```

Combining two or more Strings is referred to as ***concatenation***. The "+" operator is used to concatenate Strings. No space will be added between the Strings. They are simply joined together as shown in Ex. 2.10 below.

Ex. 2.10 – String Concatenation

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

    String word1 = "First";
    String word2 = "Second";
    String word3 = word1 + word2;

    System.out.println(word3);
}
```

```
<terminated> Ex_2dot10 [Java Application]
FirstSecond
```

To add a space between the Strings, it can be added to either of the Strings being combined or separately as shown here.

```
String word3 = word1 + " " + word2;
```

```
<terminated> Ex_2dot10 [Java Application]
First Second
```

The lines of code in Ex. 2.11 below prompt for a first and last name, and use the ***next()*** method to obtain the input. The inputs are stored in String variables, and concatenation is used in the output statement to add a space between the names, and to add a period at the end.

Ex. 2.11 – String Concatenation

```
String firstName, lastName;

Scanner in = new Scanner(System.in);

System.out.print("Enter your first name ");
firstName = in.next();

System.out.print("Enter your last name ");
lastName = in.next();

System.out.print("Hello " + firstName + " " + lastName + ".");
```

```
<terminated> Ex_2dot11 [Java Application]
Enter your first name George
Enter your last name Eliot
Hello George Eliot.
```

Technical Notes

The ability to manipulate Strings is an important skill in programming. Most interfaces receive input as Strings and perform error checking and conversion of numeric values. Some of these techniques are covered in the next chapter.

Methods, Functions, and Dot Notation

In this chapter, the examples used methods and functions to perform operations. Although the terms tend to be interchangeable, for clarification, a method typically refers to functionality associated with an object of a class, and a function is not. As an example, the String variables declared in this chapter are declared using an upper case "S" whereas the integer and double data types are not capitalized. Classes are capitalized in accordance with programming standards, and the String is a class. The String class has other methods available including *toLower()* and *toUpper()* for converting**.** To use them, the variable name

for the String is followed by the dot notation and the method name. In the statements below, a String is declared called word. The variable is actually an object of the String class and inherits the methods of that class. The methods are accessed using the dot operator as shown on the line below that uses *length()*.

```
String word = "friendly";

int len = word.length();
```

The Scanner is also a class that provides methods some of which were used in this chapter. Again, after declaring a Scanner, dot notation was used to utilize the methods. The Math functions that were used did not require declaring a Math object before using them, but did use dot notation. The Package (Library) name Math, the dot operator, and the function name were used as shown here.

```
double root = Math.sqrt(4);
```

Classes and Objects are covered in a later chapter.

Programming Style and Standards

Proper programming style and format of the code make a program easier to read and maintain, and help to prevent errors from being introduced. The cost to fix errors increases dramatically as the software development cycle progresses. Anything that decreases the chances of introducing bugs is welcome and utilized. For these reasons, Programming Standards have been developed to provide uniformity and enhance the readability and maintainability of code. They include proper spacing and indentation among others. An entire program could be written on a single line and the computer would have no problem with it, but someone trying to debug the code or add functionality would have a very difficult time. Appendix F contains a set of Programming Standards for Java.

Technical Notes

Programming Standards and Style Guides are used in industry to ensure the readability and maintainability of programs due to the time and cost associated with poorly written and ambiguous code.

Chapter 2 Review Questions

1. The characteristics and rules that must be followed when writing programs in a high-level language are ___________ and ___________.
2. Words that are reserved in a programming language are called ____________.
3. Words added to programs to explain complex areas or to add clarity and are not executed when the program runs are _______________.
4. __________ are used to store values in memory.
5. The __________ data type is used to store whole numbers.
6. Numbers with a decimal should be stored in the _________ data type.
7. The equal sign is used to ______________ a value to a variable.
8. The ______ side of the assignment statement is assigned to the ______ side.
9. A variable must be __________ before it can be used by the program.
10. Variable names (can or cannot) _______ begin with a number.
11. A ___________ should be used in place of a magic number.
12. A value passed to a method is called a(n) ___________ .
13. The __________ escape character is used to produce a tab.
14. Converting an item to a different data type is known as ___________.
15. Errors in programing are typically referred to as __________.
16. Which of the following variable names follow proper naming conventions?
    a. average
    b. 8pieces
    c. netPay$
    d. grossPay
    e. hourlyRate
17. The first character in a String occupies index ________ in the String.
18. The _________ data type can store one character.
19. Concatenation refers to _____________ two or more Strings.
20. Programming _____________ provide uniformity and enhance the readability and maintainability of code.

Chapter 2 Short Answer Exercises

21. What type of variable is defined in this expression?

```
final double INTEREST_RATE = 0.07;
```

22. What do the following lines of code display?

```
double ouncesPerCan = 8.0;
System.out.print(ouncesPerCan);
```

23. What is the resulting output when the following lines of code are executed?

```
int number = 23;
number = 52;
System.out.print(number);
```

24. What do the following lines of code display?

```
int number = 12;
number = number + 8;
System.out.print(number);
```

25. What value is assigned to num3 in the lines below?

```
int num1 = 32, num2 = 6;
int num3 = num1 + num2;
```

26. What do the following lines of code output?

```
double num1 = 2.5;
int num2 = 5;
num1 = num1/num2;
System.out.print(num1);
```

27. In these expressions, what value will be assigned to the variable result?

    a. `result = 5 / 2.0;`
    b. `result = 7 / 2;`
    c. `result = 4 * 3 / 2;`
    d. `result = 5 % 2;`

28. Express the following equations using Java expressions.

    e. $4xy$

    f. $z = 2ab$

    g. $y = b^2 - 4ac$

    h. $t = \frac{a + b}{x - y}$

29. In the expressions below, what will be the value assigned to the variable num1?

    a. num1 = Math.round(2.3);
    b. num1 = Math.pow(4,2);
    c. num1 = Math.sqrt(25);
    d. num1 = Math.sqrt(Math.sqrt(81));

30. In following statement, what does the number "8" specify?

```
System.out.printf("%8.2f", number);
```

31. After the lines below execute, what will be the output?

```
double number = 123.453;
System.out.printf("%.2f", number);
```

32. What is the output from the following statement?

```
System.out.print("She said \"hello\".");
```

33. After the lines below execute, what will be stored in the variable name?

```
String first = "Angela";
String init = "P.
String last = "Harad";
String name = first + " " + init + " " + last;
```

34. After the lines below execute, what will be stored in the variable letter?

```
String name = "Sheila";
char letter = name.charAt(1);
```

35. After the lines below execute, what will be stored in the variable part?

    String word = “cartoon”;
    String part = word.substring(0, 4);

36. After the lines below execute, what will be stored in the variable portion?

    String word = “classroom”;
    String portion = word.substring(5);

37. After the lines below execute, what will be stored in the variable len?

    String cafe = “Angelo’s”;
    int len = café.length();

38. Which of the following should be used to read a line of text including spaces to store in the variable phrase?

    a. phrase = in.nextInt( );
    b. phrase = in.nextDouble( );
    c. phrase = in.nextLine( );
    d. phrase = in.next( );

Chapter 2 Programming Exercises

39. Complete the “Hello World” exercise in Appendix C and provide a screen capture of the IDE with the output.

40. Write a line of code that displays the following text.

    Java is a high-level language.

41. Write a line of code that displays the following text with the quotes.

    The waiter said “The special is good!”

42. Write a program that prompts the user to enter their name, stores the name entered in a String variable, and then displays ‘Hello ‘, and the name that was entered.

43. Write a program with three integer variables: first, second, and third. Assign the values 5, 6, and 7 to the variables and display each on a separate line using the variable names.

44. Write a program that declares the constant below and displays "The interest rate is " followed by the constant and a percent sign.

    INTEREST_RATE = 7

45. Write a program that assigns the variable tickets the value 125 and then displays "The tickets sold today were " followed by the variable.

46. Write a program that defines three String variables named word1, word2, and word3. Assign abc to word1, def to word2, and then assign word1 and word2 combined to word3 using concatenation. Then display word3.

47. Write a program that uses a double that is assigned 12345.678 and use a format specifier to display the number with commas and two decimal places.

    12,345.68

48. Write a program using variables to display the numbers below on separate lines, with two (2) decimal places, and in fields that are eight (8) characters wide.

    123.45 1452.56 56.80

49. Write a program that uses three (3) print statements to display the words No, lines, and between all on one line with spaces between the words.

50. Write a program that prompts the user to enter their age, stores the age in a variable named age, and then displays 'Your age is' and the age that was entered.

51. Write a program that prompts the user to enter a number, and then a second number. The program will add the numbers, store the result in a variable and display "The sum of the numbers is: " and the result.

52. Write a program that computes the total cost of a meal based on the meal price entered by the user, plus a 20% tip, and 5% sales tax. The output should be displayed as shown below and include a dollar sign and two (2) decimal places.

```
Enter the meal price $29.50
Total cost is $36.88
```

53. Expand number 14 to include output of the tip, and tax amounts, before the total price. The output should include a blank line between the input prompt and the output, dollar signs, two (2) decimal, and amounts aligned right.

```
Enter the meal price $34.98

Tip:   $  7.00
Tax:   $  1.75
Total: $ 43.73
```

54. Write a program that prompts the user to enter a Fahrenheit temperature, computes the Celsius temperature, and displays "The Celsius temperature is: " and the result. The equation for the conversion is:

    C = (F – 32) / 1.8 Test data: When F = 23, C = -5

55. Write a program that prompts the user to enter the lengths of the two sides of a rectangle. The program will compute the area and perimeter, and assign the values to two variables. Then display the computed values with their titles as shown in the example below.

```
Enter side length 1 3
Enter side length 2 4

The area is: 12.0
The perimeter is: 14.0
```

56. Write the pseudocode for a program for a Yogurt vendor that computes the total sales and profit for a day's sales based on the number sold at $6.50 each, and the cost of the Yogurt to the vendor which is $4.25 each. The profit is the total sales minus the total cost.

57. Write the program for the Yogurt vendor in #56 above. The program will display the output as shown in the example below. Note the dollar signs and alignment.

```
Enter the number sold 10

Units sold: 10
Total sales: $65.00
Total cost:  $42.50
----------------------

Profit for the day: $22.50
```

58. Write a program that prompts the user for two integers (x and y) and computes a result using the equation below. Note the output when y is entered as 1.

$$answer = \frac{x+2}{y-1}$$

59. Part #1: The surface area of a sphere is given by the equation below. Write a program that prompts the user for the radius of a sphere as a double and the units of measure (feet, miles, etc.), computes the surface area, and displays the result with the square units. Use Math.pow() in the solution, and format the output to 3 decimal places. Use 3.14159 as the value for PI.

$$Surface\ area = 4\,\pi\,r^2$$

Part #2: The volume of a sphere is given by the equation below. Write a program that prompts the user for the radius of a sphere as a double and the units of measure (feet, miles, etc.), computes the volume, and displays the result with the units. Use 3.14159 as the value for PI.

$$Volume = 4\pi\frac{r^3}{3}$$

Part #3: Combine parts 1 and 2 into a single program and add the computation and output for the circumference. The program will prompt the user for a radius as a double and the units of measure. Two examples are provided using 3.14159 as the value for PI, for testing to validate the output.

$$Circumference = 2\pi r$$

Volley Ball radius: approx. 4.1 inches

```
Enter the radius: 4.1
Enter the units (feet, miles, etc.): inches

The surface area is 211.241 square inches
The volume is 288.695 cubic inches
The circumference is 25.761 inches
```

The Earth's moon radius: approx. 1,080 miles (note the commas in the output)

```
Enter the radius: 1080
Enter the units (feet, miles, etc.): miles

The surface area is 14,657,402.304 square miles
The volume is 5,276,664,829.440 cubic miles
The circumference is 6,785.834 miles
```

Chapter 2 Programming Challenge

Loan Calculator

Design and develop a program for a car dealer that computes the monthly payment, total payback amount, and total interest paid for a car loan based upon the loan amount, interest rate, and loan duration in months.

The equation for determining the monthly payment for a loan is:

Monthly Loan Payment Formula: $MP = L * (r / (1 - (1 + r)^{-N}))$.

- MP = monthly payment amount
- L = principal, meaning the amount of money borrowed
- r = effective interest rate. Note that this is usually not the annual interest rate (see below).
- N = total number of payments

Calculate the effective interest rate (r) - Most loan terms use the "nominal annual interest rate", but that is an annual rate. Divide the annual interest rate by 100 to put it in decimal form, and then divide it by the number of payments per year (12) to get the effective interest rate. Note that the user enters the interest rate as a percentage (i.e. 4 for 4%).

- Example, if the annual interest rate is 5%, and payments are made monthly (12 times per year), calculate 5/100 to get 0.05, then calculate the rate:

  Effective rate = 0.05 / 12 = **0.004167**.

Sample output:

```
Enter the Loan Amount: 10000
Enter the Interest Rate: 4.5
Enter the duration in months: 48

The monthly payment: $228.03
The payback amount:  $10,945.67
The total interest:  $945.67
```

Chapter 3

Decision Structures and Boolean Logic

Decision structures determine the statements that execute based upon a condition. A conditional expression is used to control which line or lines of code execute. Decision structures provide multiple paths through a program based on the status of the true or false condition. If the condition is true, then a statement or statements are executed, otherwise they are not executed.

The *if* Condition

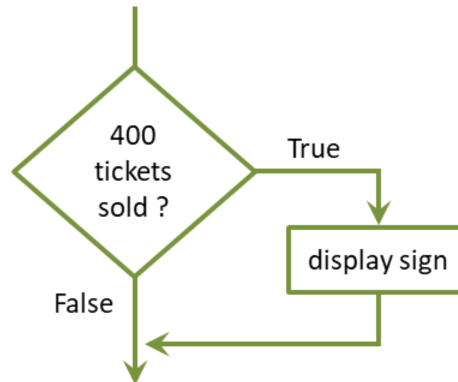
As an example, assume that a Theater has seating for 400 customers. Once the Theater has sold 400 tickets, the show has been sold out. When this occurs, the Theater displays a "Sold Out" sign at the box office. The decision to display the sign is made based upon whether or not 400 tickets have been sold. The decision structure is implemented using the ***if*** statement.

Ex. 3.1 – Conditional Statement Pseudocode

```
If 400 tickets have been sold
    -   Display the "Sold Out" sign
```

The condition tests if 400 tickets have been sold, and if it is true, the "Sold Out" sign is displayed. If the condition is false and 400 tickets have not been sold, then the sign will not be displayed.

Conditional expressions are represented in flowcharts as diamonds. The different paths that the program can take are shown using lines from the corners of the diamond, arrows indicate the direction, and text indicates the result. These paths in the program are often referred to as the ***Flow of Control*** or the ***Order of Operations***. In the example, if the condition is true then the flow of control follows the path to display the sign. Otherwise (if the condition is false), the program continues without displaying the sign.



Conditional Expression Flowchart

When programming a conditional expression in Java, the syntax includes parenthesis around the conditional expression, and braces to enclose the statement(s) executed when the condition is true. The statements to be executed are indented for readability (most IDEs automatically indent these lines). The general format is shown below with the opening brace on the line with the condition. Some programmers place the opening brace on the line following the expression, however most Java Style Guides prefer the format shown here.

```
if (conditional expression) {
    statement to execute;
}
```

Ex. 3.2 – Theater Ticket Sales "if" Condition (note use of two equal signs)

```
if(ticketsSold == 400) {
    System.out.println("Sold Out");
}
```

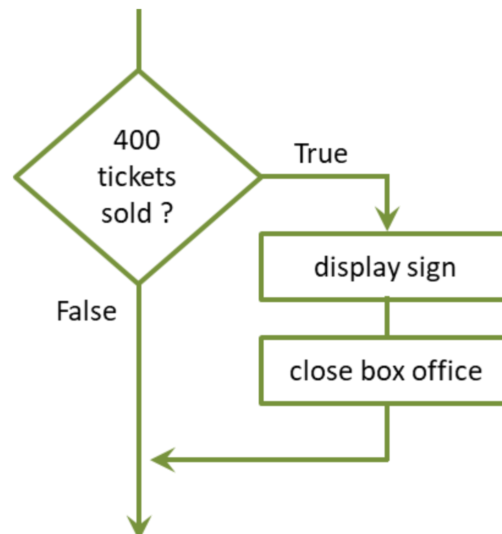
When multiple statements are associated with a condition, they form a ***block of code***. A block of code is a group of associated statements. In this case, they are associated with the conditional expression. If the condition is true, all of the statements within the braces (the block of code) will be executed. If the condition is false, all of the statements within the braces will be skipped over.

```
if (conditional expression) {
    statement1;
    statement2;
}
```

Continuing the Theater example, assume that when the show is sold out, the box office is closed in addition to the sold-out sign being displayed. A standard practice is to indent the pseudocode in line with the way that the lines would be indented in the actual code.

```
If 400 tickets have been sold
    -   Display the "Sold Out" sign
    -   Close the box office
```

The flowchart for the Theater example has been modified to include closing the box office if the condition is true.



Note how clearly the pseudocode and flowchart indicate the path taken if the condition is true. The modified code is shown in Ex. 3.3 below.

Ex. 3.3 – Theater Ticket Sales "if" Condition Expanded

```
System.out.println("Enter the tickets sold: ");
ticketsSold = in.nextInt();

if(ticketsSold == 400) {
    System.out.println("Sold Out");
    System.out.println("Box Office Closed.");

}
```

Boolean Expressions

Conditional expressions are either true or false, and are referred to as ***Boolean Expressions*** named after the mathematician George Boole (1815-1864). Boolean expressions are implemented using ***Relational Operators*** that resolve to either true or false by testing relationships. The result of the expression determines the next step or path for the program. For example, one value can be greater than another, or less than another, or equal to another. One of these three cases must be true, and the others would be false. Table 3.1 lists the Boolean operators available in Java. Note that two equal signs are used to test for equivalence (a single equal sign is the assignment operator).

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

Table 3.1 – Relational Operators

Some examples follow.

For the examples below: x = 5, y = 8, z = 5

x > y	5 > 8	False	5 is not greater than 8
x < y	5 < 8	True	5 is less than 8
x >= z	5 >= 5	True	5 is equivalent to 5
x <= z	5 <= 5	True	5 is equivalent to 5
x == y	5 == 8	False	5 is not equivalent to 8
x != y	5 != 8	True	5 is not equivalent to 8
x == z	5 == 5	True	5 is equivalent to 5

Relational Expressions

The *else* Condition

The Theater example conditionally displays "Sold Out" and "Box Office Closed" if exactly 400 tickets have been sold. Otherwise the program does nothing. To provide for other statements to execute when the condition is false, an ***else*** clause is implemented which can be thought of as an "*otherwise*" condition for when the relational expression is not true. When the "*if*" condition is true, the statements in the "*if*" block will be executed and the "*else*" block will be skipped. When the "*if*" condition is false, the "*if*" block will be skipped and the "*else*" block will execute.

```
if (conditional expression) {
    statement1;
    statement2;
}
else {
    statement1;
    statement2;
}
```

Continuing the Theater example, if 400 tickets have not been sold, tickets will continue to be sold.

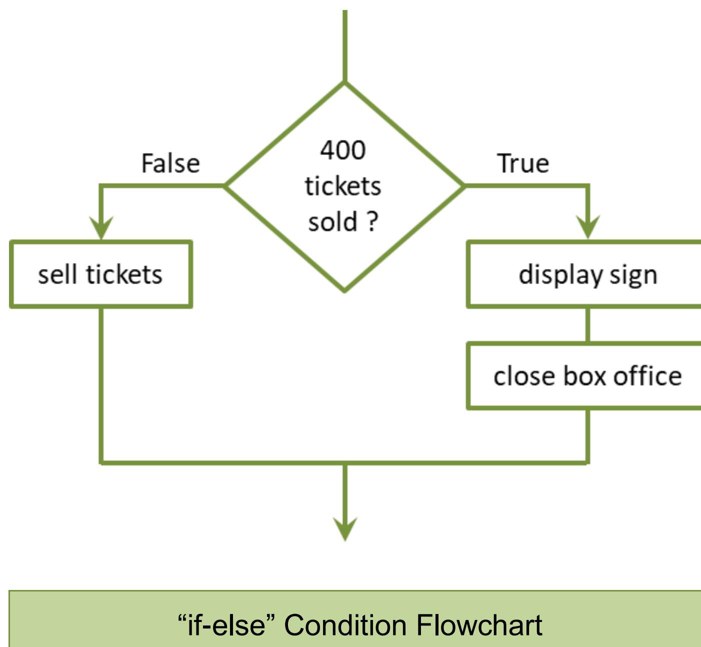
if 400 tickets have been sold

- Display the "Sold Out" sign
- Close the box office

else

- Continue to sell tickets

A flowchart highlights the two different paths that can be taken as a result of the conditional expression, and that only one path will be executed. It also shows that the two paths converge as the program continues.



"if-else" Condition Flowchart

The modified code in Ex. 3.3 below includes the Scanner and prompt as well as the additional else clause.

Ex. 3.4 – Theater Ticket Sales "if-else"

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

System.out.println("Enter the tickets sold: ");
ticketsSold = in.nextInt();

if(ticketsSold == 400) {
    System.out.println("Sold Out");
    System.out.println("Box Office Closed.");

}
else {
    System.out.println("Continue selling tickets.");
}
```

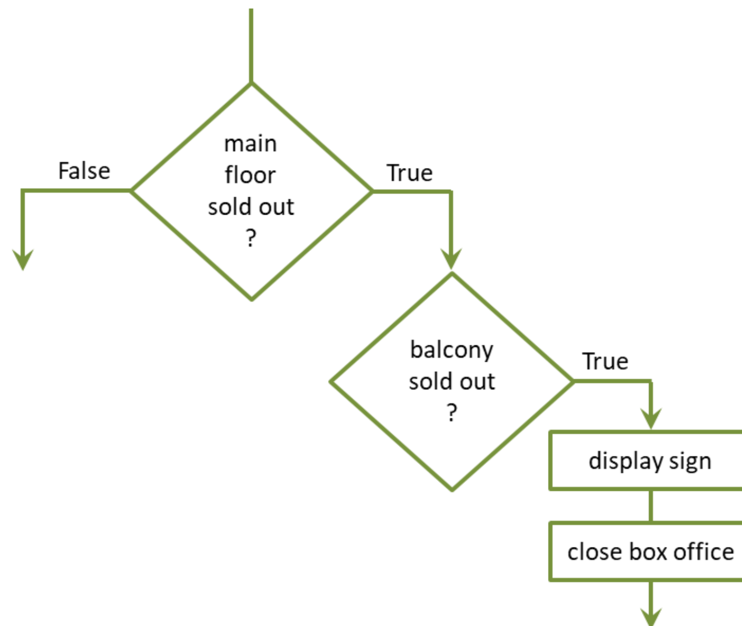
Nested if Structures

When two (or more) conditions are being tested, there are several ways to implement the logic. One of these is to use a nested if, which is an "*if*" condition inside another "*if*" condition. If *condition1* below is true, then *condition2* is tested, and if it is also true, then the statements will be executed. If *condition1* is false, then *condition2* will not be tested and the statements are skipped.

```
if (condition1) {
    if (condition2) {
        statement1;
        statement2;
    }
}
```

To illustrate this, the Theater example now includes a balcony section in addition to the main-floor seats. The tickets sales are tracked separately, and to determine if the show is sold out requires testing both areas for the tickets sold.

if the 400 main floor seats are sold
    if the 200 balcony seats are sold
- Display the "Sold Out" sign
- Close the box office



Nested "if" Condition Flowchart

The conditional expression for the balcony seats above is only tested if the main floor is sold out. Later in the text, Boolean Logic will be covered which will combine expressions and in most cases, eliminate the need for a nested if.

The if, else if, else Structure

To handle situations when different conditions result in different paths, an "if" and an "else" are inadequate because there are only two paths. Additional "if" statements may be appropriate, but more often the ***if, else-if, else*** structure is a better solution. Note that *else-if* is not an otherwise condition, but another conditional test that is only tested if the test before it is false. Once a condition is found to be true, the others are skipped over. The final else clause handles the situation when none of the conditions is true.

```
if (condition1) {
    statement(s);
}
else if (condition2) {
    statement(s);
}
else if (condition3) {
    statement(s);
}
else if (condition4) {
    statement(s);
}
else {
    statement(s);
}
```

Notice in the example below that there is no test for equivalence with zero. If the number is not positive and it is not negative then (otherwise) it must be zero.

Ex. 3.5 – Conditional Example with Three Conditions

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

When designing conditional logic, it is important to consider all possible scenarios, and use the proper condition for the test. Consider the two examples below and the results when a grade of 95 is entered. The code on the left would test each condition and eventually assign "D" to the variable letter.

```
if (grade > 90) {
    letter = 'A';
}
if (grade > 80) {
    letter = 'B');
}
if (grade > 70) {
    letter = 'C';
}
if (grade > 60) {
    letter = 'D';
}
```

```
if (grade > 90) {
    letter = 'A';
}
else if (grade > 80) {
    letter = 'B';
}
else if (grade > 70) {
    letter = 'C';
}
else if (grade > 60) {
    letter = 'D';
}
```

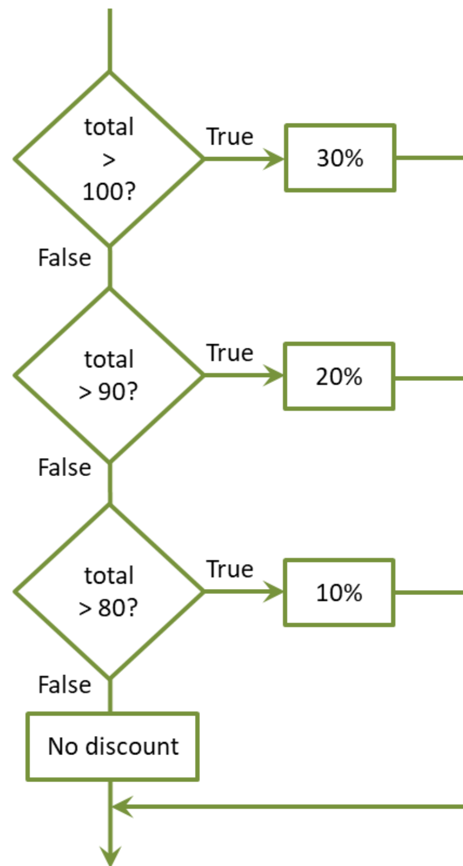
Logic Design

Ex. 3.6 – Discount Based on Total Sale

As an example, a store is having a Special that discounts the total sale price based upon the amount of the sale. The pseudocode below tests for the proper discount percentage. The logic only tests the second condition if the first condition is false, and the third condition is only tested if the first and second conditions are false. The final *else* handles the situation when all of the conditions are false. Consider that a total sale amount of $120.00 is also greater than $100.00 and greater than $90.00 and greater than $80.00, but after the first condition resolves to true, Discount is 30%, and the other conditions are skipped over. If the price were $95.00, then the first condition would be false and the second condition would be tested. Since that condition would be true, the discount would be 20%, and the other conditions would be skipped.

```
if the total sale > $100.00
    Discount is 30%
else if the total sale > $90.00
    Discount is 20%
else if the total sale is > $80.00
    Discount is 10%
else
    No discount
```

The following flowchart highlights the paths based upon the total sale, and that only one path is taken as a result of the conditions. If all of the conditions are false, then there is no discount.



"if, else-if, else" Condition Flowchart

The Java code for the discount conditions in the example are shown here.

```
if(total > 100) {
     discount = 30;
}
else if(total > 90) {
     discount = 20;
}
else if(total > 80) {
     discount = 10;
}
else {
     discount = 0;
}
```

Technical Notes

Designing conditional expressions for computer programs is an important skill to develop. Many difficult-to-find bugs are caused by incorrect conditional expressions and relational evaluations, and the use of break and continue which bypass conditional logic. Pseudocode and flowcharts are helpful design tools, and careful testing during development can eliminate most bugs.

Strings and Conditional Expressions

In addition to comparing numbers, very often strings of characters need to be compared. When a password is created or changed, it is entered a second time as confirmation. The two entries are compared to ensure that they match. In computing, what is actually being compared is the ASCII representation of the letters character by character. Recall from Chapter 1 that each character has a binary representation in the ASCII character set (Appendix A). To compare Strings in Java, the equivalence operator cannot be used. It would compare the locations in memory for the Strings. The member function ***equals()*** is used.

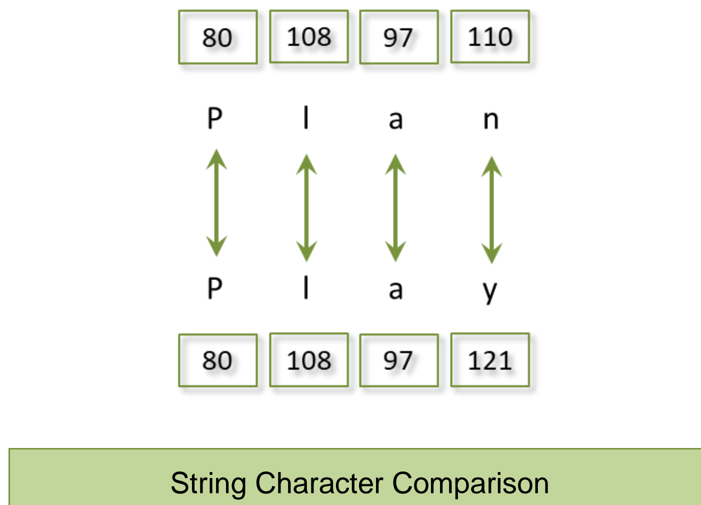
```
if(string1 == string2)          // compares memory locations

if(string1.equals(string2))     // compares the characters
```

As an example, “Play” and Plan” are compared in this code.

```
String string1 = "Play";
String string2 = "Plan";

if(string1.equals(string2)) {
    System.out.println("They are the same.");
}
else {
    System.out.println("Not the same.");
}
```

In the conditional statement, each character of each String is compared one at a time using the ASCII representation for the letter. When ‘n and ‘y’ are compared the condition returns false. The two Strings are not equivalent. The base-10

ASCII values for the letters are 110 for 'n' and 121 for 'y'. Therefore, they are not equal since 'y' has a different ASCII value than 'n'.



String Character Comparison

When a String has more characters than another, they could not be equivalent. Additional String methods and operations are covered in a later chapter, and the next two sections cover String conversions for numeric input.

String to Numeric Conversion

When a program requires a user to enter numeric values, we cannot trust that what is entered is truly a number. The methods ***nextInt()*** and ***nextDouble()*** will fail if the value is not the correct data type. One way to resolve this is by first reading the input as a String and then *parsing* the input to the data type desired. There are two methods for the conversions to numeric values. The method ***Integer.parseInt( )*** converts the data type from a String to an integer, and the method ***Double.parseDouble( )*** converts the String to a double.

```
String numString = "32";

int num = Integer.parseInt(numString);
```

To covert a String to a double, *Double.parseDouble()* is used.

```
String dblString = "123.45";

double dblNum = Double.parseDouble(dblString);
```

The code below fails and throws a *NumberFormatException* and does not complete because the parsing attempt fails. The String cannot be parsed to an integer. Handling exceptions is covered in a later chapter, but the next section provides another way of checking the input first before attempting the conversion.

```
String badString = "bad";

int badNum = Integer.parseInt(badString);
```

```
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)
```

Conversion Attempt Failed

Validating Input

In the last section, *nextInt()* and other input methods were introduced with a caution that they will fail if the value is not the expected data type. 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. The method ***hasNextInt( )*** provides this ability and returns a Boolean value based on the next input. In example Ex. 3.7, the input is only obtained if it is an integer.

Ex. 3.7 – Validating Input with *hasNextInt()*

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

System.out.print("Enter an integer ");

if(in.hasNextInt()) {
    num = in.nextInt();
    System.out.print("Integer is " + num);
}
else {
    System.out.print("Not an 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

In an earlier example, a nested-if was used to test two conditions. The second condition was tested only if the first condition was true. Both conditions need to be true for the statements to execute. The pseudocode would be *"if condition1 is true AND then if condition2 is true, execute the statements"*.

```
if (condition1) {
    if (condition2) {
        statement1;
        statement2;
    }
}
```

Both of the conditions above can be combined into a single compound expression. When a program needs to make decisions based on complex conditions, multiple conditions can be combined using the Logical Operators. The logical operators are "&&" for AND, "||" for OR, and "!" for NOT.

With a logical ***AND***, both sides of the expression must be true for the condition to be true. If either one of the conditions is false, the expression is false.

```
if(condition1 && condition2) {
    statement1;
    statement2;
}
```

Table 3.2 is a truth table for logical AND.

A	B	A && B
True	True	True
True	False	False
False	True	False
False	False	False

Table 3.2 – Logical AND Truth Table

This logic is often used to verify that a number is within a range, especially when validating input. For example, if the program requires a number between 0 and 10, the conditions can be combined in a single expression. This is referred to as *range checking*. Both conditions must be true, for the expression to be true. The number entered below must be greater than 0 and less than 10.

Ex. 3.8 – Validating Input with AND (&&)

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

System.out.print("Enter a number between 0 and 10: ");
num = in.nextInt();

if(num > 0 && num < 10) {
    System.out.print("Valid number");
}
else {
    System.out.print("Not a valid number.");
}
```

The Theater Ticket Sales example required main-floor seats and balcony seats to be sold out for the Theater to display the "Sold Out" sign and close the box office. Both conditions must be true and a nested "*if*" condition was used in the example.

If the 400 main floor seats are sold
  If the 200 balcony seats are sold
  - Display the "Sold Out" sign
  - Close the box office

This can be easily implemented with the AND operator, since both conditions must be true.

Ex. 3.9 – Validating a Range with Logical AND (&&)

```
if(floorTicketsSold == 400 && balconyTicketsSold == 200) {
    System.out.println("Sold Out");
    System.out.println("Box Office Closed.");
}
else {
    System.out.println("Continue selling tickets.");
}
```

An either-or expression can also be used to test multiple conditions. Reversing the previous logic for the Theater Ticket Sales example, if the main-floor is not sold out, or the balcony is not sold out, then the Theater is not sold out. The logical ***OR*** would be used in this case. The logical OR operator consists of two pipes "||" (shift backslash on the keyboard). With a logical OR, if either side of the expression is true then the expression is true.

```
if(condition1 || condition2) {
    statement1;
    statement2;
}
```

As shown in Table 3.3, when either condition is true, the expression is true.

A	B	A || B
True	True	True
True	False	True
False	True	True
False	False	False

Table 3.3 – Logical OR Truth Table

Using the previous example to validate a number between 1 and 9 inclusive, a logical OR can be used, but note the different values used to test outside the range instead of inside. Also, the order of the output statements is reversed. If either condition is true, then the expression is true, and the number is not within the range required.

Ex. 3.10 – Validating Input with OR (||)

```
if(num < 1 || num > 9) {
    System.out.print("Not a valid number");
}
else {
    System.out.print("Valid number.");
}
```

Short-Circuit Evaluation

For the logical AND and the logical OR operations, the computer uses what is called short-circuit evaluation. With the logical AND, both sides of the compound condition must be true for the expression to be true, so if the left side is false, then the right side is not evaluated. It wouldn't matter if the right side were true since the expression is already false.

The reverse occurs with the logical OR. When either side of a compound expression using OR is true, the expression is true. Therefore, if the left side of the compound condition is true, the right side is not evaluated. It doesn't matter whether the right side is true or false since the expression is already true.

Technical Notes

The Simple-OR operator in Java is a single pipe "|" that does not perform shirt-circuit evaluation. It evaluates both expressions and returns the result.

Some Common Logic Errors

Recall that logic errors are those errors that do not halt execution of the program but produce incorrect results. Many of these occur due to incorrect logical expressions. Pseudocode and flowcharts can help, but careful consideration of the logic is required. Table 3.4 lists some examples and the numeric values that would make the expressions true. Note the situation where any number is valid.

Expression	True when the value of x is:
if x > 0 && x < 100	any number 1 thru 99
if x >= 0 && x <= 100	any number 0 thru 100
if x > 0 || x < 100	any number
if x < 0 || x > 100	negative or above 100
if x <= 0 || x >= 100	negative, zero, or 100 or above

Table 3.4 – Logical Expressions

Boolean Variables

Boolean variables are variables that can only have a value of true or false. They are often used to store the result of a condition, or as a ***flag*** that is set as the result of some processing. The example below sets a Boolean variable to false, and if the condition is true the Boolean variable (acting as a flag) is flipped to true and is used to determine the next step in the program.

```
boolean validUser = false;

if(passWord.equals(previous)) {
    validUser = true;
}

if(validUser) {
    // access the account

}
```

Boolean Variables

Some programmers prefer to write out the conditional expression for clarity. The equivalent versions are shown here, although the version on the left is preferred.

```
if(validUser) {
    // access the account
}
```

```
if(validUser == true) {
    // access the account
}
```

The last of the logical operators is the ***NOT*** (!) operator. This operator returns the opposite of a Boolean expression or operand. If the operand or expression is true, the NOT operator returns false. If the operand is false, the NOT operator returns true.

A	! A
True	False
False	True

Table 3.5 – Logical NOT Truth Table

Caution should be exercised when using the NOT operator because it often introduces bugs and confusion. In pseudocode, the expression below would read "If x is greater than y is true, and x is greater than z is true, then return false". It isn't clear what condition is being tested.

```
if(!(x > y && x > z))
```

It is often easier to reverse the logic and remove the NOT operator. De Morgan's Law provides two forms: one for negation of an AND expression and one for an OR expression.

```
!(A && B)        !A || !B
!(A || B)        !A && !B
```

De Morgan's Law

When a Boolean variable is involved, the NOT operator can often be used without adding confusion. Consider the valid user condition below, and the second example that reverses the logic. Either one is a correct implementation of the logic.

```
if(validUser) {
    // access the account

}


if(!validUser) {
    // don't allow access to the account

}
```

A logical and methodical approach when creating conditional statements and compound expressions will save a lot of time spent debugging logic errors.

Test Cases

As programs become more complex, testing becomes more critical to ensure accurate processing and output. Test Cases are actions executed to verify a

portion of or a complete software implementation. All possible paths through the program must be verified. Consider the Theater Ticket program logic repeated below as an example.

```
if(floorTicketsSold == 400 && balconyTicketsSold == 200) {
    System.out.println("Sold Out");
    System.out.println("Box Office Closed.");
}
else {
    System.out.println("Continue selling tickets.");
}
```

The compound conditional expression requires testing, and there are two Boolean conditions within the expression that need to be tested. There is only one scenario in which the expression as a whole is true as shown below.

Test Case #	Values Entered	Expected Result
1	Floor Tickets < 400 and Balcony Tickets < 200	False
2	Floor Tickets 400 and Balcony Tickets < 200	False
3	Floor Tickets < 400 and Balcony Tickets 200	False
4	Floor Tickets 400 and Balcony Tickets 200	True

Test Cases should be simple to execute, have the end user of the program in mind, ensure 100% coverage, and be repeatable (the same results whenever run). They should be numbered or labeled for tracking purposes and regression testing to ensure that new functionality has not introduced an error.

Programming Style and Standards

Although it may be convenient in some cases to string multiple logical operators together, it can add confusion and introduce bugs. Taking a logical and methodical approach to compound conditional expressions will often produce a simpler algorithm that is easier to understand and implement.

Chapter 3 Review Questions

1. Decision structures determine the statements that execute based upon a ____________.
2. A ___________ result is one that is either true or false.
3. The Flow of Control refers to the ________ in which statements will execute.
4. In a Flowchart, decisions are represented by ____________.
5. Statements that execute when an "*if*" condition is true are below the condition and within braces forming a ________ of code.
6. Boolean expressions are implemented using ______________ operators.
7. In an *if-else* structure, when the "*if*" condition is false the ________ clause will execute.
8. When two Strings are compared for equivalence, the ____________ value of each character is compared individually.
9. Compound Boolean expressions are implemented using the _______ operators.
10. For a compound expression that uses a logical AND to be true, ________ of the conditions must be true.
11. For a compound expression that uses a logical OR" to be true, ________ of the conditions must be true.
12. The logical operator that is used to negate a Boolean value is the ____ operator.
13. Boolean variables can only be assigned a status of ______ or ______.
14. Test Cases are an important tool for _____________ the accuracy of a program.

Chapter 3 Short Answer Exercises

15. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
if (var2 > var1) {
        System.out.print("var2 is greater");
```

16. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
if (var1 < var2) {
        System.out.print("var2 is greater");
```

17. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
if (var1 <= var2) {
        var3 = var1 + var2;
        System.out.print("var3 is ", var3);
}
```

18. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
if (var1 == var2) {
        System.out.print("They are the same.");
}
else {
        System.out.print("They are not the same.");
}
```

19. What do the following lines of code output if *first* = 10, and *second* = 10?

```
if (first > second) {
        System.out.print("first is greater");
}
else {
        System.out.print("second is greater");
}
```

20. What do the following lines of code output if *val1* = 3, *val2* = 5, and *val3* = 8

```
if (val2 > val1) {
        val3 = val2 – val1;
}
else if (val3 > val2) {
        val3 = val2;
}
else {
        val3 = 99;
}
System.out.print("val3 is ", val3);
```

21. Write the word Java using the ASCII Base-10 (digit not binary) equivalences for the letters (ref. Appendix A).

22. Are the following expressions true or false if *first* = true and *second* = false?

    a. first && second
    b. first || second
    c. second || first
    d. !first
    e. !second

23. Write an "*if*" conditional expression using a logical operator that tests for a number variable temp that is greater than 32 and less than 120.

24. Write an "*if*" conditional expression using a logical operator that tests for a number variable num1 between 0 and 50, including 50 but excluding 0.

25. True or false, the following expressions test for the same condition.

```
if num > 9 && num < 21      if num >= 10 && num <= 20
```

26. What range of numbers assigned to num would make this expression true?

```
if num > 0 || num < 100
```

27. What range of numbers assigned to num would make the following expression true?

```
if num > 0 && num > 100
```

28. What numbers are excluded by the following expression?

```
if num < 0 || num > 0
```

29. What value assigned to done would execute the print statement?

```
if (done) {
        System.out.print("That's all.");
```

Chapter 3 Programming Exercises

30. Write the code for a conditional clause that tests and outputs whether a variable num1 is greater than zero.

31. Write a program that accepts an integer as input, stores the value in num1, and displays whether it is positive, negative, or zero.

32. Write a program that accepts an integer as input and stores the value in a variable named hours for the number of hours worked and executes the following algorithm. If the number of hours worked are greater than 40, then output "There is overtime", otherwise output "There is no overtime".

33. Expansion of the program in #32.

    a. Draw a flowchart for following pseudocode. Consider the order of operations and each possible condition.

```
Get the number of hours worked
Get the hourly rate of pay

if the number of hours worked > 40
  – Compute overtime pay (1.5 * hourly rate for hours > 40)
Compute regular pay (hourly rate * hours up to and including 40)
Compute total pay

Output regular pay
Output overtime pay
Output total pay
```

    b. Develop a program for the pseudocode in part (a) above. Consider each variable that is needed, the order of operations, and formatting of the output for dollar amounts. Design the solution in terms of input, processing, and then output. The output statements should not be within the conditional blocks.

<u>Sample output</u>

```
Enter hours worked 41

Enter hourly rate 10.00

Regular pay is $400.00
Overtime pay is $15.00
Total pay is $415.00
```

34. Modify Programming Exercise #33 to include double time pay (2 * hourly rate) for hours above 50, and add the additional output. The hours from 41 to 50 remain time-and-a-half pay (1.5 * hourly rate). Sample output is shown below.

```
Enter hours worked 51

Enter hourly rate 10.00

Regular pay is $400.00
Overtime 1.5x pay is $150.00
Overtime 2x pay is $20.00
Total pay is $570.00
```

35. Write a program that requests a *username* and *password* from the user, and then requests that they confirm the password. If the passwords match, output "Account created for" and the *username*, otherwise output "Password confirmation error".

36. Write a program that accepts the price for an item and computes a discounted price based on the criteria below, determines the 7% sales tax amount on the discounted price, and display the original price, discounted price, sales tax, and the total amount for the purchase. Include dollar signs, two decimal places, and right align the dollar amounts as shown in the sample output below.

    Discount criteria

        Greater than $100.00 – 25%

        Greater than $75.00 – 18%

        Greater than $50.00 – 10%

    Sample output

```
Enter the price of the item 95.50

Original price  $ 95.50
Discount price  $ 78.31
Tax             $  5.48
Total amount    $ 83.79
```

37. Write a program that prompts the user to enter a temperature and then "F or f" or "C or c" if it is a Fahrenheit or Celsius temperature to convert. Display the converted temperature and scale or "Cannot convert" if an incorrect letter was entered. The equations for the conversions are:

    C = (F – 32) / 1.8   F = (C * 1.8) + 32

38. Write a program for a Theater that computes the total sales receipts and profit for an event based on the number of tickets sold and the following criteria:

- The 200 main floor tickets are sold first, and then the 75 balcony tickets are sold once the main floor is sold out.
- Main floor tickets are $29.50 each, and Balcony tickets are $19.50 each. Use constants for these amounts
- The program will request the total number of tickets sold (assume <= 275 as input) and "M" for Matinee or "E" for evening. The cost to hold an event is $1,200.00 for Matinee and $1,450.00 for evening.
- The output will include the number of tickets sold and sales for each section, the total sales receipts, the event cost, and the profit for the event. Profit is total sales minus cost. See sample output below.

#38 (a) Write the pseudocode for the program

#38 (b) Draw a flowchart of the solution

#38 (c) Develop the program

#38 (d) Create and run two (2) Test Cases (not shown below) and screen capture the results

Sample output A

```
Enter the tickets sold 199
Enter "M" for Matinee, "E" for Evening E

Main Floor ticket sales: $5870.50
Balcony ticket sales: $0.00
Total sales: $5870.50
Event cost: $1450.00
Profit from the event: $4420.50
```

Sample output B

```
Enter the tickets sold 201
Enter "M" for Matinee, "E" for Evening M

Main Floor ticket sales: $5900.00
Balcony ticket sales: $19.50
Total sales: $5919.50
Event cost: $1200.00
Profit from the event: $4719.50
```

39. The wind chill in North America is computed using temperature in Fahrenheit and wind speed in miles-per-hour, however it is not valid for temperatures above 50 degrees or when the wind speed is 3.0 mph or less. Write a program that requests the temperature and wind speed from the user, and computes the wind chill or displays that it is not valid and the particular reason that it is not valid.

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

    $$\textit{wind chill} = 35.74 + 0.6215\, T_a - 35.75 V^{+0.16} + 0.4275\, T_a V^{+0.16}$$

    T_a is the air temperature in Fahrenheit, and

    V is the wind speed in mph

    Consider - Math.pow(windSpeed, 0.16)

    Sample output A

    ```
    Enter the temperature in Fahrenheit 25
    Enter the wind Speed in mph 10

    The Wind chill is 14.8 degrees Fahrenheit.
    ```

    Sample output B

    ```
    Enter the temperature in Fahrenheit 16
    Enter the wind Speed in mph 9

    The Wind chill is 4.5 degrees Fahrenheit.
    ```

    Sample output C

    ```
    Enter the temperature in Fahrenheit 70
    Enter the wind Speed in mph 12
    Wind chill is not valid at that temperature.
    ```

Additional Test data:

https://www.weather.gov/safety/cold-wind-chill-chart

Chapter 3 Programming Challenge

Planet-days to Earth-days

Write a program that compares Planet-days to Earth-days.

- Request the name of the planet and a number of Earth-days
- Validate that the input is one of the planets listed below
- Validate that the number of days is greater than zero
- Compute and display the number of planet-days that would pass on the chosen planet based upon the conversion values in hours provided (NASA).
- Display the result formatted to three (3) decimal places.

Planet	Earth Equivalent Hours
Earth	24.0 hours
Mercury	4222.6 hours
Venus	2802.0 hours
Mars	24.7 hours

Sample output A

```
Enter the planet: Mercury, Venus, or Mars Mars
Enter the number of days to compare 5

5.0 Earth-days is equivalent to 4.858 days on Mars.
```

Sample output B

```
Enter the planet: Mercury, Venus, or Mars Venus
Enter the number of days to compare 5

5.0 Earth-days is equivalent to 0.043 days on Venus.
```

Chapter 4

Loops and Repetition Structures

Repetition Structures (Loops)

Very often a statement or set of statements in a program need to repeat over and over to accomplish a task or compute a result. An example would be computing compound interest for a bank account over *some* period of time. The program would begin with a balance, compute the interest amount, add it to the balance, compute the interest on the new balance, add it to the balance, and so on. This would continue for as many times as needed.

```
Start with an account balance
Compute the interest amount
Add it to the balance
Compute the interest on the new balance
Add it to the new balance
Compute the interest on the new balance
And so on…
```

In addition, it may be desirable to repeat an entire program instead of restarting the program to enter different inputs. ***Repetition Structures*** or ***loops*** provide a way of repeating steps without repeating the code. The loop statements continue to execute until a final result has been reached and the loop ends. As an example, the Theater program in Chapter 3 contained a conditional statement for

closing the box office if enough tickets had been sold. The alternate path was to display "Sell Tickets" and end the program. Consider that while there are tickets to sell, they should be sold and that each time tickets are sold, the condition should be tested again until all of the tickets are sold.

```
if 400 tickets have been sold
    -   Display the "Sold Out" sign
    -   Close the box office
else
    -   Continue to sell tickets
```

The While Loop

The while loop is a ***condition-controlled*** loop. The statement or statements within the loop execute while some conditional expression is true, and each execution of a loop is referred to as an ***iteration*** of the loop. Repetition structures follow the same brace and indentation rules applied to conditional statements. A conditional expression for the loop is enclosed in parenthesis and braces form the block of code executed as long as the condition is true. Indentation of the statements adds clarity. The general format is shown below and can be read as "*while this condition is true, do these things*".

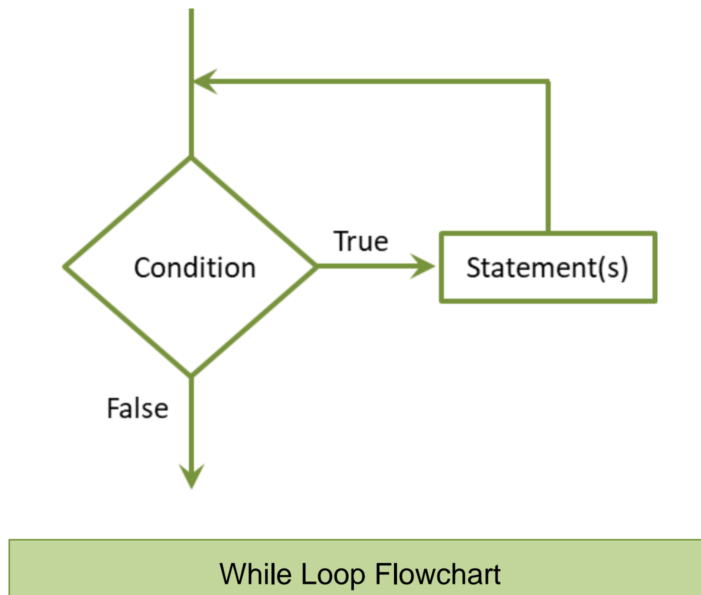
```
while (condition) {
    statement(s);
}
```

Pseudocode for the Theater example using a loop would reverse the logic used with the "if" condition shown above. While there are tickets to sell, sell tickets. When there are no more tickets to sell, display the Sold-Out sign and close the box office.

```
while (400 tickets have NOT yet been sold)
    -   Continue to sell tickets

Display the "Sold Out" sign
Close the box office
```

The flowchart depiction of a loop also uses a diamond for the conditional expression, and arrowed lines representing the order of operations. The line

after the statements returns to a point prior to the conditional expression indicating that it is tested again after the loop body executes. When the condition is false the loop ends and the program continues.



While Loop Flowchart

Since the conditional expression in a while loop is tested prior to the body of the loop executing, it is referred to as a ***pre-test*** loop. This means that a while loop may or may not execute depending on the result of the conditional expression.

Ex. 4.1 – Theater Ticket Sales Using a While Loop

```
while(ticketsSold < 400) {
    System.out.print("Sell another ticket.");
    ticketsSold = ticketsSold + 1;
}

System.out.println("Display \"Sold Out\" sign" );
System.out.println("Close the box office.");
```

While Loop Example

Notice in the actual code above that the final two output statements are outside the braces and are not part of the loop. They execute when the conditional expression is false and the loop ends. Also note that the variable ticketsSold increases inside the loop to eventually make the conditional expression false.

This is referred as an ***update*** expression, and is an important part of the loop. There must be a change that occurs inside the loop that eventually makes the condition false. Otherwise, the loop will continue to run resulting in an ***infinite loop***. When this occurs, the program must be ended to stop the loop.

Infinite loops occur due to programmer errors. As an example, the following loop is an infinite loop because the variable value never changes (it will always be less than ten) and therefore the conditional expression is always true.

```
int value = 3;

while(value < 10) {
    System.out.println("This will never end.");
}
```

Infinite Loop Example

A while loop can also be used to allow the body of a program to run multiple times without the user having to restart the program. In Ex. 4.2 the user is prompted for whether or not another temperature conversion is desired. Notice that the String variable another is set to "*y*" to start the loop. Since a while loop is a pre-test loop, the condition must be true or the loop will not be entered. At the end of the loop, the user is asked if they would like to convert another. Any character entered other than "Y" or "y" will end the loop.

Ex. 4.2 – Condition Controlled Loop

```
Scanner in = new Scanner(System.in);
String another = "y";
double tempF = 0, celsius = 0;

while(another.equals("Y") || another.equals("y")) {

    System.out.print("Enter a Fahrenheit temperature ");
    tempF = in.nextDouble();
    celsius = (tempF - 32) * 1.8;

    System.out.println("The Celsius temperature is " + celsius);
    System.out.print("Enter \"Y\" to convert another ");
    another = in.next();
}
```

The Increment and Decrement Operators

There are operators for adding one to a variable and for subtracting one from a variable. The increment operator is two plus signs (++) and the decrement operator is two minus signs (--). They are used frequently with counters in programs, especially within loops, and they simplify update statements.

The statements on the left accomplish the same as the statements on the right.

```
number++;        number = number + 1;

number--;        number = number - 1;
```

The examples above are referred to as ***postfix*** mode since the operators follow the variable. The operators also have a ***prefix*** mode as shown here.

```
++number;        number = number + 1;

--number;        number = number - 1;
```

The expressions on the right for both modes are the same because the statements simply increment or decrement the variable. But when the operators are used in other operations, there is a difference. Note the output from the following statements and when the updates occur. The first output statement uses postfix mode and `value` is accessed before the update occurs.

```
int value = 1;

System.out.println(value++);

System.out.println(value--);

System.out.println(value);

System.out.println(++value);

System.out.println(--value);
```

```
<terminated> PostFixPrefix
1
2
1
2
1
```

The For Loop

Another type of loop used in programming is the for-loop which is a ***count-controlled*** loop where the number of iterations is a specific number of times. The programmer sets the number of times that the loop will execute when designing the loop. Like the while loop, the for-loop is a ***pre-test*** repetition structure. The general format includes the initialization for the loop, the conditional expression, and the update on a single line and within parentheses.

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

When the loop above is encountered in the program, the integer `var` is declared and initialized to zero. The conditional expression is tested, and if the condition is true, the statements in the body of the loop are executed and the update occurs last. After the update occurs, the condition expression is tested again, and if it is true, the body of the loop is executed again and the update occurs again. This continues until the conditional expression is false.

initialization update

```
for(int i = 0; i < 100; i++)
```

conditional expression

The update for the loop can be an expression. The following two examples output even numbers.

```
for(int i = 2; i <= 10; i = i + 2) {
    System.out.println("Even numbers " + i);
}
```

```
int num = 2;

while (num <= 10) {
    System.out.println("Even numbers " + num);
    num = num + 2;
}
```

This example uses multiplication in the update expression to output the powers of 2 up to 1000 formatted with right alignment. Since the condition is *less than* 1000, the loop ends when the variable i reaches 1024 which is not displayed.

```
for(int i = 2; i < 1000; i = i * 2) {

    System.out.printf("\n%3d", i);
}
```

```
<terminated> Ex_
  2
  4
  8
 16
 32
 64
128
256
512
```

For Loop Example

The update for a for-loop is included in the loop header and should not be duplicated or modified within the for-loop body.

Using "i" and Scope

Many examples above use the variable "i" which is typical of for-loops. Loops are used extensively in programming and this variable is used as a control variable, sometimes referred to as throw-away variable. Since the variable is declared within the loop header, it is local to the loop. This means that another variable can be named "i" in another loop later in the program. The example below shows that a variable declared within a loop is not available outside the loop. Variable scope will be revisited in the chapter on Methods. Note the error indicator when the program attempts to access number outside the loop.

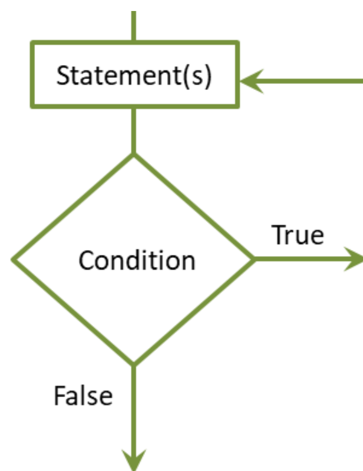
```
20
21          for(int number = 0; number < 10; number++) {
22              System.out.println(number);
23          }
24
25  number cannot be resolved to a variable intln(number);
```

The Do Loop or Do-while Loop

The third loop structure is the do-while loop. It is similar to a while loop except that the conditional expression is after the loop body. Therefore, the do-while loop is a ***post-test*** loop, and will always execute at least once. The general format is shown below and can be read as "*do these things, and if the condition is true, do them again*".

```
do {
    statement(s);
}
while(condition);
```

The do-while loop differentiates itself in several ways. It begins with the word "do" followed by the body of the loop, and ends with the word while followed by the conditional expression and a semicolon. The flowchart highlights the fact that the body of the loop is executed before the conditional expression is evaluated.



Do-While Loop Flowchart

Since the do-while loop executes the loop body first, it can be used in those cases when the statements need to execute at least once. As an example, an account login operation requests a username and password. Then, if the username or password does not match those stored by the program, the user is again requested to enter them. The statements need to execute the first time, and only again if an invalid username or password is entered.

Comparing Loop Structures

A while loop can be used to implement the operations of the other loops. A for-loop can be used to implement most of the operations that a while and do-while loop can implement, but not all. With a for-loop, the update always occurs last after all of the statements in the loop execute. With a while loop and do-while loop, the programmer can place the update anywhere within the loop. A while loop may or may not execute based upon the conditional expression, but a do-while loop will always execute the body at least once. The following three examples implement the same loop using the different structures.

While loop implementation

```
int num = 0;

while (num < 10) {
    statement(s);
    num++;
}
```

For loop implementation

```
for(int num = 0; num < 10; num++) {
    statement(s);
}
```

Do-while loop implementation

```
int num = 0;

do {
    statement(s);
    num++;
}
while(num < 10);
```

User Loop Control

For flexibility, a program can allow the user to determine the number of times a loop will iterate. Example Ex. 4.2 above was a user-controlled loop. The user was asked to enter “Y” to convert another temperature. In the example below

the user is asked to enter the number of sales amounts to total. The number is stored in the variable entries, which is used to control the loop.

Ex. 4.3 – User Controlled Loop

```
Scanner in = new Scanner(System.in);
int entries = 0;
double salesAmount = 0, total = 0;

System.out.print("Enter the number of sales amounts ");
entries = in.nextInt();

for(int i = 0; i < entries; i++) {
    System.out.print("Enter a amount ");
    salesAmount = in.nextDouble();
    total = total + salesAmount;
}
System.out.printf("The total is $%.2f", total);
```

Notice in the for-loop above that the variable "i" is initialized to zero and that the condition uses less-than the variable entries. It is important to implement the condition correctly. The program could have initialized "i" to one, and used less-than-equivalent-to entries as well. It could also be implemented using a while loop as shown here. Note the conditional statement and that the variable entries is decremented in the loop.

```
System.out.print("Enter the number of sales amounts ");
entries = in.nextInt();

while(entries > 0) {
    System.out.print("Enter an amount ");
    salesAmount = in.nextDouble();
    total = total + salesAmount;
    entries--;
}
System.out.printf("The total is $%.2f", total);
```

Loop Accumulator

When a program needs to compute a running total or accumulate values as it did in Ex. 4.3, it uses what is referred to as an ***accumulator***. The accumulator is a variable that tallies the values as the loop iterates and contains the total when the

loop finishes. As an example, the program below asks the user how many grades will be entered, and totals them as they are input. The variable totalGrades is initialized to zero and is used to accumulate the grades each time the loop executes. The output statement displays the average of the grades. Notice that numGrades is used to control the loop and to compute the average.

Ex. 4.4 – Grade Averaging Example Using an Accumulator

```
Scanner in = new Scanner(System.in);
int numGrades = 0, grade = 0;
double totalGrades = 0, average = 0;

System.out.print("Enter the number of grades ");
numGrades = in.nextInt();

for(int i = 0; i < numGrades; i++) {
    System.out.print("Enter a grade ");
    grade = in.nextInt();
    totalGrades = totalGrades + grade;
}
average = totalGrades/numGrades;
System.out.printf("The average grade is %.1f", average);
```

The accumulator inside the loop in Example Ex. 4.4 uses an expression common in programming, but impossible in Algebra. In mathematics, a value can never be equal to itself plus some value. In programming, this is an assignment statement and is perfectly acceptable.

```
totalGrades = totalGrades + grade;
```

Assignment statements that have the same variable on both sides of the assignment operator are common. It is important to understand this concept. The right-hand side of an assignment statement is evaluated first by the computer and then the result is assigned to the left-hand side.

Loop Counters

When the number of iterations a loop will execute is undetermined but is needed by the program, a ***counter*** variable is placed inside the loop. For example, a program that computes the average of a set of values needs to know the number of values that were entered in order to compute the average. The next example

uses a counter within the loop to count the iterations and then uses the count to compute the average of the values entered. Note that the variables are initialized and grouped together. This is in line with style and standards requirements. The user is asked to enter a sales amount or "Q" to quit. The loop ends when any character is entered and *hasNextDouble()* is false. Also notice that the prompt occurs before the loop and at the end of the loop. The user may quit before entering any amounts, and the loop will not execute. This is the reason for the condition at the end that tests to be sure the program does not divide by zero.

Ex. 4.5 – Counting the Iterations of a Loop

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

int numSales = 0;
double salesAmount = 0, average = 0, total = 0;

System.out.print("Enter a sales amount or \"Q\" to quit ");

while(in.hasNextDouble()) {
    salesAmount = in.nextDouble();
    total = total + salesAmount;
    numSales++;
    System.out.print("Enter a sales amount or \"Q\" to quit ");
}

if(numSales > 0) {
    average = total/numSales;
    System.out.printf("The average sale is %.2f", average);
}
else {
    System.out.printf("No sales amounts entered.");
}
```

```
Enter a sales amount or "Q" to quit 1.50
Enter a sales amount or "Q" to quit 2.50
Enter a sales amount or "Q" to quit 3.50
Enter a sales amount or "Q" to quit Q
The average sale is $2.50
```

Program Output

Common Loop Algorithms

Loops are commonly used in programming to implement algorithms including accumulating a total and computing an average of values as shown previously.

Others include validating input, finding the minimum or maximum of a set of numbers, or finding a match. Ex. 4.6 below requests a number within a specific range and the loop continues the request until a valid number is entered.

Ex. 4.6 – Input Validation

```
System.out.print("Enter a number between 1 and 10 ");
int number = in.nextInt();

while(number < 1 || number > 10) {

    System.out.println("That is not a valid number.");
    System.out.print("Enter a number between 1 and 10 ");
    number = in.nextInt();
}

System.out.print("Thank you.");
```

```
Enter a number between 1 and 10 12
That is not a valid number.
Enter a number between 1 and 10 9
Thank you.
```

Program Output

The next example determines the minimum value from a series of inputs. Notice that the first number entered is assigned to the variable smallest. Since it is the only number entered so far, it is the smallest. Any number that is input that is smaller than the one currently stored in smallest, is assigned to smallest overwriting the previous value. When a character is entered, the loop ends.

Ex. 4.7 – Finding the Minimum

```
System.out.print("Enter the integers, then \"Q\" to quit. ");
int number = in.nextInt();
int smallest = number;

while(in.hasNextInt()) {
    number = in.nextInt();
    if(number < smallest) {
        smallest = number;
    }
}

System.out.print("The smallest is " + smallest);
```

The algorithm in Ex. 4.7 could easily be modified to determine the largest value in the series, or to determine both the smallest and the largest. Note that the first number entered is both the largest and the smallest. As additional numbers are entered, the conditional statements assign new values to largest and smallest as needed.

```
System.out.print("Enter the integers, then \"Q\" to quit. ");
int number = in.nextInt();
int smallest = number;
int largest = number;

while(in.hasNextInt()) {
    number = in.nextInt();
    if(number < smallest) {
        smallest = number;
    }
    if(number > largest) {
        largest = number;
    }
}
```

Sentinels

In some of the example programs, the user was asked to enter 'y' to continue. In programming, a ***sentinel*** is often used to indicate that the end of the input has been reached by having the user enter a character or a number that could not be part of the set of values. As an example, if a program is requesting positive integers as input, the user may be prompted to enter -1 when finished. If a program is requesting numeric test grades, the user may be prompted to enter 999 when finished. The point is that a sentinel is a value that is outside the value set being used by the program. It is intentionally beyond a reasonable input value.

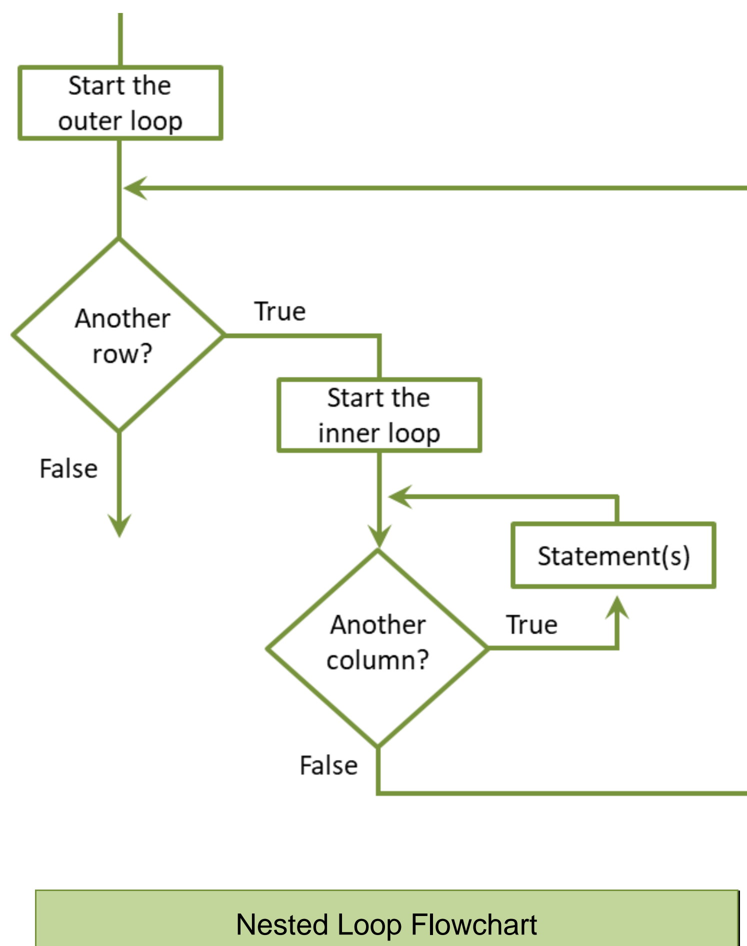
Nested Loops

When a loop is contained inside another loop, it is called a nested loop. An outer loop is entered and an inner loop executes. When the inner loop completes, if there are more outer loop iterations to complete, it again initiates the inner loop. A good example of this operation is a set of rows and columns. The output

would display a row of data across (each column), then the next row and all of its columns.

```
While there is a row of values to print
    While there is a column value to print
        print the value
```

As an example, a program that displays four (4) rows of data having three (3) values (3 columns) would have an outer loop that iterates four times (rows) and an inner loop that iterates three times (columns). For each execution of the outer loop, the inner loop executes three times.



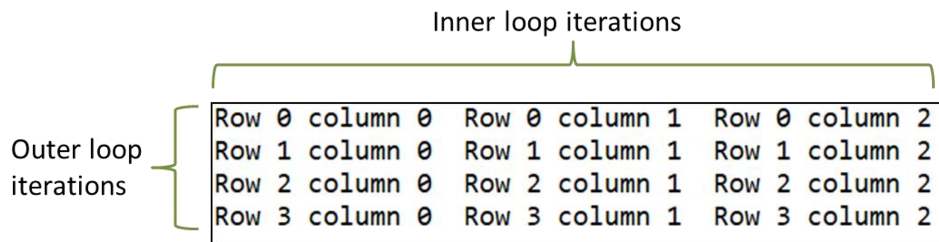
Nested Loop Flowchart

The code for the nested loop is shown below. Notice that the line feed is within the outer loop but outside the inner loop. After the iterations of the inner loop complete, the line feed executes and `row` is incremented. For each repetition of

the outer loop, the inner loop executes three repetitions. The output is shown below. The output statement includes the values of row and col for clarity.

Ex. 4.8 – Nested Loop

```
for(int row = 0; row < 4; row++) {

    for(int col = 0; col < 3; col++) {

        System.out.print("Row " + row + " column " + col + "\t");
    }
    System.out.println();
}
```



Common Loop Errors

Common errors associated with loops include off-by-one errors, where the programmer has written the conditional expression incorrectly and the loop is executing one too many or one too few times. This is easily corrected after running and testing the program. Others include confusing what should be inside the loop and what should be outside the loop. When these types of issues occur, adding print statements before, within, and after the loop can help to determine where the specific problem is located.

Proper Loop Construction

Although they are part of the language the use of the ***break*** and ***continue*** statements is not accepted by most Programming Standards. Break is used to break out of a loop at a point where continuing the loop would cause an error. Continue is used to jump to the end of the loop and bypass the loop logic, again to avoid an error. Both are typically used instead of designing a well-formed loop condition and loop body. Both increase debugging and maintenance time and should be avoided.

A Complete Example – Investment Program

Requirements:

Write a program for a Financial Adviser that computes the number of years to double an investment at a given annual interest rate.

Program Pseudocode:

Step 1 Prompt for the investment amount

Step 2 Prompt for the interest rate

Step 3 Compute the interest on the balance

Step 4 Add the interest to the balance

Step 5 Increment the number of years

Step 6 Is the balance < 2 times the investment amount

Yes, go back to Step 3

No, got to Step 7

Step 7 Display the number of years

Verbalizing and walking through the steps that the program will take while visualizing the program running can help to determine the sequence of events and the order of operations for the program.

*"Set up the program by initializing the balance and years, and obtain the investment amount and interest rate from the user. While the balance is less than 2 * investment, compute the interest on the balance and add it to the previous balance".*

*"When the balance is no longer less than 2 * the investment amount, display the number of years".*

Development

The development of the program follows the pseudocode. The number of years is initialized to zero, and the investment amount and interest rate are obtained from the user. Notice that balance is assigned the investment amount. As long as (while) the balance is less than twice the investment amount, compute the interest and add it to the balance, and increment the years. The solution below includes an output statement

within the loop for test purposes. Output statements are a great tool for quick debugging and testing programs.

```
Scanner in = new Scanner(System.in);
int years = 0;
double interest = 0, investment, balance, rate;

System.out.print("Enter the investment amount ");
investment = in.nextDouble();
balance = investment;

System.out.print("Enter the interest rate (4 for 4%) ");
rate = in.nextDouble();

while(balance < (2 * investment)) {
    interest = balance * (rate/100);
    balance = balance + interest;
    years++;
    System.out.printf("\nBalance is $%.2f", balance);
}

System.out.print("\n\nThe investment doubles in ");
System.out.println( years + " years.");
```

```
Enter the investment amount 12000
Enter the interest rate (4 for 4%) 4.5

Balance is $12540.00
Balance is $13104.30
Balance is $13693.99
Balance is $14310.22
Balance is $14954.18
Balance is $15627.12
Balance is $16330.34
Balance is $17065.21
Balance is $17833.14
Balance is $18635.63
Balance is $19474.24
Balance is $20350.58
Balance is $21266.35
Balance is $22223.34
Balance is $23223.39
Balance is $24268.44

The investment doubles in 16 years.
```

Program Output

Testing and Debugging

The development isn't complete until the program is tested and verified for accuracy. Testing can also surface questions about the requirements for the program. The output states that the balance doubled in 16 years, but the balance is actually more than double the initial amount at that point. Since interest is being computed and added annually, the program meets the requirements, but it might be a good idea to ask the Financial Advisor if this is what they had in mind.

Strings Revisited

Chapter 2 introduced Strings and the ***charAt()*** method. Loops are often used to inspect Strings by accessing the individual characters and testing for a condition or match. As an example, the program below replaces the occurrences of "t" in the String with "s" in the output.

Ex. 4.9 – String Inspection

```
String phrase = "She tells tea shells.";

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

    if(phrase.charAt(i) == 't') {
        System.out.print("s");
    }
    else {
        System.out.print(phrase.charAt(i));
    }
}
```

There are a few things to take note of in the example above. The for-loop begins with zero and ends at one less-than the length of the String. Recall that String indexes begin at zero. Second, the conditional statement is accessing each character using *charAt()* and comparing them to a char ('t' has single quotes). When a 't' is found, 's' is output, otherwise the character is output. Also note that "i" is used as a loop control and as an index for the String.

```
<terminated> Ex_4dot9 [Java Application]
She sells sea shells.
```

Program Output

To actually replace the character, the ***replace()*** method can be used which returns a copy of a String object with all occurrences of a specified character replaced by another specified character. Note the case sensitivity in the example.

```
String phrase1 = "she Sells Sea shells";
String phrase2 = phrase1.replace('S', 'T');
System.out.println(phrase2);
```

Only the uppercase occurrences of "S" would be replaced with "T" and the output would be "*she Tells Tea shells*".

The Character class provides additional String testing methods including *isDigit(), isUpperCase(), isLowerCase(), isWhiteSpace(),* and *isLetter()*. Each of these returns a Boolean value. The following program declares a String and enters a while loop to access and evaluate each character in the String.

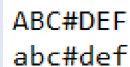
```
String myString = "ABCD12345def";
int index = 0, digit = 0, upper = 0, lower = 0;

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);
System.out.println("Uppercase " + upper);
System.out.println("Lowercase " + lower);
```

```
Digits 5
Uppercase 4
Lowercase 3
```

Program Output

The String modification methods include conversion to upper and lower case. These do not affect items that are not letters or are already in the desired case.

```java
String aString = "abc#def";
aString = aString.toUpperCase();
System.out.println(aString);

aString = aString.toLowerCase();
System.out.println(aString);
```



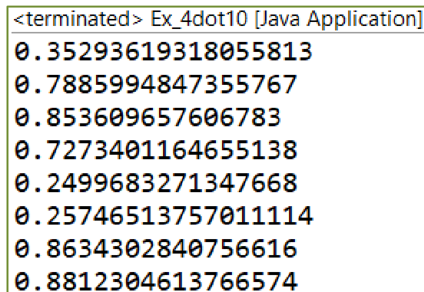
Program Output

Random Numbers

Most programming languages include a way of generating random numbers. Random numbers are used extensively in game scenarios, like card games and games that use dice, and in modeling and simulation to determine the occurrences of random events. It is often easier to simulate an occurrence than it is to have the event actually happen. Java includes random number generation within the Math class that returns a positive double from 0.0 to 1.0 inclusive.

Ex. 4.10 – Random Numbers

```java
for(int i = 0; i < 8; i++) {

    double ran = Math.random();

    System.out.println(ran);
}
```



Program Output

It might seem like these numbers couldn't really be used for anything, but the random number returned 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. 4.11, the random number is multiplied by 100 with 1 added to eliminate 0 and produce a random number between 1 and 100.

Ex. 4.11 – Random Number Ranges

```
int randInt = (int) (Math.random() * 100) + 1;
System.out.print("randInt is " + randInt);
```

To further explore this manipulation, consider one of the numbers that was produced in Ex. 4.10 (0.849430898282955).

Multiply by 100: 84.9430898282955

Next, add 1: 85.9430898282955

Next, the number is cast to an integer (discarding the fractional part).

Resulting number: 85

To show that zero can actually occur and must be removed by adding one, consider another random number (0.007128528411940449).

Multiply by 100: 00.7128528411940449

Next, the number is cast to an integer (discarding the fractional part).

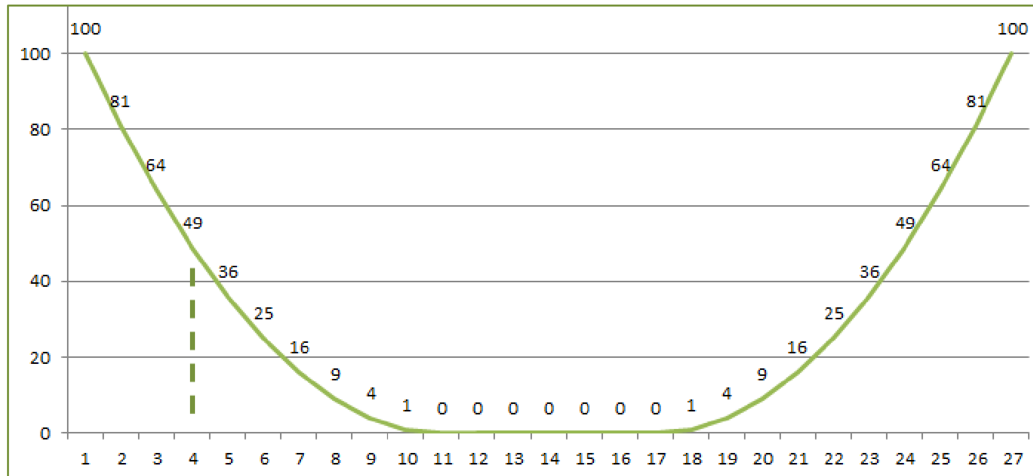
Resulting number: 0

Note that this type of manipulation can be used to generate any range of numbers. The standard algorithm for obtaining a random number within a range is shown here.

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

Simulations and digital games often use random numbers to determine random events. For example, the Oregon Trail game used a formula for snowfall that produced a U-shaped curve covering the trail from mountain range to mountain

range. A random number was generated and if it fell under the curve, which was likely at the beginning and end of the curve (near mountains), snow occurred. If the number fell above the curve, which was typical between the mountain ranges, no snow. The probability of snowfall changes as a function of location along the trail. Assume that the graph below represents the journey from mountain range to mountain range, and that the curve is the probability of snowfall. High up in the mountains it is snowing or 100% probability.



A random number between 0 and 100 would be generated at each interval.

```
int randInt = (int) (Math.random() * 100);
```

Let's say that the traveler is at step 4 along the journey and a random number is generated. Any number below 49 would result in snowfall. Any number generated that is 49 or above would not result in snowfall.

Programming Style and Standards

The use of break and continue bypass logic and are not acceptable operations. Performing computations in output statements is also ill-advised. A computed value should be assigned to a variable, and the variable should be used in the output statement. Output statements should perform output, not calculations. Both of these increase debugging and maintenance time and should be avoided.

```
double number = x * y / Math.PI;
System.out.println("Result is: %.2f", number);
```

Chapter 4 Review Questions

1. A structure that allows repeating steps without repeating code is referred to as a _______________ structure.
2. A loop that repeats while some condition is true is a ________ controlled loop.
3. Each execution of a loop is referred to as a(n) ____________.
4. A while loop is a ___________ loop and may or may not execute depending on the conditional statement.
5. A loop that repeats a specific number of times is a _________ controlled loop.
6. A loop that continues to run without a control or condition to stop it referred to as a(n) ___________ loop.
7. A variable within a loop that tallies a running total is a(n) _____________.
8. A variable within a loop that counts the number of iterations of the loop is referred to as a _____________.
9. A value entered by the user that is used by the program to indicate the end of a data set is referred to as a ___________.
10. A loop within a loop is referred to as a __________ loop.
11. A nested loop structure can be thought of as a spreadsheet with _______ and ___________.

Chapter 4 Short Answer Exercises

12. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
while (var1 < var2) {
        System.out.print(var1 + "  ");
        var1 = var1 + 1;
}
```

13. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
while (var1 < var2) {
        System.out.print(var1 + "  ");
        var1 = var1 + 2;
}
```

14. What do the following lines of code output if *var1* = 6, and *var2* = 8?

```
while (var1 <= var2) {
        System.out.print(var1);
}
```

15. What do the following lines of code output if *first* = 10, and *second* = 10?

```
while (first < second) {
        System.out.print("Enter a number ");
```

16. What do the following lines of code ensure?

```
System.out.print("Enter a positive number ");
int value = in.nextInt();
while (value < 1) {
        System.out.print("Enter a positive number ");
        value = in.nextInt();
}
```

17. How many times will the following loop display "Hello"?

```
for (int i = 0; i < 4; i++) {
        System.out.println("Hello");
}
```

18. In the following code, what term would be used to describe the variable total?

```
while (grades < 5) {
        System.out.print("Enter a grade ");
        int grade = in.nextInt();
        total = total + grade;
}
```

19. In the following code, what term would be used to describe the variable count?

```
while (count < 5) {
        System.out.print("Enter a grade ");
        int grade = in.nextInt();
        total = total + grade;
        count = count + 1;
}
```

20. In the following code, what term would be used to describe -99?

```
System.out.print("Enter a grade and -99 when done");
while (input != -99) {
        int input = in.nextInt();
        total = total + input;
        System.out.print("Enter a grade and -99 when done");
}
```

21. Rewrite the following loop as a while loop.

```
int number = 10;
for(int count = 0; count < 10; count++) {
        number = number – 1;
        System.out.println("Number is : " + number);
        System.out.println("Count is : " + count);
}
```

22. Correct the 3 errors in the following code to produce even numbers only.

```
for(int num = 1; num < 20; num--) {
        if(num < 20)  {
            num = num + 2;
        System.out.println("Number is : + num");
}
```

Chapter 4 Programming Exercises

23. Write a program that sets a variable num to 0 and uses a while loop to display the numbers 0 thru 9 separated by a space using the variable. Use the increment operator in the solution.

24. Write a program that sets a variable num to 2 and uses a while loop to display the even numbers 2 thru 20 separated by a space using the variable.

25. Write a program that sets a variable num to 2 and uses a while loop to display the even numbers 2 thru 20 separated by a space using the variable, and use the modulus operator in the solution.

26. Write a program that prompts for a negative integer, and uses a while loop to validate the input. If the number entered is not a negative number, the loop will output the error message "Invalid input", and request another number. When a valid number has been entered, output "Thank you".

27. Write a program using a for-loop that displays a heading and the columns of data shown below containing the numbers 1 thru 10 and the squares of the numbers. Use the tab escape character and width specifier as needed.

```
 Number Square
---------------
    1       1
    2       4
    3       9
    4      16
    5      25
    6      36
    7      49
    8      64
    9      81
   10     100
```

28. Write a program using a nested loop that displays the number shown below in 3 rows and 5 columns using the variable for the inner loop condition in the output statement.

```
1    2    3    4    5
1    2    3    4    5
1    2    3    4    5
```

29. Write a program that prompts for a word from the user, stores it in a String, and then displays each character in the word separated by a tab. Use the charAt() method in the solution . As an example, if the word "*Java*" were entered:

```
Enter a word Java
         J        a        v        a
```

30. Write a program that prompts the user for a temperature in Fahrenheit to convert to Celsius. Display both of the temperatures and allow the user to convert another temperature without restarting the program. Use integers for the temperature values, and a String for the loop condition. The equation for conversion is shown below.

$$C = (F - 32) / 1.8$$

31. Write a program that will generate 6 random integers between 1 and 20 inclusive. As the numbers are generated, display the results in rows of asterisks as shown below.

    Sample program run:

```
*******************
*****
***************
*
**************
*******************
```

32. When four random numbers are generated between 1 and 6 inclusive, the probability is high that at least one 6 will be produced. Write a program that produces four (4) random numbers and displays “Found a six” if a six was produced and “No six” if none was produced. Include a loop in the program that runs the scenario 20 times. Does six occur more than expected?

33. Write a program that requests a cable length (must be positive) from the user and a cable thickness (must be between 0.1 and 2.5) in inches. Validate the input and keep requesting until valid input is received. Write a loop that will simulate applying one pound of tension and stretching the cable for each repetition of the loop.

    The cable will stretch its thickness times 0.28 feet for each pound of tension applied. The cable will break when it is 112% of its original length. Output the pounds of tension on the cable and the length for each pound applied, and announce when the cable has broken.

    Sample program run:

```
Enter a cable length in feet 24
Enter a cable thickness (0.1 and 2.5) inches 1.75
Tension: 1 lbs.  length:24.5
Tension: 2 lbs.  length:25.0
Tension: 3 lbs.  length:25.5
Tension: 4 lbs.  length:26.0
Tension: 5 lbs.  length:26.4
Tension: 6 lbs.  length:26.9
The cable has broken.
```

Chapter 4 Programming Challenges

#1 Password Validation

Write a program that requests a password, and validates the password against the criteria below. If the password is valid, display "Password Accepted" and end the program.

If the password is not valid, display specifically why it is not valid (what it does not contain) and end the program.

The password must be at least 9 characters long and have:

- At least one uppercase letter
- At least one lowercase letter
- At least one digit

Sample program runs:

```
Enter a password of at least 9 characters
that contains at least one uppercase letter,
lowercase letter, and digit.
invalid
The password is not at least 9 characters.
The password does not contain a digit.
The password does not contain an uppercase letter.
```

```
Enter a password of at least 9 characters
that contains at least one uppercase letter,
lowercase letter, and digit.
nouppercase45
The password does not contain an uppercase letter.
```

```
Enter a password of at least 9 characters
that contains at least one uppercase letter,
lowercase letter, and digit.
goodPass123
Password Accepted
```

#2 Drainage Canal

A canal has a natural flow rate of 40 ft^3 /s at 3.3 feet. Rainfall increases the water level of the canal and a flood gate must be opened to remove the excess water.

Prompt the user for the water level in feet (must be > 3.3) and the number of feet to open the flood gate (must be >= 1). Validate the input with loops that alert the user of invalid input and request input again. Compute the time to lower the level to 3.3 feet while displaying the minutes passed, and the current level of the canal.

The program will simulate the discharge of water through the flood gate using a loop at a rate of 0.03 feet of water per minute for each foot that the food gate is open. This will continue until the water level in the canal has reached 3.3 feet.

The program will announce when the canal has reached the natural level of 3.3 feet and end. Note the output alignment in the sample below.

Sample program run:

```
Enter the water level (> 3.3) 4.6
Enter the gate opening in one foot increments 3

Minutes:   1  Water Level: 4.51
Minutes:   2  Water Level: 4.42
Minutes:   3  Water Level: 4.33
Minutes:   4  Water Level: 4.24
Minutes:   5  Water Level: 4.15
Minutes:   6  Water Level: 4.06
Minutes:   7  Water Level: 3.97
Minutes:   8  Water Level: 3.88
Minutes:   9  Water Level: 3.79
Minutes:  10  Water Level: 3.70
Minutes:  11  Water Level: 3.61
Minutes:  12  Water Level: 3.52
Minutes:  13  Water Level: 3.43
Minutes:  14  Water Level: 3.34
Minutes:  15  Water Level: 3.25

The water level is now at 3.3 feet.
```

Chapter 5

Methods, Modules, and Basic Graphics

Methods

As programs become longer and execute more tasks, the main method grows and code may be repeated in order to repeat operations. The design process includes dividing the program into logical sections of distinct functionality which will be developed individually. This is referred to as ***modularization***. Separating the program into distinct parts provides many benefits including the ability to: reuse portions of the code, divide the program development among multiple programmers, and simplify the task. Instead of writing one long program, logical sections are developed in methods, and the methods are ***called*** when needed, and as many times as needed. Some methods, like *next(), round(), length(), and substring()* were used in previous programs. There are many more available, and programmers write their own methods as well.

There are two types of methods, ***void methods*** that simply perform a task and ***value-returning methods*** that return a value. The method *System.out.print()* is a void method. It performs output, but does not return anything. The *nextInt()* method on the other hand, is a value-returning method that returns an integer which is received and used by the program. Note the difference in their use below. The *print()* method varieties simply display what is passed to them.

```
System.out.println("Void Method");
```

The *nextInt()* method returns a value that is assigned to a variable.

```
int number = in.nextInt();
```

Notice that we do not see the code that is executed when these methods are called. The inner-workings are hidden. We use the methods knowing the task that they perform, but it isn't necessary that we know how the methods perform their tasks. This is often referred to as the Black-box analogy. A car can be driven even if the driver does not know the specifics of how the accelerator works. The driver presses on the accelerator and the car moves.

Writing Methods

The code for a method is called the ***method definition*** and it begins with the keyword ***public*** which is followed by the keyword static, a name for the method, and a pair of parentheses. The parentheses will contain any ***parameters*** that are passed to the method that it needs to perform its task. The first line of the method definition is referred to as the ***method header***.

not associated with an object

method name

public static void someMethod( )

public access

no return value

The statements that will execute when the method is called are enclosed in braces to form a block of the code called the method ***body*** and are indented. The following void method displays the words "In displayString" when it is called.

```
public static void displayString() {

    System.out.println("In displayString");
}
```

The method is located outside of main, and is called from within main as shown in Ex. 5.1 below. Note that the IDE italicized the method call in main.

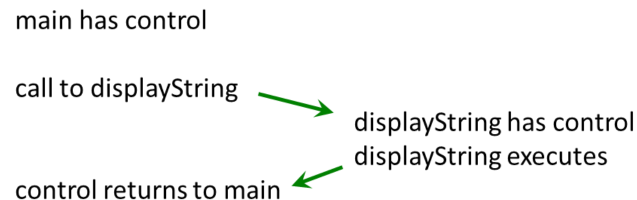
Ex. 5.1 – Void Method

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

    displayString();
}


public static void displayString() {

    System.out.println("In displayString");
}
```

```
<terminated> Ex_5dot1 [Java Application]
In displayString
```

Program Output

When a method is called, program control transfers to the method. When the method completes, control returns to the calling method at the point where the method was called. The flow of control for Ex. 5.1 is shown here.



Value Returning Methods

A void method should not have a return statement because it does not return anything, however a value returning method returns a value to the calling part of the program. The data type of the return value replaces the word void in the method header, and the method has a return statement. The general format is shown below.

```
public static data type methodName() {
        statement(s);
        return data type;
}
```

Notice that the last line of the method is the return statement. This follows programming logic: input, processing, and output. A method should have only one return statement and it should be the last statement in the method.

The method in Ex. 5.2 below obtains and returns the city of the program user. Note that the data type (String) of the returned value is in the method header, and that a variable city is declared within the method to store the input from the user. The variable is then used in the return statement.

Ex. 5.2 – Value-returning Method

```
public static String getUserCity() {

    Scanner in = new Scanner(System.in);
    String city;

    System.out.println("Enter your city ");
    city = in.nextLine();

    return city;
}
```

Calling the method in Ex. 5.2 requires receiving the return value. In the example below, it is assigned to the String userCity when it is called by the main method.

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

    String userCity = getUserCity();

    System.out.println("Your city is " + userCity);
}
```

Methods Cannot Return Multiple Values

Methods in Java cannot return multiple values. They should implement a limited number of operations and return one value or no value at all. Java is an Object-Oriented Language and Objects tend to contain multiple methods, each of which has a single task.

Passing Values to Methods

Methods cannot access variables declared in other parts of the program, but often they need to use them. When a method needs to use a variable defined somewhere else in the program, the variable is passed to the method as an ***argument***. The method receives it as a ***parameter***. Technically speaking,

arguments are passed to methods and parameters are received by them. In the following example, main calls a method and passes a variable (argument) to the method that receives it (parameter) into a local variable, and uses it in an equation and output statement. Notice that *argument* is the name of the variable passed as the argument, and *parameter* is the name of the variable receiving the parameter in the method header. This highlights that a value is passed. The computer goes to the memory location for the variable argument, finds out what is stored there and passes a copy of it to the method. The method receives the value and stores it in the memory location for parameter. This is referred to as ***pass-by-value***.

It doesn't matter what the receiving method calls the value that it receives, except that it must use that name internally.

Ex. 5.3 – Passing an Argument to a Method

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

    int argument = 5;

    computeSquare(argument);
}

public static void computeSquare(int parameter) {

    int square = parameter * parameter;

    System.out.println("The square is " + square);
}
```

The argument passed to a method does not need to be a variable. A literal can be passed as well. The method in the example could be called as shown here.

```
computeSquare(5);
```

Variables and Scope

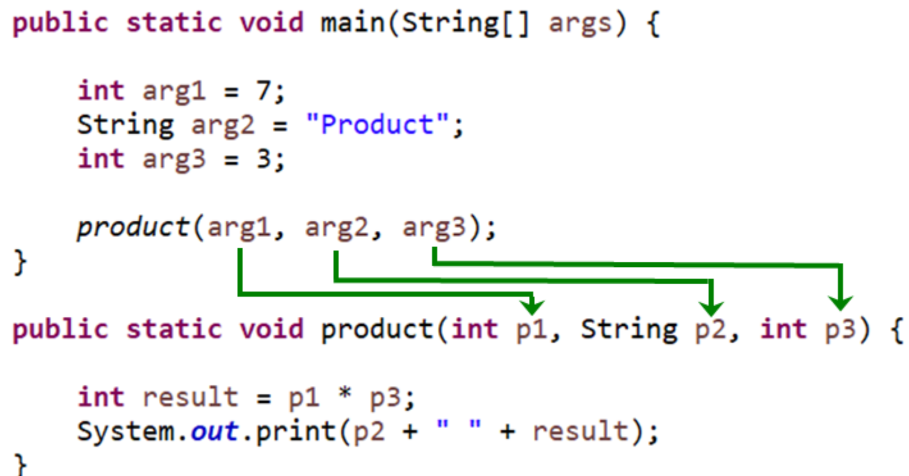
The part of a program where a variable is accessible is referred to as the variable's ***scope***. When a variable is declared within a method (including main), the scope of the variable is the method. It is referred to as a ***local variable***. A variable declared inside a method is not accessible outside that method, so

different methods could have variables with the same name without causing any conflict. The scope for each of the variables would be their particular method, and they would not be accessible by another method. If several engineers are working on the same program, but they are working on different methods, they may name a local variable the same name. This is the reason that variables must be passed to methods, and why methods return values. If a program attempts to access a variable outside of its scope, an error occurs. Chapter 2 covered global variables which have program scope, and should only be used as constants.

Passing Multiple Values to Methods

Multiple arguments can be passed to methods. The arguments can be different data types, and are received in the order that they are passed. Example Ex. 5.4 demonstrates passing three arguments to a method. Note the data types in the parameter list inside the parentheses. Each of the parameters is actually a declaration of a variable that is local to the method. These variables receive a copy of the values in the variables that are passed to them. In other words, a copy of the value in `arg1` is passed to the method and received into `p1` which is used within the method.

Ex. 5.4 – Passing Multiple Arguments to a Method



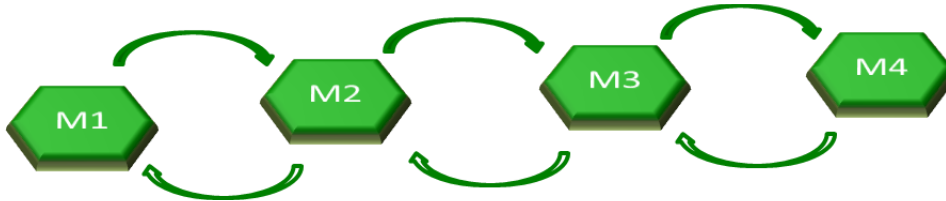
```
public static void main(String[] args) {

    int arg1 = 7;
    String arg2 = "Product";
    int arg3 = 3;

    product(arg1, arg2, arg3);
}

public static void product(int p1, String p2, int p3) {

    int result = p1 * p3;
    System.out.print(p2 + " " + result);
}
```

```
<terminated> Ex_5dot4 [Java Application]
Product 21
```

Program Output

Methods Calling Other Methods

Main is a method and as shown in the examples above, main can call other methods. Those methods can also call other methods which can call other methods. Each method will return control to the calling method. It is similar to walking forward on a few stepping stones, and then back again using the same stones.



In the example below, main calls a method which then calls the square root function. Control transfers from the main method to *getRoot()*, and then to the square root function. The square root function returns the value to the *getRoot()* method, which then returns the value to the main method.

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

    double num = 25;
    double sqRoot = getRoot(num);
    System.out.print(sqRoot);
}

public static double getRoot(double number) {

    double root = Math.sqrt(number);
    return root;
}
```

As previously noted, a method should accomplish a single task. They are used to modularize programs, break down complex tasks, and increase reuse. When multiple tasks are required, multiple methods should be used. The following example calls three methods. The method *getPassword()* is called from main and returns a String. The String is assigned to pw and is then passed to the method *longEnough()* which returns a Boolean value (true or false). The Boolean value is passed to the *output()* method which determines the correct print statement. Notice that main simply consists of three method calls, and that the method

definitions are in the same order that the methods are called by the program. This is a programming practice that enhances readability and maintainability.

Ex. 5.5 – Calling Multiple Methods

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

    String pw = getPassword();
    boolean ok = LongEnough(pw);
    output(ok);
}

public static String getPassword() {
    Scanner in = new Scanner(System.in);
    System.out.print("Enter your password ");
    String pass = in.next();
    return pass;
}

public static boolean longEnough(String pass) {

    boolean valid = false;
    if(pass.length() > 9)
        valid = true;

    return valid;
}

public static void output(boolean valid) {

    if(valid) {
        System.out.print("Valid password ");
    }
    else {
        System.out.print("Not valid.");
    }
}
```

The example demonstrates a first step toward modular programming with the majority of the code being executed in methods. The main method becomes a series of method calls to execute the program. Most of the functionality that programs execute can be placed in methods. As the requirements for a program are refined and the Design Phase progresses and the order of operations becomes clear, sections of the program will surface that should be implemented in methods. Once, the functionality is determined, the methods can be defined.

Naming and Defining Methods

The method naming convention in Java is the same as the variable naming convention. The first word is all lower case and the first letters of any additional words are uppercase. When creating a method, there are several things to consider and steps that can help in the process. First, determine what the method will do. Each method should accomplish one task, and the name of the method should describe what it does. Since methods perform an action, names with verbs are typical, like `computeNetPay` or `getTaxRate`. Next, determine what parameters the method will need. Then, determine if the method returns a value and if so, what data type will it return, and any local variables it needs.

Step 1 name the method what it does
Step 2 determine the parameters that it needs
Step 3 determine if the method will return a value
- If yes, determine the return type

Step 4 determine if the method will need local variables
Step 5 write the method
Step 6 write the method call

Attempting to access a variable in a method before it has been declared will cause an error. It is always best to declare all variables needed by a method together and first within the method. This makes readability and maintenance much easier and reduces the possibility for errors. Declaring variables where needed runs the risk of duplicating a variable with a different name, and introducing hard-to-find bugs.

Recursion

In the examples, the main method called other methods to perform operations, but methods can call themselves as well (except for main). This is referred to as ***Direct Recursion***, and it occurs when a method calls itself with an argument that is the result of the previous call to that same method. In other words, the output from the method is the input into the next call to that same method. The method will continue to call itself until a ***base case*** is reached similar to ending a loop. As an example, consider a method that receives an integer, displays the

integer, and calls itself after decrementing the integer. The method will call itself until the argument passed to it reaches zero (the base case).

```
public static void timer(int ticks) {
    System.out.println("Count is now " + ticks);
    if(ticks > 0) {
        timer(ticks - 1);
    }
}
```

When the method is called and passed 9, the output below shows the number decreasing as the recursion occurs.

```
<terminated> CH_5_recursion [Java Application]
Count is now 9
Count is now 8
Count is now 7
Count is now 6
Count is now 5
Count is now 4
Count is now 3
Count is now 2
Count is now 1
Count is now 0
```

As noted previously, after a method completes, control returns to the calling method at the point of the call. This is highlighted by moving the output statement to the line *after* the recursive call.

```
public static void timer(int ticks) {

    if(ticks > 0) {
        timer(ticks - 1);
        System.out.println("Count is now " + ticks);
    }
}
```

The output now occurs after control returns and the numbers output count up.

```
<terminated> CH_5_recursion [Java Application]
Count is now 1
Count is now 2
Count is now 3
Count is now 4
Count is now 5
Count is now 6
Count is now 7
Count is now 8
Count is now 9
```

Another type of recursion is ***Indirect Recursion*** where method 'A' calls method 'B' which then calls method 'A', and so on. Method 'A' could call 'B' which calls 'C' which calls 'A' or any variation as well.

> *Technical Notes*
>
> Any repetitive algorithm can be implemented with a loop, and the use of recursion is never required. Recursion also uses computer resources for each call to the method especially if local variables are being declared. If the solution is more easily implemented with a loop, then a loop is a better choice.

Functions and Methods - Terminology

Some confusion may arise as a result of different languages using different terminology with respect to methods and functions. For example, Java uses the terms method and function (sometimes interchangeably), C/C++ uses the term function, and Python uses both function and method. For clarification, the Java definitions follow, but if either is used the listener will know what is meant. Both are named blocks of code that execute when called.

Function – a static method (not associated with an object)

Method – associated with an object

Notice that when we use the Math.*sqrt()* function, we do not create an object of the Math class. We simply precede the call with Math and the dot operator.

```
int num = Math.sqrt(9);
```

However, when we use the String length method, a String object is first created and the method is accessed using the String variable name and the dot operator.

```
String word = "password";

int len = word.length();
```

Again, it is really a matter of semantics with Java.

Design (IPO) Documents

A tool used to design and document programs is a Design Document called an Input, Processing, Output document or ***IPO*** document. An IPO includes an overall description of the program and the methods used. The method section of the IPO includes the name of the method, a brief description of what it does, the input (parameters) needed, the processing it will accomplish, and the output or return value. The next example uses methods to obtain a number from the user, compute the square of the number, and display the result. Notice that main follows the methods in this example. In a single file program, the code for the methods can be above or below main (the next section will take a further look at modularization). An IPO for the program follows the code.

Ex. 5.6 – IPO (Input, Processing, Output) Documentation

```
public static int getInput() {
    Scanner in = new Scanner(System.in);
    System.out.print("Enter a number ");
    int num = in.nextInt();
    return num;
}

public static int getSquare(int num) {
    int square = num * num;
    return square;
}

public static void output(int square) {
    System.out.print("Number squared is " + square);
}

public static void main(String[] args) {

    int userNum = getInput();
    int square = getSquare(userNum);
    output(square);
}
```

The IPO or Design Document for the program first provides a general description of the program, the input, processing, and output of the program, and then handles each of the methods the same way. The IPO content is subjective and differs among organizations that use them, but the overall concept is consistent with its name. They provide a brief summary of the input, processing, and output of the program and methods. They are another tool that

can be used in the design process to save time and produce modularized programs.

Program IPO:

Description: the program calls three methods to obtain user input of a number, square the number, and display the square of the number.

Input: number from user

Processing: square the number

Output: display the result

Method IPOs:

public static int getInput()

Description: Obtains user input

Input: number from user

Processing: none

Output: returns the number

public static int getSquare(int number)

Description: Computes the square of the number

Input: a number

Processing: square the number

Output: return the value

public static void output(int number)

Description: Produces output

Input: the value to display

Processing: none

Output: display the result

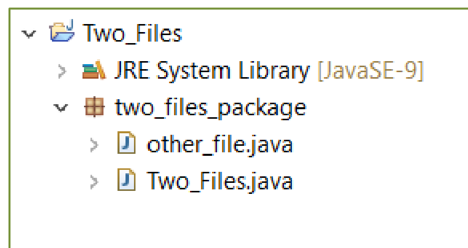
Modular Programming

Modularizing programs using methods separates operations into manageable chunks and enhances maintenance, but multiple engineers cannot easily work on the same program because it is in a single program file. Separating the program

into files (modules) allows multiple engineers to work on the same program at the same time enhancing what is referred to as ***collaborative development***. Collaborative environments like Configuration Management Systems covered in Chapter 1 facilitate the development and control of multi-file programs and development by multiple engineers. As discussed previously, large and complex program requirements are decomposed during design into manageable sections, and are then further refined into methods. Methods that are related can be developed in a file together, or if the method is large, in a file by itself. A step-by-step walk-thru of creating a second file is provided in Appendix D.

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 D). The following package contains two files with the main method in one and an example method in the other. The package explorer shows the two files.



The main method is located in the file named *Two_files.java* and contains the method call to *getSquaredVal()* which includes the second file's class name followed by the dot notation and name of the method.

```
package two_files_package;

public class Two_Files {

    public static void main(String[] args) {

        int num = 5, square;

        square = other_file.getSquaredVal(num);
        System.out.println("The square is " + square);
    }
}
```

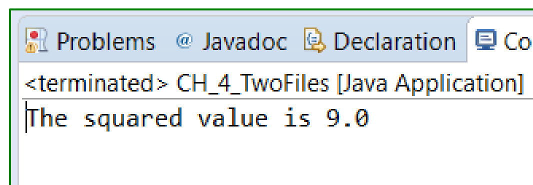
It is always best to use the "***build-a-little, test-a-little***" process when developing programs. Develop a small part of the program and test and debug that part until it is working correctly. Then, develop another small part and test and debug the program again. If there is an error, it is most likely in the part that was recently added. This process is referred to as incremental programming or ***iterative enhancement***. The method for the example program has not been completed. The second file other_file.java contains the method, but it is incomplete as shown below. It is a ***stub***. A stub is an incomplete method that returns a value that could not be a real value for the method.

```
package two_files_package;

public class other_file {

    public static int getSquaredVal(double num) {

        return -99;

    }
}
```

Stub Example

As development continues, the stub is replaced with a complete method and testing continues.

```
package two_files_package;

public class other_file {

    public static int getSquaredVal(int num) {

        int squared = num * num;
        return squared;

    }
}
```



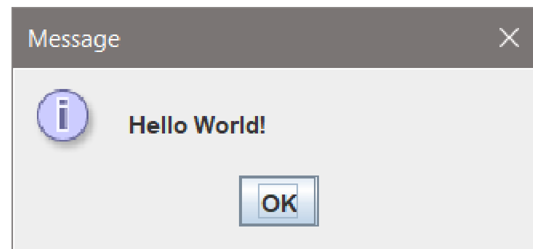
Program Output

If the method were in another package, the package and the class would be imported the same way that the Scanner is imported from the java.util library.

Dialog Boxes

The JOptionPane class which is part of the swing components provides message, information, input, and error ***dialog boxes*** for displaying messages and obtaining input. The JOptionPane must be imported from javax.swing to use them. The example below creates a ***message dialog*** box with the text "Hello World!" The first argument is "null" which would be the parent window for the program. In this this case there isn't one and null causes the message box to be located in the center of the display.

```
import javax.swing.JOptionPane;

public class CH_5_Dialogs {

    public static void main(String[] args) {

        JOptionPane.showMessageDialog(null, "Hello World");
    }
}
```



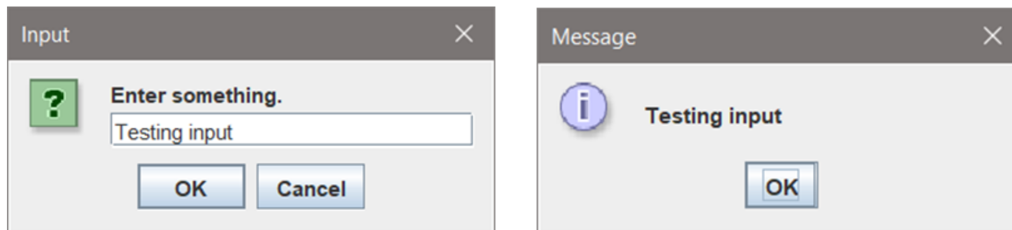
To show an error or alert, two additional arguments are passed to the dialog. The third is the title on the title bar, and the fourth is a JOptionPane constant.

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

The class also contains an ***input dialog*** with two buttons (OK, and Cancel) that returns a String when the user clicks the "OK" button. If the "Cancel" button is clicked, an empty String is returned. The following program requests input

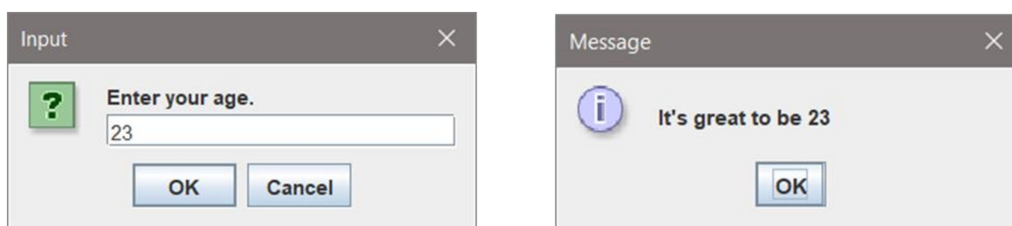
through the input dialog and assigns the return value to a variable input which is used in a message dialog.

```
public static void main(String[] args) {
    String input = JOptionPane.showInputDialog("Enter something.");
    JOptionPane.showMessageDialog(null, input);
}
```



The show input dialog returns a String. If the program is obtaining a number from the user, the String can be parsed to a number as shown below. The example then uses the result in the message dialog.

```
public static void main(String[] args) {
    String ageStr = JOptionPane.showInputDialog("Enter your age.");
    int age = Integer.parseInt(ageStr);
    JOptionPane.showMessageDialog(null, "It\'s great to be " + age);
}
```



There are many other dialog boxes including confirmation, and “YES, NO, CANCEL” dialogs that can be used to enhance programs, as well graphical user interfaces developed with frames which will be covered in a later chapter. Error dialogs are most commonly used to alert a user of missing or invalid input.

Drawing Simple Graphics

One way to draw simple graphics in Java requires creating a frame, and then adding a JComponent, JPanel, JTextComponent, or JLabel to the frame. In Ex. 5.7, a JFrame (window) is created which is a container in Java that can hold components. To create a JFrame, the swing components library is imported. Programmers often use the ***wildcard*** import javax.swing.* which imports all of the classes in the package. The example below imports only the JFrame.

```
import javax.swing.JFrame;
```

The code below creates a simple frame.

Ex. 5.7 – A Simple Frame (Window)

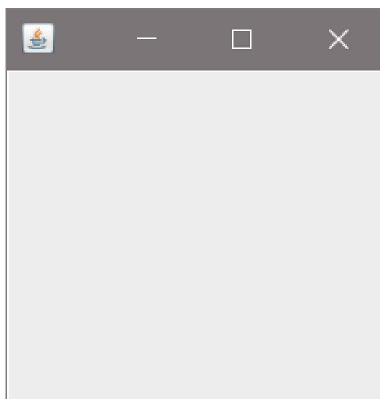
```
1. public static void main (String[ ] args)  {
2.
3.      JFrame myFrame = new JFrame();
4.      myFrame.setSize(200,200);
5.
6.      myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
7.      myFrame.setVisible(true);
8. }
```

Line 3 creates a JFrame and assigns it to myFrame

Line 4 sets the initial size of the frame in pixels

Line 6 sets the default close operation to *exit* so that if the window is closed the program ends. Other options are covered later in the text.

Line 7 makes the frame visible (by default, it would not be visible)

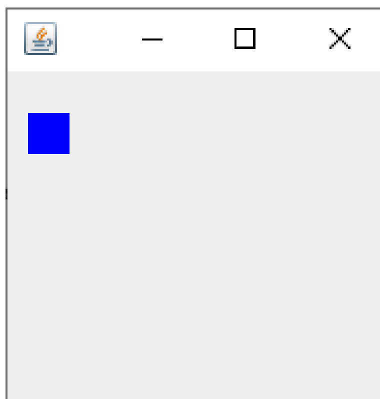


Program Output

The example above made the frame visible. By default, a frame is not visible which allows control over what is visible as a program runs. To draw on the frame, a JComponent is added and the paintComponent method does the drawing. It is called automatically when the frame is created, and when the window is resized, exposed after having been covered, or needs repainting (to be redisplayed). The paintComponent receives the Graphics object being redisplayed which has the graphics' state (color, font, etc.) and methods for drawing shapes. In Ex. 5.7A, the graphics object method *fillRect()* performs the drawing of a filled rectangle. Notice the next to last line which adds the JComponent to the frame.

Ex. 5.7A – Drawing on a Frame

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

    JFrame myFrame = new JFrame();
    myFrame.setSize(200,200);
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JComponent comp = new JComponent()
    {
        public void paintComponent(Graphics g)
        {
            g.setColor(Color.BLUE);
            g.fillRect(10, 20, 30, 30); // x, y, width, height
        }
    };

    myFrame.add(comp);
    myFrame.setVisible(true);
}
```



Program Output

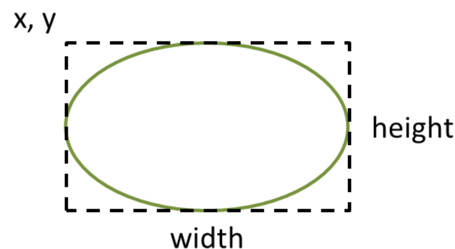
Drawing methods are provided to draw arcs, polygons, and for setting the color and fonts, as well as those listed below. Note that in graphics, the x, y coordinates 0, 0 are the top-left corner of the frame. To move the y coordinate down requires a positive number. Both coordinates are in pixels.

```
drawRect(x, y, width, height)

fillRect(x, y, width, height)

drawOval(x, y, width, height)

fillOval(x, y, width, height)

drawLine(x1, y1, x2, y2)

drawString("String to draw", x, y)
```

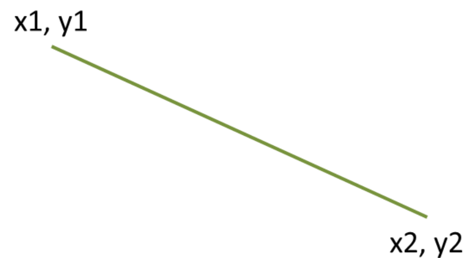
Drawing Methods

The x and y arguments passed to the methods are the starting coordinates (top-left) when drawing a shape. The ***drawString()*** method uses x and y as the coordinates for the starting lower-left (base-point) of the String.

Oval and Rectangle drawing



Line drawing



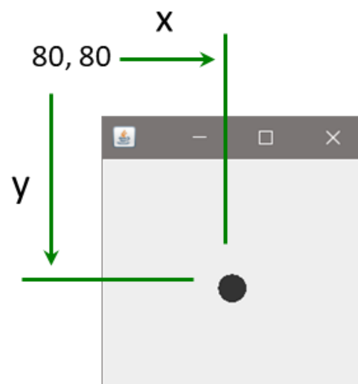
String drawing

String to draw
x, y

A method can be called to handle drawing. In Ex. 5.7B, the paintComponent calls a method named *drawShapes()* that performs the drawing. The graphics object is passed as the argument to the method. The drawing method receives the Graphics object and draws a circle (an oval with the same width and height).

Ex. 5.7B – Drawing from a Method

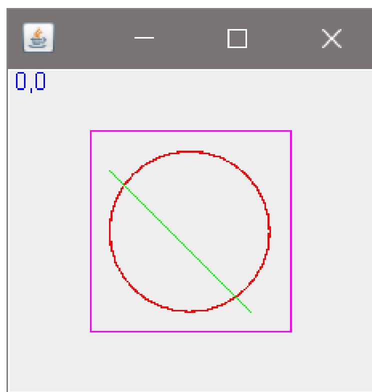
```
public static void main(String[] args) {

    JFrame myFrame = new JFrame();
    myFrame.setSize(200,200);
    myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JComponent comp = new JComponent()
    {
        public void paintComponent(Graphics g)
        {
            drawShapes(g);
        }
    };

    myFrame.add(comp);
    myFrame.setVisible(true);
}

public static void drawShapes(Graphics g)
{
    g.fillOval(80, 80, 20, 20);
}
```



Recall that x, y coordnates 0, 0 are the top-left corner of the frame, and the y coordinate is a positive number down from the top. In Ex. 5.7C, multiple draw methods were added to the previous *drawShapes()* method, and the use of ***setColor()*** is shown. Note that the y coordinate for the String is 10 because it is the basepoint of the text. If y were zero, the text wuld be ouside the frame drawing area. The *setColor()* method accepts any of the Java colors, and sets the color until it is set to another.

Ex. 5.7C – Combining Draw Methods

```
public static void drawShapes(Graphics g)
{
    g.setColor(Color.BLUE);
    g.drawString("0,0",2,10);

    g.setColor(Color.RED);
    g.drawOval(50, 40, 80, 80);

    g.setColor(Color.GREEN);
    g.drawLine(50, 50, 120, 120);

    g.setColor(Color.MAGENTA);
    g.drawRect(40, 30, 100, 100);
}
```



Program Output

Programming Style and Standards

Variables should be declared together and first in a method, and method names should be descriptive and indicate what they perform. This programming style makes readability and maintainability much easier and saves development time and reduces bugs.

Chapter 5 Review Questions

1. Separating a program into distinct sections is referred to as ____________.
2. To execute a method, it must be ___________.
3. The two types of methods are ______ methods and ____________ methods.
4. The code within a method is called the method ___________.
5. The first line of a method that contains the name and parameter list for the method is called the method ____________.
6. The area of a program where a variable is accessible is the variable's ________.
7. A variable declared inside a method is referred to as a ________ variable.
8. A void method (should/should not) _____________ have a return statement.
9. Technically speaking, a value passed to a method is called a(n) __________.
10. Technically speaking, a value received by a method is called a(n) ___________.
11. A value-returning method must have a ___________ statement.
12. When a method calls itself, it is referred to as _____________.
13. An IPO document contains brief descriptions of the _______, __________, and ___________ of a program or method.
14. A term used to describe multiple programmers working together on the same program is ______________ development.
15. An incomplete method that is used as a placeholder and usually returns a value that could not be a real value for the method is called a _________.
16. A _______ _____ is a small graphical window that displays a message or requests input.
17. In graphics, the x, y coordinates 0, 0 are located at the __________ of the frame.

Chapter 5 Short Answer Exercises

18. Write a statement that calls the following method.

```
public static void showOutput( ) {
        System.out.print("Hello from my method")
}
```

19. Write a statement that calls the following method and passes it the phrase "Hello World".

```
public static void showOutput( String phrase) {
        System.out.print(phrase);
}
```

20. What does the method below output when it is called as shown?

```
smallest(6, 3);

public static void smallest(first, second) {
        if (first < second) {
                System.out.print("first is smaller");
        }
        else {
                System.out.print("second is smaller");
        }
}
```

21. Write a statement that calls the following method and stores the return value in a variable named num.

```
public static int getValue( ) {
        int val = 200;
        return val;
}
```

22. Write a method doubleUp() that receives an integer and returns twice the number that was passed in.

23. Write a method largestOne( ) that receives three doubles and returns the largest of the three.

24. Write an IPO for the method in #23 above.

25. Write a method called displayNumber( ) that obtains a number from the user and displays "You entered ", and the number that was entered.

26. Write a method called getInput( ) that obtains an integer from the user and returns the number that was entered.

Chapter 5 Programming Exercises

27. Write a program that calls a method named *average* that accepts three (3) integers as arguments, and returns the average. The main method will store the return value in a variable and then display "The average is " and the number.

28. Write a program with two (2) methods. The first method will obtain and return the radius of a circle from the user, and the second will compute and return the circumference of the circle. Then, main will display "The circumference of the circle is " with the result formatted to two decimal places. The equation for circumference is shown here. Use Math.PI in the solution.

$$C = 2\pi r$$

29. Modify the circle program in #28 to locate the methods in a separate module (file) that is part of the same package.

30. Write an IPO for the program in #28, and include a method IPO for each of the methods.

31. Write a program with three (3) methods located in a separate module. The main method will prompt the user for the two side lengths of a rectangle and validate the input (must be > 0). The first method called will compute and return the area, the second method will compute and return the perimeter, the third method will compute and return the diagonal, and then main will display the values as shown in the sample below. Use the Pythagorean Theorem for the diagonal.

```
Enter the length of side 1 3
Enter the length of side 2 4

The area is: 12.0
The perimeter is: 14.0
The diagonal is: 5.0
```

32. Write a sales program with five (5) methods located in a separate module. The first method will obtain and return the price (double) of an item being purchased. The second method will obtain and return the quantity (integer) of the item being purchased. The third method will compute and return the total price (double) for the items. The fourth method will compute and return the tax

amount (double) at 7% (0.07) for the purchase. The fifth method will display all of the information as shown below.

```
Enter the price of the item 12.34
Enter the number of items 2

Price       $12.34
Quantity         2
Subtotal    $24.68
Sales tax   $ 1.73
------------------
Total Sale $26.41
```

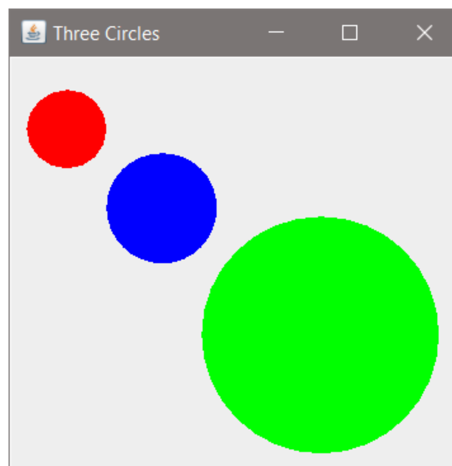
33. Write a program that requests the name of the user using an input dialog box, and displays “Hello “ and the name entered using a message dialog box.

34. Write a program that creates a frame (300 x 300) and places three circles on the frame as shown below. One circle is red and 50 pixels, one is blue and 70 pixels, and the one is green and 150 pixels. Locate them similar to the image shown. Remember to include the following import statements.

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;
```

To add the title to the frame use:

```
frame.setTitle("Three Circles");
```



Chapter 5 Programming Challenges

#1 Meteor Evacuation Status Simulation

Design and develop a program that determines the evacuation status for a city based upon the size and distance of a meteor coming toward the city. The program will accept a meteor size in meters and a distance from the city in miles, and compute and display the meteor data and evacuation status. Allow the user to enter another set of data without restarting the program.

Required five (5) methods located in a separate module:

- Method #1 - Prompt for and obtain, validate, and return the user input of the meteor size in meters (must be > 0.0 and < 10.0),
- Method #2 - Prompt for and obtain, validate, and return the distance of the meteor in miles (must be > 0.0 and < 500)
- Method #3 - Compute and return the meteor's speed (120 mph * size)
- Method #4 - Compute and return the time to impact (distance/speed) in minutes
- Method #5 - Determine and return the evacuation status as a String for the city based on the criteria below.

Display the data from main as shown below.

Evacuation Status Criteria:

If the time to impact < 45 minutes, then Evacuation CANNOT BE COMPLETED
If time to impact > 45 and <= 90 minutes, then Evacuation is POSSIBLE
If the meteor time to impact is > 90, then Evacuation is PROBABLE

Note that speed is in mph, but time to impact is in minutes.

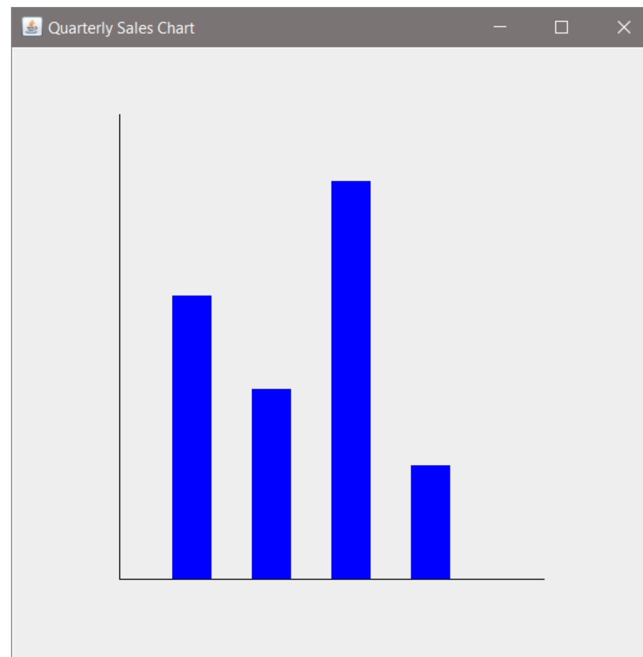
```
Enter the meteor size in meters: 3
Enter the meteor distance in miles: 400
      Meteor Data:

      Diameter in meters:    3.0
      Distance in miles:  400.00
      Speed in mph:       360.00
      Minutes to impact:   66.67
      Evacuation Status: Evacuation is POSSIBLE
```

#2 Quarterly Sales Bar Chart

Design and develop a program that accepts four sales figures in whole dollar amounts (integers) from the user and graphs the amounts using vertical bars in a frame (500 x 500). The bars should be 30 pixels wide and 30 pixels apart, and scaled so that the largest amount entered is represented with a bar that has a vertical size of 300 pixels. Add the frame title and lines as shown in the example below.

Use a method to determine the largest value for scaling the values. Drawing can be handled inside the paintComponent method.

```
// x, y is the top-left corner of the rectangle
fillRect(x, y, width, height)
```




```
Enter 1st quarter sales 245
Enter 2nd quarter sales 165
Enter 3rd quarter sales 345
Enter 4th quarter sales 98
```

Chapter 6

Arrays and ArrayLists

A variable can store a single value, but it is often necessary to use multiple values. Consider a program that obtains 10 values and determines the lowest and highest numbers. Ten variables would be needed to hold the ten values. Another solution would be to use an ***array*** which can store multiple values of the same data type. An array is an abstract data type that is defined by the programmer. The declaration can be a two-step process where the first step declares an array reference, and the second creates an array and assigns it to the reference. The line below declares an array reference to an array of integers, but it cannot store any values. The square brackets indicate that it references an array.

```
int [] valuesArray;
```

The next step creates an array (allocates memory for storage) and assigns the memory address of the array to the valuesArray reference variable. The number 20 in the square brackets indicates the size of the array. Memory for 20 integers will be allocated, and the array will be able to hold 20 integers.

```
valuesArray = new int[20];
```

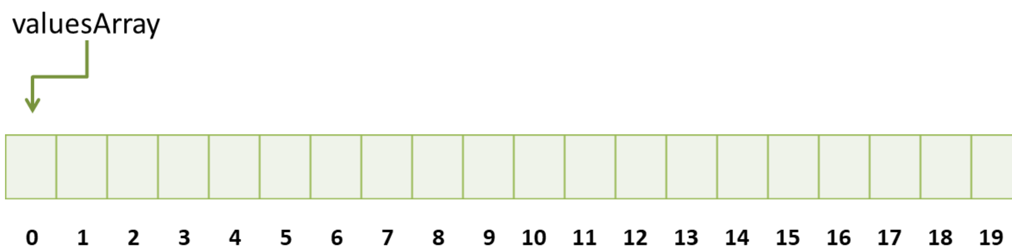
The two steps can be combined into a single statement as shown here.

```
int[] valuesArray = new int[20];
```

The computer will allocate enough memory to hold 20 integers in the example. Integers are stored using 4 bytes, so 80 bytes of adjacent memory would be allocated for an array that stores 20 integers. An array of 20 doubles would use 160 bytes since doubles are typically stored using 8 bytes of memory.

Array Indexes

The elements in an array are stored at indexes in the array in adjacent memory locations. The indexes are numbered zero through the size of the array minus one the same way that String characters are numbered. Notice that the name of the array (valuesArray) references the zero index of the array.



Array Indexes – valuesArray[20]

Technical Notes

The terminology associated with the array indexes varies. They are sometimes referred to as *slots* or *subscripts*. The use of the term *index* is common and is implied when using "i" in loops that access array elements.

To store a value in the array, the index must be used. This statement assigns 12 to the first index in the array.

```
valuesArray[0] = 12;
```

Loops are typically used to access the indexes in an array. The following loop assigns 33 to each index in the valuesArray array. Note that "i" controls the

loop and is also used as the index for the array. The conditional expression stops the loop at 20 since 19 is the last index in the array.

```
for(int i = 0; i < 20; i++) {
    valuesArray[i] = 33;
}
```

33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33

An array can contain other data types including Strings, doubles, and even arrays. The statements below declare an array of 100 doubles and an array of 60 Strings.

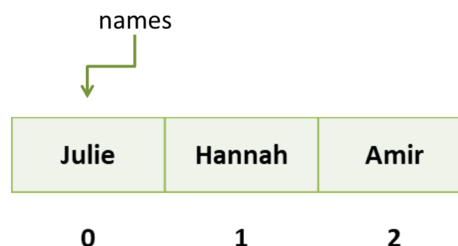
```
double[] amounts = new double[100];

String[] names = new String[60];
```

A variable can be used for the size of an array, but it must be an integer. Here a constant is declared and used as the size (number of indexes) for the array.

```
final int SALES_ITEMS = 16;

int[] sales = new int[SALES_ITEMS];
```

Arrays can also be initialized with an assignment statement when they are declared as shown below. The size of the array will automatically be the number of items assigned. The indexes will be zero through the size minus one.

```
String[] names = {"Julie", "Hannah", "Amir"};
```



Example Ex. 6.1 below initializes an array and displays the elements. The program also uses the ***length*** attribute of an array, and uses it to control the loop. Notice that the loop condition used is less than the length (indexes begin at zero).

Ex. 6.1 – Initializing an Array, Displaying the Contents, and Using *length()*

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

    int size;
    double[] sales = {10.45, 2.99, 4.76, 3.89};

    size = sales.length;

    System.out.println("There are " + size + " elements.");

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

        System.out.println(sales[i]);
    }
}
```

```
There are 4 elements.
10.45
2.99
4.76
3.89
```

Program Output

Arrays are fixed-size containers. That is, the size or number of elements that an array can hold cannot be changed once it is declared. This is one of the limitations of arrays since a programmer may not know exactly how many elements an array will need to store when a user runs the program. One solution is to ask the user how many items will be entered and declare an array of that size. However, if the user enters the wrong number, the array will be too large or too small and the program will not run correctly.

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

System.out.println("How many sales entries?");
numItems = in.nextInt();

double[] sales = new double[numItems];
```

Another solution is to declare an array size larger than what could be needed, and keep track of the number of indexes used. As an example, if a payroll program is written to process data for 42 employees, but the number of employees may vary (the company may be hiring), an array of size 50 might be

declared for the program. When the array is populated with data, only the needed indexes will be populated. The other indexes would have invalid data in them. This is referred to as a partially filled array, and the algorithm requires counting the number of indexes that are populated with valid data. The count would then be used to stop any loop in the program so that it only accesses indexes with valid data.

Technical Notes

A common error associated with arrays is an off-by-one error. A loop iterates one too many or one too few times while accessing the array. One too few will show up in the output, but if an index beyond the array boundary is used in a program, it will cause an error and the program will terminate. The ArrayList covered later is a similar container that will simplify many tasks associated with arrays.

Copying Arrays

When copying an array, an assignment statement would only copy the reference, not the array contents. The statements below would copy the reference.

```
int[] grades = {90, 95, 89};
int[] points = grades;
```

There are two ways to copy and array. One is to visit each index from one array and copy the contents to the corresponding index of the second array. Another solution is to use the ***copyOf()*** method which accepts two arguments. The first is the name of the array to copy and the second is the number of elements in the second array. This means that the second array does not have to be the same size as the one being copied. It can be larger to accommodate additional data. As an example, to double the size of an array the second argument could be two times the length of the first. The statements below create a copy of the array named grades to tempGrades which is twice the size of grades. The second line assigns the reference of the new array to grades. Since it is a copy, all of the data from the original grades array is in the new array and it can now hold additional data.

```
int [] tempGrades = Arrays.copyOf(grades, 2 * grades.length);

grades = tempGrades;
```

Passing Arrays to Methods

When passing an array to a method, the name of the array is the argument passed to the method. In the receiving method, the square brackets are used in the method header for the parameter. Since the array name is a reference to the location in memory, the method has access to the actual memory locations and can change the items stored in the array. Recall that the array name is a reference to the zero index of the array. The computer can locate the other indexes since they are in adjacent memory.

Ex. 6.2 – Passing Arrays to Methods

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

    int [] numArray = new int[6];

    for (int i = 0; i < 6; i++) {
        numArray[i] = i;
    }

    squareThem(numArray);
}

public static void squareThem(int[] nums) {

    int square;

    for(int i = 0; i < nums.length; i++) {
        square = nums[i] * nums[i];
        System.out.println(square);
    }
}
```

A method can also return an array as shown below. Note the return type in the method header. The size of the array is passed to the method, and the method creates an array of that size. The array is populated, and the method returns a reference to the array.

```
public static int [] createNumberArray(int size) {

    int [] tempArray = new int[size];

    for(int i = 0; i < size; i++) {
        tempArray[i] = i;
    }

    return tempArray;
}
```

The method above would be called with an assignment statement to receive the reference to the array.

```
int [] numArray = createNumberArray(20);
```

Two-dimensional Arrays

The examples so far have dealt with one-dimensional arrays, but arrays can be two, three, or more dimensions. Just as a spreadsheet has rows and columns, a two-dimensional array can be used for storing related data or to represent a matrix. When declaring a two-dimensional array, there are two sets of brackets and two size declarators. The following statement declares a two-dimensional array named table that has 4 rows and 3 columns.

```
int [][] table = new int[4][3];
```

table[0][0]	table[0][1]	table[0][2]
table[1][0]	table[1][1]	table[1][2]
table[2][0]	table[2][1]	table[2][2]
table[3][0]	table[3][1]	table[3][2]

Two-dimensional Array Indexes

Notice that both dimension indexes begin at zero, and to assign a value, two indexes are required. The following statement assigns a value to the second row, second column of the array.

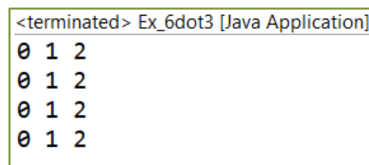
```
table[1][1] = 35;
```

table[0][0]	table[0][1]	table[0][2]
table[1][0]	35	table[1][2]
table[2][0]	table[2][1]	table[2][2]
table[3][0]	table[3][1]	table[3][2]

With a two-dimensional array, populating the array or accessing the data in the array requires a nested loop to increment one dimension in the outer loop and one dimension in the inner loop. Example Ex. 6.3 populates the table array using constants named ROWS and COLS for clarity. The array is populated with the value of "j" as the inner loop executes. The nested loop for output adds a space after each element and a line feed each time that the inner loop completes.

Ex. 6.3 – Populating a Two-dimensional Array

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

    final int ROWS = 4;
    final int COLS = 3;
    int[][] table = new int[ROWS][COLS];

    for(int i = 0; i < ROWS; i++) {

        for(int j = 0; j < COLS; j++) {
            table[i][j] = j;
        }
    }

    for(int i = 0; i < ROWS; i++) {

        for(int j = 0; j < COLS; j++) {
            System.out.print(table[i][j] + " ");
        }
        System.out.println();
    }
}
```


```
<terminated> Ex_6dot3 [Java Application]
0 1 2
0 1 2
0 1 2
0 1 2
```

Program Output

A modification to the output statement displays the row and column numbers as the loops iterate.

```
for(int i = 0; i < ROWS; i++) {

    for(int j = 0; j < COLS; j++) {
        System.out.print("Row " + i + " Column " + j + "\t");
    }
    System.out.println();
}
```

```
<terminated> Ex_6dot3 [Java Application]
Row 0 Column 0  Row 0 Column 1  Row 0 Column 2
Row 1 Column 0  Row 1 Column 1  Row 1 Column 2
Row 2 Column 0  Row 2 Column 1  Row 2 Column 2
Row 3 Column 0  Row 3 Column 1  Row 3 Column 2
```

To display by column and row, the algorithm is modified so that the outer loop uses the columns and the inner loop uses the rows. The order of the variables changes in the output statement as well.

```
for(int i = 0; i < COLS; i++) {

    for(int j = 0; j < ROWS; j++) {
        System.out.print("Row " + j + " Column " + i + "\t");
    }
    System.out.println();
}
```

```
<terminated> Ex_6dot3 [Java Application]

Row 0 Column 0  Row 1 Column 0  Row 2 Column 0  Row 3 Column 0
Row 0 Column 1  Row 1 Column 1  Row 2 Column 1  Row 3 Column 1
Row 0 Column 2  Row 1 Column 2  Row 2 Column 2  Row 3 Column 2
```

A two-dimensional array can also be initialized when it is declared. Braces surround the initialization and the rows of data. Commas separate the rows of data, and a semicolon follows the closing brace. The dimensions aren't necessary within the brackets as shown because of the initialization.

```
int [][] keys = {
                    { 7, 8, 9 },
                    { 4, 5, 6 },
                    { 1, 2, 3 }
                };
```

7	8	9
4	5	6
1	2	3

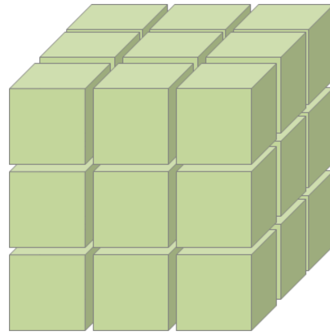
When passing a two-dimensional array to a method, the name of the array is passed, but both sets of brackets are required in the receiving method header.

```
public void displayArray(int[][] arr)
```

Multi-dimensional Arrays

Arrays can be declared with more than two dimensions. Although a four- or five-dimensional array might be difficult to picture and complex to implement, a three-dimensional array can be seen as a cube with a height, width, and depth. A three-dimensional array requires three sets of brackets as shown below.

```
final int HEIGHT = 3;
final int WIDTH = 3;
final int DEPTH = 3;

int[][][] cube = new int[HEIGHT][WIDTH][DEPTH];
```



Accessing the elements in a three-dimensional array would require an additional nested loop within the other two.

```
for(int i = 0; i < HEIGHT; i++) {
    for(int j = 0; j <  WIDTH; j++) {
        for(int k = 0; k < DEPTH; k++) {
            System.out.print(cube[i][j][k]);
        }
    }
}
```

Technical Notes

Programmers commonly use "i" as the temporary variable with loops especially when indexing an array. When a second variable is needed for a nested loop, "j" is commonly used. With a three-level loop, as in the case of a three-dimensional array, a "k" would be used for the third variable.

Array Algorithms

Common array algorithms include populating an array and sorting an array. Examples have already shown populating an array. To sort an array, the method **Arrays.*sort()*** is provided in the java.util.Arrays class. The following statement would sort the `values` array.

```
Arrays.sort(values);
```

To sort only a portion of the array, the indexes are passed as well as the name of the array.

```
Arrays.sort(arrayName, fromIndex, toIndex);
```

Other array algorithms include finding a value, computing the sum and average, finding the minimum or maximum value in an array, and adding or removing an element. Each of these algorithms requires visiting each index of the array. The ***enhanced for loop*** will access each element and provide a copy of the value in each index. In example Ex. 6.4, the loop declares a temporary variable `num` which receives a copy of each value in the `numbers` array and adds that value to the variable `sum`. Note that the loop is controlled by the enhanced for loop and will end when the end of the array is reached.

Ex. 6.4 – Enhanced for-loop

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

    int [] numbers = { 1, 2, 3, 4, 5};
    int sum = 0;

    for(int num : numbers) {

        sum = sum + num;
    }

    System.out.print("The total is : " + sum);
}
```

```
<terminated> Ex_6dot4 [Java Application]
The total is : 15
```

Program Output

Modifying this program to compute the average simply requires dividing the sum by the number of elements in the array using the length attribute.

```
System.out.print("The total is : " + sum);

average = sum / numbers.length;
System.out.print("Average is: " + average);
```

Finding the maximum or minimum value in an array requires traversing the indexes while comparing the values. The following example sets a variable for the minimum value to the first element in the array and then compares the remaining values to determine the minimum. Finding the maximum value would require a minor modification to the code.

Ex. 6.5 – Finding the Minimum Value

```
int [] numbers = { 1, 2, 3, 4, 5};

int min = numbers[0];

for(int i = 1; i < numbers.length; i++) {

    if(numbers[i] < min) {
        min = numbers[i];
    }
}

System.out.print("The minimum value is : " + min);
```

There are several ways to search for a value in an array. One routine is called ***Linear Search*** which inspects each element in the array until the value is found or until the end of the array is reached. This can be inefficient with large arrays especially when the value is not found. A while loop with a logical AND can stop the iterations when the value is found.

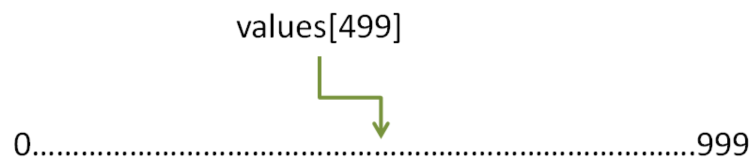
```
boolean found = false;
int i = 0;

while(i < numbers.length && found == false) {

    if(numbers[i] == findNum) {
        found = true;
    }
    i++;
}
```

Notice how similar this code is to the code for finding the minimum value above.

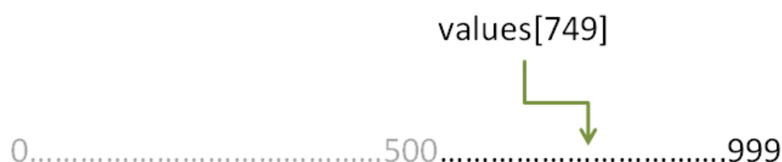
Ex. 6.6 – Linear Search

```
int [] numbers = { 1, 2, 3, 4, 5};

int findNum = 4;
int i = 0;
boolean found = false;

while(i < numbers.length && found == false) {

    if(numbers[i] == findNum) {
        found = true;
    }
    i++;
}
if(found) {
    System.out.print("The value was found.");
}
```

For larger arrays that are sorted, a search algorithm called ***Binary Search*** is more efficient. With Binary Search, the array is divided in half as elements are compared to the desired value. Consider a sorted array named `values` of size 1000 that is populated with the numbers 0 to 999, and the task of determining if the number 567 is in the array. Binary Search would access index [499] to see if it contains the desired number. If it does, the search ends. If it does not, the algorithm determines if it is smaller or larger than the desired number.



Since `values[499]` contains 499, it is less than the desired number and since the array is sorted, the desired number cannot be in the lower half of the array. The lower half of the array can be eliminated from the search. The algorithm then accesses the center index of the upper portion of the array to compare with the desired number.



Again, if it contains the desired number the search ends. If it does not, it is determined whether it is smaller or larger than the desired number. Recall that the example is searching for 567, which is less than 749 so the algorithm continues.

Note that the array is not affected by the algorithm. It is the middle, lower, and higher indexes being used for the search that are changing each time as shown in the example below.

```
int [] values = new int [1000];

int findNum = 567;

boolean found = false;
int lowIndex = 0;                       // starting lower index
int highIndex = values.length - 1;      // starting high index
int midIndex = 0;                       // midpoint set in loop

while( lowIndex <= highIndex && found == false) {

    midIndex = (lowIndex + highIndex) / 2;

    if(values[midIndex] == findNum) {
        found = true;
    }
    else if(values[midIndex] < findNum) {
        lowIndex = midIndex + 1;
    }
    else {
        highIndex = midIndex - 1;
    }
}
```

When using arrays and array algorithms, modifications are often needed to produce the desired result, and their limitations need to be considered in the design. Consider that Linear Search ends when the desired value is found, but there may be duplicates of that value in the array. Also, an array is a fixed size container, so adding or removing elements requires careful design as well. The ArrayList class and the methods that are provided eliminate some of the issues associated with using arrays.

The ArrayList

The ArrayList class in Java is similar to an array and allows storing multiple items of the same data type, but an ArrayList automatically expands as items are added to it, and will shrink in size if items are removed. 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. These features make the ArrayList easier to use than an array. To use an ArrayList in a program, the following import statement is required.

```
import java.utils.ArrayList;
```

To declare an ArrayList, angled brackets surround the data type. The example below declares an ArrayList of Strings named myList on the first line and creates an ArrayList object on the second. The size of an ArrayList is not required.

```
ArrayList<String> myList;

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

The two-step process can be handled with one statement as shown below.

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

Items are added to an ArrayList using the ***add()*** method. In Ex. 6.7, three names are added to the ArrayList myList. The items are added in order to the end of the ArrayList, and the enhanced for loop displays the contents of the ArrayList.

Ex. 6.7 – Adding Items to an ArrayList

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

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

    myList.add("Betty");
    myList.add("James");
    myList.add("Devon");

    System.out.println("Initial ArrayList");

    for(String name : myList)
        System.out.println(name);
}
```

```
<terminated> Ex_6dot7 [Java Application]
Initial ArrayList
Betty
James
Devon
```

Program Output

To insert an item at a specific index, the *add()* method is passed the index as an argument before the item to be added. The statement below adds "Allison" to index[0] and shifts the indexes of the other items in the ArrayList.

```
myList.add(0, "Allison");
```

There are several other methods and features of ArrayLists shown in Ex. 6.8 including the use of the ***size()*** method to control the loop, and the use of the ***get()*** method to access the items in the ArrayList.

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

myList.add("Betty");
myList.add("James");
myList.add("Devon");

myList.add(0, "Allison");

for(int i = 0; i < myList.size(); i++) {
    System.out.print("Index " + i + " contains: ");
    System.out.println(myList.get(i));
}
```

```
<terminated> Ex_6dot7 [Java Application]
Index 0 contains: Allison
Index 1 contains: Betty
Index 2 contains: James
Index 3 contains: Devon
```

Program Output

When removing an item, the index for the item to remove is needed. The following statement would remove Betty from the example ArrayList.

```
myList.remove(1);
```

The algorithm for removing an item would include finding the index for the item. Example Ex. 6.8 below includes both operations.

Technical Notes

When using a loop to locate an item and remove it, the design of the loop must consider that the indexes are shifted after the removal of the item. If the item occurs twice, the loop could inadvertently skip the second occurrence when the index variable is incremented.

The algorithms to find an item, the minimum, and the maximum in an ArrayList are similar to those for an array except for the ArrayList methods and syntax. Example Ex. 6.8 locates and removes "Allison" from an ArrayList named myList.

Ex. 6.8 – Finding and Removing an Item

```
int i = 0;
boolean found = false;
String name = "Allison";

while(i < myList.size() && found == false) {

    if(name.equals(myList.get(i))) {
        found = true;
        myList.remove(i);
    }
    i++;
}
```

To replace an item in the ArrayList, the ***set()*** method is used. The index is passed as an argument along with the item, and the item replaces the item at the index. Before and after graphics of the ArrayList are shown below for the following statements.

```
myList.add("Allison");
myList.add("James");
myList.add("Devon");

myList.set(1, "Pavin");
```

Before	After
Allison	Allison
James	Pavin
Devon	Devon

Primitive Data Types and Wrapper Classes

The primitive data types cannot be used with ArrayLists. Instead, the Wrapper classes are used. Wrapper class names begin with uppercase letters, and Integer and Character are spelled out. Conversion between primitive types and wrapper classes is automatic (called 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

Table 6.1 – Wrapper Classes

For an ArrayList of floating-point numbers, the wrapper class "Double" is used in place of "double" within the angled brackets for the ArrayList. The methods covered previously are used to access, add, remove, and replace items when using the wrapper classes.

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

When passing ArrayLists to methods, the name is passed and the parameter for the method is an ArrayList and the data type as shown here.

Ex. 6.9 – Method Receiving an ArrayList

```
public static void displayList(ArrayList<Double> list) {

    for(double num : list) {
        System.out.println(num);
    }
}
```

A Complete Example – Random ArrayList

Requirements:

Write a program that creates an ArrayList of 20 random integers between 0 and 100 inclusive. Call a method that removes the lowest and highest numbers, and then call a method to compute and return the average. Then display the average from the main method.

Program Pseudocode:

Step 1 Create the ArrayList

Step 2 Populate the list with random numbers

Step 3 Remove the highest and lowest numbers

- Consider how these could be easily found

Step 4 Call a method to determine and return the average

Step 5 Display the average

Design

Considering the requirements, removal of the lowest and highest values is simplified if the list is sorted first. The lowest would be at index 0 and the highest would be at index[size – 1]. The ***sort()*** method for an ArrayList requires the import statement below and is preceded by Collections.

```
import java.util.Collections;
```

The decision regarding where to sort the list is subjective. It could be done in main prior to the list being passed to the method that will remove the lowest and highest numbers, or in the method itself. In the example, it will be performed in the method.

Development

The development of the program follows the pseudocode step-by-step, and includes the design consideration. The ArrayList is declared and created, and then populated. The list is then passed to the method that will sort the list and remove the lowest and highest values. Next, the list is passed to the method which computes and returns the average, and then the main method will display the average.

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

    ArrayList<Integer> randList = new ArrayList<Integer>();
    int rand = 0, average = 0;

    for(int i = 0; i < 20; i++) {
        rand = (int) (Math.random() * 100);
        randList.add(rand);
    }

    removeMinAndMax(randList);

    average = getAverage(randList);

    System.out.println("The average is: " + average);
}

public static void removeMinAndMax(ArrayList<Integer> list) {

    Collections.sort(list);
    list.remove(0);
    list.remove(list.size() - 1);
}

public static int getAverage(ArrayList<Integer> list) {

    int total = 0, avg;

    for(int num : list) {
        total = total + num;
    }

    avg = total / list.size();

    return avg;
}
```

Testing and Debugging

The development isn't complete until the program is tested and verified for accuracy. To determine that the program is performing as designed requires actually seeing the numbers generated and manually checking the output. To do this, output statements can be added that will display the original list, show which numbers were removed as the lowest and highest, and the resulting list for comparison. Displaying the list at each interval would provide this information and a display list method can be written quickly. The average will be calculated manually.

```
public static void displayList(ArrayList<Integer> list) {

    for(int num : list) {
        System.out.print(num + " ");
    }
    System.out.println();
}
```

The display method will be called prior to and after the method that removes the lowest and highest numbers.

```
displayList(randList);

removeMinAndMax(randList);

displayList(randList);
```

When the program runs, the output can be compared. The lowest number in the first row is 3, and 92 is the highest. The second row shows that they have been removed. The sum of the numbers in the second row is 848 and there are 18 numbers so 47 is the average (declared as int).

```
<terminated> Ex_6_complete [Java Application]
70 23 46 51 89 79 34 3 51 62 7 55 31 16 78 60 12 92 53 31
7 12 16 23 31 31 34 46 51 51 53 55 60 62 70 78 79 89
The average is: 47
```

Other designs and implementations can be used without sorting the ArrayList. Below is an algorithm to determine and remove the lowest number by locating the index. Removing the highest would be similar.

```
int min = numList.get(0);
int index = 0;

for(int i = 1; i < numList.size(); i++) {

    if(numList.get(i) < min) {
        index = i;
        min = numList.get(i);
    }
}

numList.remove(index);
```

Arrays vs ArrayLists

Although any solution using an array could be implemented using an ArrayList, arrays are common in programming. When the number of elements is a fixed size, an array is efficient. For all other implementations, an ArrayList is usually preferred because it is considered to be easier to use.

Array and ArrayList Boundaries

Java will not allow an index to be used that is negative or beyond the range for an array or ArrayList. This is referred to as ***Bounds Checking*** and occurs at runtime not compile time. The compiler will not display an error, but when the program runs an exception will be thrown and the program will terminate. In the example below, the loop condition tries to access beyond the ArrayList boundary. Note the error message.

```
ArrayList<Integer> numList = new ArrayList<Integer>();

numList.add(11);
numList.add(12);
numList.add(13);
numList.add(14);

for(int i = 0; i < 5; i++) {
    System.out.println(numList.get(i) + " ");
}
```

```
11
12
13
14
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 4 out-of-bounds for length 4
```

Programming Style and Standards

Since arrays and ArrayLists are passed to methods using their names, it is clearer what is being passed if the name indicates that it is a collection of items. Adding a word such as array or list to the name increases readability.

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

Chapter 6 Review Questions

1. An array declared with a size [10] can hold _________ elements.
2. The items stored in an array must be the ________ data type.
3. Once declared, an array (can/cannot) _________ change size.
4. The _________ attribute of an array is the size of the array.
5. The ___________ method is used to copy the contents of an array.
6. When an array is passed to a method, the array _______ is passed.
7. A two-dimensional array requires ________ subscripts/indexes.
8. An ArrayList (can/cannot) ________ grow and shrink as needed.
9. ArrayLists can store primitive data types using the __________ classes.

Chapter 6 Short Answer Exercises

10. What data type can be stored in the array declared in this statement?

```
int [] values = new int [20];
```

11. How many items can be stored in the array declared by this statement?

```
int [] values = new int [30];
```

12. What value is assigned to x by the statements below?

```
int [] numbers = {34, 35, 36, 37};
x = numbers[1];
```

13. Rewrite the loop below using the enhanced for loop.

```
for(int i = 0; i < array.length; i++) {
        System.out.print(array[i]);
}
```

14. Write a method header for a void method called displayAll that receives an array of Strings.

15. Write a method called arraySum that receives an array of integers and returns the sum of the values in the array. Do not use the enhanced for loop in the solution.

16. Rewrite the method in #15 above using the enhanced for loop.

17. Declare a two-dimensional array of integers named matrix with 3 rows and 5 columns.

18. Write a declaration for an ArrayList of Strings called names.

19. Write a statement that adds the name "Jennifer" to the ArrayList declared in #18 above.

20. Write a statement that removes "Dillon" from an ArrayList named contacts that contains "Allison", "Dillon", "Coleen", and "Olivia".

21. Write a declaration for an ArrayList of doubles called sales.

Chapter 6 Programming Exercises

22. Write a program that declares an array of integers named numbers. Using a loop, populate the array with the numbers 1 thru 20, then use an enhanced for loop to display the array contents separated by spaces.

23. Using the array in #22 above, display only the even numbers in the array.

24. Declare the two arrays below in a program and write a loop to add the numbers from the two arrays that occupy the same indexes and display the results.

```
int [] arr1 = { 1, 2, 3, 4, 5};

int [] arr2 = { 1, 2, 3, 5, 6};
```

25. Write a program that declares an ArrayList named values and populate the list with numbers below. Display the numbers, and then call a method named average that returns the average of the numbers and then display the average. Use the length attribute in the method for division and two decimal places in the output.

```
11.98    5.67    8.47    2.99    6.94
```

26. Using the ArrayList in #25 above, display the contents separated by spaces, sort the ArrayList using *Collections.sort()* (reference the Complete Example in the chapter), remove the lowest value using the index, and display the contents separated by spaces.

27. Write a program that populates a 3 x 3 two-dimensional array with the numbers 1 thru 9 and display the array by row and column as shown below using a tab between columns.

```
1	2	3
4	5	6
7	8	9
```

28. Write a program that populates an ArrayList with 20 random numbers from 1 to 6 inclusive, and call a method that displays the ArrayList separated by spaces and display number of sixes that occur in the ArrayList.

29. Write a program that creates an ArrayList named contacts that contains "Allison", "Dillon", and "Coleen". Write a display method that displays the ArrayList on a single line separated by commas as shown below (note there are no commas at the ends of the lines). Then complete the steps below.

    a. call the method to display the initial ArrayList
    b. add "Olivia" to the list and call the display method
    c. replace "Colleen" with "Reece", and call the display method
    d. remove "Dillon" and call the display method

```
Allison, Dillon, Colleen
Allison, Dillon, Colleen, Olivia
Allison, Dillon, Reece, Olivia
Allison, Reece, Olivia
```

30. Write a program that creates an ArrayList of integers named numbers and passes it to a method named *addEvens()* that populates it with the even numbers from 2 to 20. Then call a method named *addOdds()* that adds the odd numbers from 1 to 19 to the ArrayList. Then sort the ArrayList using *Collections.sort()* and call a method that displays the contents of the ArrayList separated by a space and colon as shown below.

```
1: 2: 3: 4: 5: 6: 7: 8: 9: 10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20
```

Chapter 6 Programming Challenges

#1 Sales Data ArrayList

Write a program that creates an ArrayList named sales populated with the sales data in the left column below, and create another ArrayList named cost populated with the cost data in the center column below. Call a method to display the data in columns including a third column with the profit for each pair (sale price minus cost). Include column headers for the data and the column totals as shown below.

Sale Price	Cost	Profit
3.55	3.27	
12.34	10.61	
2.67	2.46	
4.99	4.59	
15.95	13.72	
39.50	34.65	

#2 Theater Seating Program

Design and implement a computer program for a Theater that allows users to select a seat by row and column. The program will display the seating price in row column format with the Stage indication (see below). When a seat is selected the program will redisplay the theater seating replacing the price of the seat selected with an "X" to indicate that the seat has been sold. The program will not allow selected seats (those indicated with an X) to be selected. The program will end when all of the seats have been sold.

The program will have the following methods:

- A method to display the Theater seating
- A method that determines if the seat selected has already been sold
- A method to determine if the Theater is sold out

The main loop for the program can be in the main method.

The two-dimensional price array is provided below.

```
int[][] priceArray = { { 30, 40, 50, 50, 50, 50, 50, 50, 40, 30 },
                       { 20, 30, 30, 40, 50, 50, 40, 30, 30, 20 },
                       { 20, 20, 30, 30, 40, 40, 30, 30, 20, 20 },
                       { 10, 10, 20, 20, 20, 20, 20, 20, 10, 10 },
                       { 10, 10, 20, 20, 20, 20, 20, 20, 10, 10 },
                       { 10, 10, 20, 20, 20, 20, 20, 20, 10, 10 },
                       { 10, 10, 10, 10, 10, 10, 10, 10, 10, 10 },
                       { 10, 10, 10, 10, 10, 10, 10, 10, 10, 10 },
                       { 10, 10, 10, 10, 10, 10, 10, 10, 10, 10 } };
```

The program will display the theater with higher priced seating at the bottom, a line, and the word "Stage". Note that higher priced seating is near the stage and is row 1 to the user. After displaying the Theater seating, the program will prompt the user to select a row, and then a column for a seat. Note that row 1 is closest to the stage.

```
                    Available Theater Seating

 $10   $10   $10   $10   $10   $10   $10   $10   $10   $10
 $10   $10   $10   $10   $10   $10   $10   $10   $10   $10
 $10   $10   $10   $10   $10   $10   $10   $10   $10   $10
 $10   $10   $20   $20   $20   $20   $20   $20   $10   $10
 $10   $10   $20   $20   $20   $20   $20   $20   $10   $10
 $10   $10   $20   $20   $20   $20   $20   $20   $10   $10
 $20   $20   $30   $30   $40   $40   $30   $30   $20   $20
 $20   $30   $30   $40   $50   $50   $40   $30   $30   $20
 $30   $40   $50   $50   $50   $50   $50   $50   $40   $30

 ____________________________________________________________
                            STAGE

      Please Select a Theater Seat Row : 1

      Please Select a Theater Seat Column : 5
```

The program will acknowledge the selection and price in the output, and redisplay (refresh) the seating available showing the selected seat as "X" and prompt for the next user to select a seat. Sold seats cannot be selected again.

```
Thank you for choosing seat 1 : 5
Your ticket price is : $50

                    Available Theater Seating

 $10   $10   $10   $10   $10   $10   $10   $10   $10   $10
 $10   $10   $10   $10   $10   $10   $10   $10   $10   $10
 $10   $10   $10   $10   $10   $10   $10   $10   $10   $10
 $10   $10   $20   $20   $20   $20   $20   $20   $10   $10
 $10   $10   $20   $20   $20   $20   $20   $20   $10   $10
 $10   $10   $20   $20   $20   $20   $20   $20   $10   $10
 $20   $20   $30   $30   $40   $40   $30   $30   $20   $20
 $20   $30   $30   $40   $50   $50   $40   $30   $30   $20
 $30   $40   $50   $50    X    $50   $50   $50   $40   $30

 ____________________________________________________________
                            STAGE

      Please Select a Theater Seat Row :
```

If the user attempts to select a seat has already been sold, output that it is not available and prompt for another selection.

```
        Please Select a Theater Seat Row : 1

        Please Select a Theater Seat Column : 5

        ATTENTION: THAT SEAT HAS ALREADY BEEN SOLD

                Please choose another seat.

                Available Theater Seating

  $10   $10   $10   $10   $10   $10   $10   $10   $10   $10
  $10   $10   $10   $10   $10   $10   $10   $10   $10   $10
  $10   $10   $10   $10   $10   $10   $10   $10   $10   $10
  $10   $10   $20   $20   $20   $20   $20   $20   $10   $10
  $10   $10   $20   $20   $20   $20   $20   $20   $10   $10
  $10   $10   $20   $20   $20   $20   $20   $20   $10   $10
  $20   $20   $30   $30   $40   $40   $30   $30   $20   $20
  $20   $30   $30   $40   $50   $50   $40   $30   $30   $20
  $30   $40   $50   $50    X    $50   $50   $50   $40   $30

   _______________________________________________________
                          STAGE

        Please Select a Theater Seat Row :
```

If all seats are sold (i.e. all “X”s), display “Theater Sold Out”, display the theater seating, and do not accept input (end the program).

Chapter 7

File Operations and Exceptions

Recall that the data stored in RAM does not persist between runs of the program or when the computer is turned off. Data is saved on secondary storage in files which allow information to be stored until it is needed, changed when required, and deleted when no longer needed. All files have what is referred to as a ***file extension***. This is the three or four letters that follow the period in the file name. File extensions are used by most operating systems to associate an application with the file. When a file is double-clicked, the operating system determines the application to launch based upon the file's extension and the application that was used to open that type of file previously. For example, double-clicking a file named "song.mp3" will launch an audio player because the audio player application has been associated with the mp3 file extension. The example below has a "txt" file extension which is typical for text files which are usually opened with Notepad or Notes by the computer's operating system.

file name

some_data_file.txt

file extension

File Names and Extensions

Different file types are usually opened by different applications, although some applications like Notepad can open a variety of file types. Table 7.1 lists some common file extensions with descriptions.

Extension	Description
.docx	Microsoft Word document file
.exe	executable file
.html	web page file
.java	Java source code file
.jpg	JPEG image file
.mov	movie file
.mp3	audio file
.pdf	Adobe Portable Document File
.py	Python source code file
.zip	ZIP compressed archive

Table 7.1 – Common File Extensions

Files being read from are typically referred to as ***input files***, and files being written to as ***output files***. There are three steps to using a file in a computer program:

- the file is opened
- the file is processed (read from, written to, or both)
- the file is closed

File handling in Java uses the File class and two different classes depending upon whether reading from or writing to the file. For reading from files, a file object is created and assigned the actual name of the file that will be used. The name is in quotes and includes the file extension.

```
File inputFile = new File("input.txt");
```

The line above creates a file object named inputFile and assigns it the file *input.txt* (a text file containing some data). This is the only time that the actual file name (input.txt) is used in a program. It has been assigned to the file object inputFile which will be used in the program.

Reading from a File

When reading from a file, the Scanner class is used similar to getting input from the keyboard. Instead of *System.in*, the Scanner object is created using the file object name (not the actual file name) as shown below.

```
File inputFile = new File("input.txt");

Scanner inFile = new Scanner(inputFile);
```

Note that the two statements above can be combined into one.

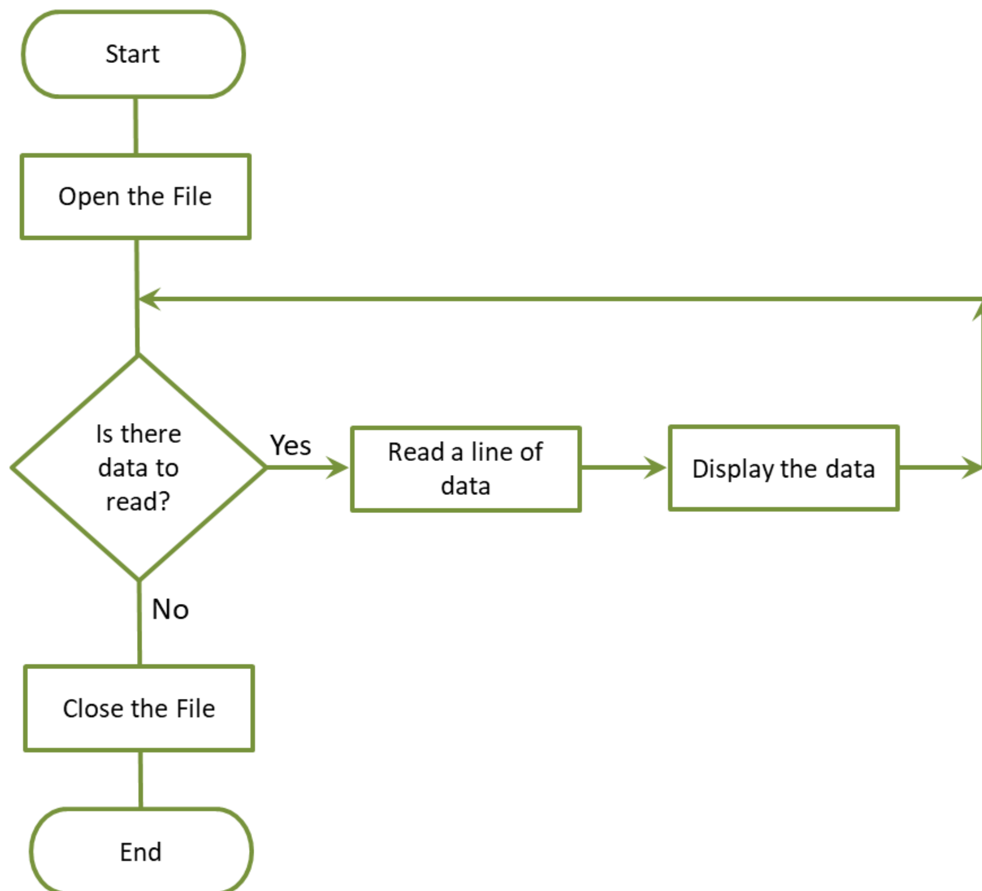
```
Scanner inFile = new Scanner(new File("input.txt"));
```

The Scanner methods used for keyboard input include the variations of *next()*, as well as the variations of *hasNext()*. Recall that *next()* consumes any leading white-space and reads until it encounters white space. The method *nextLine()* will read a complete line of text from a file including white space. The methods used for reading and handling data from a file are dependent upon the data format and operations being performed. The following code opens a file for reading named *input.txt*, and reads the lines in the file and displays them until the end-of-file is reached. The file is then closed.

```
File inputFile = new File("input.txt");

Scanner inFile = new Scanner(inputFile);

String someText;

while (inFile.hasNextLine()) {
    someText = inFile.nextLine();
    System.out.println(someText);
}

inFile.close();
```

Opening a File and Reading Text

In the example, the loop would read from the file until there are no more lines to read. A flowchart of the operation is shown below.



File Reading Flowchart

Technical Notes

When the file name is used, the program will search the default directory (which is where the program is running) to find the file. If the file is in another directory, the full path must be used. Since a single backslash in a literal string is the escape character, two backslashes are needed in the file path as shown here.

```
File inFile = new File("C:\\Data_files\\input.txt");
```

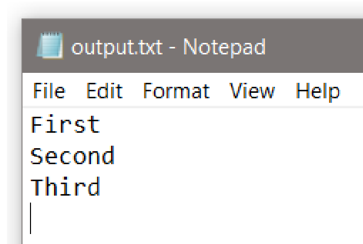
Writing to a File

The standard way to create and write to a file is with a ***PrintWriter*** which requires importing java.io.PrintWriter. The PrintWriter object is created and assigned the name of the file in quotes as shown below. If the file "output.txt" does not exist, it will be created (if possible). If the file exists, it will be emptied before writing (appending to a file is covered later).

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

The PrintWriter can use the System.out methods *print(), println(),* and *printf(),* and is closed in the same way that the Scanner is closed. The following code opens a file for writing named *output.txt*, and writes the three words on separate lines in the file using *println()*. The file is then closed.

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

outFile.println("First");
outFile.println("Second");
outFile.println("Third");

outFile.close();
```



Opening a File and Writing Text

Closing Files

Data being written to a file is queued in a ***buffer*** (a holding area in memory) for efficiency. If a program does not close a file, the operating system will eventually close it, but will not check the buffer first. Using the ***close()*** method ensures that anything in the write buffer is written to the file before it is closed.

File Not Found Exceptions

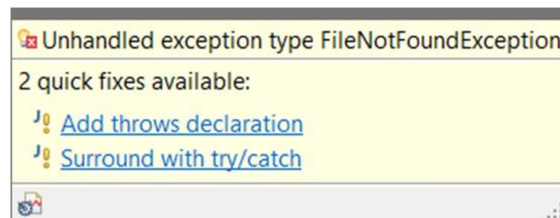
If a program attempts to open a file for reading and the file does not exist, the program will terminate and throw what is called an exception. To highlight this, the Eclipse IDE will show an error when the Scanner is declared.

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

The same issue exists when declaring a PrintWriter, since it will attempt to create the file if it does not exist and the user may not have permission to create a file on the system or there may not be enough space on the device to create one.

```
PrintWriter outFile = new PrintWriter("data.txt");
```

Hovering over the errors in the IDE will display an "Unhandled exception type FileNotFoundException", and two quick fixes.



The first of the two quick fixes is "Add throws declaration" which indicates the type of exception that might be thrown by a method. Adding it as shown below removes the error indicator but does not handle the issue.

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

    Scanner inFile = new Scanner (new File ("data.txt"));
    String text = inFile.nextLine();
    System.out.print(text);
    inFile.close();

}
```

Throws Declaration

The second of the "quick fixes available" is to "Surround with try/catch" which is the format for an exception handler in Java. A *try* block that includes statements that may throw an exception is followed by a *catch* block for handling the

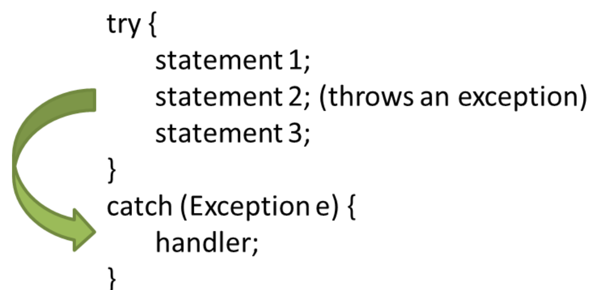
exception. In this case, a FileNotFoundException may be thrown. Clicking on the quick fix option will add the try/catch code or it can be entered manually.

The general format for a ***try/catch*** is shown here.

```
try {
    statement;
    statement;
}
catch (Exception Type e) {
    statement;
    statement;
}
```

Try/catch Exception Handling

The *try* block is entered and if a statement throws an exception, the *catch* block (handler) for that exception type is entered. Control is transferred to the catch block matching the exception thrown, the handler statement(s) executes, and the program continues. No other statements in the *try* block following the one that threw the exception will execute including closing a file. As shown below, if statement 2 throws an exception, the try block is exited, the catch block will execute, and statement 3 will never execute.



Technical Notes

Catch clauses are exception specific and are a way of handling issues that may arise without ending the program. A thrown exception that is not handled will halt execution of the program. There are two general types, and a variety of exceptions that can be thrown. Other exceptions are covered later in the chapter.

Example Ex. 7.1 adds the try/catch blocks and exception handlers for writing to a file and then reading from that file. Note that the *close()* methods are within the try block and will not execute if an exception is thrown. This will be addressed going forward.

Ex. 7.1 – File Writing and Reading with Exception Handling

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

    try {
        PrintWriter outFile = new PrintWriter("data.txt");
        outFile.println("Writing to a file.");
        outFile.close();
    }

    catch (FileNotFoundException e) {       // catch block
        System.out.print("Output file cannot be opened.");
        e.printStackTrace();
    }

    try {
        Scanner inFile = new Scanner (new File ("data.txt"));
        String text = inFile.nextLine();
        System.out.print(text);
        inFile.close();
    }
    catch (FileNotFoundException e) {       // catch block
        System.out.println("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. 7.1 when the input file is not found.

```
Input file cannot be opened.
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_7do1p.Ex_7dot1.main(Ex_7dot1.java:24)
```

Error Stack Trace

As mentioned, when a statement in the *try* clause throws an exception, no other statements in the *try* block following the one that threw the exception will execute including closing a file. The file cannot be closed in the *catch* block because of scope issues. The error message below is "inFile cannot be resolved" because it is declared within the try block and is not accessible in the *catch* block.

```
catch (FileNotFoundException e) {        // catch block
    System.out.println("Input file cannot be opened.");
    e.printStackTrace();
    inFile.close();
}
```

Unable to Use *close()* in the Catch Block

One way of executing clean-up statements is with a ***finally*** clause. The finally clause will execute whether an exception is thrown or not, and can be used for statements that must execute. However, the file cannot be closed in the finally clause since the file is declared within the *try* block and is out of scope. The general format is shown below.

```
try {
    statement;
    statement;
}
catch (Exception Type e) {
    statement;
    statement;
}
finally {
    statement;
    statement;
}
```

Try-catch-finally

An efficient way to ensure that resources used will be closed if an error occurs is to declare and instantiate the PrintWriter and Scanner resources within the try clause (after the word "try" and prior to the opening brace). This is called a ***try-with-resources*** statement. The resources will be closed automatically even if an

exception is thrown. Example Ex. 7.2 uses the try-with-resources statements to write a line of text to a file and then read and display the line.

Ex. 7.2 – File Writing and Reading Using *try-with-resources*

```
try (PrintWriter out = new PrintWriter("data.txt"))  {
    out.println("Writing to a file.");
}
catch (FileNotFoundException e) {
    System.out.print("Output file cannot be opened.");
}

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

Try-with-resources

Other Exceptions

There are many exceptions included in the Java Library that could be thrown including NumberFormatExceptions, FileNotFoundExceptions, IOExceptions and StringIndexOutOfBoundsExceptions. There are two types of exceptions: ***checked*** and ***unchecked***. A checked exception will be highlighted by the compiler and require a solution (try/catch or throws statement) before compiling. An unchecked exception occurs at runtime like accessing an out-of-bounds index or an illegal argument used by the program. The compiler will not highlight these. When an exception is thrown, the program will look for a catch clause that matches the specific exception thrown. If it does not find one, the program will terminate. Many exceptions are preventable with code depending on the operation. The line below could throw a NumberFormatException because *hasNextInt()* is not used to ensure that the String can be parsed to an integer.

```
String input = inFile.next();
int num = Integer.parseInt(input);
```

Instead of surrounding the code with a try/catch and writing an exception handler, the code can be rewritten to ensure that the data is an integer before attempting the assignment. The ***trim()*** String method can be used to remove any leading or trailing whitespace, and the ***isNumeric()*** method will return true or false and can be used as a conditional expression as shown here.

```
String input = inFile.next();

if(isNumeric(input)) {
    int num = Integer.parseInt(input);
}
```

A generic exception handler can also be written to catch any exceptions that are not specifically handled. The catch clause could then output the message that is contained in the exception as shown below. Note that this catch clause does not handle the exception; it simply displays the error information.

```
catch (Exception e) {
    System.out.println(e.getMessage());
}
```

Append an Existing File

When a file is opened by the PrintWriter, any data in the file 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 then the PrintWriter methods can be used for writing.

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

PrintWriter out = new PrintWriter(fwriter);

out.append("some text");
```

Reading and Writing Numeric Data

Numeric data is often stored in files, and since the Scanner is used for reading, *nextInt()* and *nextDouble()* can be used to read integers and doubles. However,

they may run into trouble if what is read is not a numeric value. Using *hasNextInt()* and *hasNextDouble()* will look ahead to ensure that the expected data type is read. Recall from Chapter 3 that *Integer.parseInt()* and *Double.parseDouble()* will convert the text to a numeric value if possible.

Example Ex. 7.3 below writes integers to a file on separate lines, then a String, and closes the file. The program then reads the integers using *hasNextInt()* which does not read the line feeds or the String. *(Note: Exception handling is omitted.)*

Ex. 7.3 – Writing and Reading Numeric Values from a File

```
PrintWriter outFile = new PrintWriter("numbers.txt");
int x = 4;

while(x < 20) {
    outFile.println(x);
    x = x + 4;
}

outFile.println("Finished Writing");
outFile.close();


Scanner inFile = new Scanner (new File ("numbers.txt"));
int num = 0;

while(inFile.hasNextInt()) {
    num = inFile.nextInt();
    System.out.println(num);
}

inFile.close();
```

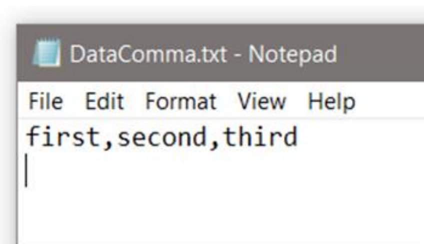
Program Output

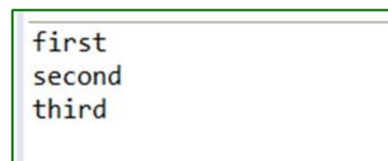
Delimiters

An earlier example read an entire line from a 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 encounters whitespace. A delimiter (data separator) can also be used for reading using the Scanner method. As an example, a comma delimited file could be read as shown below. A Scanner named *inFile* is declared, and it is assigned the delimiter (",") to use. The *next()* method will read until the delimiter and consume the delimiter just as it would a space or line feed. Note that the commas are not in the output.

```
Scanner inFile = new Scanner(new File ("dataComma.txt"));
String myString = "";
inFile.useDelimiter(",");

while(inFile.hasNext()) {
    myString = inFile.next();
    System.out.println(myString);
}
```





Data File Output

Reading Using a Delimiter

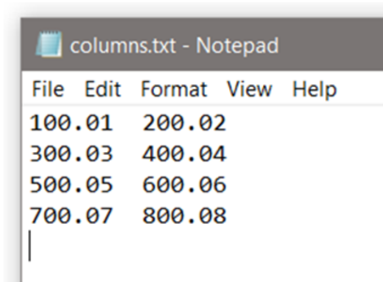
A delimiter does not have to be a character. In some files, the data may be tab or space delimited. The *next()* methods including *nextInt()* and *nextDouble()* will read until whitespace, and consume leading white space. Consider a data file that has two columns of data separated by a tab. The requirement is to read the two values on each line in the file and display their sum. The *nextLine()* method could be used to read the line containing the two values and then split them apart, but since *next()* will read until the tab, and then consume it, reading twice for each line simplifies the solution.

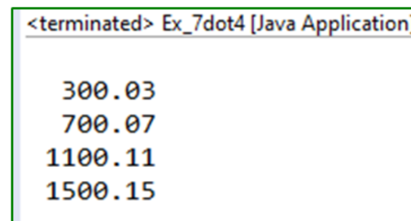
Example Ex. 7.4 reads two values from a tab delimited file, adds the values together, and displays their sum. Notice that the program uses try-with-

resources and a conditional statement before reading the second value to ensure that it is a double before reading.

Ex. 7.4 – Two-column Tab Delimited File Reading

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

    double num1 = 0, num2 = 0, sum = 0;

    try (Scanner inFile = new Scanner(new File("tabs.txt"))) {

        while (inFile.hasNextDouble()) {
            num1 = inFile.nextDouble();
            if(inFile.hasNextDouble()) {
                num2 = inFile.nextDouble();
            }
            sum = num1 + num2;
            System.out.printf("\n%8.2f", sum);
        }
    }
    catch (FileNotFoundException e) {
        System.out.println("The File was not found");
    }
}
```





Data File Output

Data Format

Designing the data format for file storage is an important task that effects program design and operation, data handling, and the scalability of the data and the program. When designing data files, the format, text/binary, delimiters, and any encryption would be considered. Many large-scale, data intensive programs require a formal **Data Dictionary** which is a file separate from the data that

contains the data descriptions, format, delimiters, the ordering of the data, and often additional information and comments. 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.

When a program stores and retrieves the information, the format for storing the data can accommodate the way that the program retrieves and uses the data. Consider the program that stores customer reservations for a restaurant in the complete example below.

A Complete Example – Customer Reservations

Requirements:

Write a program that retrieves and displays customer reservation information for the "Finest Dining" restaurant. The restaurant serves prix fixe (fixed price), four course meals for $75.00 per person. A deposit of 20% confirms the reservation, otherwise it is pending confirmation. The data will include the customer last name, contact phone number, number in the party, deposit amount, balance due, and the date and time of the reservation. The data is comma delimited.

Program Pseudocode:

While there is a line of data in the file to read

- Read a line of the data for a reservation
- Compute the balance due
- Display the information

Design

Since the data file format is known (shown below), the program will read each piece of data, compute the balance and display the information. The design would include parsing the deposit amount to determine the

reservation status and to compute the balance due. The data format when written to the file is shown below. Consider the different ways that the data could be written to the file.

```
Reservation date        mmddyy
Reservation time        hhmm
Customer name           string
Contact number          (ddd) ddd-dddd
Number in party         d
Deposit                 dd.dd
```

File data (space delimited):

```
Finest.txt - Notepad
File  Edit  Format  View  Help
05/10/22 07:00 Mahir (555)858-3753 4 60.00
05/10/22 07:30 Gaeff (555)868-4226 2 00.00
05/10/22 07:30 Anderson (555)876-7945 2 30.00
05/10/22 07:45 Daffney (555)776-1793 4 00.00
05/10/22 08:00 Scritney (555)868-1007 2 30.00
05/10/22 08:00 Eldridge (555)876-3086 2 30.00
```

Development

The development of the program follows the pseudocode and includes a Scanner for reading the data. The file is space delimited so *next()* will be used for reading. A loop will read the data and convert the values needed for computations and the output as follows:

> Number in party – convert to integer to determine total bill amount, deposit required, and balance due
>
> Deposit – convert to double to determine balance, reservation status, and balance due

The variables are declared and initialized.

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

    String date = "", time = "", name = "", contact = "";
    int partyNum = 0;
    double deposit = 0, balanceDue = 0, totalBill = 0;
```

The try block uses try-with-resources and the loop reads the file data and parses the integer and double.

```
try (Scanner inFile = new Scanner(new File("Finest.txt"))) {

    while (inFile.hasNext()) {
        date = inFile.next();
        time = inFile.next();
        name = inFile.next();
        contact = inFile.next();
        partyNum = Integer.parseInt(inFile.next());
        totalBill = partyNum * 75;
        deposit = Double.parseDouble(inFile.next());
        balanceDue = totalBill - deposit;
```

The output is formatted for display and the program is tested.

```
System.out.printf("\nDate%12s   Time%6s", date, time);
System.out.printf("\nName%12s   Contact%12s", name, contact);
System.out.printf("\nParty of%2s", partyNum);
System.out.printf("\nDeposit%8s%5.2f", "$", deposit);
System.out.printf("    Balance Due%3s%5.2f", "$",balanceDue);
```

A conditional expression determines the reservation status based on the deposit amount.

```
if(deposit >= totalBill * 0.2) {
    System.out.printf("\nReservation *** Confirmed ***\n\n");
}
else {
    System.out.printf("\nReservation - PENDING\n\n");
}
```

Testing and Debugging

Some testing would have been performed during development to ensure that the file was being opened and read. The loop would be developed incrementally with testing along the way. As an example, simply reading and displaying data proves that the development is on the right track.

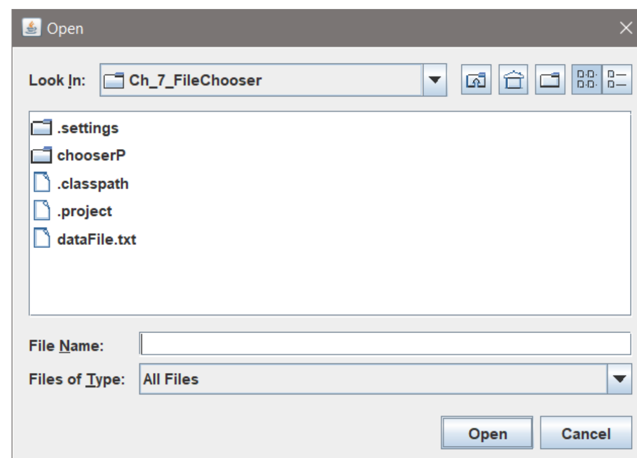
```
while (inFile.hasNext()) {
    date = inFile.next();
    System.out.printf("\nDate", date);
```

The program is tested and the output is checked against the file data to ensure that it is producing the correct output for each of the reservations.

```
Date     05/10/22   Time 07:00
Name         Mahir   Contact(555)858-3753
Party of 4
Deposit        $60.00    Balance Due  $240.00
Reservation *** Confirmed ***


Date     05/10/22   Time 07:30
Name         Gaeff   Contact(555)868-4226
Party of 2
Deposit        $ 0.00    Balance Due  $150.00
Reservation - PENDING
```

File Selection - JFileChooser

Java provides a swing component called the JFileChooser to select files. Rather than type a file name, the component is used to obtain the filename and path.



JFileChooser

The following lines of code create the file dialog shown above, and include line numbers for explanation. The program first obtains the directory the user is currently working in on line 10, and then creates a JFileChooser named chooser on line 12. Line 14 sets the directory for the file chooser to the working directory, and line 16 creates the file chooser window (null is used in place of a parent window for positioning). Line 16 also declares an integer to obtain the return value from the file chooser. Line 18 tests the return value from the chooser for

the APPROVE_OPTION (a file was selected), and line 20 obtains the file name including the path. Line 22 displays the path of the file selected for testing the example.

```
10 File workingDir = new File(System.getProperty("user.dir"));
11
12 JFileChooser chooser = new JFileChooser();
13
14 chooser.setCurrentDirectory(workingDir);
15
16 int status = chooser.showOpenDialog(null);
17
18 if(status == JFileChooser.APPROVE_OPTION) {
19
20     File file = chooser.getSelectedFile();
21
22     System.out.println(file);
23
24 } else {
25     System.out.println("Cancelled Operation");
26 }
```

To assign a Scanner in the program above, it is passed the file path assigned by the file chooser as shown here.

```
if(status == JFileChooser.APPROVE_OPTION) {
    File file = chooser.getSelectedFile();
    Scanner inFile = new Scanner(file);
```

To open the file using an application, use the Desktop, and open method.

```
Desktop dt = Desktop.getDesktop();
if(file.exists()) {
    dt.open(file);
}
```

Filtering Selectable Files

To filter the selectable files on a file type, a filter can be assigned to the JFileChooser. The following code creates a file chooser and then a filter which includes only .jpg file types. The filter is assigned to the JFileChooser using *setFilter*(). The only file types displayed for selection will have the .jpg extension.

```
JFileChooser chooser = new JFileChooser();
FileNameExtensionFilter f = new FileNameExtensionFilter("JPG", "jpg");

chooser.setFilter(f);
```

To save a file, the showSaveDialog() is used.

```
int status = chooser.showSaveDialog(null);
```

Programming Style and Standards

Data file naming recommendations include the use of all lowercase letters with underscores between words, and not using spaces or special characters in the name. Short names are considered best, but should adequately describe the contents of the file.

Exceptions should be handled and not squelched with catch blocks that simply catch the exception. A generic catch block (shown below) that omits the exception type and will catch any exception is acceptable if it is included last in the series. It should be considered a default catch block for testing and debugging purposes.

```
try {
    // Protected code
}
catch (ExceptionType1 e1) {
    // Catch block
}
catch (ExceptionType2 e2) {
    // Catch block
}
catch (ExceptionType3 e3) {
    // Catch bloc

catch (Exception e ) {
    e.printStackTrace();
}
```

Multiple exceptions can be caught by the same catch block as shown below, but they would have the same handler which would not be specific to either one.

```
catch (IOException | FileNotFoundException e ) {
    e.printStackTrace();
}
```

Programmers can also create their own exceptions by extending the Exception class.

```
class customException extends Exception {
}
```

Chapter 7 Review Questions

1. The characters that follow the name of a file are referred to as the file ______________.
2. A file opened for reading is referred to as an _____________ file.
3. A file opened for writing is referred to as an _________________ file.
4. In order to use a file in a program, the file must be ____________.
5. When a file is opened for writing, the data in the file is ____________.
6. For file handling, the directory where the program is running is referred to as the _________ directory.
7. The _________ object is used to read data from a file.
8. The _____________ object is used to write data to a file.
9. An area in memory where data to be written is temporarily stored is referred to as a ____________.
10. When a program is finished using a file, it should ___________ the file.
11. Adding to the end of a file's contents is referred to as _____________ to the file.
12. The _________________ class can be used to append data to an existing file.
13. A ______________ is a character used to mark the beginning or end of an item of data, or as a data separator.
14. When an error occurs because a file cannot be found for reading or created for writing, it is referred to as throwing a(n) ______________.
15. When a statement in a try block throws an exception, any remaining statements in the try block (will/will not) _________ execute.
16. When an exception occurs and there is no catch clause matching the exception thrown, the program will ______________.

Chapter 7 Short Answer Exercises

17. Write the statement(s) required to open a file named "dataFile.txt" for reading and associate it with the reference inFile.
18. Write the statements required to open a file named "dataFile.txt" for writing and associate it with the reference outFile.

19. Write the statements required to open a file named "dataFile.txt" for writing and associate it with the reference outFile, and write "This is a test." to the file.

20. Write the statements required to open a file named "dataFile.txt" for writing and associate it with the reference outFile, write the statement "This is a test." to the file, and close the file.

21. Write the statements to open a file named "numbers.txt" for writing that do not erase the existing data in the file. Associate the file with the name outFile.

22. Write the statements required to open a file named "data.txt" for reading in a try block, and the exception handler for a FileNotFoundException that displays "File not opened". Use try-with-resources in the statements.

23. Write the statements required for a try block to read the contents from a file named "names.txt" that has names on separate lines, display the contents on separate lines, and handle a FileNotFoundException.

Chapter 7 Programming Exercises

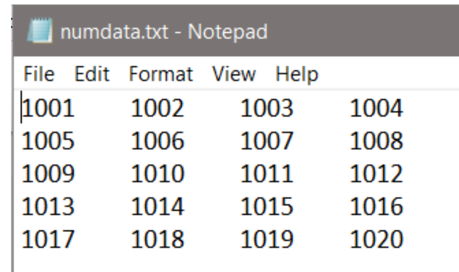
24. Write a program that creates a file for writing called "data.txt" and write the following lines to the file on separate lines and close the file.

    The first line
    The second line
    The third line
    The fourth line
    The fifth line
    The sixth line

25. Write a program that reads the "data.txt" file created from #24 above and display the contents of the file with a line number and colon as shown below.

```
1: The first line
2: The second line
3: The third line
4: The fourth line
5: The fifth line
6: The sixth line
```

26. Write a program that creates a text file named "num_data.txt" and writes the numbers 1001 thru 1020, separated by a tab with four numbers per line.



numdata.txt - Notepad

File Edit Format View Help

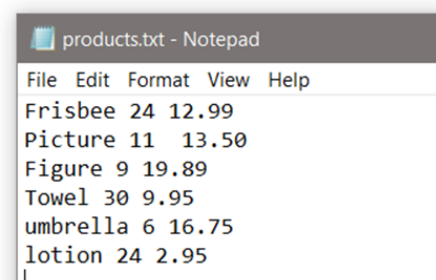
```
1001	1002	1003	1004
1005	1006	1007	1008
1009	1010	1011	1012
1013	1014	1015	1016
1017	1018	1019	1020
```

27. Create a text file named "sales_data.txt" with the sales data listed below each on a separate line. Write a program that reads one value from the file at a time, computes the discount price (20% off), and display the original and discount prices in two (2) columns separated by a tab as shown, with headers for the columns.

Sales data: 19.64, 3.56, 9.87, 16.33, 12.95, 6.50

```
Price             Discount
-------------------------
 19.64              15.71
  3.56               2.85
  9.87               7.90
 16.33              13.06
 12.95              10.36
  6.50               5.20
```

28. Create a text file named "products.txt" with the product names, units, and prices separated by a space as shown below left. Write a program that reads the data from the file and displays the data for each item in a column with a header as shown below right.



products.txt - Notepad

File Edit Format View Help

```
Frisbee 24 12.99
Picture 11  13.50
Figure 9 19.89
Towel 30 9.95
umbrella 6 16.75
lotion 24 2.95
```

```
Item             Units    Price
----------------------------
Frisbee             24    12.99
Picture             11    13.50
Figure               9    19.89
Towel               30     9.95
umbrella             6    16.75
lotion              24     2.95
```

Data File   Output

29. Using the data file from #28 above, write a program that adds the products below to the file without deleting the existing products.

    Post Card 50 1.99

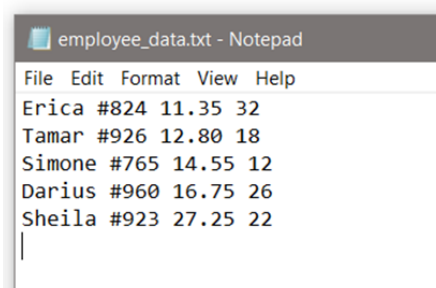
    Cooler 8 9.98

Chapter 7 Programming Challenges

#1 Employee Data File

Design and develop a program for a local company payroll that uses the employee data file information shown. Create the data file shown below left and write a program that will read the file and display the name and ID for the employee, and the gross pay for each employee based upon the input file data (hourly rate * hours worked). The format for the output is shown below. Include an exception handler in the solution. (To add the dollar sign, insert it before the % sign in the format specifier)

The data format for the input file is: name, ID number, hourly rate, and hours worked.



```
employee_data.txt - Notepad
File  Edit  Format  View  Help
Erica #824 11.35 32
Tamar #926 12.80 18
Simone #765 14.55 12
Darius #960 16.75 26
Sheila #923 27.25 22
```

```
Employee        ID#      Gross Pay
----------------------------------
Erica           #824     $363.20
Tamar           #926     $230.40
Simone          #765     $174.60
Darius          #960     $435.50
Sheila          #923     $599.50
```

Data File   Output

#2 Employee Data File - Dialog

Modify the program in Programming Challenge #1 to use a File Open dialog to obtain the name of the file.

Chapter 8

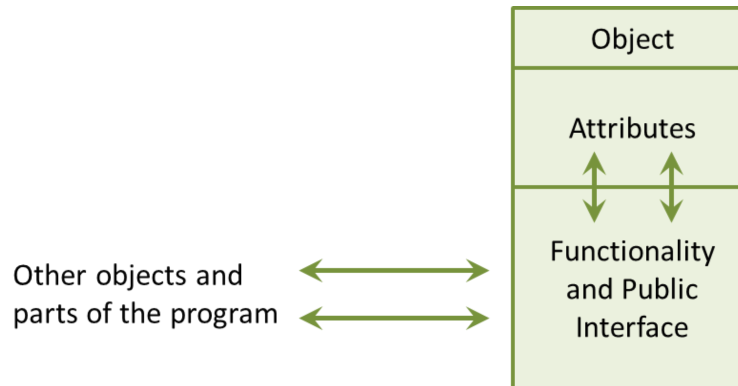
Classes and Objects

In ***Object-oriented Programming*** (OOP), data and functionality are combined in an object and are *hidden* from the rest of the program. The data items stored by an object are referred to as ***attributes*** or members (sometimes member variables or fields), and the ***methods*** within an object perform operations (sometimes referred to as behaviors or procedures) on the data. Object oriented terminology has changed over the decades and different terms are used interchangeably and sometimes depend on the programming language. This is unfortunate, but essentially objects have attributes (data elements) and methods that operate on the data elements and provide an interface for other objects and parts of the program.

Objects generally model real-world entities that have characteristics such as a house that has doors and windows. The doors and windows would be the attributes of the house. This is often referred to as the "has a" relationship. A house *has a* door, and the house *has a* window.

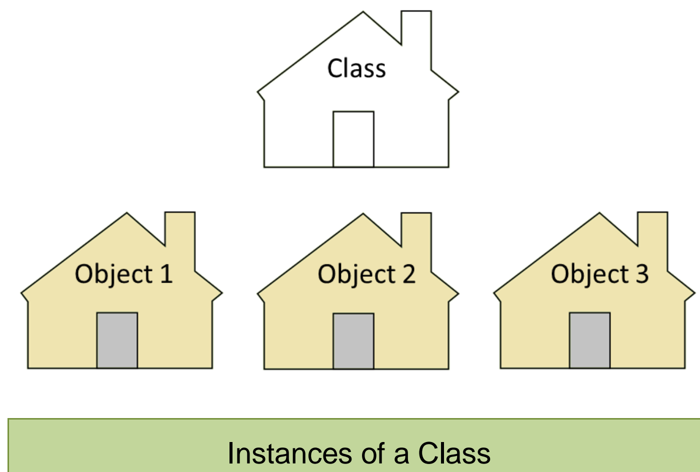
The attributes of an object are typically hidden from outside the object to protect the data from being corrupted or changed arbitrarily. This is called ***encapsulation***. Parts of a program and other objects are provided access to the attributes through a ***public interface*** made up of methods that provide protection for the attributes while allowing access to them when necessary. Since interaction with an object is provided through the public interface, only

knowledge of the interface is required to use them. This is referred to as ***information hiding*** since programs can use an object without knowing the inner workings. Any future changes to an object internally do not necessarily require changes to a program that uses that object. Unless the public interface has changed, there is typically no need to modify programs that use the object.



Classes

To create an object, there must be a class. A ***class*** is a framework or blueprint of what an object will contain when one is created. As an example, an architect can provide a detailed drawing of a building that shows a door, but the door cannot be opened. It doesn't exist. A building must be built from the drawing, and then the physical door of the building can be opened. The building would be an instance of the drawing the same way that an object is an ***instance*** of a class. In addition, multiple buildings could be built from the same drawing and they would all be identical, and they would each have their own door. Multiple objects can be ***instantiated*** from a single class, and they would each have their own set of class attributes.



The ***class definition*** describes the data elements and methods for the class. The general format for a class definition is shown below. The class definition begins with the public access specifier, the ***class*** key word and the name of the class. Class names begin with an uppercase letter and then begin each additional word with an uppercase letter. The third line below is the ***constructor*** which has the exact same name as the class, and looks like a method but has no return type.

```
public class ClassName {

    -- Attributes --

    public ClassName() {

        -- Attribute Initializers --

    }

        -- Methods and Public Interface --

}
```

Class Definition Format

Constructors

Every class has a constructor which is called when an object (instance of the class) is created using the ***new*** operator. The constructor allocates memory and *constructs* an instance (object) of the class. If a constructor is not written, the compiler will generate one without parameters, and the attributes of the object will be set to default values. Numbers will be set to zero, Boolean variables will be initialized to false, and array and object references will be set to null. However, a constructor is typically written to initialize the ***instance variables*** (attributes) of the object. A written constructor may have parameters that receive values to initialize the attributes, or it may initialize them with default values.

Technical Notes

Since the attributes (data elements) only exist when an object (instance of the class) exists, the attributes are often referred to as instance variables.

As an example, consider a *Reservation* class that has attributes for name, day, time, and number of guests. When a reservation object is created, it could be initialized with the reservation information passed to the constructor. The class definition in example Ex. 8.1 includes the class declaration, and the attributes (instance variables) for the class, followed by the constructor. The constructor includes the parameters for name, day, time, and guests. When an object of this class is created, four arguments must be passed to the constructor. The constructor ***initializes*** the attributes with the values received in the parameters.

Ex. 8.1 – Reservation Class Definition

```
public class Reservation {

    String name, day;
    int time, guests;

    public Reservation(String n, String d, int t, int g) {

        name = n;
        day = d;
        time = t;
        guests = g;
    }
}
```

Example Ex. 8.2 below creates a *Reservation* object (instance of the class) and passes "Walker", "Mon", 1730, and 5 to the constructor. A *Reservation* object is created and assigned to `r1`. The output statements access the instance variables directly using the name of the object, the dot operator, and the attribute name.

Ex. 8.2 – Reservation Class Object

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

    Reservation r1 = new Reservation("Walker", "Mon", 1730, 5);

    System.out.printf("%8s%4s", r1.name, r1.day);
    System.out.printf("%5d%2d", r1.time, r1.guests);

}
```

```
  Walker Mon 1730 5
```

Program Output

Each *Reservation* object created from the class will have its own set of attributes with their own values. The values stored in the instance variables are referred to as the object's ***state***. The state for r1 is shown below.

Reservation r1	
name	Walker
day	Mon
time	1730
guests	5

Object State

To highlight this, example Ex. 8.3 creates two *Reservation* objects (r1 and r2) with different values and accesses their instance variables.

Ex. 8.3 – Multiple Objects (instances of the class)

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

    Reservation r1 = new Reservation("Walker", "Mon", 1730, 5);
    Reservation r2 = new Reservation("Bethea", "Tue", 1800, 4);

    System.out.printf("%8s%4s", r1.name, r1.day);
    System.out.printf("%5d%2d", r1.time, r1.guests);
    System.out.printf("%8s%4s", r2.name, r2.day);
    System.out.printf("%5d%2d", r2.time, r2.guests);

}
```

```
  Walker Mon 1730 5  Bethea Tue 1800 4
```

Program Output

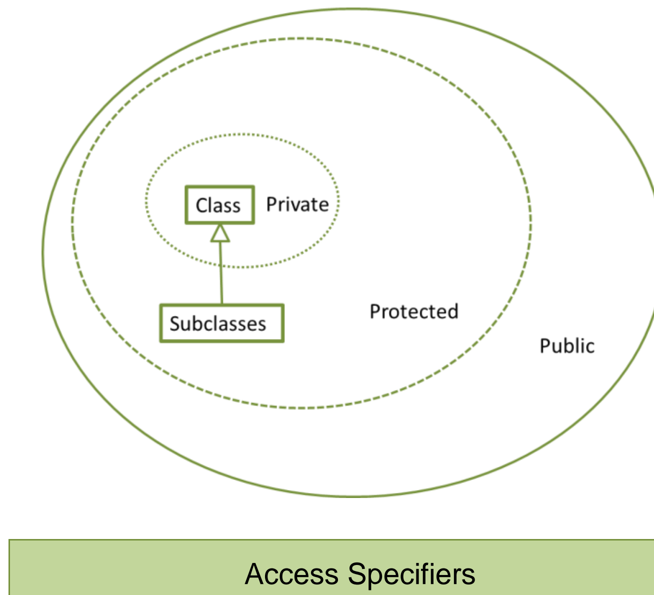
Access Specifiers

In the examples, the class and constructor were preceded by the *public* access specifier. Access specifiers provide control over those parts of a class that can be accessed by other objects and other parts of a program. The three access specifiers are private, protected, and public and provide different levels of protection and access.

Public: visible everywhere.

Protected: visible to the package and to subclasses.

Private: visible only to other members of the class.



Access Specifiers

Class attributes in OOP should be specified as ***private*** to enforce encapsulation and information hiding. This protects the attributes from being accessed directly and changed arbitrarily. The code below includes the private access specifiers for the Reservation attributes. It also lists the instance variables on separate lines which is typical and enhances readability.

```
public class Reservation {

    private String name;
    private String day;
    private int time;
    private int guests;

    public Reservation(String n, String d, int t, int g) {

        name = n;
        day = d;
        time = t;
        guests = g;
}
```

Private Access Specifiers

Public Interface

The *Reservation* example has a constructor that receives parameters to initialize the instance variables of the object. Some classes do not receive parameters in the constructor and there are methods within the class to assign values to them. Also, consider that a value in an object may need to be changed such as a *Reservation* time or day. Methods are added to the class definition, which allow access to the private attributes to change the state of the object. Methods that provide access to or the ability to change the attributes in a class form the public interface. There are two types of methods in the public interface. Methods that change the attributes of an object are referred to as ***mutator*** methods. Methods that access an object's attributes without changing them are referred to as ***accessor*** methods. Mutators *set* and accessors *get*. For this reason, some programmers refer to them as setters and getters.

```
getSomeValue();        // accessor

setSomeValue();        // mutator
```

Mutator and accessor methods should have names that indicate what they change or access. Example Ex. 8.4 adds a method to the Reservation class to change the time. The method validates the input to protect time from being set to an invalid number. This is an important role for the public interface.

Ex. 8.4 – Public Interface Methods

```
public class Reservation {
    private String name;
    private String day;
    private int time;
    private int guests;

    public Reservation(String n, String d, int t, int g) {
        name = n;
        day = d;
        time = t;
        guests = g;
    }

    public void setTime(int t) {
        if (t > 0 && t % 100 <= 59) {
            time = t;
        }
    }
```

The accessor methods for the Reservation class are shown in example Ex. 8.4A below. The accessors simply provide the current state (value) of the attributes.

Ex. 8.4A – Public Interface Methods for Reservation

```
public String getName() {
    return name;
}

public String getDay() {
    return day;
}

public int getTime() {
    return time;
}

public int getGuests() {
    return guests;
}
```

Public Interface Accessor Methods

The output statements in the example program have been changed since the attributes are now private. They must be accessed through the public interface of the class. The example below uses the accessors and demonstrates changing the time for a reservation using the mutator method.

Ex. 8.5 – Public Interface Methods for Reservation

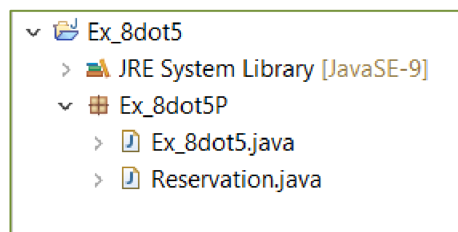
```
public static void main(String[] args) {

    Reservation r1 = new Reservation("Walker", "Mon", 1730, 5);

    System.out.printf("%8s%4s", r1.getName(), r1.getDay());
    System.out.printf("%5d%2d", r1.getTime(), r1.getGuests());

    r1.setTime(1800);

    System.out.printf("%8s%4s", r1.getName(), r1.getDay());
    System.out.printf("%5d%2d", r1.getTime(), r1.getGuests());
}
```

```
  Walker Mon 1730 5  Walker Mon 1800 5
```

Program Output

Modularization and Class Files

Modularization, which was introduced in Chapter 5, also applies to classes. Class definitions should be located in separated files. This aligns with the process of modularizing programs which provides many benefits including the ability to: reuse portions of the code, divide the program development among multiple programmers, and simplify the overall project. Classes should be in their own files, the file name should be the class name, and there can only be one public class per file. If the file is in another package, the package is imported just as java.util is imported. The file structure for Ex. 8.5 is shown below.



Ex. 8.5 File Structure

Class Variable Types

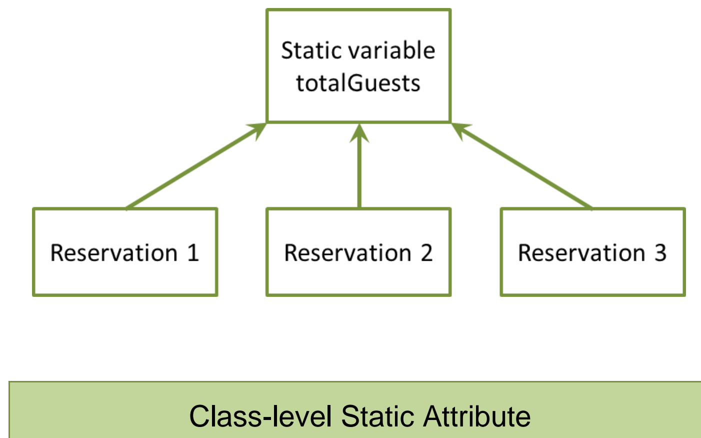
A class can contain any of the following variable types.

- **Local variables** – Variables defined inside methods, constructors or blocks of code. The variable will be declared and initialized within the method and destroyed when the method has completed.
- **Instance variables** – Variables within a class but outside any method, and are initialized when an object is created. Instance variables can be accessed from any method, constructor, or block of code within the class.
- **Class variables** – Variables declared within a class with the static keyword, and outside any method (further explained in the next section).

Class-level Variables – Static

An attribute that is declared as ***static*** and outside any method in a class is a class-level attribute and is shared by all objects of the class. They are useful for class constants, tracking data across all instances of the class (similar to static variables), and for defining default values. A class attribute can be accessed

using the class name or an objects name. As an example, the restaurant using the Reservation program may want to tally the total number of guests combined for all reservations. A static variable could be added to the class that would be shared by all instances (objects) of the class. When a new reservation object is created, the constructor could add the number of guests for that reservation to the class variable, and a method could be added to obtain the total number of guests. Each Reservation would share the same variable.



Class-level Static Attribute

Example Ex. 8.6 includes the modifications to add the static variable, update the variable in the constructor, and provide an accessor method for the variable.

Ex. 8.6 – Static Variable for Total Number of Guests

```
public class Reservation {

    private String name;
    private String day;
    private int time;
    private int guests;
    private static int totalGuests;

    public Reservation(String n, String d, int t, int g) {

        name = n;
        day = d;
        time = t;
        guests = g;
        totalGuests = totalGuests + g;
    }


    public int getTotalGuests() {

        return totalGuests;
    }
```

The main method for the program has been modified below to create two Reservation objects and display the total number of guests after the first object is created and then again after the second is created. Note that the second call to the method is also made using the name of the first object. Since totalGuests is a shared variable, each of the objects would access the same value.

```
public static void main(String[] args) {
    Reservation r1 = new Reservation("Walker", "Mon", 1730, 5);
    System.out.println("Total guests are " + r1.getTotalGuests());
    Reservation r2 = new Reservation("Bethea", "Tue", 1800, 4);
    System.out.println("Total guests are " + r1.getTotalGuests());
}
```

```
Total guests are 5
Total guests are 9
```

Program Output

Static Methods

Classes can also define static methods that are not invoked on an object. As an example, the getTotalGuests has been redefined as a static method.

```
public static int getTotalGuests() {
    return totalGuests;
}
```

The method can now be called using the class name instead of the object name, and can be called even when there are no objects.

```
System.out.println("Total guests are " + Reservation.getTotalGuests());
Reservation r1 = new Reservation("Walker", "Mon", 1730, 5);
System.out.println("Total guests are " + Reservation.getTotalGuests());
```

```
Total guests are 0
Total guests are 5
```

Program Output

Overloaded Constructors

To provide different initializations of an object, a class can have more than one constructor. When a method (a constructor is a method) is defined more than once it is referred to as an overloaded method (the parameter lists must be different). The method's ***signature*** is the name, parameter list, and return type. This is the way that the methods are differentiated. As an example, consider that reservations are frequently made at the restaurant for large groups and the exact time is not known and will be provided later. A constructor could be added that does not require the time argument. The proper constructor will be called based upon the arguments that are passed when the object is created.

```
public Reservation(String n, String d, int t, int g) {

    name = n;
    day = d;
    time = t;
    guests = g;
}

public Reservation(String n, String d, int g) {

    name = n;
    day = d;
    time = 0;
    guests = g;
}
```

Overloaded Constructor

The program below creates two Reservation objects using the two different constructors. In each case, the proper constructor is called.

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

    Reservation r1 = new Reservation("Walker", "Mon", 1730, 5);

    System.out.printf("%8s%4s", r1.getName(), r1.getDay());
    System.out.printf("%5d%2d", r1.getTime(), r1.getGuests());

    Reservation r2 = new Reservation("Bethea", "Wed", 3);

    System.out.println();
    System.out.printf("%8s%4s", r2.getName(), r2.getDay());
    System.out.printf("%5d%2d", r2.getTime(), r2.getGuests());
}
```

Notice in the output that the time attribute for r2 was initialized to zero by the constructor.

```
Walker Mon 1730 5
Bethea Wed    0 3
```

Program Output

Passing Objects as Arguments

Objects can be passed to methods as arguments. The parameter of the method receives a reference to the object which is the object's memory address and it provides access to the object's methods and attributes. The example below creates a *Reservation* object and passes it to a method.

Ex. 8.7 – Objects as Arguments to Methods

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

    Reservation r1 = new Reservation("Walker", "Mon", 5);
    showReservation(r1);

}
```

The method receives a *Reservation* object reference in the parameter which allows access to the methods of the object. Note that the type of the parameter for the method is a Reservation. The name used by the method to receive the reference can be anything, and is used to access the attributes of the object by the method.

Ex. 8.7A – Objects as Arguments to Methods

```
public static void showReservation(Reservation r) {

    System.out.printf("Name:%8s Day:%4s", r.getName(), r.getDay());
    System.out.printf("\nTime:%5d Guests:%2d", r.getTime(), r.getGuests());

}
```

Designing Classes

Before determining the classes for a project, a complete understanding of the requirements is needed to ensure that all aspects of the project are known. The

requirements and all possible real-world entities (classes), data elements (attributes) and events that could occur (methods) are then listed. This is referred to as the ***Problem Domain***. Once the problem domain is established, the nouns are considered to determine the class candidates, and the verbs are considered to determine the methods. The Reservation project's problem domain might be as simple as the description below.

Reservation Project Problem Domain

A reservation system for a restaurant including name, day, time, and guests
Allow for updates and display of the reservation information

Before writing a class definition, time should be spent designing the class. This includes considering the data attributes and the public interface (methods) that will be needed to access and change the data. Methods should be included that provide the necessary operations on objects of the class. A program that uses the class should not have to include functionality pertaining to the object. Most classes model real-world objects and should represent a clear and single ***abstraction***. That is, the class should not include functionality that is outside its responsibilities like getting user input, or anything that is specific to a particular program. One of the goals of OOP is reuse of the class. If a program provides functionality that should be in the class, the class is not easily used in another program. Another goal is ***cohesion*** which refers to the degree to which a class represents a single abstraction without external dependencies. The degree to which a class depends on another class or another part of the program is referred to as ***coupling***. A class should have cohesion (represent a stand-alone entity), and loose coupling (no external dependencies).

One of the tools used in the design of classes is the brain-storming session. A project design team meets, discusses, and documents ideas and suggestions for classes, and the nouns and verbs used when describing the classes. The nouns represent attributes (data elements) and the verbs represent the methods that will act on the data including the public interface. The goal is to include all possible elements, and refine the list as design continues. The lists of nouns and verbs for the Reservation project would include at least those shown below.

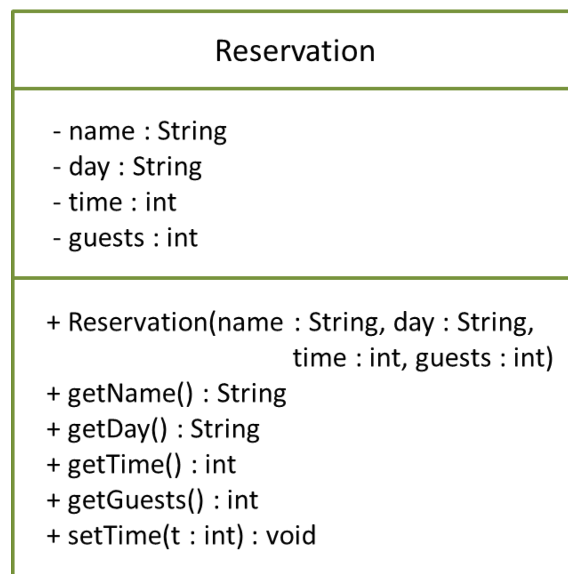
Reservation Project

Nouns – reservation, customer name, day, time, and number of guests

Verbs – create a reservation, get the name, get/set day, get/set time, get/set guests, and display the reservation information

Unified Modeling Language (UML)

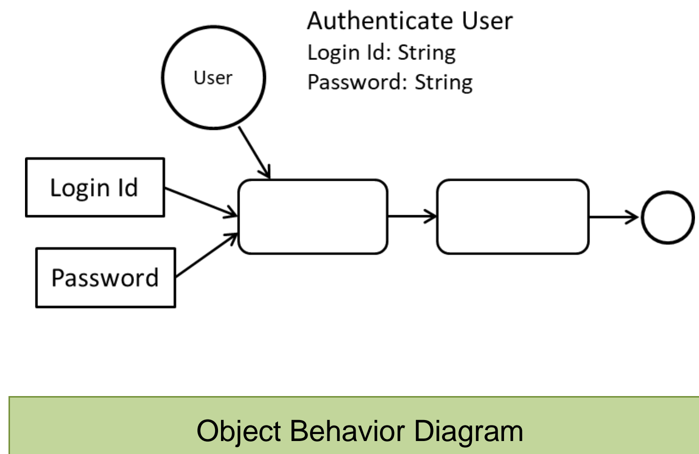
Another tool used for designing and documenting classes is the UML diagram which describes a class, including the attributes, and methods. The top section of a UML diagram contains the name of the class, followed by a section that describes the data attributes, and the bottom section lists the methods for the class. A UML diagram includes symbols and a specific format for identifying data types and access specifiers. The plus sign represents public access, the pound sign for protected, and a minus sign is used to represent private access. Attribute data types follow a colon after the attribute name. Methods include the name, and the parameters are listed with name, colon, and data type. The return data type, including void, is listed after the closing parenthesis and a colon. For the Reservation, the attributes are private and the methods including the constructor for the class are public. A static variable would be underlined.



Unified Modeling Language (UML) Diagram

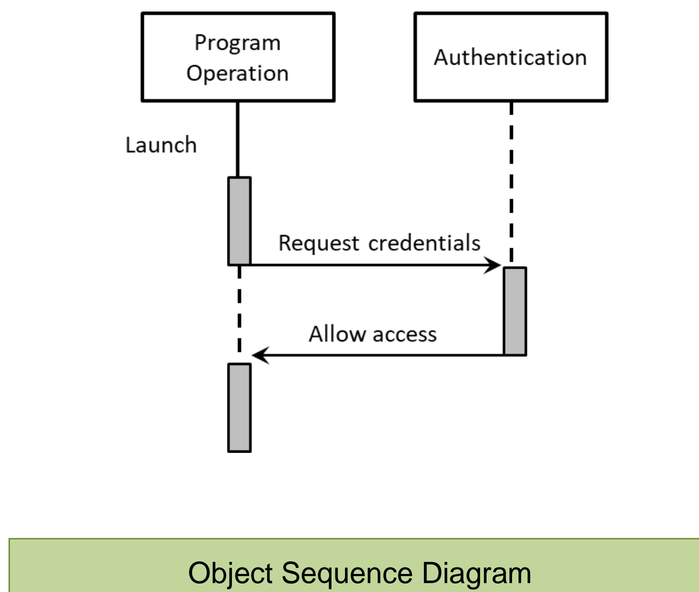
The UML format may differ slightly across industries or organizations, but all capture the data attributes and methods and can be used in the design process. UML behavior diagrams (object activity diagrams) are used to show the flow of control, data, and transactions. UML Superstructure Specifications provide a standard for object interaction depiction.

The Behavior Diagram below illustrates the authentication of user activity with Login Id and Password.



Object Behavior Diagram

The Object Sequence Diagram adds the chronological aspect to the operations as well as the program sequence and order of operations.



Object Sequence Diagram

The Reservation Class Revisited

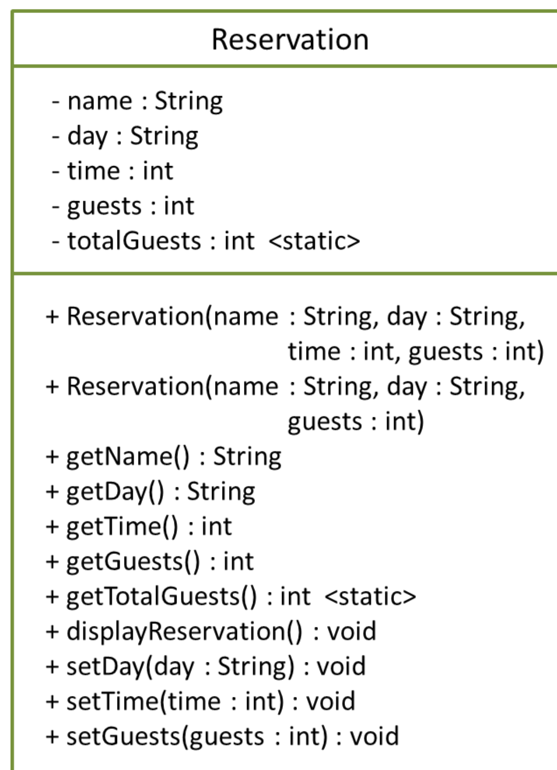
The Reservation class definition below has been modified to include the accessor and mutator methods. Due to the increase in lines of code, most standards allow single line method bodies for accessors and when appropriate.

Ex. 8.8 – Reservation Class

```
public class Reservation {

    private String name;
    private String day;
    private int time;
    private int guests;
    private static int totalGuests;

    // Constructor
    public Reservation(String n, String d, int t, int g) {

        name = n;
        day = d;
        time = t;
        guests = g;
        totalGuests = totalGuests + g;
    }


    // Overloaded Constructor
    public Reservation(String n, String d, int g) {

        name = n;
        day = d;
        time = 0;
        guests = g;
        totalGuests = totalGuests + g;
    }


    // Accessors
    public String getName() {
        return name; }

    public String getDay() {
        return day; }

    public int getTime() {
        return time; }

    public int getGuests() {
        return guests;  }

    public void displayReservation() {
        System.out.printf("Name:%8s   Day:%4s", name, day);
        System.out.printf("\nTime:%5d   Guests:%2d", time, guests);
    }
```

```
    // Mutators
    public void setDay(String d) {
        day = d; }

    public void setTime(int t) {
        time = t; }

    public void setGuests(int g) {
        guests = g; }


    // Static method
    public static int getTotalGuests() {

        return totalGuests;
    }
}
```

The UML Diagram for the Reservation Class below has been updated to include the additional methods, the overloaded constructor, and the static variable. The static designation uses angled brackets.

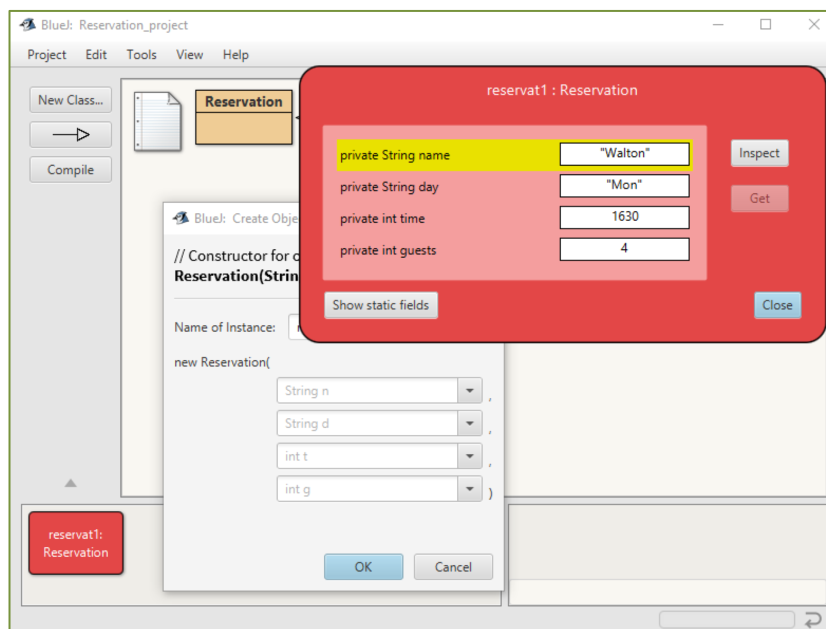


Updated Reservation Class UML Diagram

Testing Classes

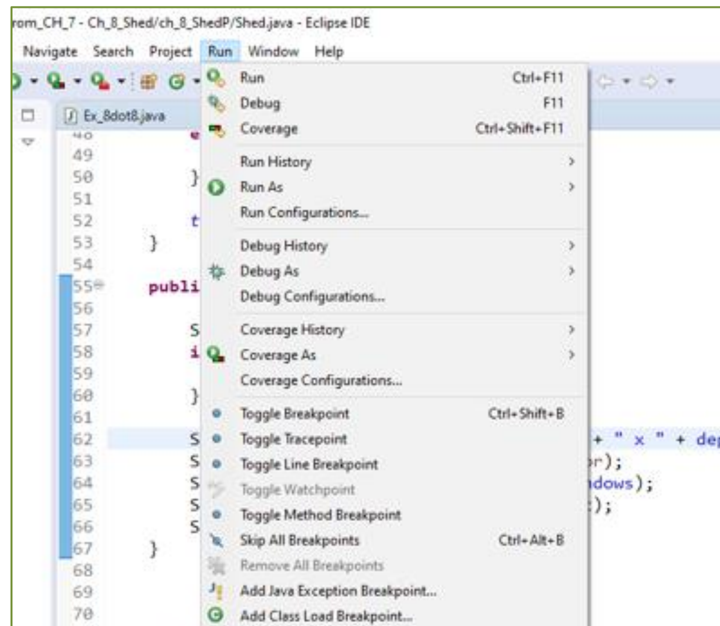
Classes are often part of a larger program that may contain multiple classes. Before introducing a class into a larger program, the class should be thoroughly tested. Testing a class in isolation is referred to as ***Unit Testing***. The examples created instances of Reservations from a main method and invoked some of the methods. This approach is often used to test a class and might be considered a ***driver program***. Programs and classes can be designed to test parts or even complete programs to automate the process. Automated testing reduces human error, saves time, and makes testing more likely.

When testing classes, how the class will be used in the application determines the level and type of testing. Each part of the class should be tested separately and as a unit, and testing should be carried out during development as well. Recall from Chapter 1 that the cost of fixing errors increases through the Software Development Lifecycle. Developers should be aware of multiple test strategies and tools, and choose the appropriate one for that point in development. When a modification is made to an existing program, ***regression testing*** involves re-running tests that previously passed to ensure that they still pass, or to compare current program results with previous results. JUnit and BlueJ are popular frameworks for regression and unit testing.



BlueJ Integrated Development Environment

When a bug (error) is detected, there are multiple strategies for ***debugging*** as well. Debugging tools are provided in integrated development environments that highlight errors and often suggest corrective action. They also include what are called breakpoints that allow stopping execution of the code at certain points in the program to view variables and the state of the attributes.



Eclipse Debugging Tools Menu

Other debugging techniques include manual and verbal walkthroughs of the code. Very often verbalizing the steps in the program or explaining what is happening to someone else will surface the issue. Adding output statements is another popular debugging tool because it does not require any special tools and provides a quick look at what is happening at any point in the program.

Technical Notes

Testing detects the presence of errors and debugging searches for the source of errors. The error may be visible far from the source and require extensive debugging.

Thoroughly testing a class can reveal areas requiring additional consideration. For example, the Reservation class will have user interaction for data entry. A restaurant employee could accept reservation information over the phone, or a customer could enter information on the restaurant website. Both of these scenarios could introduce invalid data. The Reservation constructor and methods in the examples did not all validate the entered data before assigning values to the attributes. The mutators modify the attribute values, and they should test the data before any updates as well. The Reservation class mutators are modified below to include validation of the input values.

```
// Mutators
public void setDay(String d) {
    if(d.contentEquals("Mon") || d.contentEquals("Tue") ||
            d.contentEquals("Wed") || d.contentEquals("Thu") ||
            d.contentEquals("Fri") || d.contentEquals("Sat") ||
            d.contentEquals("Sun")) {
        day = d;
    }
    else {
        System.out.println("Invalid day entered.");
    }
}

public void setTime(int t) {
    if(t <= 0 || t > 2400) {
        System.out.println("Invalid time entered.");
    }
    else {
        time = t;
    }
}

public void setGuests(int g) {
    if(g > 0) {
        guests = g;
    }
    else {
        System.out.println("Invalid number of guests.");
    }
}
```

Updated Reservation Class Mutators

Rather than repeat input validation in the constructor, the constructor can call the mutators which validate the data and assign the values. The constructor

below has been modified to use the updated mutator methods and protect the private attributes from erroneous values.

```
// Constructor
public Reservation2(String n, String d, int t, int g) {

    name = n;
    setDay(d);
    setTime(t);
    setGuests(g);
}
```

Updated Reservation Class Constructor

Modifications to the mutator methods and constructor were performed after the fact, and could have been considered during the design phase. Very often testing will reveal things that may have been overlooked or could be improved. The goal is to provide a solution that is as error-free as possible.

A Complete Example – Garden Shed Company

Requirements:

Design and develop a program for a company that builds wooden garden sheds. The program will determine the cost to build sheds based upon the size of the shed, door type, and number of windows. The program should allow the information for multiple sheds to be input, and provide the total cost for all of the sheds.

Shed Options and Pricing:

Shed size options as width/depth and price:

10 x 8 - $900, 12 x 10 - $1,800, and 16 x 12 - $2,600

Window options and price:

One (1) large $90.00 or two (2) medium at $80.00 each

Door options and prices:

Single door $160.00, Double door $240.00

Ramp:

Double door requires entry ramp $90.00

Design

The program will have a Shed Class with a constructor that accepts the size, the door type, and number of windows. The main method will test the class by creating shed objects and displaying the total cost.

The constructor for the Shed class will receive the size of the shed as integers for width and depth (to accommodate additional size sheds in the future). The option for door type will be a String ("S" or "D") which will determine if the ramp cost is included, and an integer (1 or 2) will be used for the number of windows.

After assigning values to the attributes, the constructor will call a method to compute the cost for the shed that will also update a static variable for the total cost.

The Shed Class (public interface methods omitted)

```
public class Shed {

    private static double totalCost = 0;

    private int width;
    private int depth;
    private String doorType;
    private int windows;
    private double cost;


    public Shed(int w, int d, String door, int wins) {

        width = w;
        depth = d;
        doorType = door;
        windows = wins;
        computeCost();
    }


   private void computeCost() {

       if(width == 10 ) {       // cost from the size
           cost = cost + 900;
       }
       else if(width == 12) {
           cost = cost + 1800;
       }
       else {
           cost = cost + 2600;
       }
```

```
        if(doorType.contentEquals("S")) {  // cost from door type
            cost = cost + 160;
        }
        else {
            cost = cost + 240 + 90; // includes ramp
        }

        if(windows == 1) {          // cost for windows
            cost = cost + 90;
        }
        else {
            cost = cost + 160;
        }

        totalCost = totalCost + cost;
    }


    public void displayCost() {

        String door = "double";
        if(doorType.contentEquals("S")) {
            door = "single";
        }

        System.out.print("Shed size: " + width + " x " + depth);
        System.out.print("   Door type: " + door);
        System.out.println("   Windows: " + windows);
        System.out.printf("Cost: $%,8.2f", cost);
        System.out.println("\n");
    }
```

The static method for the total cost

```
public static void displayTotalCost() {
    System.out.printf("\nTotal Cost: $%,8.2f", totalCost);
}

}
```

Testing and Debugging

The class must be tested and verified. A few different shed options should be tested and verified for accuracy using calculated data. The main method below creates three shed objects of different sizes and with different options for doors. The expected results are calculated for comparison with the output from the program.

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

    Shed s1 = new Shed(10, 8, "S", 1);
    s1.displayCost();

    Shed s2 = new Shed(12, 10, "D", 2);
    s2.displayCost();

    Shed s3 = new Shed(16, 12, "D", 2);
    s2.displayCost();

    Shed.displayTotalCost();
}
```

Test data:

Shed 1	10 x 8	$900.00	
	Single door	$160.00	
	One window	$90.00	
	Total	$1,150.00	
Shed 2	12 x 10	$1,800.00	
	Double door	$240.00	
	Ramp	$ 90.00	
	Two windows	$160.00	$80.00 each
	Total	$2,290.00	
Shed 1	16 x 12	$2,600.00	
	Double door	$240.00	
	Ramp	$90.00	
	Two windows	$160.00	$80.00 each
	Total	$3,090.00	

```
Shed size: 10 x 8   Door type: single   Windows: 1
Cost: $1,150.00

Shed size: 12 x 10   Door type: double   Windows: 2
Cost: $2,290.00

Shed size: 16 x 12   Door type: double   Windows: 2
Cost: $3,090.00


Total Cost: $6,530.00
```

Program Output

Object References

In the Reservation examples, an instance or object of the class was created using the `new` operator, and the class name. The object was created and a reference to the object in memory was assigned to a variable (`r1` below). The variable does not hold the object.

```
Reservation r1 = new Reservation("Walker", "Mon", 1730, 5);
```

This is different from an integer or any of the primitive data types. For example, when an integer is declared and assigned a value, the variable holds the value (not a reference to the variable).

```
int value = 23;
```

The computer uses 4 bytes to store an integer, but an object could be very large and require a lot of memory. It is more efficient to use a reference to an object. A method that returns an object would return a reference to the object.

Methods Returning Objects

A method can return an object (actually a reference to an object). The receiving data type would be the class name and the method would call the constructor. Consider a method that returns a Reservation object. The method would call the constructor and return a reference to the object.

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

    Reservation r1 = createReservation();
    r1.displayReservation();

}

public static Reservation createReservation() {

    Reservation obj = new Reservation("Name", "Day", 1500, 4);

    return obj;
}
```

The example above may seem like a two-step process that could be avoided by having the main method call the constructor directly, but consider a file of Reservations that is read by the program, an object for each Reservation is

created, and then the objects are added to an ArrayList of Reservations. The program can then use the ArrayList of Reservations for processing.

The *this* Reference

A reference variable that an object can use to reference itself is the *this* reference variable. Instance methods implicitly receive it and it refers to the current object. Programmers can used it to overcome ***shadow*** variables which can occur when variables have the same name. As an example, if the Reservation constructor parameter for name were modified to accept the argument as name, the instance variable would cause the IDE to indicate that it is not used and that the constructor assignment has no effect (note the yellow line under name = name).

```
public class Reservation {

    private String name;
    private String day;
    private int time;
    private int guests;

    // Constructor
    public Reservation(String name, String d, int t, int g) {

        name = name;
        day = d;
        time = t;
        guests = g;
    }
```

When the program runs and an attempt to access the name variable is made, the output is null.

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

    Reservation r1 = new Reservation("Bethea", "Wed", 1730, 4);

    System.out.println("Name is: " + r1.getName());
}
```

```
<terminated> Ch_8_this [Java Application]
Name is: null
```

Instance Variable Assigned null

By using the key word ***this*** in the constructor, the assignment is made to the attribute of the object. The variable on the left side of the assignment statement is associated with the class. Using *this* is not required, but it can be used to avoid shadow variables or to add clarity and highlight that the variable is an instance variable and belongs to the object. It can also be used with class methods.

```
// Constructor
public Reservation(String name, String d, int t, int g) {

    this.name = name;
    day = d;
    time = t;
    guests = g;
}
```

```
<terminated> Ch_8_this [Java Application]
Name is: Bethea
```

The *this* Key Word

Since a descriptive name should be used for parameters as well as attributes, they may have the same name. Some programmers prefer to use the this key word and avoid any issues with shadowing. In the Reservation example, the first letter of the attribute was used for the constructor parameters. Since the assignments and method calls are in view, it is easy to identify what the letters indicate and the abbreviated parameter names are acceptable.

```
// Constructor
public Reservation2(String n, String d, int t, int g) {

    name = n;
    setDay(d);
    setTime(t);
    setGuests(g);
}
```

The null Reference

An issue that should be avoided occurs when an instance variable is not set by a constructor. The variable reference will be set to null by default, and operations

on that variable could cause a null pointer exception when the program runs. This is why the previous example displayed null for the name when the assignment in the constructor could not be made and the attribute was accessed. If the Reservation constructor did not accept arguments, or a default constructor was used, the instance variables would be assigned their default values (numbers to zero and objects to null). A String is an object so the default initialization is null. To avoid this situation, the default constructor could be modified to initialize some or all instance variables. As shown below, the instance variable name can be assigned the empty String which has a length of zero, but is a valid String.

```
public class Reservation {

    private String name;
    private String day;
    private int time;
    private int guests;

    // Constructor
    public Reservation() {

        name = "";
    }
```

Initializing Instance Variables

Technical Notes

Technically speaking, null is seen by the computer as memory address zero. There is no address zero and therefore null is assumed to be nothing. Recall in Chapter 5, null was used as an argument for a dialog box to indicate that there was no parent window.

The Garbage Collector

The Java Virtual Machine periodically runs Garbage Collection to automatically remove unreferenced objects from memory. The garbage collector performs a mark operation that identifies memory in use and memory no longer in use. It then performs a sweep operation that removes items marked as unused. This automatically frees up memory that is no longer being used.

Chapter 8 Review Questions

1. Hiding the implementation of a class is referred to as ______________.
2. The instance variables of a class are called ________________.
3. Program statements and other objects access an object's attributes through the _________ _________.
4. An object is a(n) _____________ of a class.
5. The class ____________ declares the data and methods for a class.
6. The method that creates an instance of the class and may initialize an object is called the ____________.
7. The values stored in an object's data attributes are referred to as the object's ___________.
8. The three access specifiers are __________, _________, and __________.
9. Class attributes should be specified as _________ to enforce encapsulation.
10. Other objects and methods should access an object's attributes through the ________ _________.
11. Methods that access an object's data attributes are called ____________.
12. Methods that set or change an object's data attributes (state) are called ________________.
13. Modularization requires classes to be in separate __________.
14. Variables declared within a class with the static key word and outside any method are __________ variables.
15. ___________ variables are shared by all of the objects of a class.
16. A _________ method is not invoked on an object and is called using the class name.
17. A constructor with the same name as another constructor, but has a different parameter list is a(n) _____________ constructor.
18. When an object is passed to a method, the method receives a ____________ to the object.
19. ____________ refers to the degree to which an object represents a single abstraction without external dependencies.
20. __________ refers to the degree to which an object is dependent upon another.
21. A _________ diagram describes the attributes and methods of a class.

22. Testing a class in isolation is referred to as __________ testing.
23. The _________ reference is often used to overcome variable shadowing by indicating the current object is being referenced.
24. A String attribute that is not initialized by a constructor will be initialized to _________ when an instance of the class is created.

Chapter 8 Short Answer Exercises

25. What is the difference between a class and an object?
26. Why do classes have a public interface?
27. What is included in the public interface of a class?
28. What are the attributes of the following class?

```
public class Account {

    private String name;
    private int accountNumber;
    private double balance;

    public Account(String n, int num, double bal) {

        name = n;
        accountNumber = num;
        balance = bal;
     }
```

29. How many parameters does the constructor require in the following class?

```
public class Account {

    private String name;
    private int accountNumber;
    private double balance;

    public Account(String n, int num, double bal) {

        name = n;
        accountNumber = num;
        balance = bal;
     }
```

30. Write an accessor method for the balance attribute of the Account class above.

31. Write a mutator method for the balance attribute of the Account class above.

32. Write a default constructor that has no parameters, but initializes the attributes of the Account class above.

33. Rewrite the assignments in the constructor below without changing the names to eliminate the shadow variable issue.

```
public Rectangle(int side1, int side2) {

    side1 = side1;
    side2 = side2;
}
```

Chapter 8 Programming Exercises

34. Create a program using the class definition below for a Circle class. In the main method for the program, create an instance of the class called c1 with 2.0 as the radius, and then display the diameter of the circle using the accessor method.

```
public class Circle {

    private double radius;
    private double diameter;

    public Circle(double rad) {

        radius = rad;
        diameter = 2 * radius;
    }


    public double getDiameter() {
        return diameter;
    }
}
```

35. Using the Circle class above, add an accessor *getCircumference()* that uses the radius to compute and return the circumference. Do not add an attribute for circumference. Add a display statement in main to display the result.

Circumference = 2 * Math.PI * radius

36. Using the Circle class in #34 above, add an accessor method *getArea()* that uses the radius to compute and return the area. Do not add an attribute for area.

Area = Math.PI * radius * radius

From the main method, create a Circle object with a radius of 3.0 and display the diameter, circumference, and area as shown below using 2 decimal places.

```
Diameter is 6.0
Circumference is 18.85
Area is 28.27
```

37. The value for the diameter is set when an object of the Circle class above is created. There is also an attribute for diameter in the class.

    a. Should there be an attribute for diameter?

    b. If there were a mutator for radius, would the diameter be updated?

    c. To resolve the issue, would it be preferred to have diameter computed in the accessor, and remove the attribute?

    d. Redesign the class to eliminate the attribute for diameter, and modify the constructor and *getDiameter()* method.

38. Implement a Product class that has private data attributes for description, price, and inventory. Write a constructor that accepts parameters for the attributes and initializes them, and a method to display a product's information. Write a program to create the product objects below and display them as shown.

```
Description      Price    Inventory
Mug               8.50       23
T-Shirt          12.95       45
Towel            18.50       36
```

39. Design and implement a Fuel Pump class that has private data attributes for price per gallon for regular $4.29 and premium $4.44, fuel grade, and sale amount. The constructor will accept a dollar amount, a "1" for regular or a "2" for premium fuel, and will set the prices for the two fuel grades (the input does not need to be validated). The class will have an accessor method to compute and return the gallons pumped, and an accessor method for the price per gallon based on the fuel grade selected. Create a UML diagram for the class.

Write a main program that will prompt for the sale amount and fuel grade. It will then create a Fuel Pump object and call the methods to display the number of gallons that were pumped and the price per gallon. A sample constructor is shown below.

```
public FuelPump(double amount, int fg) {

    priceReg = 4.29;
    pricePrem = 4.44;
    fuelGrade = fg;
    totalSale = amount;
}
```

Chapter 8 Programming Challenges

#1 Umpire Indicator

Many softball umpires use an Umpire Indicator to keep track of strikes, balls, outs, and innings. Design and implement an Indicator class that allows incrementing strikes to 3, balls to 4, outs to 3, and innings to 7. The class should have a constructor that initializes the attributes and the following methods:

displayAll() - displays the current values for all of the attributes

newBatter() - resets balls and strikes

setstrikes() - increments strikes, at 3 it increments outs and is reset to 0

setBalls() – increments balls, at 4 is reset to 0

setOuts – increments outs, and at 3 is reset to 0

newInning - increments the inning and resets balls, strikes, and outs

When the attributes reach their maximum values (i.e. 3 strikes), the methods should automatically update other attributes. In other words, after 3 strikes, *setOuts()* and *newBatter()* should be called, and the total number of outs should be monitored to automatically call *newInning()*.

Sample output portion:

```
Balls:   0      Strikes: 1      Outs:    2      Inning:  1
Balls:   0      Strikes: 2      Outs:    2      Inning:  1
Strike 3 - the batter struck out.

Three outs.
Starting Inning #2

Balls:   0      Strikes: 0      Outs:    0      Inning:  2
Balls:   1      Strikes: 0      Outs:    0      Inning:  2
```

Write a main method for unit testing and consider testing using nested loops as shown here.

```
Indicator i1 = new Indicator();

for(int i = 0; i < 12; i++) {

    for (int j = 0; j < 3; j++) {
        i1.setStrikes();
        i1.displayAll();
    }

    for (int j = 0; j < 4; j++) {
        i1.setBalls();
        i1.displayAll();
    }
}
```

#2 – Elevator Class

Implement an Elevator Class with attributes for elevator number (1, 2, 3), floor (1-3), and direction (going up or going down). The class should have a method to set the elevator number, a method to travel to the next floor, and a display method that displays the elevator number, current floor, and current direction (UP/DOWN). Elevators travel one floor at a time and travel up until reaching the top floor and then down until reaching the ground floor, and so on.

Write a program that creates three (3) elevators, and starts them each at the ground floor. Using a loop with 14 iterations, randomly select one of the elevators to move. The same elevator cannot be moved two times in a row. Display the current state for each of the elevators at each execution of the loop in sets of three as shown.

```
# 1  Floor 3  UP
# 2  Floor 3  UP
# 3  Floor 3  UP

# 1  Floor 3  UP
# 2  Floor 3  UP
# 3  Floor 2  DOWN

# 1  Floor 2  DOWN
# 2  Floor 3  UP
# 3  Floor 2  DOWN
```

#3 – ArrayList of Product Objects

Create a Product Class with attributes for name, price, units, and reorder point. Write a program that creates an ArrayList of the six (6) products listed below, and:

1. Display the products and information.
2. Prompt the user for the name of the product they would like to change.
3. Prompt the user for the change – 'p' = price, 'u' = units, 'r' = reorder point
4. Update the product information with the change
5. Display the products and information

Include class methods as needed to accomplish the steps.

Product data

T-Shirt, 19.50, 24, 12

Mug, 5.99, 10, 3

Clock, 14.50, 6, 2

Frame, 12.99, 12, 4

Poster, 4.99, 20, 5

Lamp, 16.50, 6, 2

Sample Program Run

```
                The list of products is:

        Name        Price     Units    Reorder Point
        T-Shirt     19.50      24           12
        Mug          5.99      10            3
        Clock       14.50       6            2
        Frame       12.99      12            4
        Poster       4.99      20            5
        Lamp        16.50       6            2

Enter the product to change: Clock

Enter p, u, or r to change price, units, or reorder point:p

Change that to 15.99

                The list of products is:

        Name        Price     Units    Reorder Point
        T-Shirt     19.50      24           12
        Mug          5.99      10            3
        Clock       15.99       6            2
        Frame       12.99      12            4
        Poster       4.99      20            5
        Lamp        16.50       6            2
```

Chapter 9

Inheritance and Interfaces

Inheritance

Objects are sometimes specialized versions of a more general class. For instance, a restaurant is a business, a clothing store is a business, and so is a theater. They have many things in common like employees, sales, and expenses, but they also have some differences. The common characteristics could be implemented in a Business class, and then each specific business type could be derived from that class. The derived classes would inherit the common characteristics from the Business class, and would implement the characteristics that are specific to them. In other words, they would ***extend*** the Business Class, and would exhibit what is called an "is a" relationship. This relationship between classes is established through ***inheritance***. The specialized classes (derived or subclasses) inherit the characteristics of the general class (base or super class).

In the diagram, the Business Class is the base class, and the specific businesses are the derived classes. As an example, suppose that a program is needed for a

company with different businesses. Since all of the companies have a name and employees, these could be implemented in the base class (Business) along with common accessor and mutator methods. Notice that the class definition below does not have a constructor. The default constructor will be used.

Ex. 9.1 – Business Class (base or super class) *partial*

```
public class Business {

    private String name;
    private int employees;

    protected void setName(String n) {
        name = n;
    }

    public void setEmployees(int e) {
        employees = e;
    }
```

A Clothing Store would have inventory, but could inherit the attributes from the Business class (name and employees). This is implemented by defining the Clothing Store Class as *extending* the Business Class using the ***extends*** key word. The ClothingStore constructor would use the mutator methods of the Business class to assign values to the Business class name and employees attributes.

Ex. 9.2 – ClothingStore Class extends Business Class

```
public class ClothingStore extends Business{

    private int inventory;

    public ClothingStore(String n, int e, int inv) {

        setName(n);
        setEmployees(e);
        inventory = inv;
    }

    public int getInventory() {
        return inventory;
    }
}
```

Notice in the constructor for the ClothingStore above that there is no indication that the methods *setName()* and *setEmployees()* are in the base class. It might seem

obvious, but programmers often add the ***super*** key word to highlight that an attribute or in this case the methods belong to the base class (shown below).

```
public class ClothingStore extends Business{

    private int inventory;

    public ClothingStore(String n, int e, int inv) {

        super.setName(n);
        super.setEmployees(e);
        inventory = inv;
    }

    public int getInventory() {
        return inventory;
    }
}
```

Using *super* to Indicate Base Class Methods

When an object of the ClothingStore class is instantiated, the constructor for the Business class is called first. The ClothingStore constructor executes next. When a ClothingStore object is destroyed, it is destroyed first and then the Business object. When a base class constructor requires parameters, the derived class must provide the required arguments to the base class constructor and call the constructor. In the example below, the Rectangle class constructor requires two integers as parameters for the two side lengths.

```
public class Rectangle {

    private int side1;
    private int side2;

    public Rectangle(int s1, int s2) {

        side1 = s1;
        side2 = s2;
    }

    public int getSide2() {
        return side2;
    }

    public int getSide1() {
        return side1;
    }
}
```

The Square class below which extends the Rectangle class only requires one integer as a parameter, but must call the Rectangle constructor and pass the two required arguments. Also note that the *getArea()* method obtains the base class attributes for the side lengths through the accessor methods since they are private.

```
public class Square extends Rectangle {

    public Square (int s) {

        super(s,s);     // call Rectangle constructor

    }

    public int getArea() {
        return super.getSide1() * super.getSide2();
    }
}
```

Note in the example above that there is no area attribute declared to store the value. It is always computed. If there were an attribute that stored the value, and a side length was changed, the area attribute would need to be updated or it would be incorrect. This is referred to as ***stale*** data. Some values should be calculated when needed and not stored in attributes.

Protected Access

In the Rectangle/Square example above, the Square class used the Rectangle accessor methods to obtain the values for the side length attributes to compute the area. The Square class could not access them directly because they were declared as private. The protected access specifier can be used in place of private to allow access by derived classes. However, this should be done with caution. If there is a public mutator in the derived class, the method would provide direct access to the private attribute from outside the derived class. As an example, the access specifier for the `name` attribute has been changed in the Business class.

Ex. 9.3 – Business Class Protected Access Specifier

```
public abstract class Business {

    protected String name;
    private int employees;
```

A derived Theater class can now assign a value to name directly in its constructor since it is protected and derived classes can access it. The employees attribute is still private and the mutator method is needed to assign a value.

```
public class Theater extends Business {

    private int numSeats;

    public Theater(String n, int e, int seats) {

        super.name = n;
        super.setEmployees(e);
        numSeats = seats;
    }

    public void setName(String n) {
        name = n;
    }
```

However, there is also a public mutator in the Theater class to change the name. This provides direct access by the program and removes the protection for the name attribute as shown by this example.

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

    Theater t1 = new Theater("Cinema 360", 18, 210);
    t1.display();

    t1.setName("CHANGED");

    t1.display();
}
```

```
The Cinema 360 Theater has 210 seats
The CHANGED Theater has 210 seats
```

Program Output

When the protected access specifier is used, public mutators should be reviewed in the base class and any derived classes to ensure that the attributes are truly protected. This is a consideration during initial design and when changes are made to the class or derived classes. Of the three access specifiers for attributes, protected is used the least and private is used the most.

Overriding Inherited Methods

When designing derived classes, a review will be made of the base class to determine what is being inherited. In some cases, the derived class may inherit a method that is inadequate. Modifying the base class' method may not be possible because subclasses have attributes that are specific to them. As an example, a display method has been added to the Business class below and is called after an object of the class is created. Since inheritance is a "one-way street", the Business class does not know anything about the derived class and cannot display the inventory for the clothing store.

Ex. 9.4 – Business Class Display Method Omits the Inventory

```
public class Business {

    private String name;
    private int employees;

    public void display() {
        System.out.println(name + " has " + employees + " employees");
    }

public static void main(String[] args) {

    ClothingStore c1 = new ClothingStore("Jenny's", 12, 24);

    c1.display();

}
```

```
Jenny's has 12 employees
```

Program Output

To resolve this situation, a display method can be added to the ClothingStore class that would include the inventory and would ***override*** the base class method. A method that overrides another method has the same signature (name, parameter list, and return type). This is not the same as overloading a method where the parameter lists are different. When the method is invoked on a ClothingStore object, the ClothingStore class method is used not the method in the Business class. This is referred to as ***polymorphism***, or the ability to take on multiple forms. The method that is called is always determined by the type of the actual object at runtime. This is referred to as ***dynamic method lookup***, and

allows objects to be treated the same way, even though the actual action taken may be different. The display method below would override the display method inherited from the Business class.

Ex. 9.5 – Business Class Display Method Overridden in ClothingStore

```
public class ClothingStore extends Business{

    private int inventory;

    public ClothingStore(String n, int e, int inv) {

        super.setName(n);
        super.setEmployees(e);
        inventory = inv;
    }

    public void display() {
        System.out.println(getName() + " has " +
                                getEmployees() + " employees");
        System.out.println("Inventory is " + inventory);
    }


public static void main(String[] args) {

    ClothingStore c1 = new ClothingStore("Jenny's", 12, 24);

    c1.display();

}
```

```
Jenny's has 12 employees
Inventory is 24
```

Polymorphism

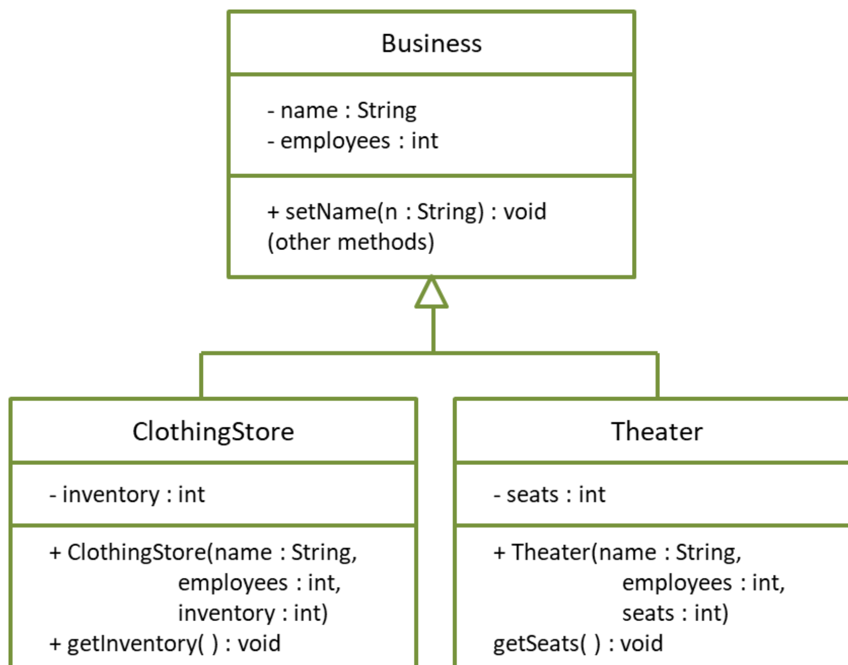
Technical Notes

To prevent a base class method from being overridden by derived classes, the method can be declared with the *final* key word similar to a constant variable declaration.

Another way of handling the display method issue is to call the base class method from the overriding method first before adding the inventory output. This eliminates the need to access the base class attributes through the accessors, and the output would be the same. The *super* key word is required in this case.

```
public void display() {
    super.display();
    System.out.println("Inventory is " + inventory);
}
```

In the Unified Modeling Language (UML), inheritance is represented with an open arrow head pointing to the base class. The base class diagram would include all of its' attributes, and each of the derived classes would contain theirs. A partial UML diagram for the Business Class hierarchy is shown below.



UML Diagram - Inheritance

For the Business examples above, the classes were modified multiple times to introduce various aspects of OOP. Typically, the design would be completed

before the implementation including determining the nouns and verbs for classes, and which class should contain which attributes.

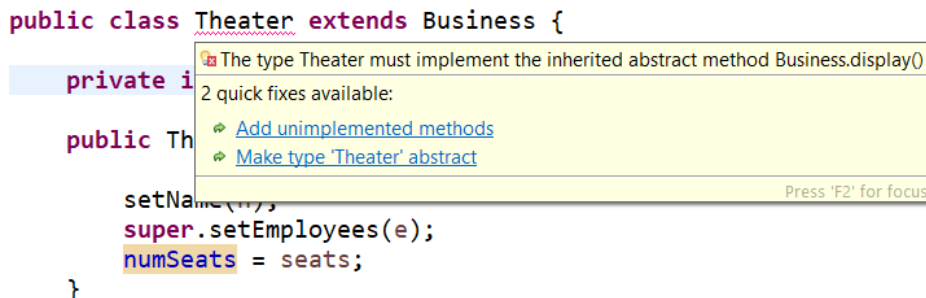
Abstract Classes and Methods

To force a subclass to override a method of the base class, the method can be declared as abstract. An abstract method does not have a method body. No implementation is provided, only the declaration. As an example, consider that the Business program has grown to include more businesses and the display method in the Business class is now considered completely inadequate. The method can be declared as abstract which would force each of the derived classes to include a display method of their own. Since the abstract method does not have an implementation, the Business class becomes abstract. An instance of the base class cannot be created since there is a method within it that has no implementation. In example 9.5, the display method of the Business class has been declared as abstract which makes the class abstract as well.

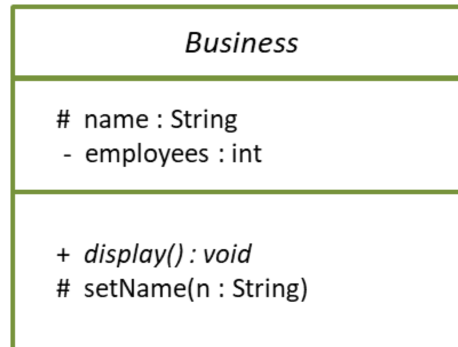
Ex. 9.5 – Abstract Business Class and Method

```
public abstract class Business {

    protected String name;
    private int employees;

    public abstract void display();


    protected void setName(String n) {
        name = n;
    }
```

Creating a Theater class that extends Business now requires the abstract method to be implemented or the Theater class would have to be an abstract class as well.



Abstract classes in UML are shown with italicized class and method names. The abstract Business class is shown below. Note that a pound sign precedes the protected attribute and method.



UML Diagram – Abstract Class and Method

Final Classes and Methods

To prevent a class from being extended, it can be declared with the final modifier. This prevents derived classes from being created and inheritance is not possible. Note that a class cannot be both abstract and final, since an abstract class must be extended and a final class cannot be extended.

```
public final class myClass () {

    // cannot be extended
}
```

To prevent overriding of a method in a class, it can be declared with the final key word as well.

```
public class anotherClass() {

    public final double someMethod () {

        // cannot be overridden
    }
}
```

Final Classes and Methods

Aggregation

Objects are sometimes made up of other objects. This aligns with the practice of code reuse but establishes an association between the classes. When an instance of a class is an attribute of another class, aggregation occurs. This is referred to as the "has a" relationship. As an example, when creating a Student class an existing Address class could be added as an attribute of the Student class as shown below. The constructor of the Student class creates the Address object.

```
public class Student {

    private Address address;

    public Student () {

        address = new Address();
    }

    public String getAddress() {
        return address.getStuAddress();
    }

}
```

The Address class attribute of the Student class is shown below. Note that it is a public class and in a separate file.

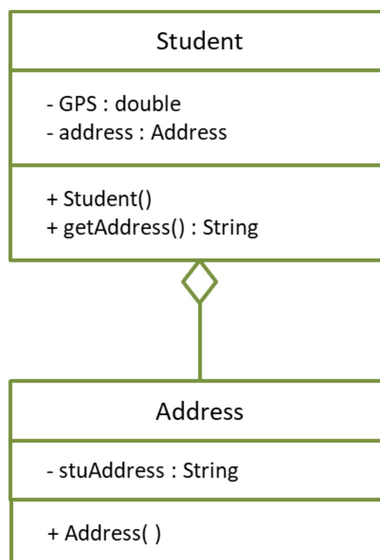
```
public class Address {

    private String stuAddress;

    public Address() {

        setStuAddress("123 Main Street");
    }

}
```

Another way of accomplishing the "has-a" relationship is to add the Address class to the Student class as an ***inner class*** (sometimes referred to as a subclass). The Address class would be declared private (in Java there can be only one public class per file), and the attributes would only be accessible from the outer class. In example Ex. 9.6, the Address is a private attribute of the Student class. The Student class constructor calls the Address constructor creating the object. Note that the *getAddress()* method accesses the stuAddress attribute of the Address class.

Ex. 9.6 – Student Class with Address Inner or Subclass

```java
public class Student {

    private double GPA;

    private Address address;

    public Student () {

        address = new Address();
    }

    public String getAddress() {
        return address.stuAddress;
    }


   private class Address {

       private String stuAddress;

       public Address() {

           stuAddress = "123 Main Street";

       }

   }
```

Aggregation in UML is represented with an open diamond and a line connecting the classes. The diagram below shows the Student class as the aggregate, and the Address class as the subclass which is also listed in the Student class attributes.



UML Diagram – Aggregation

Class Substitution

Since a derived class inherits everything from a base class, a derived class can be passed to a method that has the base class as a parameter. As an example, consider the following simple base class and derived class.

```
public class BaseClass {

    public BaseClass() {

    System.out.println("Base class");

    }
}

public class DerivedClass extends BaseClass {

    public DerivedClass() {

        System.out.println("Derived class");

        }
}
```

The main method creates an instance of the derived class, and passes it to a method that has the base class as the parameter. Notice the output also shows the order of operations. The constructor for the base class executes first, and then the derived class constructor.

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

    DerivedClass d = new DerivedClass();
    acceptEither(d);
}

static void acceptEither(BaseClass b) {

    System.out.println("In acceptEither. ");
}
```

```
Base class
Derived class
In acceptEither.
```

Program Output

Determining Class Identity

As classes become more complex and methods override other methods, it is often necessary to determine if an object is an instance of the base class or a derived class. Java provides the *instanceof* operator which returns true or false. The general format is shown below.

```
if(stu instanceof Student) {

    System.out.println("IS A STUDENT OBJECT");

}
```

Technical Notes

Experts agree that programmers have a tendency to overuse Inheritance. Technically, inheritance should be used when objects behave differently, not because their values differ. If the solution can be implemented easily with a single class, then that is preferred.

Interfaces

The interface is an abstract type that is used to designate a set of abstract methods for classes to implement. All of the methods in an interface are static methods and have no implementation (an exception is a default method shown later). An interface cannot be instantiated as an object, and classes that use an interface must inherit all of the abstract methods declared and implement (override) them. All of the attributes declared in an interface are final (constants) and must be initialized in the interface. In the example below, an interface has been defined with the abstract *animalMove()* method. The word *public* can be omitted because all interface methods must be public. Any class that implements the interface must have an implementation of the *animalMove()* method with the same signature (name, parameters, and return type). The relationship between an interface and a class that uses it is known as a *realization relationship*. The class ***realizes*** the interface.

```
public interface AnimalInterface {

    public void animalMove();

}
```

The Rabbit and Bird classes below use (realize) the interface and include ***implements*** and the name of the interface in their class declarations. The classes must implement the *animalMove()* method since they inherit it and it is abstract. If a class inherits an abstract method and does not override it, the class then becomes abstract as well and cannot be instantiated.

```
public class Rabbit implements AnimalInterface {

    public void animalMove() {

        System.out.println("hop...hop...hop...");
    }
}

public class Bird implements AnimalInterface {

    public void animalMove() {

        System.out.println("flap...flap...fly...");
    }
}
```

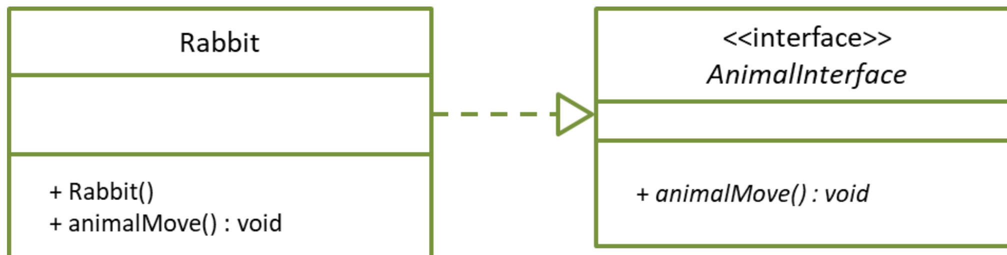
The main method for the program creates an object from each class and the appropriate method for that object is called.

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

    Rabbit rabbit =  new Rabbit();
    Bird bird = new Bird();

    rabbit.animalMove();
    bird.animalMove();
}
```

```
hop...hop...hop...
flap...flap...fly...
```

Program Output

In UML, the << interface >> tag is included with the name of the interface, the interface name and methods are italicized, and a dotted line connects classes that use the interface.



UML Diagram – Interface

As mentioned above, all of the attributes declared in an interface are final (constants) and must be initialized. When declaring an interface constant, the standard for all uppercase letters is used, and there is no need to include public static final since all variables in an interface are automatically public static final.

```
public interface InterfaceName {

    int SOME_CONSTANT = 10;
}
```

To use the constant in a program, the name of the constant is preceded by the name of the interface, and the dot operator.

```
InterfaceName.SOME_CONSTANT
```

Static methods in an interface include the implementation and are called using the name of the interface and the dot operator.

```
public interface InterfaceName {

    static double someMethod () {
        // method body
    }
}

double returnVal = InterfaceName.someMethod();
```

Default methods include the implementation and the default key word. If a class that uses the interface does not override the method, it inherits the default method as is. Note that this particular type of interface method has a body. To use the method, the interface name and dot operator precede the method name.

```
public interface InterfaceName {

    default double someMethod () {
        // method body
    }
}
```

Default methods can be added to an existing interface without affecting classes that implement the interface. Class methods can be modified to use or override the default method, but it would not cause an error if they simply ignored the default method.

Technical Notes

Interfaces are often used to implement common operations on different types of objects, and although Java does not support multiple-inheritance, a class can implement multiple interfaces achieving a similar result.

Functional Interfaces and Lambda Expressions

A functional interface has a single abstract method. Rather than define a class that implements this interface, a lambda expression can be used to create an object that implements the interface and overrides the abstract method. A lambda expression is a short block of code that receives parameters and returns a value. Lambda expressions are similar to methods, but they do not need a name and the implementation can be the body of a method. A simple lambda expression has a single parameter and an expression.

```
param -> expression
```

When there are multiple parameters, they are enclosed in parentheses.

```
(param1, param2) -> expression
```

Lambda expressions cannot have variables, assignment statements, or conditional expressions, and they can only be assigned to a reference of a functional interface.

```
param -> { return param * 2 ;};
```

To use a lambda expression, an abstract method is declared within an interface that includes the return type and parameter(s). The name of the method is used when calling the overridden method (shown below).

```
public interface MyLambda {

    double circ(double r);

}
```

To define the lambda expression, the parameter and method statements are assigned to a reference to the interface. Recall that an interface cannot be instantiated. This is where the body of the method is defined.

```
InterfaceName reference = param -> method statement(s)

MyLambda lamb = c-> {return 2 * Math.PI * c;};
```

In the statement above, a reference to the interface is declared as lamb, and the parameter as c which is also used in the method body. An example that uses the lambda is shown below.

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

    Scanner in = new Scanner(System.in);

    MyLambda lamb = c-> {return 2 * Math.PI * c;};

    System.out.print("Enter the radius ");
    double r = in.nextDouble();

    System.out.print("Circumference is " + lamb.circ(r));
}
```

```
Enter the radius 3
Output is 18.84955592153876
```

Program Output

A lambda can be redefined as long as the parameter list and return type are the same. As an example, the following interface is defined that accepts an integer as a parameter and returns an integer.

```
public interface NewLambda {

     int newLambda (int y);
}
```

The lambda is defined first to square the number and then to cube the number.

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

    Scanner in = new Scanner(System.in);
    System.out.print("Enter an integer. ");
    int t = in.nextInt();

    NewLambda lamb = r-> {return r * r;};

    System.out.println("The value squared is " + lamb.newLambda(t));


    NewLambda lamb2 = g-> {return g * g * g;};

    System.out.println("The value cubed is " + lamb2.newLambda(t));
```

```
Enter an integer. 3
The value squared is 9
The value cubed is 27
```

Program Output

Programming Style and Standards

One of the purposes of employing Programming Standards and Style Guides is to ensure readability and maintainability. Introducing unnecessary derived classes, subclasses or interfaces is counterproductive and should be avoided.

Removing Clutter

As programs are enhanced and modified, they tend to grow in size and level of complexity, and the code becomes cluttered with duplication and inefficiency. Programmers tend to be reluctant to revisit the design and implementation of a program and clean it up. ***Refactoring*** is a tool that is used to improve existing

code without changing it functionally. The goal is to enhance and simplify the code to reduce the time and effort required to change or add functionality in the future. Refactoring can also extend the life of code that has become increasing difficult to maintain.

Lambda Expression Guidelines

A lambda expression should be concise, self-explanatory, and single-line as opposed to a block of code. The compiler is able to determine the data types of the parameters by *type inference* so adding them is optional, but clearer.

```
(a, b) -> a.length() + b.length();

(String a, String b) -> a.length() + b.length();
```

With one-line lambdas, braces and return statements are optional and can be omitted, but adding them enhances readability.

```
a -> a.length();

a -> {return a.length() };
```

Overriding and Overloading Guidelines

One common error is overloading a method instead of overriding. This occurs when a parameter is changed or is added or omitted when overriding, and the method is then overloaded. The definitions are repeated here for convenience.

Overloading – two methods have the same name, but different parameter lists.

```
void display(double a, double b) {

}

void display(double a, int b, String c) {

}
```

Overriding - when a derived class defines a method with the same name and exactly the same parameters as the base class (ref example Ex. 9.5). Overriding is required when a derived class inherits an abstract method.

Chapter 9 Review Questions

1. _____________ is the relationship between generalized and specialized classes.
2. A _________________ extends the base or super class. This is referred to as the "is a" relationship.
3. Three access specifiers for classes are ___________, ___________, and ___________
4. The ____________ access specifier prevents direct access to an attribute.
5. The _____________ access specifier allows access to base class attributes by derived classes.
6. The ____________ access specifier allows universal access to an attribute or method.
7. A derived class can ____________ an inherited method and implement it a different way.
8. The ability to take on many forms is called __________________.
9. __________ _________ ___________ allows objects to be treated the same way, but with different actions.
10. To prevent a class method from being overridden, it can be declared with the __________ modifier similar to a constant variable.
11. In a UML diagram, inheritance is represented with an open ______________ pointing to the base class.
12. A method declares as ____________ in the base class must be implemented in a derived class.
13. When an instance of a class is an attribute of another class, ______________ occurs.
14. Since a derived class inherits all of the attributes of the base class, it can be ______________ as an argument to a method that has the base class as a parameter.
15. The ______________ method can be used to verify class identity.
16. A(n) _____________ is an abstract type that is used to designate a set of abstract methods for classes to implement.
17. A ___________ expression is a short block of code that receives parameters and returns a value.

18. _______________ is a tool that is used to remove duplication and inefficiencies, and improve existing code without changing it functionally.

Chapter 9 Short Answer Exercises

19. In the following declaration, what is the base class?

```
public class Racoon extends Animal {
```

20. In the following declaration, what is the derived class?

```
public class Racoon extends Animal {
```

21. In the class below, what key word can be used to indicate that the methods in the constructor for ClothingStore are inherited from the Business class?

```
public class ClothingStore extends Business{

    private int inventory;

    public ClothingStore(String n, int e, int inv) {

        setName(n);
        setEmployees(e);
        inventory = inv;
    }
```

22. In the Square class below, why is there a method *getArea()* and not an area attribute?

```
public class Square extends Rectangle {

    public Square (int s) {

        super(s,s);     // call Rectangle constructor

    }

    public int getArea() {
        return super.getSide1() * super.getSide2();
    }
}
```

23. What attribute in the class below would be directly accessible by a derived class?

```
public abstract class Business {

    protected String name;
    private int employees;
```

24. What attribute in the Business class above would not be accessible by a derived class?

25. In the Business class above, what does the word abstract indicate?

26. What is the abstract method in the Business class below?

```
public abstract class Business {

    protected String name;
    private int employees;

    public abstract void display();


    protected void setName(String n) {
        name = n;
    }
```

27. Why can an object of the Business class above not be instantiated?

28. Can an object derived from the Business class above access the *setName()* mutator method, and why or why not?

29. In the Business class above, should the name attribute access specification be changed to private, or should the setName() method be deleted?

30. In the following class, what is the Address attribute?

```
public class Student {

    private double GPA;

    private Address address;

    public Student () {

        address = new Address();
    }
```

31. If a Student object "st1" were created from the class definition below, how would the *getAddress()* method be called?

```
public class Student {

    private double GPA;

    private Address address;

    public Student () {

        address = new Address();
    }

    public String getAddress() {
        return address.stuAddress;
    }


    private class Address {

        private String stuAddress;

        public Address() {

            stuAddress = "123 Main Street";

        }

    }
```

32. In the following class, what is the name of the interface?

```
public class Message implements Analyzer {
```

Chapter 9 Programming Exercises

33 (a) Create a program with the class definition below for Aircraft and include accessors and mutators for the attributes, and the *display()* method shown.

```
public class Aircraft {

    private int range;
    private int fuelCapacity;

    public Aircraft() {

        range = 0;
        fuelCapacity = 0;
    }
}
```

```
public void display() {
        System.out.println("Aircraft range: " + range);
        System.out.println("Aircraft fuel capacity: " + fuelCapacity);

}
```

33 (b) Add a derived class to the program named Commercial with a private instance variable named seats and a *display()* method that overrides the display method in Aircraft (see output below). Write a constructor for the derived class with parameters for range, fuel capacity, and seats. In the main method, create an instance of the Commercial class with 23 seats, a range of 1500 miles, and a fuel capacity of 665 gallons, and call the *display()* method.

```
Commercial Aircraft
Aircraft range: 1500
Aircraft fuel capacity: 665
Aircraft seating: 23
```

33 (c) Add a class definition to the program for a class named Cargo that is derived from the Aircraft class. Include an instance variable for payload that is initialized by the constructor, and a *display()* method to override the inherited method as shown. In the main method, create an instance of the Cargo class with a payload of 3680 pounds, a range of 870 miles, and a fuel capacity of 335 gallons, and call the *display()* method.

```
Cargo Aircraft
Aircraft range: 1500
Aircraft fuel capacity: 665
Aircraft payload: 3680
```

33 (d) Modify the Aircraft base class by deleting the body of the *display()* method and make it abstract, and add the abstract modifier to the class.

33 (e) Add a static variable called planes to the Aircraft base class and modify the constructors of the derived classes to increment the variable. Add an output statement to the main method to display the number of planes before and after they are created.

33 (f) Draw a UML diagram of the Aircraft and Commercial classes, their attributes and methods, and indicate their relationship.

Chapter 9 Programming Challenges

#1 Base and Subclasses

Create class named Base that has a static integer objects and two (2) private subclasses named Sub1 and Sub2. The constructors for the subclasses will increment the static variable in the Base class. The constructor for the Base class will create an instance of each of the subclasses.

Use the main method below to create instances of the Base class and display the value of the static variable.

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

    Base b1 = new Base();
    Base b2 = new Base();

    System.out.println("Instances : " + Base.objects);

}
```

#2 Lambda Expression

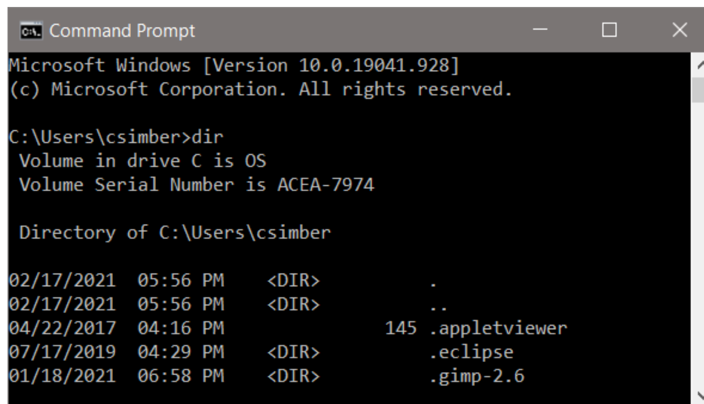
Write a program with the public interface named addTwo shown below, and implement the expression in the main method as a lambda to accept two integers and return the sum. Use the expression in an output statement.

```
public interface addTwo {

    int add(int n1, int n2);
}
```

Chapter 10

Graphical User Interfaces

Graphical User Interfaces (GUIs) were originally created by researchers at the Xerox PARC (Palo Alto Research Center), were adopted by OS developers, and quickly became the user-preferred choice for interfacing with computers in the 1980’s. Prior to this, command line interfaces (shown below) were used to interact with computers, and in some areas they continue to be used.



Command Console Interface

The graphical user interfaces commonly used are event driven by the user. A user may click on a button or tab, scroll information, or resize a window. The

program responds to user input and the user determines the sequence of many of the events. Therefore, careful design is required to control access to the events. For instance, a user may click a button to compute a result before entering required values. Scenarios like this should be considered during interface design, since they increase the input validation aspects of a program. A value needed for computation must be entered by the user before allowing computation, and the value entered must be within the correct range of values 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 must be a number
3. The radius input must be a positive number

The graceful handling of incorrect input is required for a user-friendly and well-engineered solution. In a non-GUI program, we might use a loop that iterates until a correct value is entered. It would display an error message to alert the user, and re-prompt for input inside the loop. The same operations are used in a GUI program, but with the added requirement of graphically handling the tasks. Generating a main GUI interface requires creating a window and positioning ***components*** (or controls) on it. A control is an element that enables a user to accomplish some function or to access an area of the program. They are commonly referred to as components.

Java is well suited for creating graphical user interfaces, and many Java packages and libraries contain components that are easy to use including buttons, frames, labels, panels, and more to develop user friendly interfaces.

The **Abstract Window Toolkit** (AWT) was Java's first package for creating Graphical User Interfaces (GUIs). It was available in Java 1.0 (1996) and uses a peer approach. 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 in the javax.swing package are part of Oracle’s Java Foundation Classes which provide a user interface for Java programs. It is much more extensive than the AWT and matches the look and feel of various platforms.

The Java swing components include:

Component	Description
Button	causes an action or event when clicked
Check Button	On or Off position check boxes
Text Area	display area for text
Text Entry	single line entry control
Frame	container that can hold components
Label	area that displays one line of text
List Box	user selection list (option-list)
Menu	list exposed when a menu button is clicked
Panel	rectangular area for frames and components
Radio button	select/deselect component

Java Swing Components

Before selecting components, a preliminary design should be completed as a sketch. This allows for changes to the layout for the window and an idea of how it will look and operate prior to writing any code. Walking through the program operations step-by-step as a user is also helpful at this stage, and can surface design changes prior to development. The user will interact with the program through the GUI, and it should be user friendly, have intuitive controls, and labels for instruction. A user should not have to wonder how to use the program or what units to enter.

The components required for the interface depend on what the program does and the user interaction. A few labels and buttons may be adequate, or radio buttons or option lists may be best. These considerations during the design phase will save time redesigning or reconfiguring an inadequate or problem interface. Programmers often overlook essential aspects of the interface since they know what the program does, how it functions, and the inputs required. The Agile development process involves stakeholder reviews and in some cases

the customer. This provides an opportunity for people not familiar with the planned design to offer suggestions for improvement, and eliminates surprises when the final product is delivered.

Interface Design Example

Consider a weather program with a GUI that receives user input for temperature and wind speed, and computes the wind chill factor when a button is clicked. The pseudocode for the program lists the steps in the program and helps to identify the order of operations and the components needed for the interface.

Ex. 10.1 – GUI program – Wind Chill Factor

```
Step 1  the user enters temperature            data entry
Step 2  the user enters wind speed             data entry
Step 3  the compute button is clicked          button
Step 4  the input is validated
          -   if the input is valid
                o   compute the wind chill
                o   display the result         label
          -   otherwise
                o   alert the user to the error   dialog box
                o   clear the inputs
Step 5  go to Step 1
```

A sketch of the planned interface helps to verify the controls and components that will be needed, and where they should be located.

Enter the temperature in Fahrenheit:

Enter the wind speed in miles-per-hour:

The wind chill factor is:

Compute

Preliminary GUI Sketch

Windows (Frames)

Programmers should use an object-oriented approach to GUI development. The example below defines a class with a window (frame) using a JFrame which is a container in Java that can hold components. Inheritance is used and the JFrame class will be *extended*, allowing the new class to inherit all of the members of the JFrame class such as *setSize()* and other methods. The line numbers are included for the explanations that follow.

Ex. 10.1 – A Simple Window

```
 1. public class SimpleWindow extends JFrame {
 2.
 3.     JFrame myFrame;
 4
 5.     public SimpleWindow() {      // Constructor
 6.
 7.         myFrame = new JFrame();
 8.         myFrame.setSize(300,400);
 9.         myFrame.setLocationRelativeTo(null);
10.         myFrame.setVisible(true);
11.         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12.     }
13. }
```

Line 1 begins the class definition and extends JFrame

Line 3 defines the JFrame

Line 5 is the constructor for the class

Line 7 creates a frame and assigns it to myFrame

Line 8 sets the initial size of the frame – *setSize(width, height)*

Line 9 places the frame in the center of the display area

Line 10 makes the frame visible (the default is false)

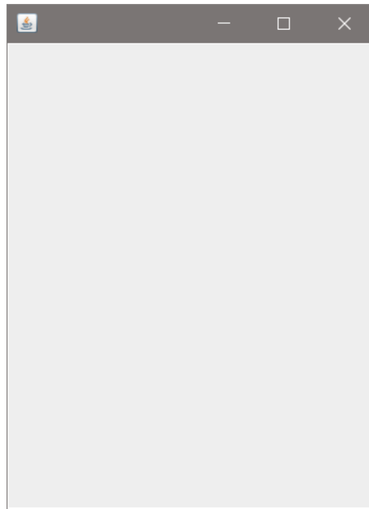
Line 11 sets the default close operation to *exit* so that if the window is closed the program ends

Technical Notes

Java will display components as small as possible to fit whatever is placed on them. There are multiple size setting methods to resolve this including setSize(), setPreferredSize(), and setMinimumSize() which are used depending upon the component.

Example Ex. 10.1 defined the class for the window. To create an instance of this class, an object of the class is created using 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();

}
```



Program Output

Example Interface Window

There are three lines of text, two text entry boxes, and a button to compute the wind chill factor planned for the example window. These components will need to be defined, created, and positioned on the frame (window) for the interface.

Enter the temperature in Fahrenheit:

Enter the wind speed in miles-per-hour:

The wind chill factor is:

Compute

Positioning Components

To position the components for an interface, they are placed on a JPanel, and positioned with Java layout managers. The default is the Flow layout manager which locates components left to right in the order they are added. Each of the layout managers has benefits and limitations, but all provide control over the locations of components. A partial list of layout managers is shown below, and Appendix G demonstrates the use of multiple layout managers and panels. Other layout managers are being introduced in JavaFX as well. The ***setBounds()*** method can be used for quickly locating components and accepts four arguments for positioning. The first two arguments are the x and y coordinates of the top left corner of the component, the third is the width, and the fourth is height. The panel layout must be set to null to override the default flow layout.

Java Layout Managers (partial list)

- BorderLayout – North, South, East, and West
- BoxLayout – vertical or horizontal arrangement
- FlowLayout – left to right
- GridBagLayout – row and column positioning and size variation
- SpringLayout – arrange using constraints

Depending on the layout manager, resizing a frame can shift the locations of the components. The ***setResizable()*** method is used to prohibit resizing the window.

```
myFrame.setResizable(false);
```

Iterative Enhancement

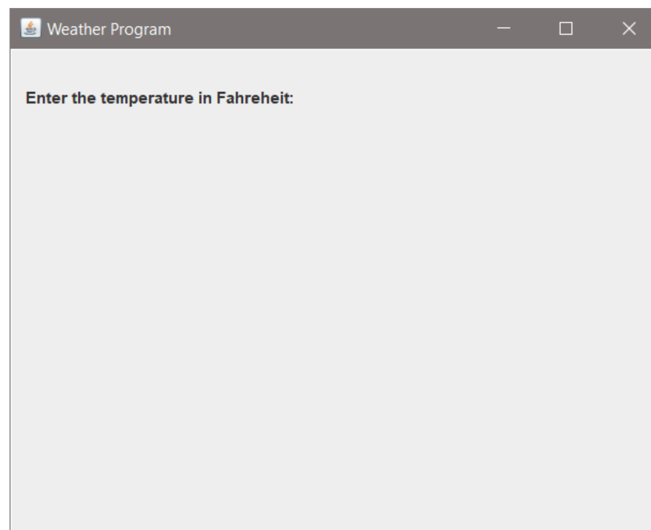
A project should be developed in steps while testing periodically. This is often referred to as iterative enhancement. A portion of code is developed and tested, and as new code is added, any errors that surface would be in the added code. The example below places a single label on a panel, and the panel on a frame. Proper naming conventions for variables and components, and using a step-by-step approach will save time and effort as the program becomes more complex.

In example Ex. 10.2 below, the frame, panel, and label are declared first. In the constructor, the frame is created, sized, and a title is added, the panel is created with the layout assigned null since *setBounds()* will be used for locating the components, and the label is created last and positioned. Finally, the label is

added to the panel, the panel to the frame, and the frame is made visible. The main method creates an instance of the class. Note the methodical approach to the code.

Ex. 10.2 – A Frame with a Panel and Label

```
public class WeatherWin {

    JFrame wFrame;
    JPanel wPanel;
    JLabel tempLbl;

    public WeatherWin() {          // Constructor

       wFrame = new JFrame();
       wFrame.setSize(500,400);
       wFrame.setTitle("Weather Program");

       wPanel = new JPanel();
       wPanel.setLayout(null);

       tempLbl = new JLabel("Enter the temperature in Fahrenheit:");
       tempLbl.setBounds(10,10,200,50); // x, y, width, height

       wPanel.add(tempLbl);
       wFrame.add(wPanel);

       wFrame.setVisible(true);

    }
```

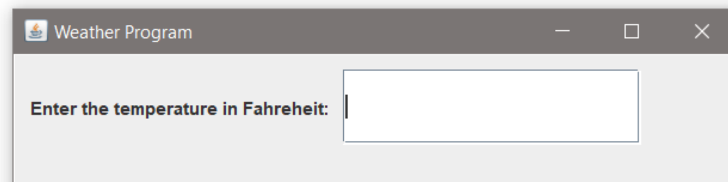


Program Output

Text Entry Controls

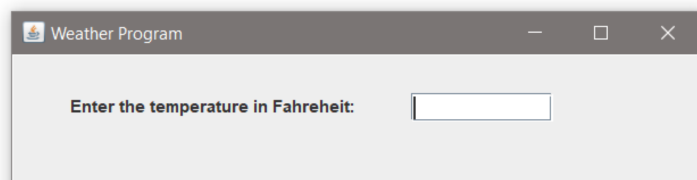
The text entry control in Java is a JTextField and it will be positioned next to the label requesting the input in the example. The text field constructor can accept two arguments. The first argument is optional and can contain default text to show in the field. The second is the character width of the field to display. In the code below, the text field is positioned at x = 220 to move it to the right of the prompting label, and y = 10 which is the same y coordinate for the label. The width and height arguments are the same as well.

```
tempLbl = new JLabel("Enter the temperature in Fahrenheit:");
tempLbl.setBounds(10,10,200,50); // x, y, width, height

tempField = new JTextField(10);
tempField.setBounds(220,10,200,50);

wPanel.add(tempLbl);
wPanel.add(tempField);

wFrame.add(wPanel);
```



The display shows that the height argument of 50 for the label pushed the text down and centered it vertically within the 50 pixels, and that it is not the preferred height for the text field. The width argument for the text field was also overridden by the width argument for *setBounds()*. The code modifications to correct this are shown below. Some trial and error is to be expected.

```
tempLbl = new JLabel("Enter the temperature in Fahrenheit:");
tempLbl.setBounds(40,10,200,60); // x, y, width, height

tempField = new JTextField(10);
tempField.setBounds(280,27,100,20);
```



The other labels are created and positioned the same way, and the code is added in the order they are created for ease of maintenance. If Flow Layout were being used, the order in which the components were added would determine their position. The complete class without attributes is shown below.

Ex. 10.3 – The WeatherWin Class

```
public class WeatherWin {

        // Attributes omitted

        public WeatherWin() {       // Constructor

           wFrame = new JFrame();
           wFrame.setSize(500,400);
           wFrame.setTitle("Weather Program");

           wPanel = new JPanel();
           wPanel.setLayout(null);

           tempLbl = new JLabel("Enter the temperature in Fahrenheit:");
           tempLbl.setBounds(40,10,200,50);     // x, y, width, height

           tempField = new JTextField(10);
           tempField.setBounds(280,27,100,20);

           windLbl = new JLabel("Enter the wind speed in MPH:");
           windLbl.setBounds(40,60,200,50);     // x, y, width, height

           windField = new JTextField(10);
           windField.setBounds(280,77,100,20);

           wPanel.add(tempLbl);
           wPanel.add(tempField);

           wPanel.add(windLbl);
           wPanel.add(windField);

           wFrame.add(wPanel);

           wFrame.setVisible(true);
    }
}
```

Buttons

The JButton component has a label for text and various customizing methods: *setSize()*, *setBackground()*, and *setForeground()*. They are created and positioned similar to other components. The example has a button that will cause the wind

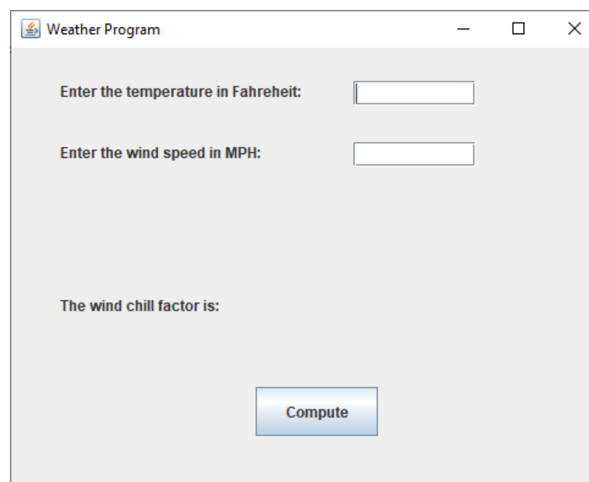
chill to be computed from the inputs in the text entry fields. This will require that the program react when the button is clicked and obtain the text entered by the user. JButtons generate an event when clicked that can be "listened" for using an ActionListener interface.

> *Technical Notes*
>
> Every mouse movement or action in a GUI generates an event. Specific events can be captured and responded to by the interface. Others can be disregarded. Listeners are used to *catch* specific events from specific event sources.

The code for creating the button with "Compute" on its label, and positioning the button on the panel is shown below with the updated display. The third and fourth arguments passed to the *setBounds()* method determine the size of the button in pixels, and will override other size methods used with buttons.

Ex. 10.4 – Adding the JButton

```
compBtn = new JButton("Compute");
compBtn.setBounds(190,277,120,40);

wPanel.add(tempLbl);
wPanel.add(tempField);
wPanel.add(windLbl);
wPanel.add(windField);
wPanel.add(chillLbl);
wPanel.add(compBtn);
```



Program Output

The interface is now complete, although nothing happens when the button is clicked. It will require an ActionListener. An action listener must implement the ActionListener interface and the *actionPerformed()* method that receives the event object which contains information about the event.

```
public interface ActionListener {

    void actionPerformed(ActionEvent event) {

    }
}
```

ActionListener Interface

Ex. 10.5 – Implementing the Listener for the Button

The lines below declare a listener class named ClickListener that implements the interface. Notice that the action is an output statement indicating that the button was clicked for testing purposes.

```
public class ClickListener implements ActionListener {

    public void actionPerformed(ActionEvent event) {

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

Once an action listener has been written, it must be assigned to the button. This code constructs an instance of the ClickListener class above and assigns it to the button for the example using the *addActionListener()* method.

Ex. 10.6 – Assigning the Listener to the Button

```
ActionListener Compute = new ClickListener();

compBtn.addActionListener(Compute);
```

Note that the listener must be implemented before it can be assigned and is typically written as an inner class (subclass). This is covered in the next section.

Subclass ActionListener

The code for the implementation of the listener class, the creation of the object, and the assignment of the listener to the button can be located in various places within the program. In most cases, an ActionListener that is an inner class or subclass of the GUI is appropriate. The listener is defined within the class and assigned to the button as shown below. Note that the class definition does not have an access specifier. Different implementations may use different specifiers.

Ex. 10.7 – The Button ActionListener as a Subclass of the GUI

```
        wPanel.add(chillLbl);
        wPanel.add(compBtn);

        wFrame.add(wPanel);

        wFrame.setVisible(true);


    class ClickListener implements ActionListener {

        public void actionPerformed(ActionEvent e) {

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

    ActionListener Compute = new ClickListener();

    compBtn.addActionListener(Compute);

    } // end of constructor
} // end of class
```

The GUI for the example is now complete in terms of the components and a working button that displays "The button was clicked." when it is clicked. The code for obtaining and validating the input, and computing and displaying the wind chill will be written next. This code could be implemented within the *actionPerformed()* method, but that would place a large amount of code within the listener. Writing a method and calling it from within the action listener is a better approach, and aligns with modularizing the program. Depending on the size of the method, it could be located in another module.

For the example, a method named *computeWeatherData()* will be called when the button is clicked. The call replaces the output statement used previously.

```
class ClickListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        computeWeatherData();
    }
}
```

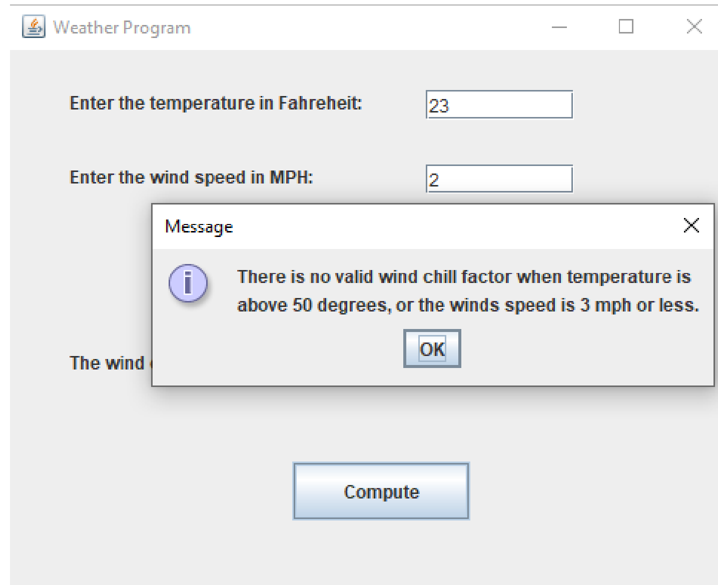
Method Call from ActionListener

The method will obtain the user input from the JTextFields using the *getText()* method which returns the text as a String. The Strings will need to be converted to numeric data for the computations using *Double.parseDouble()* and a try/catch block is required. The try block for the method is shown below.

```
public void computeWeatherData() {

    double tempF = 0;
    double windSpeed = 0;
    double wc = 0;

    try {
        String tempStr = tempField.getText();
        String windStr = windField.getText();

        tempF = Double.parseDouble(tempStr);
        windSpeed = Double.parseDouble(windStr);

        if(tempF > 50 || windSpeed <= 3.0) {
            JOptionPane.showMessageDialog(null,
                "There is no valid wind chill factor when temperature is \n" +
                    "above 50 degrees, or the winds speed is 3 mph or less.");
            chillLbl.setText("A valid wind chill factor cannot be computed. ");
        }
        else {

            double wsMod = Math.pow(windSpeed, 0.16);
            wc = 35.74 + 0.6215 * tempF - 35.75 * wsMod + 0.4275 * tempF * wsMod;
            chillLbl.setText("The wind chill factor is:   "
                    + String.format("%,.2f", wc) + " dF");
        }
    }
```

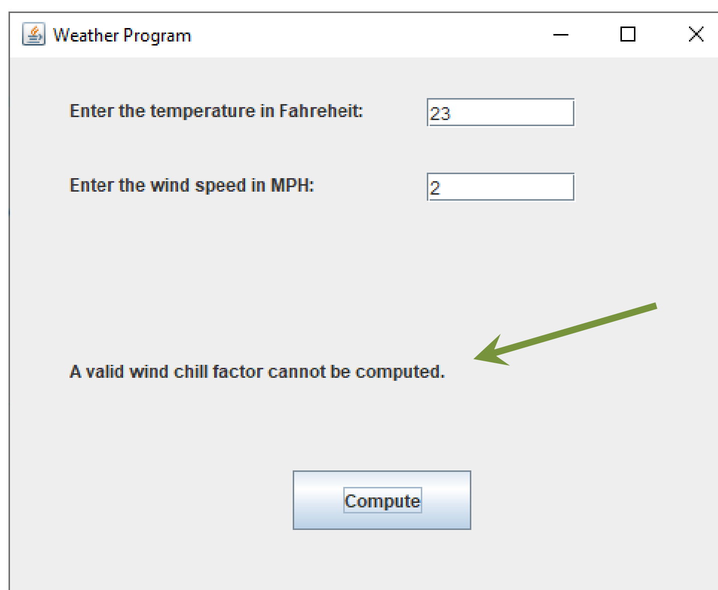
If the temperature or wind speed value is not within the proper range, a message dialog box appears to alert the user and the user is given the opportunity to correct the error without restarting the program. The code for the dialog is repeated below with sample output.

```
JOptionPane.showMessageDialog(null,
    "There is no valid wind chill factor when temperature is \n" +
        "above 50 degrees, or the winds speed is 3 mph or less.");
```



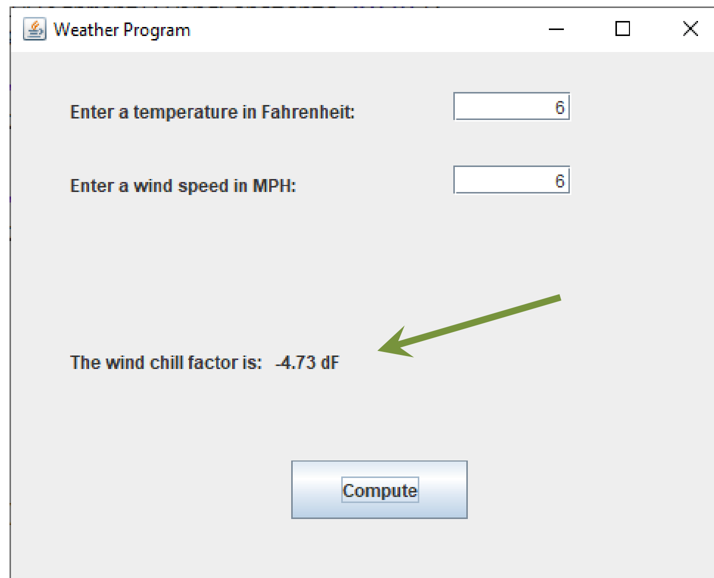
The output label is updated using the *setText()* method to indicate that the value could not be computed.

```
chillLbl.setText("A valid wind chill factor cannot be computed. ");
```



If the try clause executes with no errors and the wind chill factor is computed, the output label is updated using the *setText()* method to include the wind chill factor.

```
chillLbl.setText("The wind chill factor is:    "
        + String.format("%,.2f", wc) + " dF");
```



If the input cannot be converted by *Double.parseDouble()*, the try block will be exited and the catch clause will execute. A message dialog box will be displayed, and the output label is updated.

```
catch (NumberFormatException e) {
    JOptionPane.showMessageDialog(null,
            "An invalid value has been entered.",
            "Invalid Input", JOptionPane.WARNING_MESSAGE);
    chillLbl.setText("A valid wind chill factor cannot be computed. ");
}
```

The method used to right align the input in the text field is shown below. There are a variety of options for JTextFields including foreground and background color, fonts, and reacting to input directly with ActionListeners.

```
windField.setHorizontalAlignment(SwingConstants.RIGHT);
```

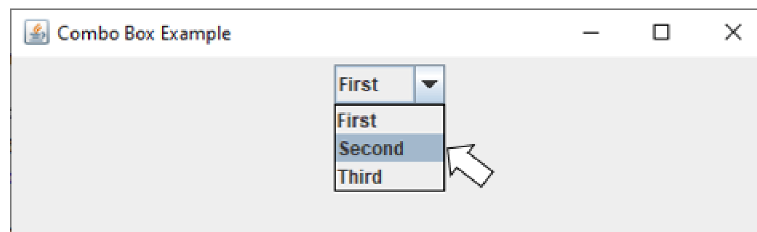
Other Input Controls

Many programs require that a user make a single selection or multiple selections from a specific set of options. Allowing users to enter choices manually invites mistakes and requires input validation and error handling. This can be avoided using a control that allows a single selection from choices that are mutually exclusive, or by allowing multiple selections from a specific set of options.

The Combo Box (option-list)

A combo-box provides mutually exclusive selections for a GUI application and can be positioned like other components. Items can be added to the combo box using the *addItem()* method and are listed in the order that they are added. When an item is selected, it replaces the read-only text field at the top of the box.

```
JComboBox<String> jcb = new JComboBox <String>();

jcb.addItem("First");
jcb.addItem("Second");
jcb.addItem("Third");
```



Combo Box Example

An array of Strings can also be used to provide the options. The array is passed to the JComboBox constructor as shown here.

```
String[ ] choices = { "First", "Second", "Third" };
JComboBox<String> dBox = new JComboBox<String>(choices);
```

To obtain the user selection, *getSelectedItem()* is used and is cast to a String.

```
String selection = (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 can be handled with an action listener. This may add complexity since the user may want to change their mind and select a different button. The *isSelected()* method resolves this and obtains the input at a determined time. In the example below, two radio buttons are created, added to a group, and then to a panel. To set a radio button by default, add *true* as a second argument after the text.

```
JRadioButton rad1 = new JRadioButton(“First”);
JRadioButton rad2 = new JRadioButton(“Second”);
```

Next, a group is created for the radio buttons.

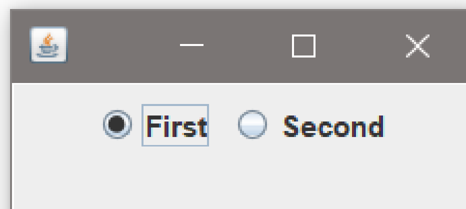
```
ButtonGroup radGroup = new ButtonGroup();
```

The buttons are added to the group to make them mutually exclusive. Only one button in a button group can be selected at a time.

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

And finally, they would be located using a layout manager and added to the panel. Note that the buttons are added, not the group.

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



Radio Button Example

To obtain the user selection in an event handler, the *getSource()* method is used to determine the source of the event. This code reacts to a button being clicked.

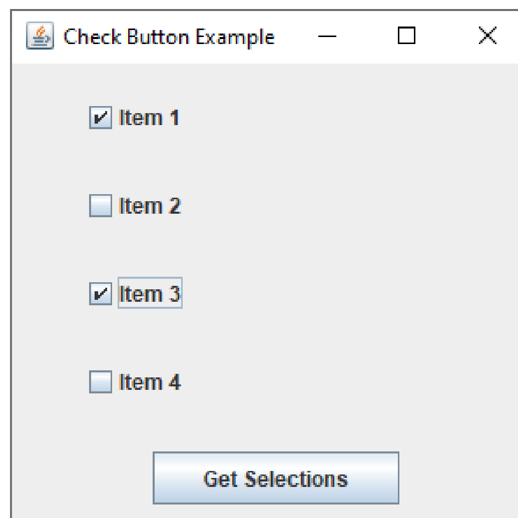
```
public void actionPerformed(ActionEvent e) {
    if(e.getSource() == rad1)
        doSomething;
    else if(e.getSource() == rad2)
        doSomethingElse;
```

To obtain the user selection within code, each radio button would be tested individually. This code would be in an event handler for a button that the user clicks after all of the selections have been made.

```
if(rad1.isSelected())
    doSomething;
if(rad2.isSelected())
    doSomething;
```

Check Boxes

Check Boxes are components that can be mutually exclusive, but typically allow the user to select multiple options. The boxes generate an action event when selected by the user which can be handled with an action listener, or their state can be accessed with the *isSelected()* method. The code for the example below follows.



Check Button Example

In the code below, after the panel is created and the layout is set to null to allow *setBounds()* to be used for positioning, the four check boxes are created and positioned, and they are added to the panel.

```
JButton btn = new JButton("Get Selections");
btn.setBounds(80, 220, 140, 30);

JPanel cPanel = new JPanel();
cPanel.setLayout(null);

JCheckBox cb1 = new JCheckBox("Item 1");
JCheckBox cb2 = new JCheckBox("Item 2");
JCheckBox cb3 = new JCheckBox("Item 3");
JCheckBox cb4 = new JCheckBox("Item 4");

cb1.setBounds(40, 20, 60, 20);
cb2.setBounds(40, 70, 60, 20);
cb3.setBounds(40, 120, 60, 20);
cb4.setBounds(40, 170, 60, 20);

cPanel.add(cb1);
cPanel.add(cb2);
cPanel.add(cb3);
cPanel.add(cb4);

cPanel.add(btn);
```

To obtain the selections for the example program, the ActionListener for the button includes inspecting each check box using *isSelected()* as shown below.

```
class ClickListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {

        if(cb1.isSelected()) {
            System.out.println("Item 1 is checked.");
        }
        if(cb2.isSelected()) {
            System.out.println("Item 2 is checked.");
        }
        if(cb3.isSelected()) {
            System.out.println("Item 3 is checked.");
        }
        if(cb4.isSelected()) {
            System.out.println("Item 4 is checked.");
        }
    }
}
```

A Complete Example – Garden Shed Company

Requirements:

Design and develop an interface for the Garden Shed Company program in Chapter 8 that determines the cost to build sheds based upon the size of the shed, door type, and number of windows. The interface will accommodate selection of the options using an option list for the sizes, radio buttons for the window choices and door choices (including the ramp). The interface will compute the total price *whenever* a selection is made or changed, and a purchase button will acknowledge the purchase.

Shed Options and Pricing:

Shed size options as width/depth and price:

10 x 8 - $900, 12 x 10 - $1,800, and 16 x 12 - $2,600

Window options and price:

One (1) large $90.00 or two (2) medium at $80.00 each

Door options and prices:

Single door $160.00, Double door $240.00

Ramp:

Double door requires entry ramp $90.00

Planning the components and positions, and considering the use of options can save time after the fact. Although the types of controls were provided in the requirements, the arrangement and positions were left to the developer. A top-down and left-right configuration is typical as shown above.

Developing the program using a methodical approach, organizing the code, and using descriptive names for variables and methods will save time as the program becomes complex. The code below declares variables for the individual prices, and includes the constructor with the frame, frame title, and panel. The panel layout is assigned null to permit *setBounds()* to be used for positioning.

```
public class CH_10_GardenShed {

    double doorPrice = 0;
    double winPrice = 0;
    double sizePrice = 900;
    double totalPrice = 900;

    public CH_10_GardenShed() {

        JFrame cFrame = new JFrame();
        cFrame.setSize(500,500);
        cFrame.setTitle("Garden Shed Company");
        cFrame.setLocationRelativeTo(null);

        JPanel cPanel = new JPanel();
        cPanel.setLayout(null);
```

The next section declares and initializes the title label, shed size label, and the combo box for the size selection.

```
        JLabel titleLabel = new JLabel("Garden Shed Company");
        titleLabel.setFont(new Font("Serif", Font.PLAIN, 28));

        JLabel shedSizeLabel = new JLabel("Shed size selections:");
        shedSizeLabel.setFont(new Font("Consolas", Font.ITALIC, 16));

        JComboBox<String> sizesBox = new JComboBox <String>();
        sizesBox.setName("Sizes");
        sizesBox.addItem("10 x 8 - $900");
        sizesBox.addItem("12 x 10 - $1,800");
        sizesBox.addItem("16 x 12 - $2,600");
```

The section of code below declares and initializes the window selection label and radio buttons, and the door selection label and radio buttons.

```
JLabel windowLabel = new JLabel("Window selections:");
windowLabel.setFont(new Font("Consolas", Font.ITALIC, 16));

JRadioButton win1Radio = new JRadioButton("Single Large");
JRadioButton win2Radio = new JRadioButton("Two Medium");
ButtonGroup winRadGroup = new ButtonGroup();
winRadGroup.add(win1Radio);
winRadGroup.add(win2Radio);


JLabel doorLabel = new JLabel("Door selections:");
doorLabel.setFont(new Font("Consolas", Font.ITALIC, 16));

JRadioButton door1Radio = new JRadioButton("Single");
JRadioButton door2Radio = new JRadioButton("Double w/ Access Ramp");
ButtonGroup doorRadGroup = new ButtonGroup();
doorRadGroup.add(door1Radio);
doorRadGroup.add(door2Radio);
```

The code in this section declares and initializes the purchase button and total label.

```
JButton btn = new JButton("Purchase");


JLabel totalLabel = new JLabel("Total price:  ");
totalLabel.setFont(new Font("Consolas", Font.BOLD, 16));
```

After establishing the components, they are all positioned in the next section to allow for adjustments without scrolling through lines of code to locate a specific component for adjustment.

```
// locate the components - x, y, width, height

titleLabel.setBounds(100,30,400,30);
shedSizeLabel.setBounds(70,100,200,50);
sizesBox.setBounds(280,110,120, 30);
windowLabel.setBounds(70,150,200,50);
win1Radio.setBounds(80,180,100,50);
win2Radio.setBounds(200, 180, 100, 50);
doorLabel.setBounds(70, 230, 200, 50);
door1Radio.setBounds(80,260,100,50);
door2Radio.setBounds(200, 260, 200, 50);
btn.setBounds(180, 340, 140, 30);
totalLabel.setBounds(160,  380, 200, 50);
```

Next the components are added to the panel, and the panel to the frame. The close operation is assigned and the frame is made visible.

```
// Add the components to the panel

cPanel.add(titleLabel);
cPanel.add(shedSizeLabel);
cPanel.add(sizesBox);
cPanel.add(windowLabel);
cPanel.add(win1Radio);
cPanel.add(win2Radio);
cPanel.add(doorLabel);
cPanel.add(door1Radio);
cPanel.add(door2Radio);
cPanel.add(btn);
cPanel.add(totalLabel);

cFrame.add(cPanel);


cFrame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
cFrame.setVisible(true);
```

The listener for the button is created, declared, and assigned to the button next.

```
class ClickListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {

        JOptionPane.showMessageDialog(null, "Order Confirmed",
                "Thank you!", JOptionPane.INFORMATION_MESSAGE);

    }
}


ActionListener purchaseListener = new ClickListener();

btn.addActionListener(purchaseListener);
```

Recall that the total price is to be recomputed anytime the user changes any selection. In the next section, a generic listener for the controls is declared that will be assigned to all of the components. They each generate an event, and the event will be tested to determine which control was clicked. The result will change the pricing for that item, and the total will be recomputed.

The output statements help with testing through development. Note that the code inside the ActionListener could be placed in a method.

```
class controlListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {

        if(e.getSource() == win1Radio) {
            System.out.println("WIN 1");
            winPrice = 90;
        }
        if(e.getSource() == win2Radio) {
            System.out.println("WIN 2");
            winPrice = 160;
        }
        if(e.getSource() == door1Radio) {
            System.out.println("DOOR 1");
            doorPrice = 160;
        }
        if(e.getSource() == door2Radio) {
            System.out.println("DOOR 2");
            winPrice = 240 + 90;
        }


        if(e.getSource() == sizesBox) {
            int size = sizesBox.getSelectedIndex();

            System.out.println(sizesBox.getSelectedIndex());

            if(size == 0) {
                System.out.println("SMALL SHED");
                sizePrice = 900;
            }
            else if(size == 1) {
                System.out.println("MED SHED");
                sizePrice = 1800;
            }
            else if(size == 2) {
                System.out.println("LARGE SHED");
                sizePrice = 2600;
            }

        }
```

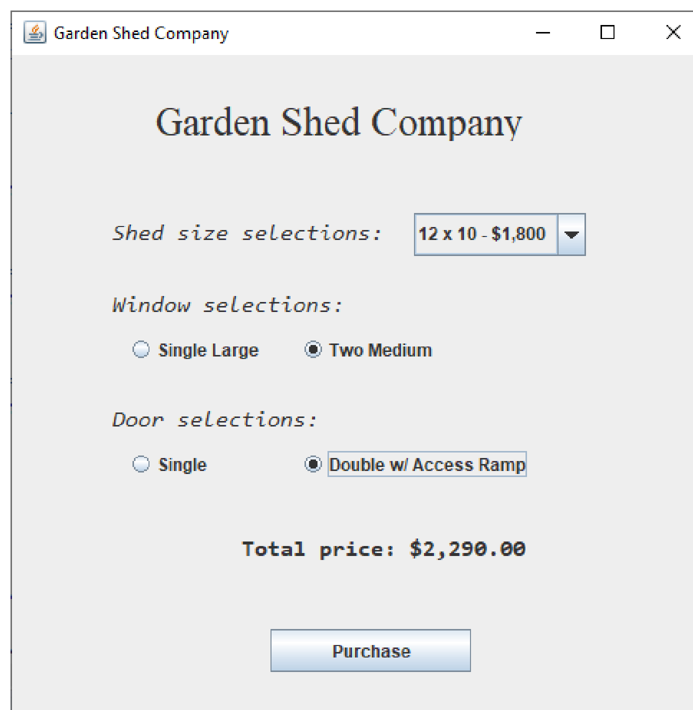
(Continued on next page)

The remaining code for the ActionListener (shown below) recalculates the total price, formats the result, stores the value as a String using *String.format()*, and then overwrites the total price label.

```
        totalPrice = winPrice + doorPrice + sizePrice;

        String price = String.format("%,.2f", totalPrice);

        totalLabel.setText("Total price: $" + price);

    }
```

A single listener is declared and assigned to all of the controls. When any control is selected or changed by the user, the total is recomputed.

```
controlListener cl1 = new controlListener();

sizesBox.addActionListener(cl1);
win1Radio.addActionListener(cl1);
win2Radio.addActionListener(cl1);
door1Radio.addActionListener(cl1);
door2Radio.addActionListener(cl1);
```



Program Output

The design of GUI requires a combination of controls, dialogs, and operations that satisfy the requirements of program. The program determines the different interactive components that will be used, and the user determines the order of

operations to a degree. A control can be disabled until a specific value is entered or enabled and used to validate input and display error dialogs. In the Complete Example above, the window was a class, and the main method simply created an instance of the class. This is the appropriate design and structure for a GUI program and accommodates unit testing the class.

Assigning Different Fonts

Different fonts can be used to highlight specific text or to introduce a group or section of text. To assign a font to a label (or any component), use the *setFont()* method with the font type in quotes, the style constant can be italic, bold, and plain, and the point size is an integer. Several styles are used in the Complete Example above.

```
JLabel myLabel = new JLabel("Italic text");
myLabel.setFont(new Font("Consolas", Font.ITALIC, 14));
```

Multiple styles can be assigned as shown below with italic and bold.

```
JLabel myLabel = new JLabel("Bold and Italic text");
myLabel.setFont(new Font("Consolas", Font.ITALIC | Font.BOLD, 14));
```

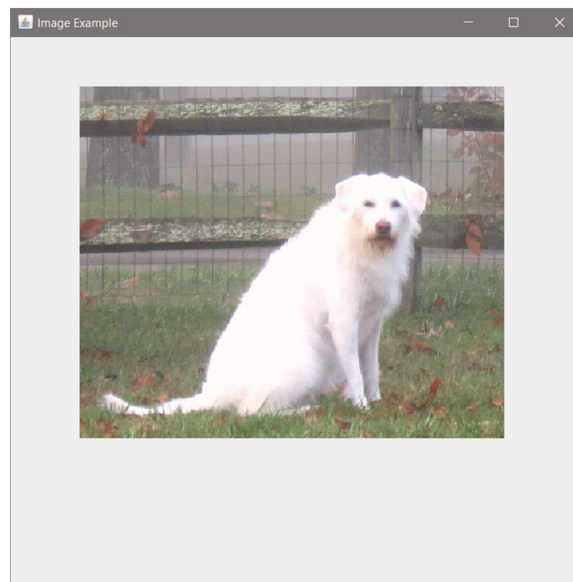
When displaying columnar data, it is preferred that monospaced fonts be used to simplify alignment. Monospaced font characters are fixed-width and occupy the same amount of space. Monospaced fonts include Consolas and Courier among others.

Adding Images

To add an image to a panel, it can be placed on a label which is then placed on a frame. A try block surrounds the statements due to file handling. Image files can be added as resources to the project or placed in the project file for access by the program. There are several ways to incorporate images depending on the application, and the size of the image, label, and panel need to be considered. The example below uses the BufferedImage subclass of the Image class and ImageIcon which paints icons for buttons and labels. The actual size of the image for this example is 434 x 360 pixels. The frame and panel are set a bit larger. The default directory is assumed for the image file location.

Ex. 10.8 – An Image on a Panel

```
JFrame frame = new JFrame("Image Example");
frame.setSize(600,600);
JPanel panel = new JPanel();
panel.setSize(500,500);
panel.setLayout(null);


BufferedImage dogImage;
JLabel picLabel = new JLabel();

try {
    dogImage = ImageIO.read(new File("Gracie.png"));
    picLabel.setIcon(new ImageIcon(dogImage));
    picLabel.setBounds(70,50,434,360);

} catch (IOException e) {
    e.printStackTrace();
}
```



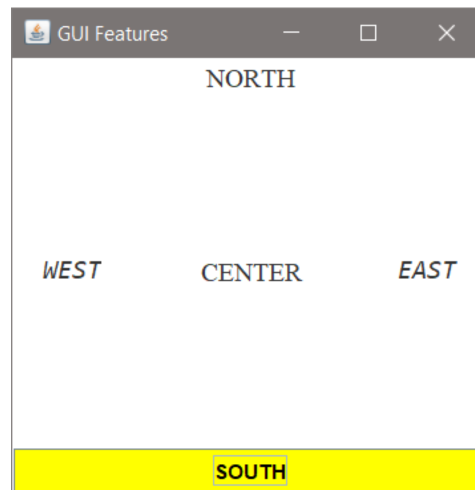
"Gracie at the fence" courtesy of Elaine Simber

Program Output

GUI Options and Layouts

Additional features that can be added to GUIs include borders (titled, etched, raised, lowered, and others), background or component color, and images. The example below uses the border layout manager with labels positioned North,

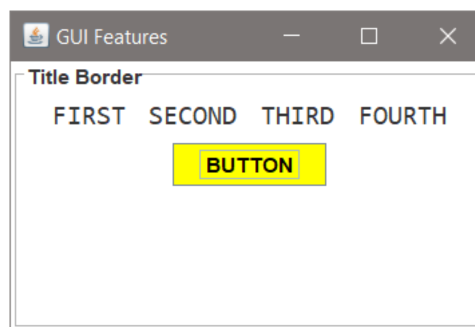
West, East, and Center, and a button located in the South area. Note that the North and South sections span the width of the panel.



The example below uses the default flow layout and an etched title border. The *setBorder()* option and button color settings are shown below.

```
JButton btn = new JButton(" BUTTON ");
btn.setForeground(Color.BLACK);
btn.setBackground(Color.YELLOW);
```

```
cPanel.setBorder(BorderFactory.createTitledBorder(
        BorderFactory.createEtchedBorder(), "Title Border"));
```



An example implementation of an interface with multiple panels and various layout managers is shown in Appendix G.

Chapter 10 Review Questions

1. GUIs are event driven by _________ input.
2. GUI program design requires carefully considering the order of events and how the user will ____________ with the program.
3. A _________ of the interface with component locations before writing code is a helpful design tool for GUIs.
4. The GUI component used to generate a window in Java is the _________.
5. The GUI component used to obtain a single line of user input from the keyboard is a ____________.
6. The GUI component used to display a line of text on a frame is a __________.
7. Using the setDefaultCloseOperation() method ensure that the program will _______ if the window is closed.
8. For a window to be displayed, the frames ______________ method must be set to true.
9. The __________ ____________ are used to position components on a frame.
10. For the click of a button to cause a reaction, a(n) _________________ must be written and assigned to the button.
11. ______________ are often used in GUI programs to alert the user to an error.
12. A combo box provides multiple ___________ for a user to choose from.
13. Mutually exclusive means that ________ selection can be made by the user.
14. Check boxes allow _____________ selections to be made by the user.

Chapter 10 Short Answer Exercises

15. Write a statement that creates a label called label1 with the text “Java is fun!”
16. Write a statement that creates a JTextField called userInput that allows for 10 characters of input.
17. Write a statement that creates a JPanel called myPanel.
18. Write a statement that creates a JFrame called myFrame.

19. Write the statements required to add the components from #15 and 16 above to the panel in #17, and to add the panel to the frame in #18 above.

20. In the following button creation, what is the text on the button?

```
JButton compBtn = new JButton("Compute");
```

21. In the following statement, why is null used?

```
myFrame.setLocationRelativeTo(null);
```

22. What does the following statement accomplish?

```
String myString = userInputField.getText();
```

23. Write a statement for a show message dialog box with the title "Error" and the text "Invalid data".

24. What does the following statement accomplish?

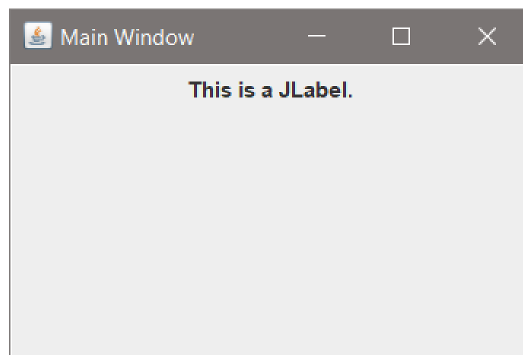
```
myLabel.setText("New data");
```

25. What does the following statement accomplish?

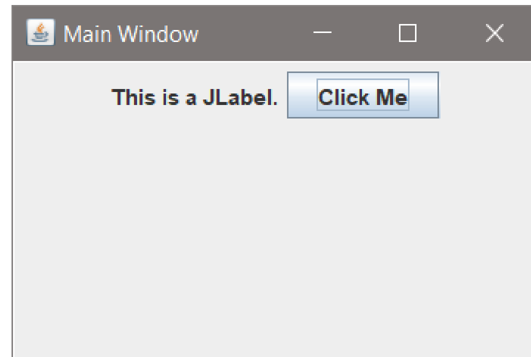
```
myLabel.setFont(new Font("Consolas", Font.BOLD, 14);
```

Chapter 10 Programming Exercises

26. Implement a class definition that generates the window below which is 300w x 200h with a title "Main Window" and a panel with a label on the panel that says "This is a JLabel." Write a statement in the main method that creates an instance of the class and displays the window.

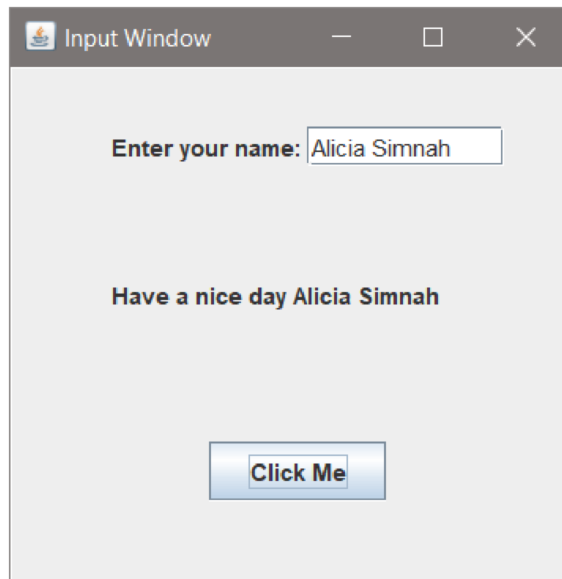


27. Using the Class from #26 above, add a button to the panel with the words "Click Me" on the button. Write a statement in the main method that creates an instance of the class and displays the window.



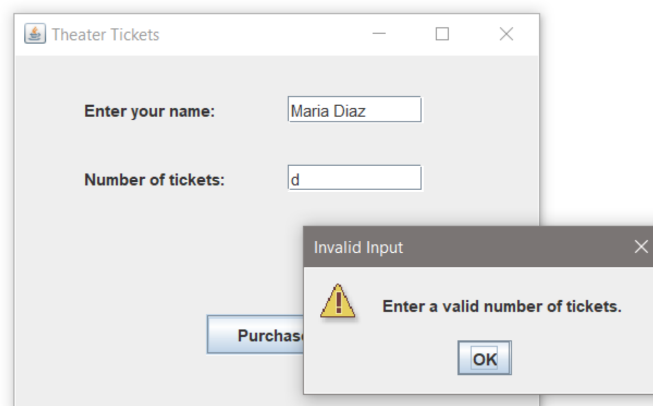
28. Using the Class from #27, set the panel layout to null, and position the label using *setBounds(100,30,100,30)* and the button using *setBounds(100,90,90,30)*. Set the frame so that the user cannot resize the window. Write a statement in the main method that creates an instance of the class and displays the window.

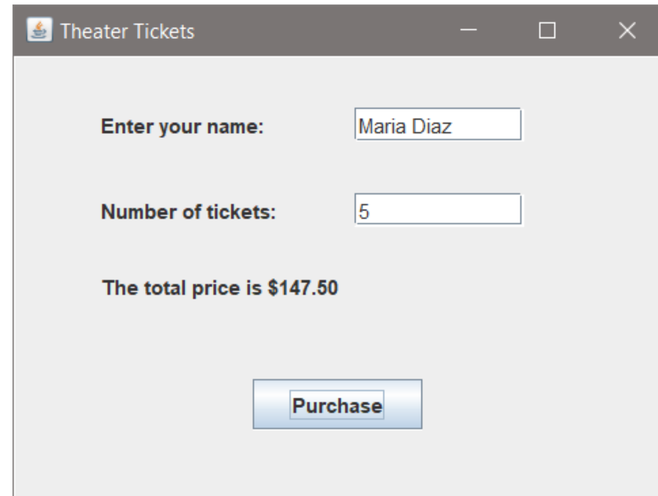
29. Implement a class InputWin that creates a GUI 300 x 300 with the title "Input Window" and a panel. Add a label to the panel that prompts the user to input their name into an entry component that is 20 characters wide. Add a button that obtains the input and updates a second label and displays "Have a nice day " and the name that was entered. Position the components using *setBounds()* as shown below, and use setLocationRelativeTo() so that the window appears in the center of the display area. Write a statement in the main method that creates an instance of the class and displays the window.



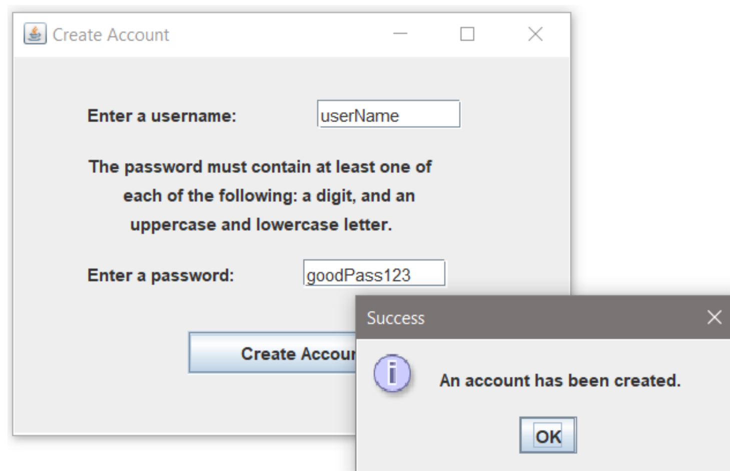
30. Implement a class TheaterWin that creates the GUI shown below. Use two entry components to obtain the users name and the number of tickets being purchased. The button text should be "Purchase", and the output label text should be "Thank you " and the name entered. If the number of tickets entered is not a positive integer, an error dialog box should appear. Display the Total cost based on a ticket price of $29.50 as shown below. Note the dollar sign and two decimal places in the output. To use *setText()* with a formatted value, it is best to convert to a formatted String as shown here.

```
String outPrice = String.format("%.2f", price);

priceLabel.setText("The total price is $" + outPrice);
```





31. Implement a Create Account class with a GUI that obtains a user name and password from the user and validates the password (at least 9 characters, at least one digit, upper, and one lower case letter). If the password is valid, display "An account has been created." in a dialog box, otherwise use a dialog box to display "Invalid Data Entered". The title on the window should be "Create Account", and the window should display the password requirements to the user.



32. Modify the program in #31 above to display the specific error when an invalid password is entered.

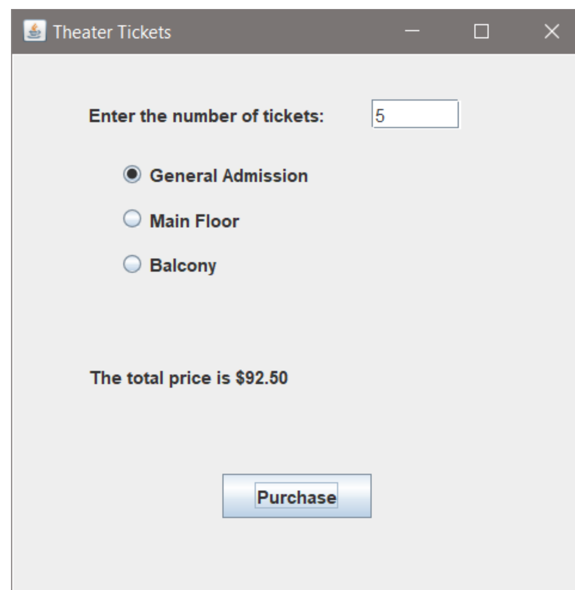
Chapter 10 Programming Challenges

#1 – Theater Ticket GUI with Seat Level Pricing

Design and implement a Class for a Theater GUI that obtains the number of tickets being purchased from the user and allows seat selection: General Admission, Main Floor, and Balcony at the prices shown below. The program should use an option list or radio buttons for seat selection, which must be mutually exclusive. When the "Purchase" button is clicked, the program should obtain the number of tickets and the seat selection, and compute the total price. The program should display an error dialog if invalid data is entered for the number of seats. Use a label to display the total price as shown below.

Seating prices:

General Admission	$18.50
Main Floor	$37.50
Balcony	$26.00



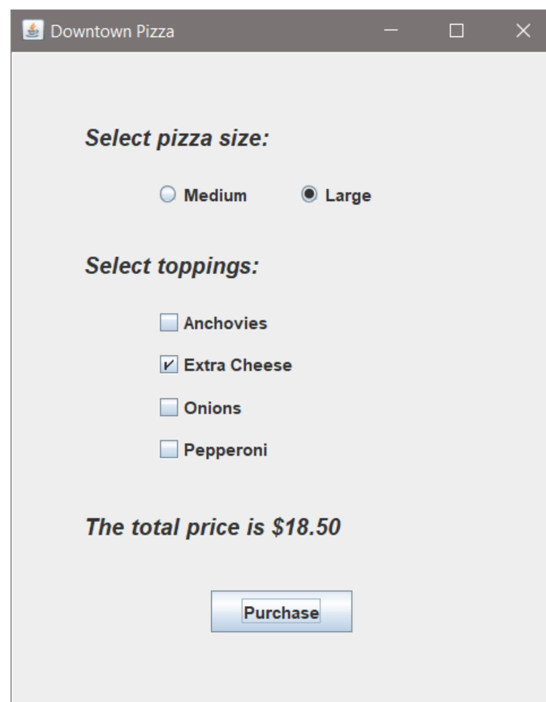
#1A – Theater Ticket GUI with Seat Level Pricing and Image

Add an appropriate image to the Theater Ticket program.

#2 – Pizza Size and Topping GUI

Design and implement a GUI for the Downtown Pizza shop that obtains an order for a pizza by size and topping. The size selection is mutually exclusive and should be implemented with radio buttons or an option list, and the topping selection with check boxes to accommodate multiple selections. A purchase button will compute the price and display it to the user based upon the prices below.

Medium	$12.50
Large	$15.50
Anchovies	$2.50
X-cheese	$3.00
Onions	$2.50
Pepperoni	$3.50



#2A – Pizza Size and Topping GUI with Image

Add an appropriate image to the Pizza program.

Chapter 11

GUI Programs

Although computer programs today perform many of the same duties they had in the past, today there is a graphical component to all user interactive programs. The interactive interface is one aspect of the program, and there are display and data representation aspects as well. The information resulting from program operations may be written to a file and displayed in a window either as text, a chart, or both. The data displayed may also be updated in real-time as the user interacts with the program. The design and layout of the interface should complement the operation of the program and user interaction. For example, a Theater Ticket program running on a kiosk may provide real-time updates of ticket sales to the theater manager. The updates may be in text, graphics, or both.

The data may also be saved to a file periodically and when selected by the manager. A confirming email or text message may be sent to a user when tickets are purchased, and an alert when the theater show time is sold out may include playing a sound. This chapter covers some typical operations that are often required and some that can be used to enhance a program.

Programs often display prior calculations for reference as well as new values being computed. Adding a new label for the new data to the interface each time a new value is computed is not practical. A better solution is to display the output in a scrollable area that is updated with each new set of data. Java provides the JTextArea and JScrollPane for this purpose.

JTextArea

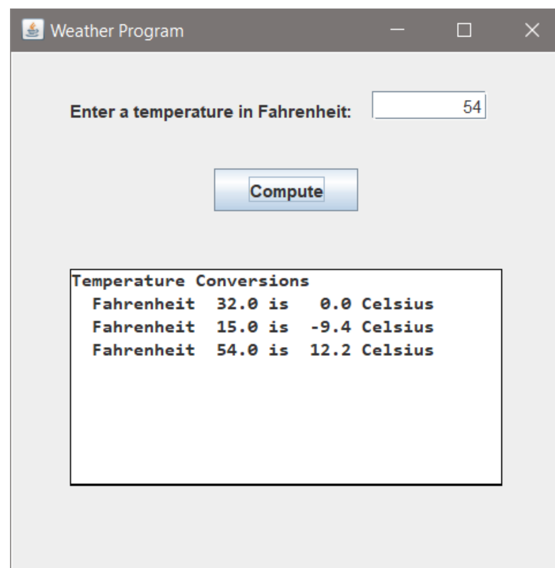
The JTextArea class provides a multi-line area that allows editing and display of multiple lines of text. The constructor accepts three optional arguments. The String (first) argument is the top row of text for a title or headers for columns.

```
JTextArea area = new JTextArea(String, rows, columns);
```

It allows appending which provides for adding data as it is computed without erasing previous values. The JTextArea can be added to a panel or directly to a frame. In example Ex. 11.1, two panels are placed on the frame. The top panel contains the label, entry component, and a button. The bottom panel (dataPanel) contains the JTextArea which is appended with each new computation (see example Ex. 11.4). This code includes the bottom panel and text area.

Ex. 11.1 – Multi-line Output with JTextArea

```
dataPanel = new JPanel();
dataPanel.setBounds(40,150,400,300);
dataPanel.setLayout(null);

outArea = new JTextArea("Temperature Conversions");
Font outFont = new Font("Consolas", Font.BOLD, 13);
outArea.setFont(outFont);
outArea.setEditable(false);
outArea.setSize(300, 150);
outArea.setBorder(BorderFactory.createLineBorder(Color.black));

dataPanel.add(outArea);
```



Program Output

Multiple Windows

When a second window is used for data display and updates, there are a variety of designs and implementations in terms of when and where the second window appears. The user will interface and enter data in one window and the resulting values will be displayed in the other. The program will need access to both in order to obtain the input and to update the data being displayed. The second window can be an attribute of the main window class and created in the constructor for the main window. The following example expands the Weather Program from Chapter 10 to include updating the text on a label that is located on a second window. The second window is in a separate class and file. An instance of the second window is created in the constructor for the main window as shown below.

Ex. 11.2 – Main Window Attribute and Constructor

```
CH_11_WeatherOuputWin outWin;

public CH_11_Weather() {        // Constructor

    outWin = new CH_11_WeatherOuputWin();
```

Ex. 11.2A – Output Window Class

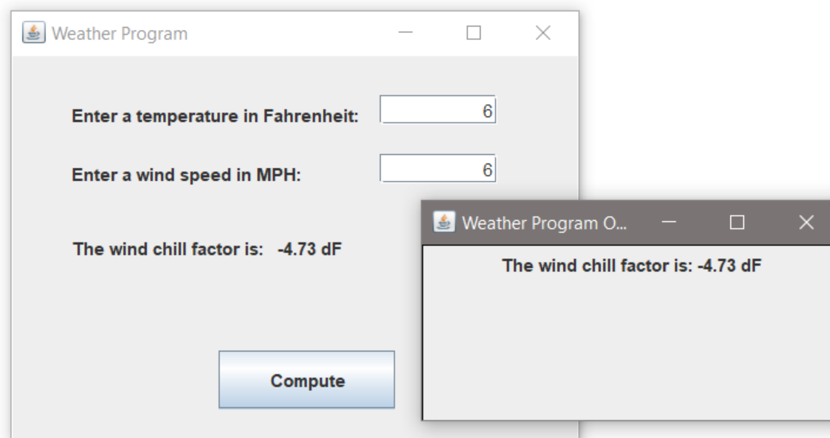
```
public class CH_11_WeatherOuputWin {

    public JFrame outFrame;
    public JPanel outPanel;
    public JLabel outLabel;

    public CH_11_WeatherOuputWin() {

        outFrame = new JFrame();
        outFrame.setSize(300,300);
        outFrame.setTitle("Weather Program Output Window");

        outPanel = new JPanel();
        outLabel = new JLabel("");

        outPanel.add(outLabel);
        outFrame.add(outPanel);

        outFrame.setVisible(true);
    }
}
```

The update to the output window can be added to the action listener after the wind chill is computed as shown below, or in a separate method. Since the output display window is an attribute (subclass) of the main window, the label can be accessed and updated directly.

Ex. 11.3 – Listener Modification to Update the Label

```
class ClickListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {

        computeWeatherData();
        outWin.outLabel.setText("The wind chill factor is: " +
                    String.format("%,.2f", wc) + " dF");
    }
}
```

When the program runs, the windows are created and the compute button updates the label on the main interface and the display window.



Program Output

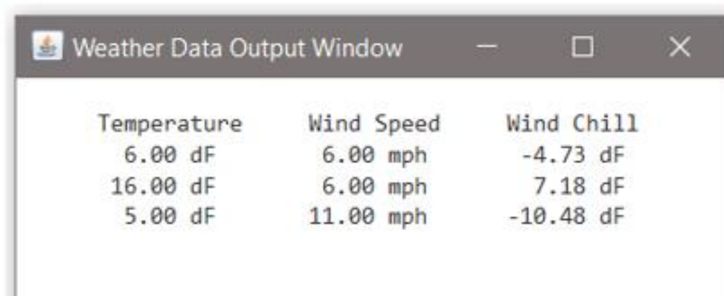
To provide the user with previous results for reference, a JTextArea can be added to the output display window either on a panel or directly on the frame. The code below adds the JTextArea to the output display window and sets options for the size, background color, whether or not the user can edit the text area (set to false), and assigns a monospaced font for consistent spacing and alignment of the output text. A header is formatted for the first row in the area and is passed to the JTextArea as the first argument.

Ex. 11.4 – Adding the JTextArea

```
public CH_11_WeatherAreaOutputWin() {

    outFrame = new JFrame();
    outFrame.setSize(360,300);
    outFrame.setTitle("Weather Data Output Window");

    header = String.format("%17s %14s %14s", "Temperature",
            "Wind Speed", "Wind Chill");

    outArea = new JTextArea("\n" + header, 40, 1);
    outArea.setSize(360, 300);
    outArea.setEditable(false);
    outArea.setBackground(Color.white);

    Font outFont = new Font("Consolas", Font.PLAIN, 12);
    outArea.setFont(outFont);

    outFrame.add(outArea);
    outFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    outFrame.setVisible(true);
}
```

The update to the JTextArea is added to the listener and includes formatting a String and appending it to the text area after the wind chill calculation. The line feed is included with the String.

```
String outString = String.format("%12.2f", tempF) + " dF"
        + String.format("%12.2f", windSpeed) + " mph"
        + String.format("%12.2f", wc) + " dF";

outWin.outArea.append("\n" + outString);
```



Program Output

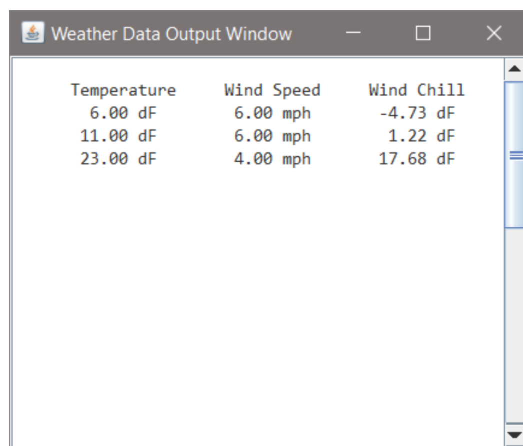
Java also provides a JTable that accepts row/column data, a JTextPane, and JEditorPane. Each of these components has benefits and limitations.

Scrollbars

If the output data could exceed the viewable area of the window, scrollbars can be added. The JScrollPane provides vertical and horizontal options for scroll bars including "Always", "Never", and "As Needed". The JTextArea is added to the JScrollPane and the pane is added to the frame as shown here.

```
scrPane = new JScrollPane(outArea);
scrPane.setVerticalScrollBarPolicy(
        ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

outFrame.add(scrPane);
```

Scrollbars can be added directly to panels, but need to be positioned on the panel. Since multiple panels can be used on a frame, placing a JTextArea on a JScrollPane which is then added to a panel eliminates positioning the scrollbar.



Program Output

Closing Programs and Multiple Windows

The use of the *setDefaultCloseOperation()* method is recommended to end a program when the user closes the interface. However, when multiple windows are used in a program the user could close any one of the windows. Depending on which window is closed, the program may need to continue or it may need to end. If the program is to end when any of the windows is closed, the default close operation must be added to each frame. Closing one frame would then end the program and close all of the frames. The JFrame constants provide various options for what should happen when a window is closed.

The statement below ends the program when a frame is closed.

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

For other situations including the case when some additional processing must be completed, there are other options available. The close operation can be set to dispose of the frame but allow the program to continue to run, or it can be set to do nothing and a window listener can be used to react to the window closing.

The statement below disposes of the frame when it is closed, but allows the program to continue to run.

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

Another option is to have the program do nothing when the frame is closed, and allow the program to continue to run. The window listener handles the reaction to the window being closed and disposes of the frame and ends the program. This provides a way to complete other operations as the program ends.

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

frame.addWindowListener((WindowListener) new WindowAdapter() {

    @Override
    public void windowClosing(WindowEvent e) {
        // do some processing;
        frame.dispose();    // dispose of the frame
        System.exit(0);     // end the program
    }
});
```

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 alerts the compiler that this method is overridden.

Drop-down Menus

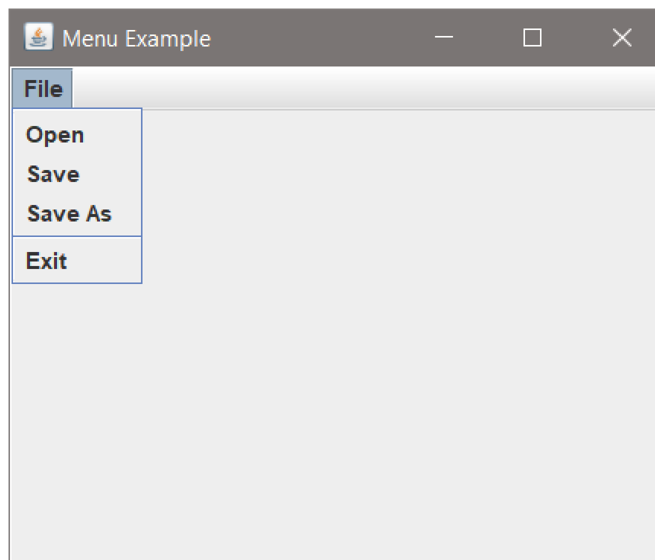
Drop-down menus on the border of a window are typically used for file handling and program features selected by users. Java provides classes for menus including the menu bar, menu, and menu item. A menu bar is created and the menus are created and added to the bar. Individual menu items are

created and assigned to the respective menu. The code below creates a file handling menu on a window. The menu items including the separator are added in the order they will appear. Comments are included to highlight the steps.

Ex. 11.5 – File Menu

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

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

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

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

The separator for the menu example was added between "Save As" and "Exit". Additional drop-down menus would be added the same way and are positioned left to right in the order they are added to the menu bar.



Program Output

To react to the selection of a menu item, each item could have an ActionListener assigned to it that would perform the proper operation.

```
class MenuItemListener implements ActionListener {

    public void actionPerformed(ActionEvent e) {

        System.out.println("SAVE AS WAS SELECTED.");
    }
}

saveAsItem.addActionListener(new MenuItemListener());
```

Since the menu items create an event, the items can be assigned the same listener and the *getSource()* method can be used in the action listener to determine the item that was selected.

```
if(e.getSource().equals(saveAsItem)) {

    System.out.println("SAVE AS WAS SELECTED.");
}
```

Recall from Chapter 7 that dialog boxes are typically used for file choosing and save/save as operations. The dialog code is repeated here for convenience. The code obtains the working directory of the user, creates a file chooser, sets the working directory, opens the dialog, and if the user selects a file, opens the file using the appropriate application on the system.

```
File workingDir = new File(System.getProperty("user.dir"));
JFileChooser chooser = new JFileChooser();
chooser.setCurrentDirectory(workingDir);
int status = chooser.showOpenDialog(null);

if(status == JFileChooser.APPROVE_OPTION) {
    File file = chooser.getSelectedFile();
    try {
        Desktop dt = Desktop.getDesktop();
        if(file.exists()) {
            dt.open(file);
        }

    } catch (IOException e1) {

        e1.printStackTrace();
    }
}
```

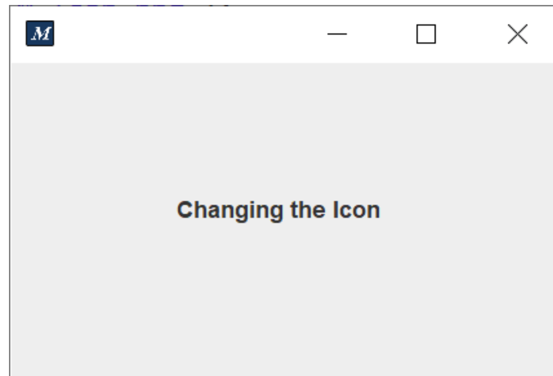
For the "Save" and "Save As" operations, writing to a file requires a PrintWriter, but the selection of the file and code are similar to the open operation.

```
File workingDir = new File(System.getProperty("user.dir"));
JFileChooser chooser = new JFileChooser();
chooser.setCurrentDirectory(workingDir);

int status = chooser.showSaveDialog(null);

if(status == JFileChooser.APPROVE_OPTION) {
    File file = chooser.getSelectedFile();
    try {
        PrintWriter w = new PrintWriter(file);
        w.print("testing writing");
        w.close();

    } catch (IOException e1) {

        e1.printStackTrace();
    }
}
```

Changing Title Bar Icons

The icon on a frame can be changed using the ***setIconImage()*** method. In the statements below, an image is assigned to an icon using the ImageIcon() class which creates images from GIF, JPEG, and PNG files. The *setIconImage()* method assigns it to the frame. The image is adjusted automatically for the size, but generally the icon sizes are 16x16 or 32x32 pixels.

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

    JFrame frame = new JFrame();
    frame.setSize(300, 200);
    frame.setLayout(null);

    Image icon = new ImageIcon("M_Icon.png").getImage();
    frame.setIconImage(icon);

    JLabel lbl = new JLabel("Changing the Icon");
    lbl.setBounds(85,50,120,50);
    frame.add(lbl);
    frame.setLocationRelativeTo(null);
    frame.setVisible(true);

}
```



Program Output

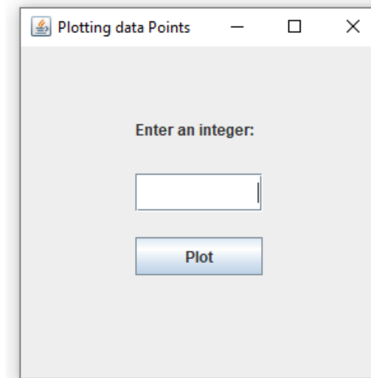
Charts

Drawing shapes was covered in Chapter 5, and data can be displayed graphically as a bar or line chart. As a reminder, drawing cannot be done on a JPanel object. A JComponent, JFrame, JTextComponent, or JLabel are used for drawing. The data for the chart must be available to the paintComponent method to perform the drawing, and whenever new data is available the output must be repainted. These considerations are covered in the example in the next section on plotting.

Plotting Data

To plot values as the user enters data in real-time requires a call to redraw (repaint) the plot as data is entered. The solution requires having the paint component update the frame each time a value 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, with a loop to access them for repainting. Assuming that the x coordinate is used for horizontal spacing, the y coordinates could be the values in the ArrayList. Some scaling may be required depending on the data set. Consider that the height of the plot area would be 100% and all heights for plotting would be adjusted to whatever value being plotted was largest. The Complete Example below uses a simple window to obtain integer input and a second window to plot the values. The input is added to an ArrayList that is an attribute of the plot window, and the plot window is an attribute of the main interface. The code to generate the interface is shown below and the code for for the PlotWin class follows.

Ex. 11.6 – Plotting in a Second Window




```
public class CH_11_Plot {

    JFrame mainFrame = new JFrame();
    JPanel panel;
    JButton btn;
    JLabel lbl;
    JTextField field;

    PlotWin pWin;        // plot window


   public CH_11_Plot() {

       pWin = new PlotWin();

       mainFrame = new JFrame();
       mainFrame.setSize(300, 300);
       mainFrame.setTitle("Plotting data Points");

       panel = new JPanel();
       panel.setLayout(null);;

       lbl = new JLabel("Enter an integer: ");
       lbl.setBounds(90, 50, 100, 30);

       field = new JTextField();
       field.setHorizontalAlignment(JTextField.RIGHT);
       field.setBounds(90,100,100,30);

       btn = new JButton("Plot");
       btn.setBounds(90, 150, 100, 30);

       panel.add(lbl);
       panel.add(field);
       panel.add(btn);

       mainFrame.add(panel);
       mainFrame.setVisible(true);
       mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

The button listener code below obtains the input from the text field which returns a String, converts it to an integer (which would be in a try block), adds it to the ArrayList, and then calls *repaint()* for the plot window. The plot window is repainted using the paintComponent method shown later.

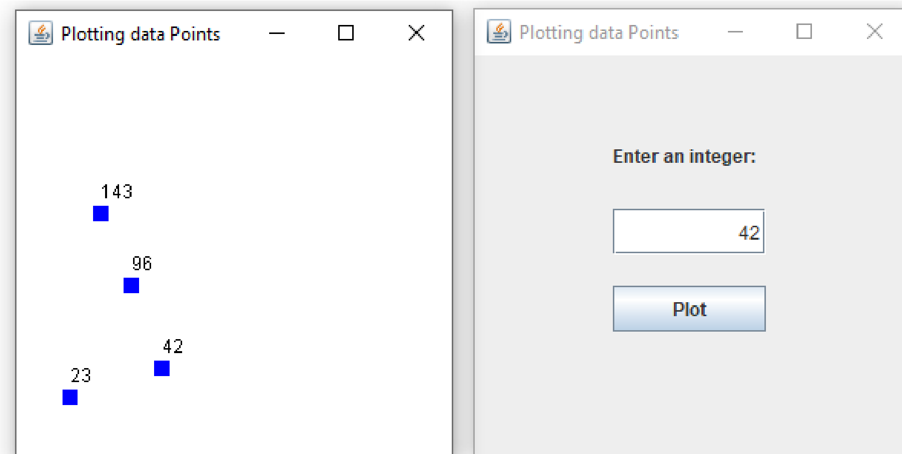
```
        class BtnListener implements ActionListener {

            public void actionPerformed(ActionEvent e) {

                String intStr = field.getText();
                int num = Integer.parseInt(intStr);

                pWin.values.add(num);
                pWin.outWin.repaint();
            }
        }

        BtnListener btl = new BtnListener();
        btn.addActionListener(btl);
    }
```

The ArrayList is an attribute of the plot window and is created in the constructor when an instance of PlotWin is created. Note the use of *getContentPane()* which is required to modify the background color. It returns the contentPane object for the plot window.

```
public class PlotWin {

    JFrame outWin;
    ArrayList<Integer> values;

    public PlotWin() {

        outWin = new JFrame();
        outWin.setSize(300, 300);
        outWin.setTitle("Plotting data Points");
        outWin.getContentPane().setBackground(Color.white);

        values = new ArrayList<Integer>();
```

The code that does the drawing (shown below) includes the *fillRect()* method for the markers and the *drawString()* method which is used to display the values associated with the markers. The algorithm includes an offset to locate the text values above and slightly to the right of the rectangles in the display area. Axis lines could be added using the *drawLine(x1, y1, x2, y2)* method. For the example, the numbers 23, 143, 96, and 42 were entered in that order, and no scaling factor was implemented. The heights are the number of pixels for that value. The frame is 300x300 and the y-offset is 240 pixels. Recall that the y coordinate is a positive number down from the top-left corner of the display area.

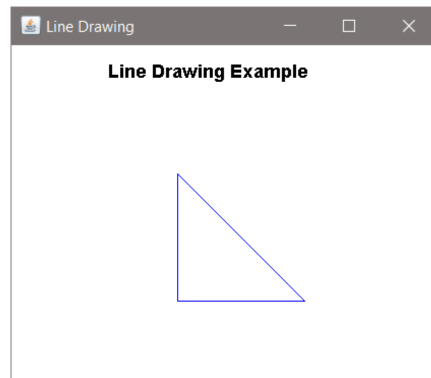
```
        JComponent comp = new JComponent() {

            public void paintComponent(Graphics g)
            {
                for(int i = 0; i < values.size(); i++) {

                    int x = i * 20 + 30;
                    int y = values.get(i);

                    g.setColor(Color.BLUE);
                    g.fillRect(x, 240 - y, 10, 10);
                    g.setColor(Color.BLACK);
                    g.drawString(String.valueOf(y),x+5,235-y);
                }
            }
        };

        outWin.add(comp);
        outWin.setVisible(true);
        outWin.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
```



Program Output

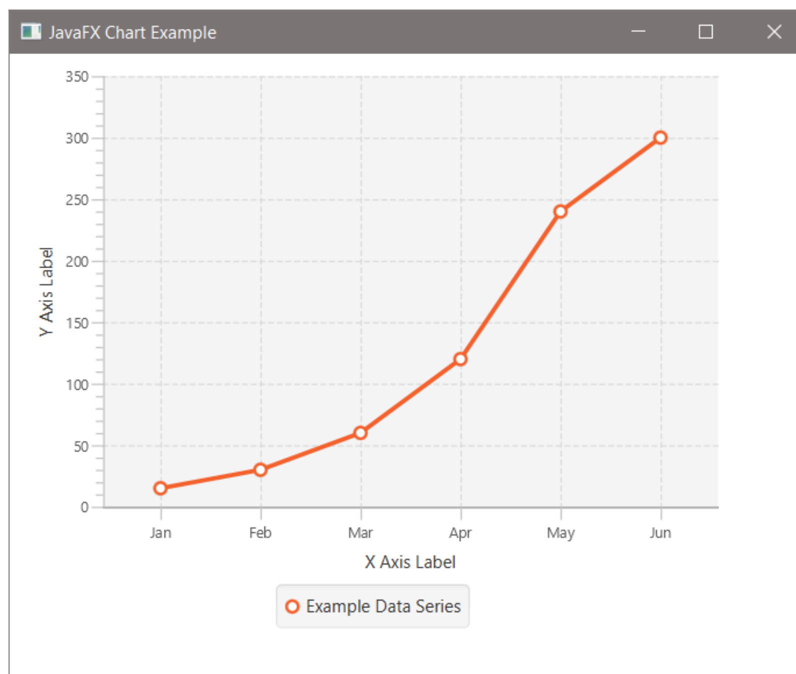
Drawing line charts from data sets would be a bit more complex since each line in the chart would require start-x, start-y, end-x, end-y specifiers. The lines below draw a simple triangle.

```
g.drawLine(130, 200, 130, 100); // x1, y1, x2, y2
g.drawLine(130, 100, 230, 200);
g.drawLine(230, 200, 130, 200);
```



Charting Tools

There are Java charting tools available including JFreeChart which is free to download and use, and the JavaFX charts and methods in the javafx.scene.chart package. The JavaFX charts are maturing into comprehensive charting tools with extensive capability, and all standard chart types; Pie, Line, Area, Scatter, Bar charts are included.



javafx.scene.chart Example

The following code with comments develops the basic line chart above using JavaFX.

```
public class CH_11_ChartFX extends Application {

    public void start(Stage s) {

        //Defining the X axis that uses Strings
        CategoryAxis xAxis = new CategoryAxis();
        xAxis.setLabel("X Axis Label");

        //Defining the Y axis that uses numbers
        // - lower bound, upper bound, tic unit
        NumberAxis yAxis = new NumberAxis(0, 350, 50);
        yAxis.setLabel("Y Axis Label");

        // Creating the line chart
        LineChart <String, Number> lineChart =
                new LineChart<String, Number>(xAxis, yAxis);

        // Setting the series
        XYChart.Series<String, Number> series =
                new XYChart.Series<String, Number>();
        series.setName("Example Data Series");

        // Defining the Series
        series.getData().add(new XYChart.Data<String, Number>("Jan", 15));
        series.getData().add(new XYChart.Data<String, Number>("Feb", 30));
        series.getData().add(new XYChart.Data<String, Number>("Mar", 60));
        series.getData().add(new XYChart.Data<String, Number>("Apr", 120));
        series.getData().add(new XYChart.Data<String, Number>("May", 240));
        series.getData().add(new XYChart.Data<String, Number>("Jun", 300));


        //Setting the data to the line chart
        lineChart.getData().add(series);

        // Create a group object
        Group group = new Group(lineChart);

        // Create a scene object - root, width, height
        Scene scene = new Scene(group,550,430);
        s.setTitle("JavaFX Chart Example");

        // Add the scene to the stage
        s.setScene(scene);
        s.show();

    }


        public static void main(String[] args) {

            Application.launch(args);

        }
```

Date and Time

The java.util package provides date and time classes for the current date and time including Date, LocalTime, and LocalDateTime.

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

The output of this code is: Sun Jan 23 15:29:17 EST 2022

The DateTimeFormatter class can provide specific date/time formatting. Some common formats are shown below.

```
"dd/MM/yy"              05/04/22
"dd MMM yyyy"           04 May 2022
"yyyy-MM-dd"            2022-05-04
"dd-MM-yyyy h:mm a"     05-04-2022 9:36 AM
```

A DateTimeFormatter is created and assigned the format using the *ofPattern()* method. The LocalDateTime is obtained and assigned to the now LocalDateTime object, and the *format()* method is used to covert the date/time and assign the result to a String.

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

The output of this code is: 01/23/22

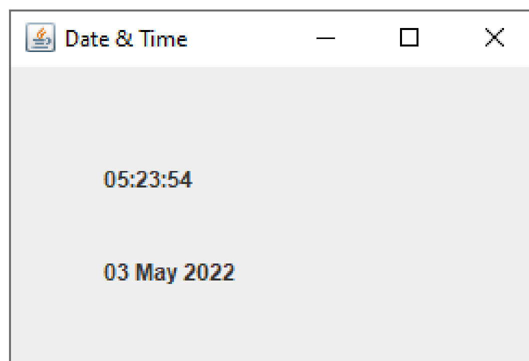
```
DateTimeFormatter df2 =
        DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
LocalDateTime now2 = LocalDateTime.now();
String formattedDate2 = df2.format(now2);
System.out.println(formattedDate2);
```

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

To include time that is continually updating, a Timer is available that accepts two arguments, a tic interval in milliseconds and an ActionListener. In the code

below, the listener obtains the current date and time and updates two labels. The timer interval is set to one second (1000 milliseconds).

```
sdfDate = new SimpleDateFormat("dd MMM yyyy");
dateLabel.setText(sdfDate.format(
        new GregorianCalendar().getTime()));

sdfTime = new SimpleDateFormat("hh:mm:ss");
timeLabel.setText(sdfTime.format(
        new GregorianCalendar().getTime()));

class TimerListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        timeLabel.setText(sdfTime.format(
                new GregorianCalendar().getTime()));
        dateLabel.setText(sdfDate.format(
                new GregorianCalendar().getTime()));
    }
};

ActionListener tListener = new TimerListener();
Timer timer = new Timer(1000, tListener);
timer.start();
```



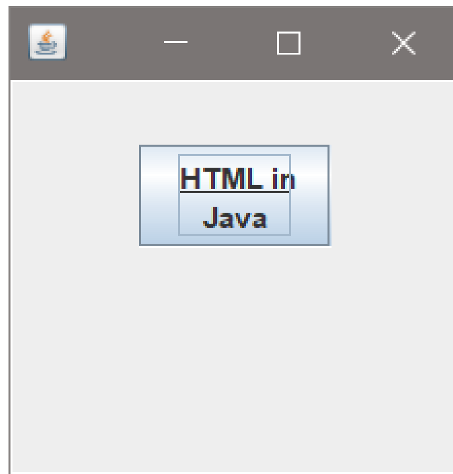
Program Output

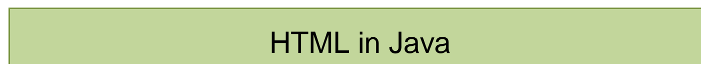
HTML in Java

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, the <html> tag is placed at the beginning of the text. Below is an example that uses HTML for the text on a button.

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





Playing Sound

One way to play sound in a program is to assign a String that includes the executable application that will play the file, and the path to the sound file to be played. 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 (wmplayer.exe in the example). The escapes are for the quotes on both portions of the String.

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

try {
    Process p = Runtime.getRuntime().exec(command);
} catch (IOException e2) {
    e2.printStackTrace();
}
```

Sound…Another Way

Another way to play sound in a program uses an AudioInputStream and a Clip resource. The input stream is assigned the sound file information and the Clip is used to open the sound file and start play. The Clip is a special kind of data line that allows audio data to be loaded before playback (instead of real-time). The Clip methods include: open, start, stop, loop, and close among others.

```
File soundFile = new File("myWaveFile.wav");
AudioInputStream audioInputStream;

Clip clip;

try {
    audioInputStream = AudioSystem.getAudioInputStream
            (soundFile.getAbsoluteFile());
    clip = AudioSystem.getClip();
    clip.open(audioInputStream);
    clip.start();
```

The rest of the try block and the exception handling, and a 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.

```
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 |
        LineUnavailableException |IOException e1) {

    e1.printStackTrace();
}
```

Launching a Browser, E-Mail, and Applications

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. 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.

Examples below include: launching the user-default browser, launching the user-default mail client, and launching a registered application to open, edit or print a specified file.

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

    } catch (IOException | URISyntaxException e) {

        e.printStackTrace();
    }
}
```

```
if (Desktop.isDesktopSupported()) {
    try {
        Desktop.getDesktop().mail(
                new URI("mailto:YourEmailAddress@gmail.com?subject=TEST"));

    } catch (IOException | URISyntaxException e) {

        e.printStackTrace();
    }
}
```

```
if (Desktop.isDesktopSupported()) {
    try {
        Desktop.getDesktop().open(new File("E:/Test.docx"));

    } catch (IOException e) {

        e.printStackTrace();
    }
}
```

Animation

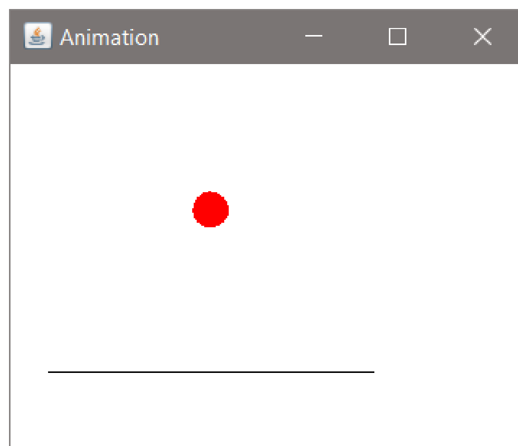
Implementing animation in Java can be accomplished with a sequence of images and an AWT Timer, or with the JavaFX 2.2 Animation class. The code below includes the declaration of a timer with a 125 millisecond delay between ticks. At each tick of the timer, an ActionListener() reacts to the tick of the timer by calling the *repaint()* method. The paintComponent can update images or call methods to update whatever is being displayed. The Timer has a *stop()* method to end the animation.

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

    }
};

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


JComponent component = 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

    }
};
```

For a bouncing ball animation that lowers the bounce height each time, only the y coordinate needs to change as the ball moves up and down. With each bounce, the height of the bounce would be reduced.

```
public void paintComponent(Graphics g) {

    g.setColor(Color.RED);
    g.fillOval(100, y, 20,20);
    g.setColor(Color.BLACK);
    g.drawLine(20, 170, 200, 170);
}
```

Recall that the y coordinate system for graphics has 0 at the top and a positive number down. Assuming a frame that is 250 pixels in height, the ball might initially start at a height of 100 pixels and travel down to 200 pixels (the baseline or lower y limit). The ball would then travel up to perhaps 90 pixels simulating the bounce, and then back down to the baseline (200 pixels). The direction of the ball determines whether the y coordinate is increased or decreased, and the direction is changed when y reaches one of the limits. The timer is stopped when the ball settles at the baseline.



In JavaFX, a Timeline, Duration, and Bounds control the specifics and sequence of operations. The JavaFX packages also provide for drawing basic shapes and transitioning including fade, fill, and rotate.

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 another task is taking up most of its time.

Painting via the paint() method is either System or App-triggered. When the AWT determines that a component needs to be repainted, it causes paint() to be invoked on the component. It is not recommended that paint() be invoked directly by a program.

Chapter 11 Review Questions

1. The design of the ____________ should complement the operation of the program and user interaction.
2. Data displayed by a GUI program may be updated in real-time for the user through the _______________ of a component.
3. A multi-line area in Java used to display multiple lines of text is the _____________.
4. The _________ method is used to add a line of output to a JTextArea.
5. A ______________ is used to add scrollbars to a display window.
6. Setting the setDefaultCloseOperation for a JFrame to EXIT_ON_CLOSE will ensure that the program will ________ if the window is closed.
7. The _____________ provides the ability to place program level menu items on a window title bar.
8. The ______________ method can be used to determine the source of an event.
9. The _____________ method forces a refresh/redisplay of a display window.
10. A __________ can be used to implement incremental actions for animation.

Chapter 11 Short Answer Exercises

11. Write a statement that creates a JTextArea named area with the header text “Five columns” that has 30 rows and 5 columns.
12. Write a statement that creates a JScrollPane named scroll and assigns the JTextArea named area to it.
13. Write a statement that sets the scroll policy for the JScrollPane scroll to never contain a horizontal scroll bar.
14. Write a statement that creates a JMenuBar named mBar.
15. Write the statement required to have the program do nothing when a JFrame named frame is closed.
16. Write the statements required to obtain the current date and time and display the result.

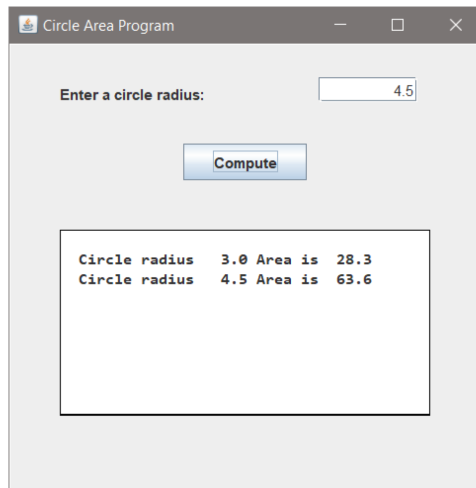
17. Write the statements required to obtain the current time using a DateTimeFormatter and display the current time as HH:mm:ss.

18. Write a statement to launch a browser that opens to https:www.java.com.

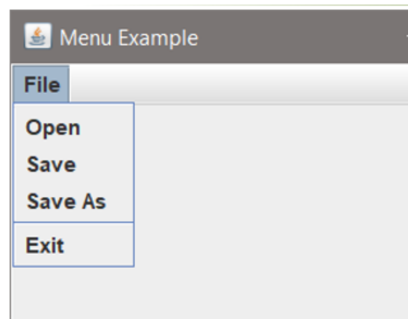
Chapter 11 Programming Exercises

19. Implement a class called CircleArea with a frame, label, entry component, and button. The program will create an instance of the class which will accept a radius through the entry component, compute the area of the circle, and display the radius and area in a JTextArea as shown below. Append each new entry.

Area = Math.PI * radius * radius



20. Implement a class called FileMenu that has a frame 300x300, with a frame title “Menu Example”, and a menu on the title bar with the items shown below. The main method will create an instance of the frame, and when the menu items are clicked, the program will print the name of the item clicked. A single listener may be used.



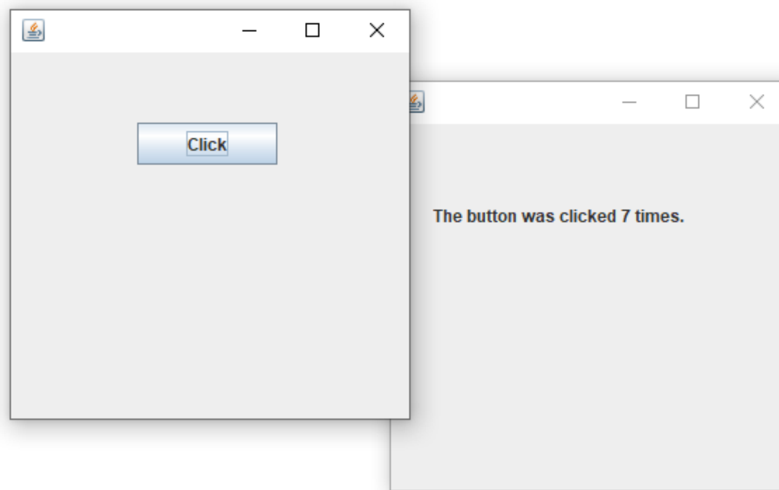
21. Add the statements required in program #20 above to prevent the window from being resized, and to have the frame centered in the display area when the program runs.

22. Implement a class MainWin with a frame 300x300 that has a button labelled "Click". Implement a subclass of MainWin called SubWin that is created in the constructor of MainWin. When the button on MainWin is clicked, a label on SubWin will display how many times the button was clicked. Add the statements required to end the program when either of the windows is closed. The main method should create an instance of the class.

    The count variable and label should be private members of the SubWin class. Therefore, a public mutator method is needed to update the counter and label.

    Consider:

```
public void updateCount() {

    count++;
    subLabel.setText("The button was clicked " + count + " times.");

}
```



23. Develop a program with a frame that is 375 x 340, and plot (drawstring) the text below at those coordinates. Make the font for the text Consolas, 12, and bold.

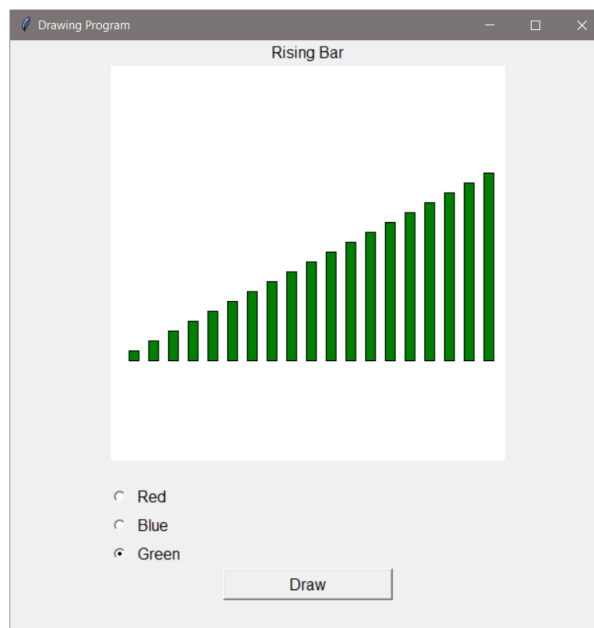
50, 50	250, 50	150,150
50, 250	250, 250	

    What do you notice about the frame size versus the coordinates?

Chapter 11 Programming Challenges

#1 – Draw Rising Bars

Implement a 500x500 window with “Rising Bars” on the title bar. Add a panel to the lower section of the frame that is 400x150, has three (3) radio buttons that select a color (red, blue, and green), and a “Draw” button. When the button is clicked, draw a rising set of 19 bars in the color selected. The bars should be 10 pixels wide, 10 pixels apart, and increase in height by 10 pixels from left to right.



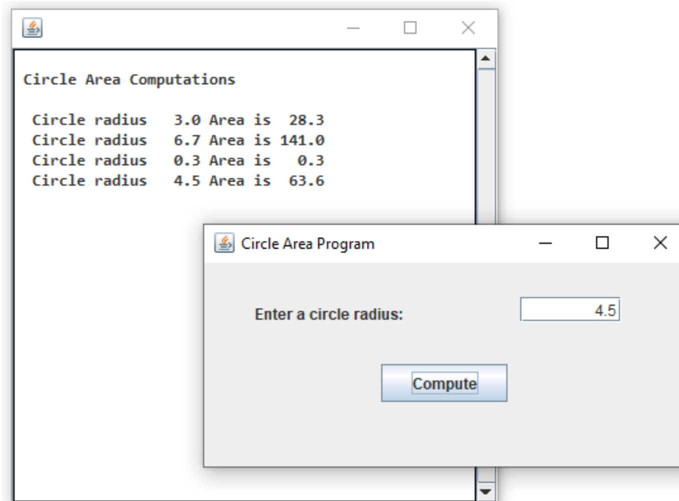
#2 – Output Display Window

Modify the program from #19 (repeated below) to display the output in a second display window that has a vertical scrollbar. The output display window should be a subclass of the main interface window.

(Recall that a scroll pane can be placed directly on a frame)

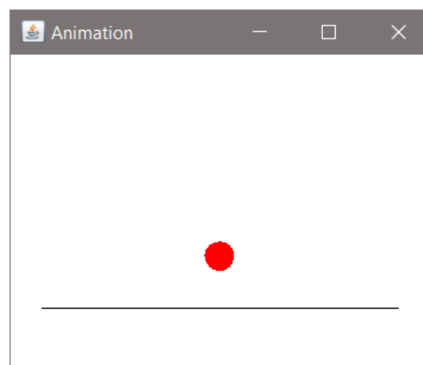
Implement a class called CircleArea that has a frame, label, entry component, and button. The program will accept a radius in the entry component, compute the area of the circle, and display the radius and area in a JTextArea as shown below.

Area = Math.PI * radius * radius



#3 – Bouncing Ball Animation

Implement the Bouncing Ball program mentioned at the end of the chapter using a class for the frame and a timer and timer listener for animation. The display should be a 300 x 250 frame. Each time the ball bounces, the height of the bounce will decrease by 10 pixels until the ball settles on a line drawn at 170 pixels. When the ball settles, stop the timer. The main method should create an instance of the frame with the ball.

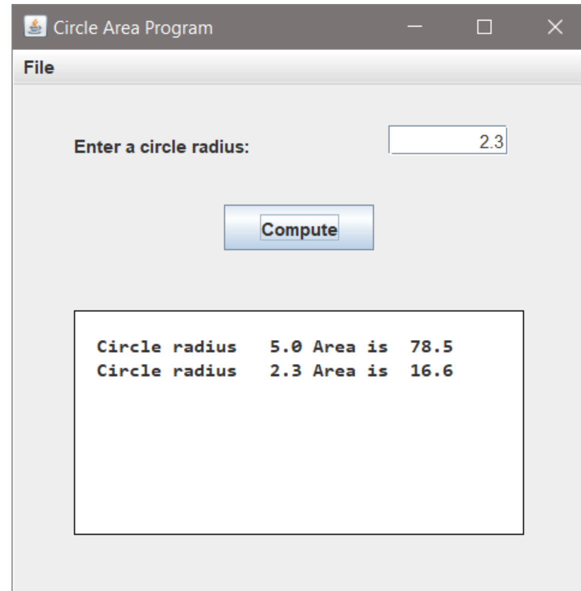


#3A – Bouncing Ball Animations

Modify the constructor to accept a color for the ball, and modify the main method to create two (2) instances of the frame with different color balls.

#4 – Circle Program with File Menu

Implement or modify the Circle program from #19 above, add the file menu from #20, and add the statements to execute the file menu items. Use JFileChooser dialogs (ref Chapter 7) for the operations, and for “Save” and “Save As”, write the data in the text area to a text file. The open item should open the text file for viewing and the exit item should end the program.



Appendix A

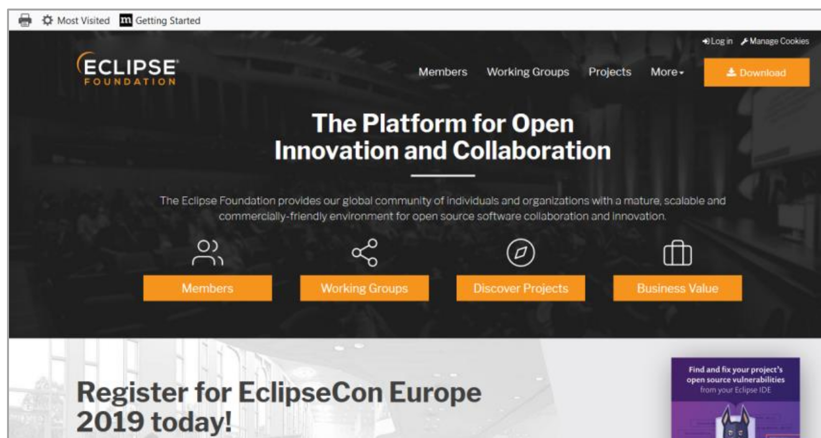
Decimal	Binary	ASCII	Decimal	Binary	ASCII	Decimal	Binary	ASCII
32	0010 0000	space	64	0100 0000	@	96	0110 0000	`
33	0010 0001	!	65	0100 0001	A	97	0110 0001	a
34	0010 0010	“	66	0100 0010	B	98	0110 0010	b
35	0010 0011	#	67	0100 0011	C	99	0110 0011	c
36	0010 0100	$	68	0100 0100	D	100	0110 0100	d
37	0010 0101	%	69	0100 0101	E	101	0110 0101	e
38	0010 0110	&	70	0100 0110	F	102	0110 0110	f
39	0010 0111	‘	71	0100 0111	G	103	0110 0111	g
40	0010 1000	(	72	0100 1000	H	104	0110 1000	h
41	0010 1001	)	73	0100 1001	I	105	0110 1001	i
42	0010 1010	*	74	0100 1010	J	106	0110 1010	j
43	0010 1011	+	75	0100 1011	K	107	0110 1011	k
44	0010 1100	,	76	0100 1100	L	108	0110 1100	l
45	0010 1101	-	77	0100 1101	M	109	0110 1101	m
46	0010 1110	.	78	0100 1110	N	110	0110 1110	n
47	0010 1111	/	79	0100 1111	O	111	0110 1111	o
48	0011 0000	0	80	0101 0000	P	112	0110 0000	p
49	0011 0001	1	81	0101 0001	Q	113	0110 0001	q
50	0011 0010	2	82	0101 0010	R	114	0110 0010	r
51	0011 0011	3	83	0101 0011	S	115	0110 0011	s
52	0011 0100	4	84	0101 0100	T	116	0110 0100	t
53	0011 0101	5	85	0101 0101	U	117	0110 0101	u
54	0011 0110	6	86	0101 0110	V	118	0110 0110	v
55	0011 0111	7	87	0101 0111	W	119	0110 0111	w
56	0011 1000	8	88	0101 1000	X	120	0110 1000	x
57	0011 1001	9	89	0101 1001	Y	121	0110 1001	y
58	0011 1010	:	90	0101 1010	Z	122	0110 1010	z
59	0011 1011	;	91	0101 1011	[	123	0110 1011	{
60	0011 1100	<	92	0101 1100	\	124	0110 1100	|
61	0011 1101	=	93	0101 1101	]	125	0110 1101	}
62	0011 1110	>	94	0101 1110	^	126	0110 1110	~
63	0011 1111	?	95	0101 1111	_	127	0110 1111	DEL

Appendix B

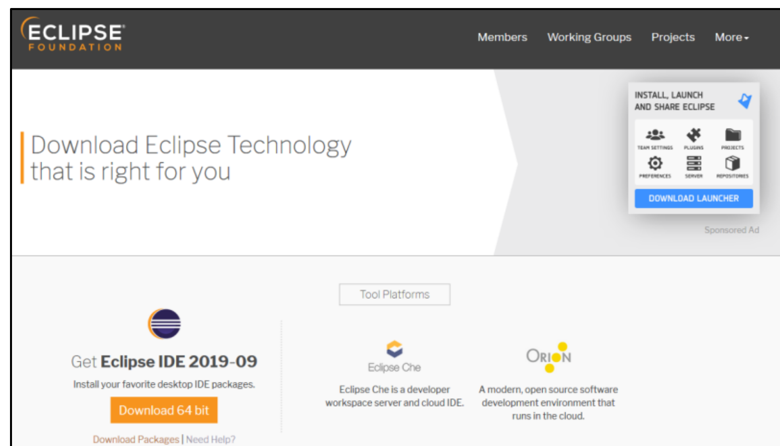
Obtaining Eclipse

- 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 easy access
    - Copying the JRE to the flash drive simplifies running

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



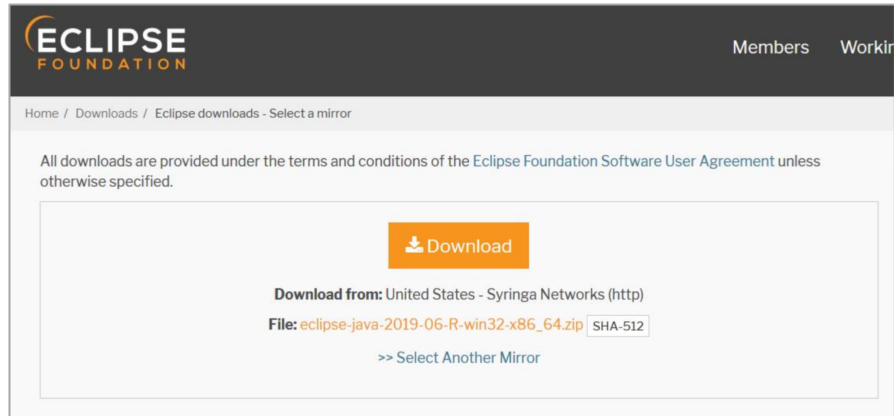
From the Downloads window shown select the appropriate version for your computer. On most Windows machines, select the "Download 64 bit" button.



Eclipse Foundation screenshots from eclipse.org used under Fair Use

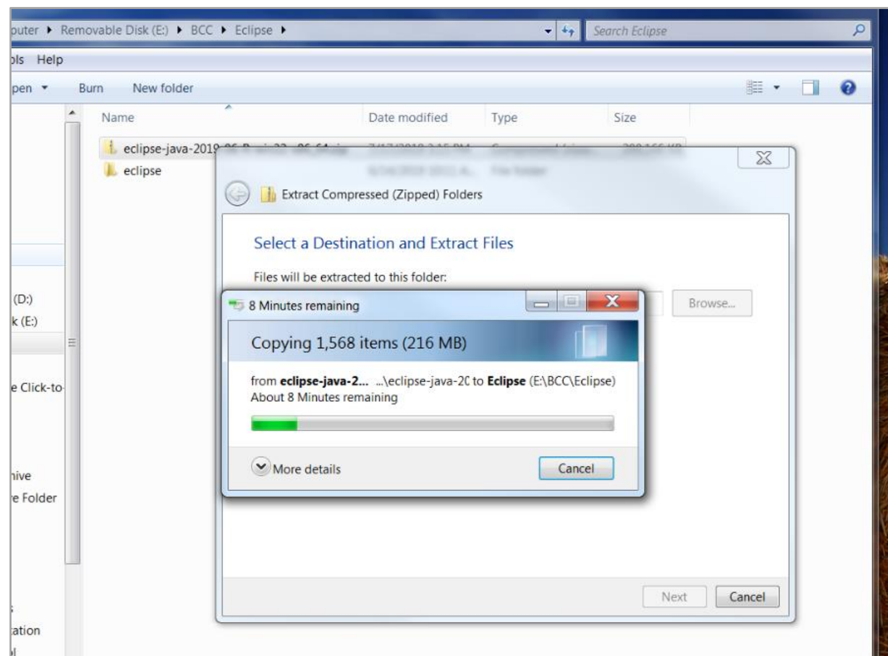
Appendix B

The appropriate download page will be displayed.



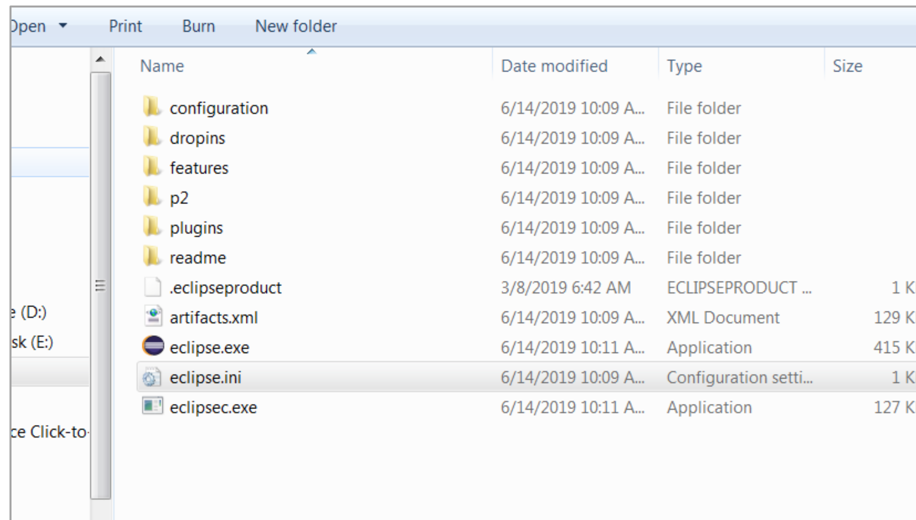
Download or save the zip file. (Eclipse will run fine on a flash drive and can be installed there if you prefer.)

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…



Appendix B

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



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.

Appendix B

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**

Error…exit code=13

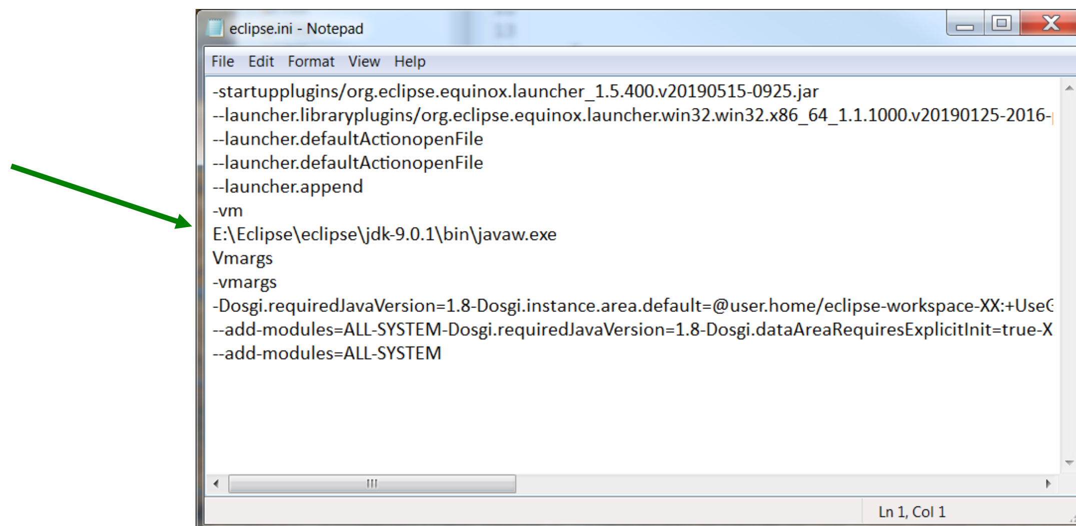
Appendix B

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

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.**



Appendix B

- You may 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 if running from a flash drive:

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

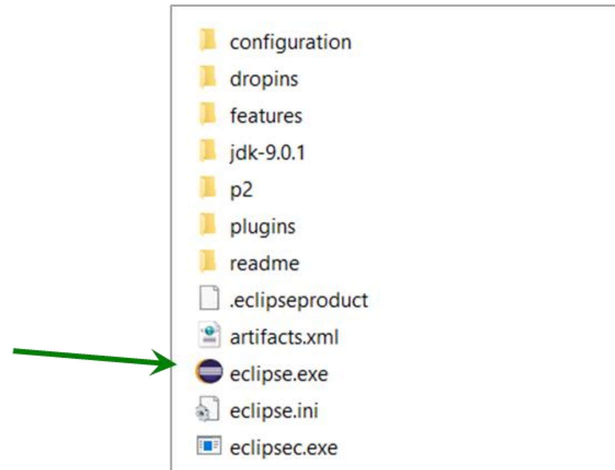
- **Help Documentation is available:**

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

Appendix C

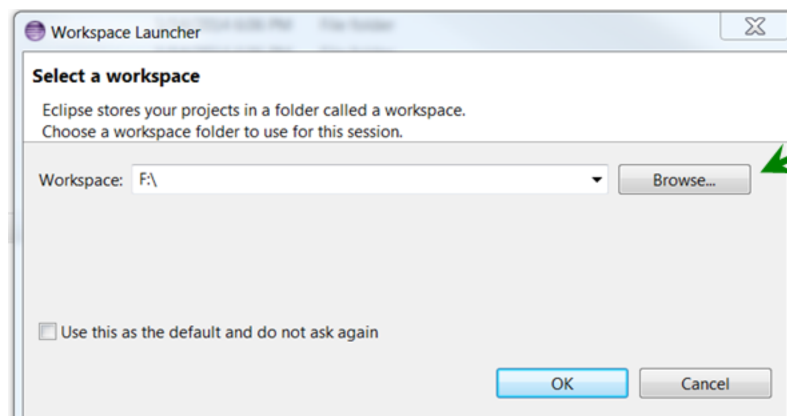
Getting Started in Eclipse

To launch Eclipse, use the short-cut on the desktop that was created when Eclipse was installed, or double-click the Eclipse icon from the Eclipse directory.



The Workspace

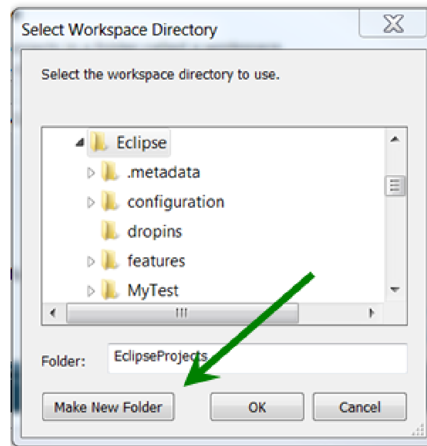
When Eclipse is launched for the first time, a "Workspace" needs to be created. The Workspace organizes programs and projects and adds supporting files. Choose the "Browse" button, and decide where the program files will be stored. It is important to keep files organized in directories and the Workspace will help with this.



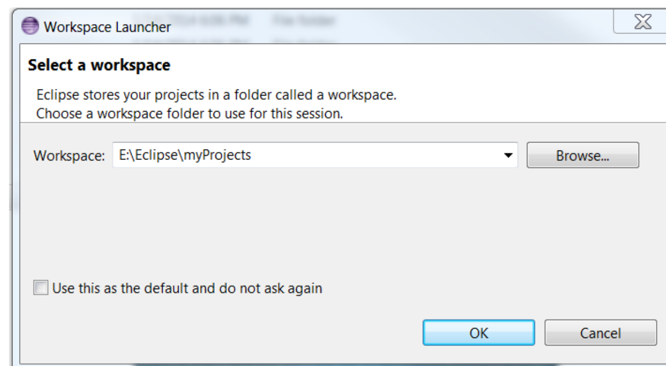
After choosing a location for the projects, click "OK" and the "Select Workspace Directory" window will appear (shown below). This is where the actual directory or folder is created.

Appendix C

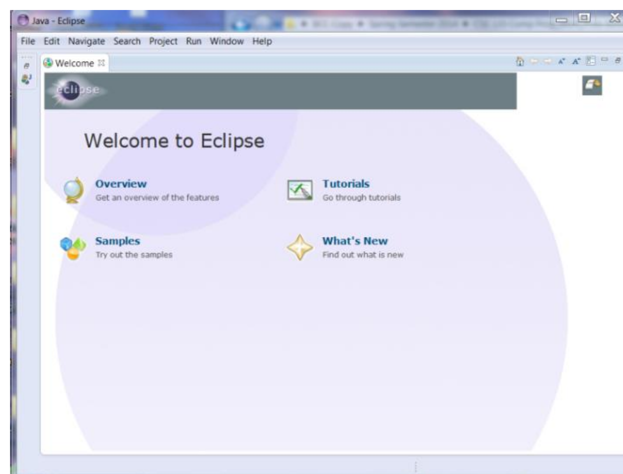
Select the "Make New Folder" button. Name the folder something appropriate for your projects and select the "OK" button.



Then, back in the "Workspace Launcher" window, select 'OK' again.

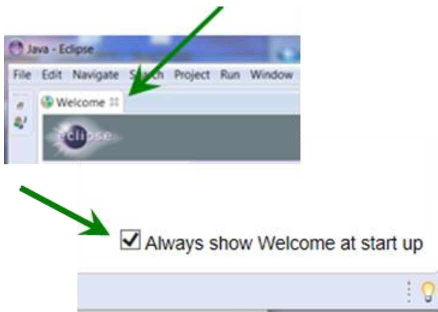


Eclipse will open to the "Welcome" window.

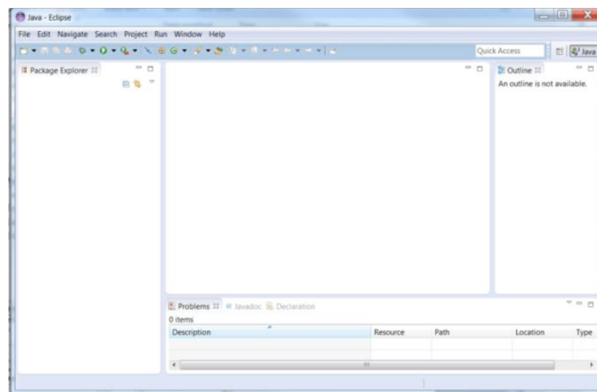


Appendix C

After unchecking the "Always show" box (bottom right), close the "Welcome" window by clicking the "X" (top left).



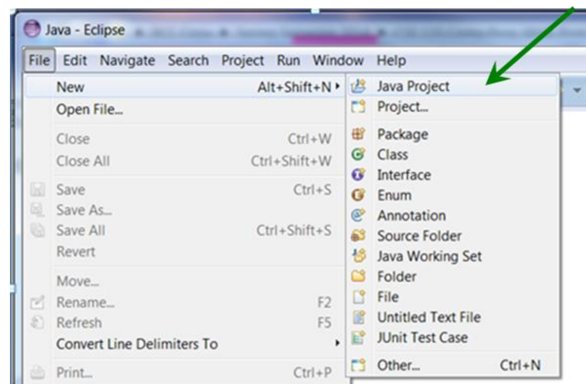
The IDE will be displayed.



Creating a Project

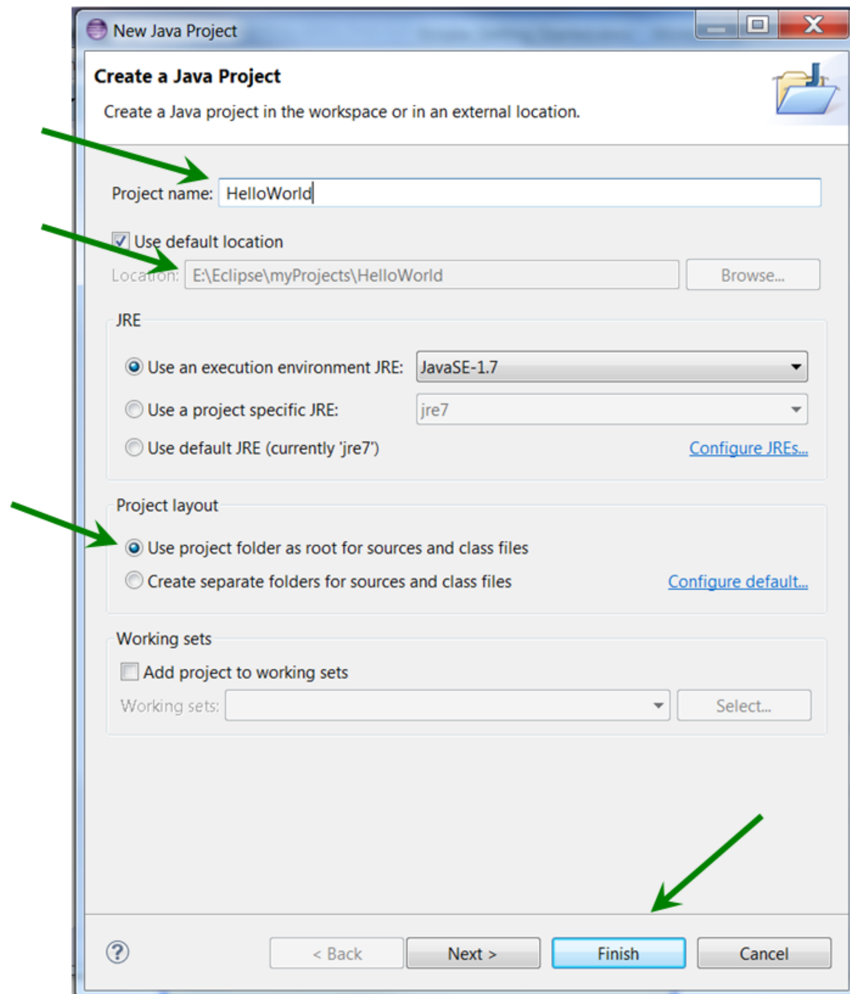
When creating a new program, always create a "**Project**" and not a "File". Eclipse creates important supporting files for a project in the Workspace. The next time Eclipse is started and the Workspace is selected, all of the projects in the Workspace and their supporting files will be loaded automatically.

Select **File** | **New** | **Java Project** from the menu

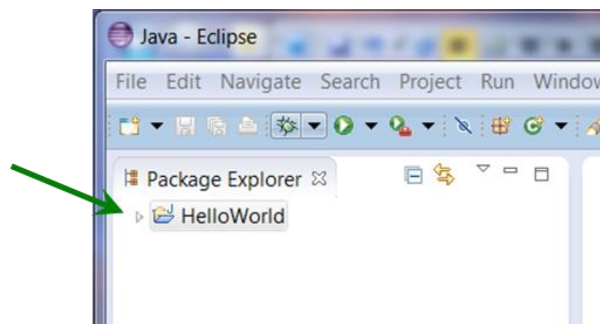


Appendix C

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".



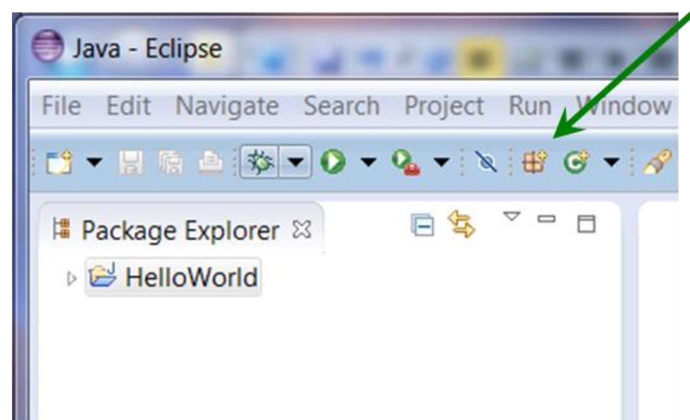
The project will appear in the **Package Explorer** on the left side of the IDE.



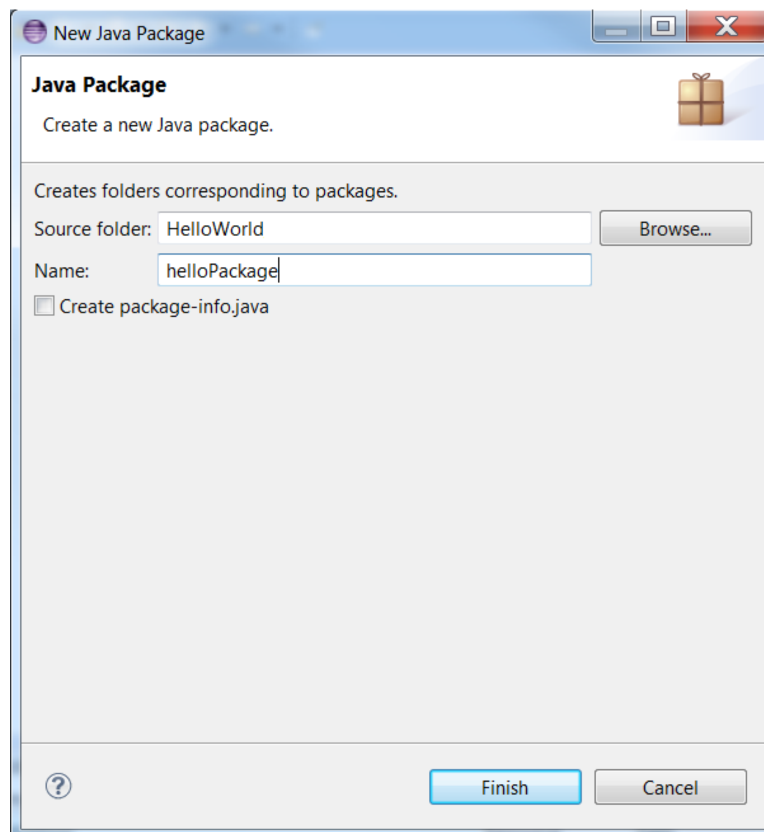
Appendix C

Creating a Package

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.



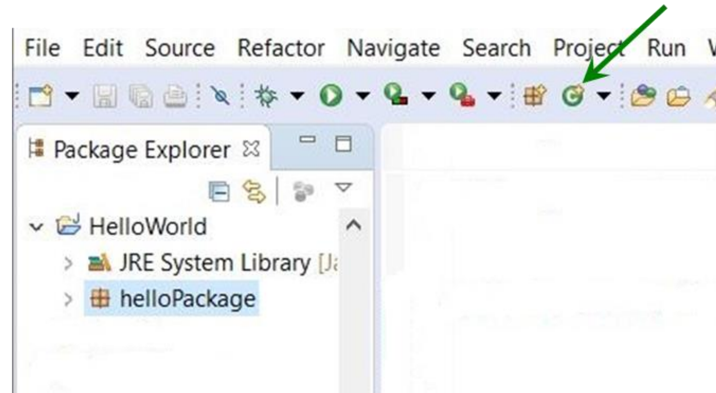
The New Java Package window will appear. Give the package a name that is relevant to the project (helloPackage shown here). Then click "Finish". Note: if the Source folder is not shown, click the Browse button to select the one you created.



Appendix C

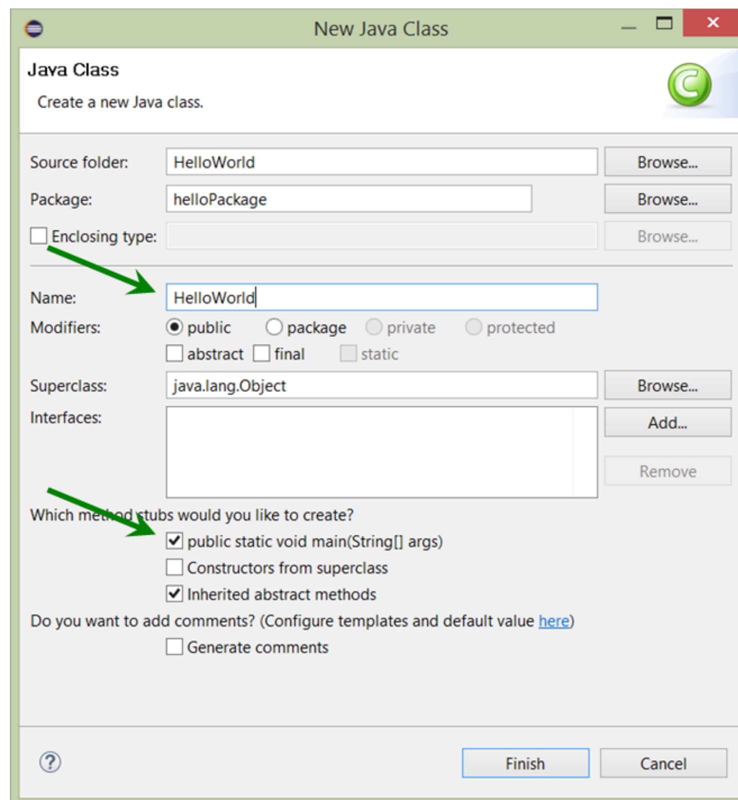
Creating a Class

The package will now appear in the project explorer. Be sure that the **package is still highlighted** and select the class icon (green circle with a gold star.



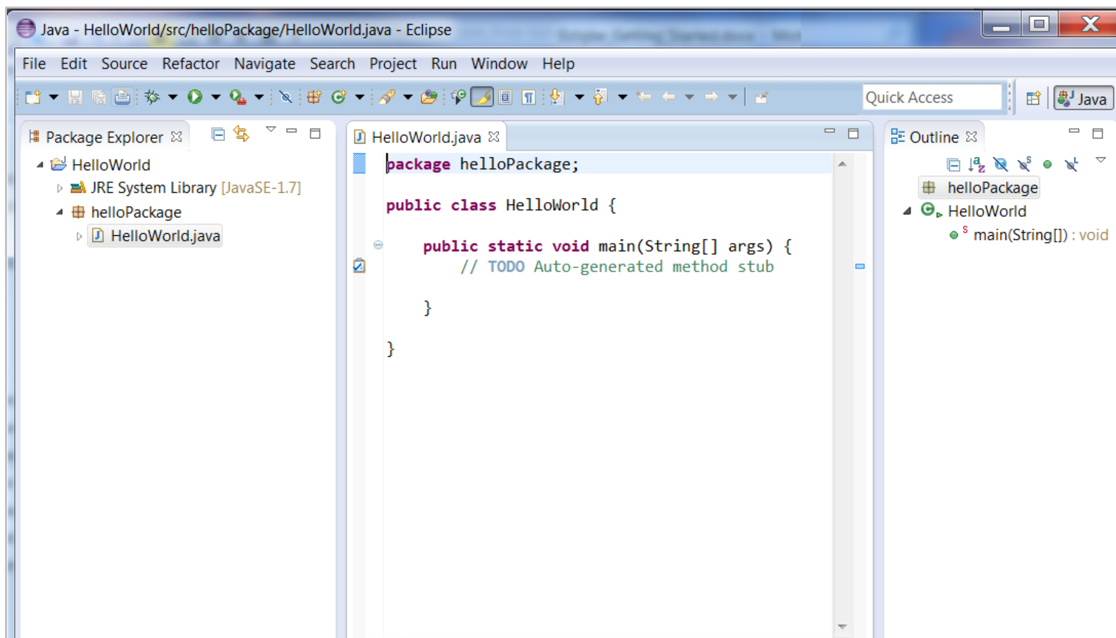
The class creation window will appear. **Give the class the same name as the project name** that was chosen earlier. In the screen capture below, notice that the "Source Folder" name and the class name are the same.

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



Appendix C

The project is now created with a package and a class, and the main method has been added to the program. The main method was created by Eclipse and is where the code for the program will be written.



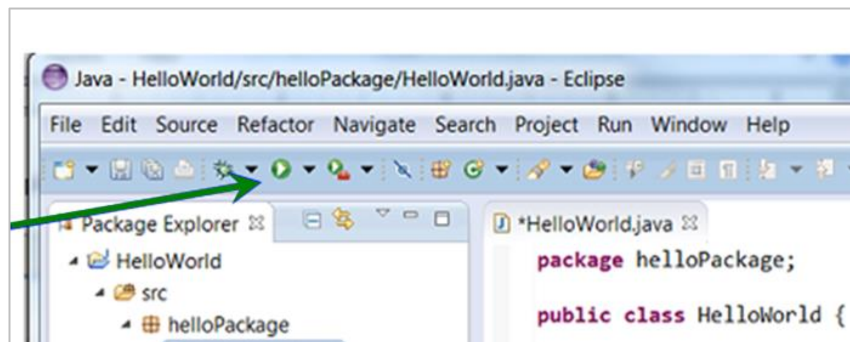
Running a Program

Creating the customary "Hello World" program first accomplishes writing, compiling, and running a program to produce output. Add the line of code shown below on line 10 exactly as it is written. This line uses the Java Utility Class *System*, the *out* object of the *System Class*, and the *print* method which sends characters to the console display area.

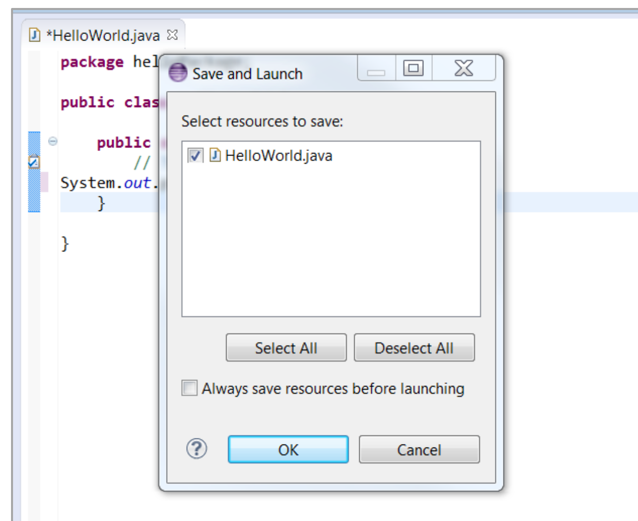
```
1  package helloPackage;
2
3
4  public class HelloWorld {
5
6
7      public static void main(String[] args) {
8          // TODO Auto-generated method stub
9
10             System.out.println("Hello World!");
11
12         }
13
14 }
15
```

Appendix C

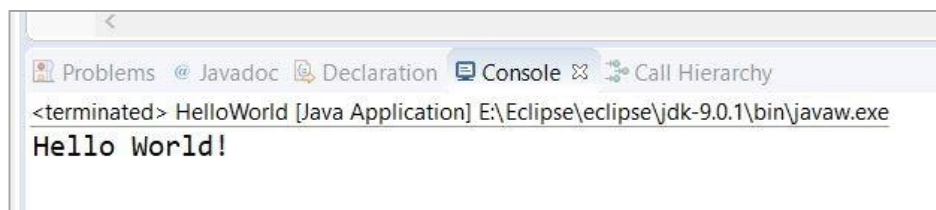
After adding the line of code, run the program by clicking on the green circle with a white triangle inside.



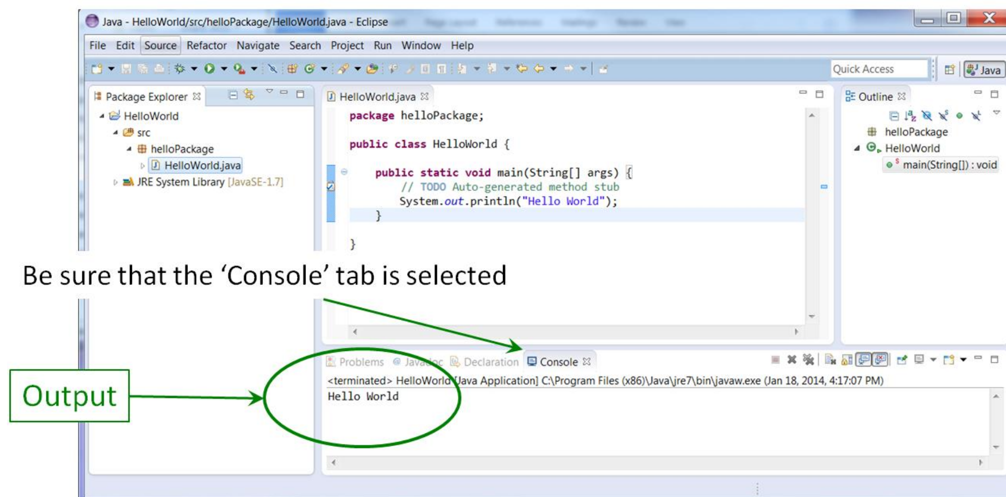
The 'Save and Launch' window will be displayed. Eclipse ensures that changes are saved before the program runs. Click "OK" and the program will run.



The output will be displayed in the console area at the bottom of the IDE.

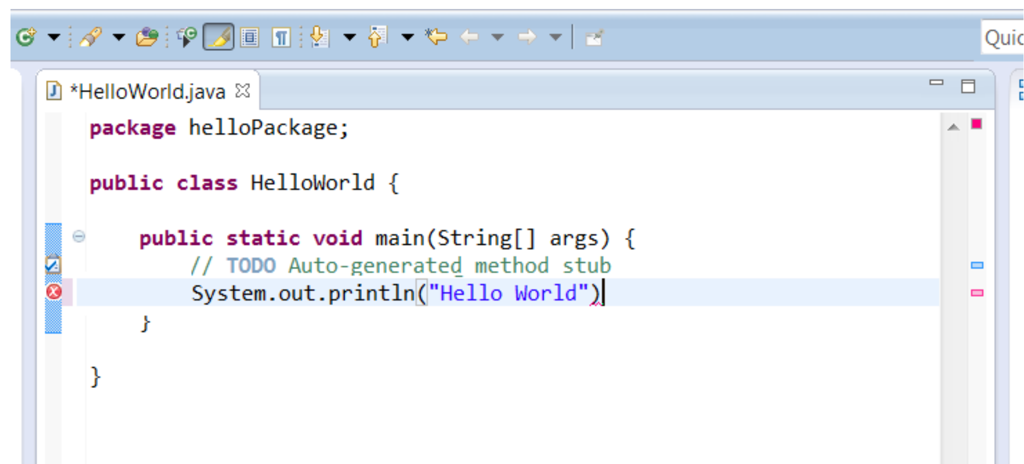


Appendix C



Programming Errors

Errors in Eclipse are shown in several ways. To highlight this, remove the semicolon from the end of the line of code from the example.

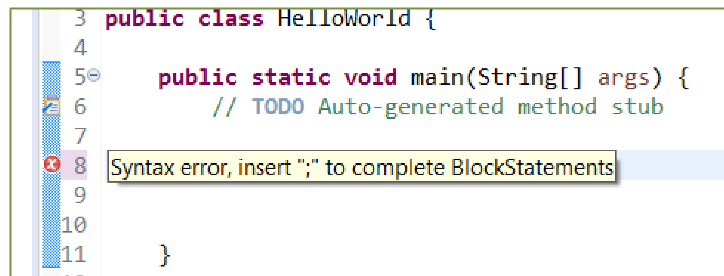


The red circle containing the white "x" at the margin indicates an error on the line (the semicolon at the end of the line is missing), and at the location where the semicolon should be there is a red wavy underline.

Hovering over either error indicator with a mouse will display a pop-up message with suggestions for correcting the error. 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. The most common errors in programming include misspelled words, forgetting closing quotes, and forgetting

Appendix C

closing parentheses. The IDE will highlight this type of error, but logic errors must be found by thoroughly testing the program.



Exiting Eclipse

To leave Eclipse, save any changes by choosing "File" on the menu bar, and "Save" from the drop-down menu, or just use Control-S, and then close the program. Workspace information will be saved as the program shuts down.

Eclipse – Quick Start

- Launch Eclipse, select the workspace folder from the list, Eclipse will start
- 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

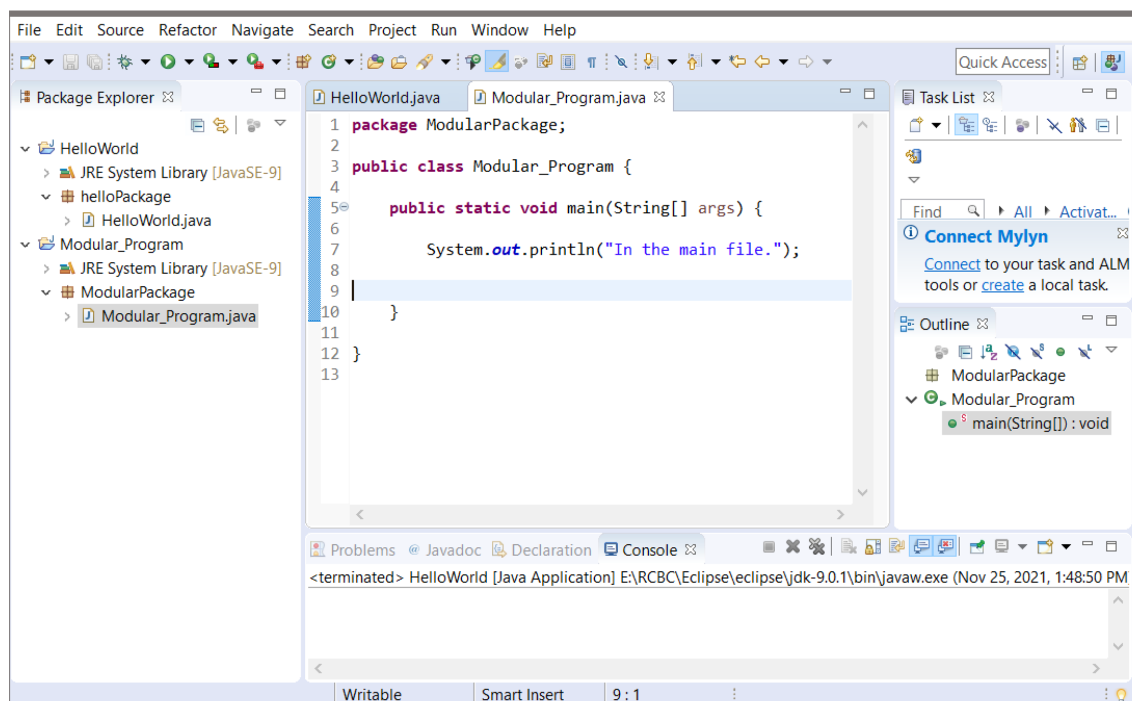
Appendix D

Modular Programming - Creating a Second File in Eclipse

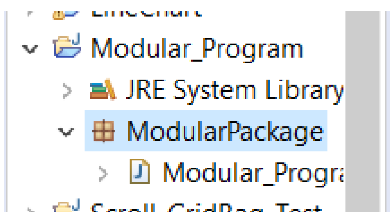
Multiple files are used to separate various parts of the program. This is referred to as modularization. By creating modules (files), the program is easier to maintain and add functionality, portions can be easily reused, and multiple engineers can work on various parts of a large program at the same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

The example creates a project with a main method, and a second file that is part of the project package. The second file contains a method that is used by the main program.

Create a project and main program. For the example, “Modular_Program” is the project and the package is “ModularPackage”. The Class is also “Modular_Program”.

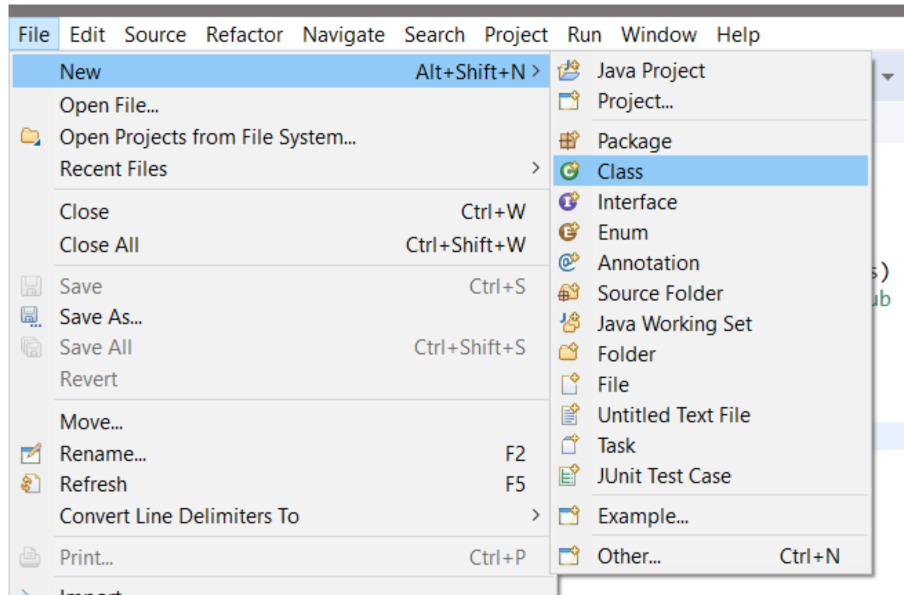


To create the second file, the package for the program should be highlighted in the Package Explorer (click on it if it isn’t highlighted).

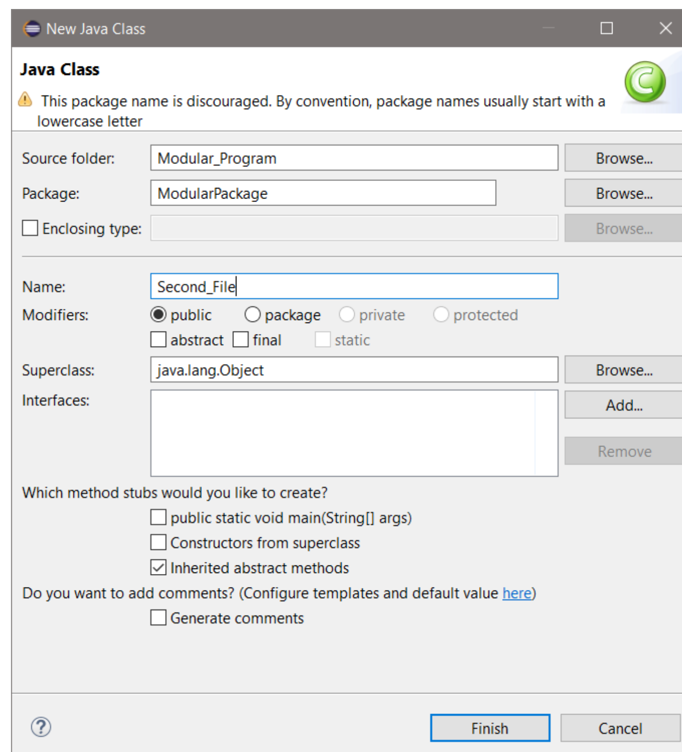


Appendix D

Then select File | New | Class from the menu.

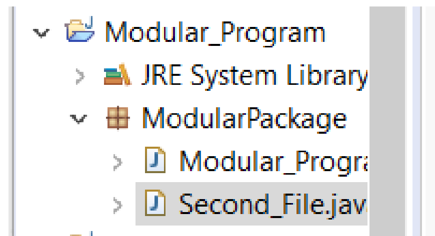


The Java Class window will appear. For the example, the class has been named Second_File and the check box for public static void main(String[] args) is not checked.

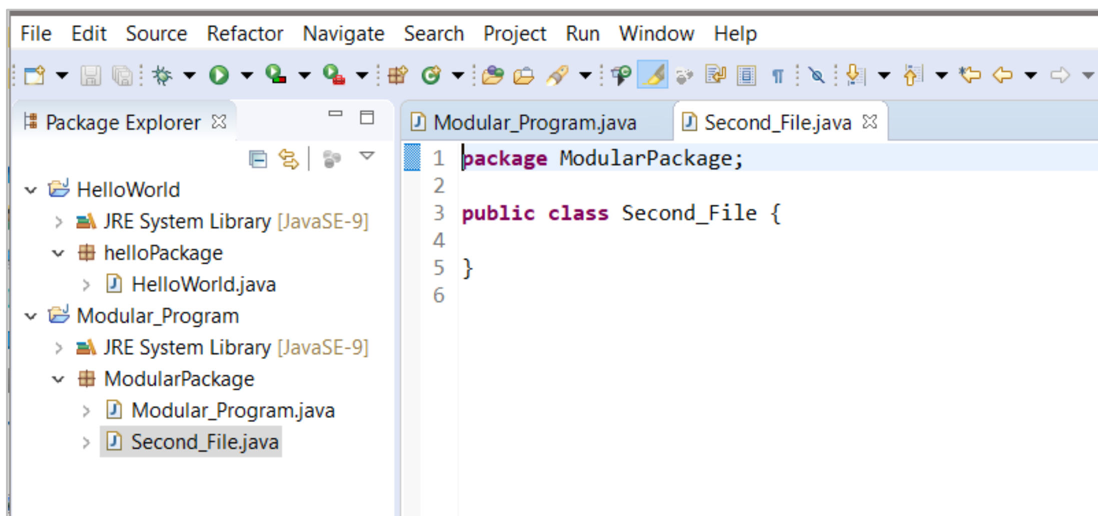


Appendix D

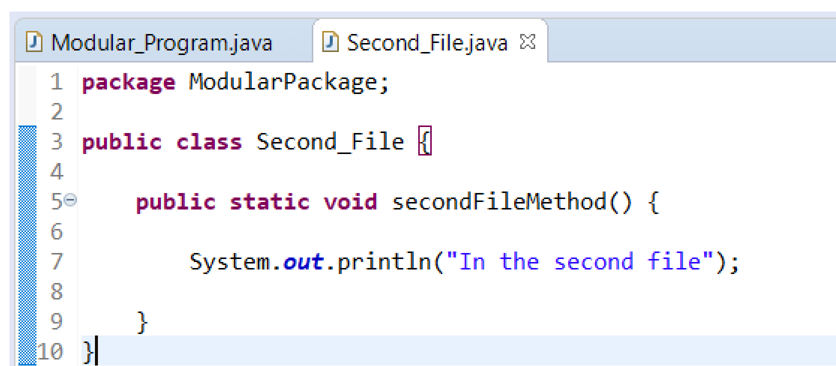
Click the Finish button, and the file is added to the program as shown in the Package Explorer.



There are now two tabs for the program in the IDE. Clicking on the tabs is a quick way of switching between files when writing code.



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



Appendix D

The method in the second file will be called from the main method 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 }
```

To see both files at once, click on the Second_File.java tab and drag it to the right of the edit panel and release the mouse.

Book_Programs_Java - Modular_Program/ModularPackage/Second_File.java - Eclipse IDE

Modular_Program.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 }
14
```

Second_File.java

```
1  package ModularPackage;
2
3  public class Second_File {
4
5      public static void secondFileMethod() {
6
7          System.out.println("In the second file");
8
9      }
10 }
11
```

Console

```
<terminated> Modular_Program [Java Application] E:\RCBC\Eclipse\eclipse\jdk-9.0.1\bin\javaw.exe (Nov 22, 2020, 6:17:51 PM)
In main.
In the second file
```

As the output shows, the main method successfully calls the method in the second file.

Appendix E

Resource Links

Eclipse.org

https://www.eclipse.org/

Eclipse User Guide

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

Java Development Guidelines – Carnegie Melon 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 F

Java Programming Guidelines and 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 the upperCasing format and descriptive names. A single letter or ambiguous abbreviation is unacceptable unless local to a method when no ambiguity is introduced (see below). Variables should be declared together whenever possible. Declaring a variable when needed increases maintenance time.

Unacceptable:

```
int hrs = 34;
float rt = 18.75;                  // Use double not float
String n = "Alicia Harad";
```

Acceptable:

```
int hoursWorked = 34;
double hourlyRate = 8.75;
String emp = "Alicia Harad";
```

Constants

Constants shall be named using all uppercase letters with underscores between words.

```
final int MIN_HEIGHT = 90;
final double TAX_RATE = 0.075;
static final double EARTH_RADIUS = 3863;
```

Appendix F

Braces

Opening and closing braces may be on separate lines and aligned horizontally, although the current preference is to have the opening brace on the first line of the code and the closing brace as the last line aligned with the first line as shown below.

Unacceptable:

```
if (value < num) { System.out.print("Status is: " + value);
```

Acceptable:

```
while (value < num)
{
   System.out.print("Status is: " + value);
   value = value + 2;
}
```

Preferred:

```
while (value < num) {

    System.out.print("Status is: " + value);
    value = value + 2;
}
```

Indentation, White Space, and Alignment

Programming style is critical to the readability and maintainability of source code. The following standards apply:

1. The tab key shall be used for indentation and alignment
2. Blocks of code associated with conditions, loops, and methods shall be indented and aligned
3. Blank lines shall be used to separate logical sections of code
4. Blank spaces shall surround operators and variables, and be placed after a comma

Unacceptable:

```
area=length*width;
```

Acceptable:

```
area = length * width;
```

Appendix F

Unacceptable:

```
for(int i=0;i<20;i++)
```

Acceptable:

```
for(int i = 0; i < 20; i++)
```

Unacceptable:

```
public static void main(String[] args) {
Scanner in=new Scanner(System.in);
int length, width, area, perimeter, diagonal;
System.out.print("What is the length of the rectangle?");
length=in.nextInt();
System.out.print("What is the width of the rectangle?");
width=in.nextInt();
        perimeter=(2*width)+(2*length);
        area=length*width;
        diagonal=(int)Math.sqrt(length+width);
System.out.println("Area:" +area);
System.out.println("Perimeter:" +perimeter);
System.out.println("Diagonal:" +diagonal);
}
```

Acceptable:

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

    Scanner in = new Scanner(System.in);
    int length, width, area, perimeter, diagonal;

    System.out.print("What is the length of the rectangle?");
    length = in.nextInt();

    System.out.print("What is the width of the rectangle?");
    width = in.nextInt();

    perimeter = (2 * width) + (2 * length);
    area = length * width;
    diagonal = (int)Math.sqrt(length + width);

    System.out.println("Area:" + area);
    System.out.println("Perimeter:" + perimeter);
    System.out.println("Diagonal:" + diagonal);
}
```

Appendix F

Using while (true), while (1), break, and continue are unacceptable

These expressions and operations show an inability to design and use functional logic and shall not be used. Break and continue bypass logic hindering debugging and readability.

Loop conditions shall consist of logical Boolean expressions.

Unacceptable:

```
while(true)

while(1)
```

Acceptable:

```
while(number < 100)

while(valid == true)

while(valid)
```

Unacceptable:

```
while (true) {

   if(number < 10) {

        System.out.println("$" + number);
        number++;
   }
   else {
        break;  // unacceptable use of break
   }
}
```

Acceptable:

```
while (number < 10) {

   System.out.println("$" + number);
   number++;
}
```

Note that it takes less code to implement the logic correctly.

Appendix F

Comments

Comments shall be used to introduce and explain complex areas of the code and equations. Comments may appear before code or in line with code. Inline comments should be tabbed for alignment.

```
// Computes the area of a sphere
area = 4 * PI * radius * radius;


area = 4 * PI * radius * radius;        // area of a sphere
```

Methods

Method names shall follow the upperCasing style and describe the operation the method performs or the value that it returns.

```
getBalance

doTaxCalculation

setCustomerName
```

The body of the method shall be indented and follow the standards above for style. A comment should be included when an explanation adds clarity.

Acceptable:

```
// Returns the square of the value passed in
public static double getSquaredVal(double val)
{
        double squaredVal = val * val;
        return squaredVal;
}
```

Preferred:

```
public static double getSquaredVal(double val) {

        double squaredVal = val * val;

        return squaredVal;
}
```

Appendix F

Methods that return a value shall have one return statement. Void method shall not have a return statement. Returning from multiple lines or to bypass code is unacceptable.

Unacceptable:

```
public static double computeArea(double radius) {

        double area = 0;

        if(radius <= 0) {
                return 0;
        }
        else {
                area = Math.PI * radius * radius;
                return area;
        }
}
```

Acceptable:

```
public static double computeArea(double radius) {

        double area = 0;

        if(radius > 0) {
                area = Math.PI * radius * radius;
        }

        return area;
}
```

Unacceptable:

```
public static void displayName(String name) {

        if(name.length() == 0) {
                return;
        }
        else {
                System.out.println("Hello " + name);
        }
}
```

Appendix F

Acceptable:

```
public static void displayName(String name) {

        if(name.length() != 0) {
                System.out.println("Hello " + name);
        }
        else {
                System.out.println("Name was not entered");
        }
}
```

Package names

Package names shall be descriptive and all lowercase letters

Class names

Class names shall begin with an uppercase letter followed by uppercasing.

```
public class StudentInfo
{
        ...
}
```

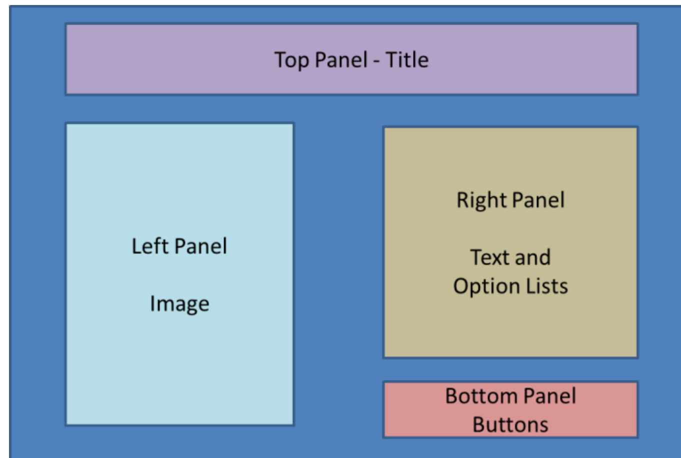
Program Layout/Logic

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 G

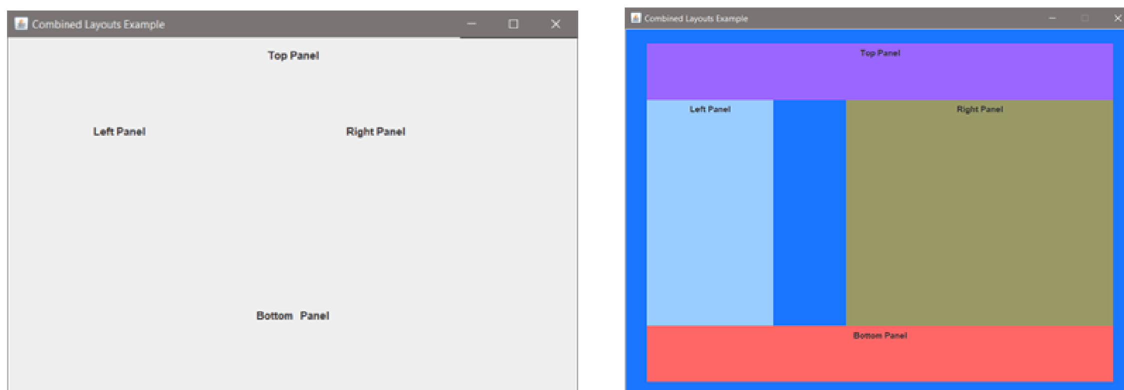
Multiple Panels and Layout Managers Example

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 G

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 the 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 G

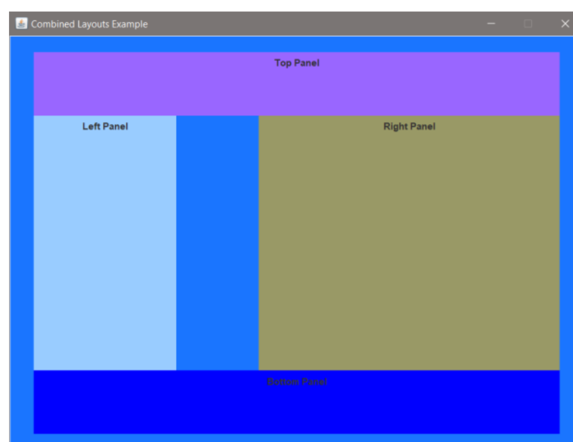
```
        // add the main panel to the Frame.
        mainFrame.add(mainPanel);

        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 G

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.

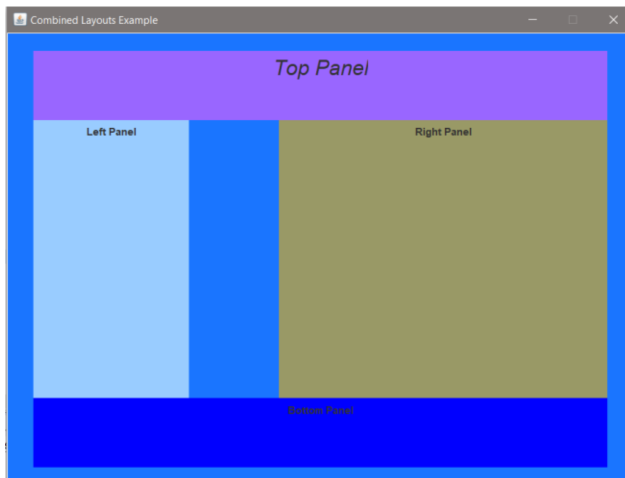
```
public CombinedLayouts() {                    // constructor

    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.

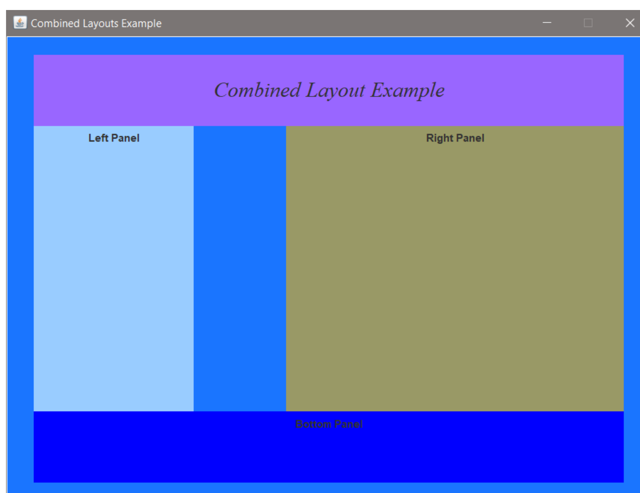
Appendix G



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.

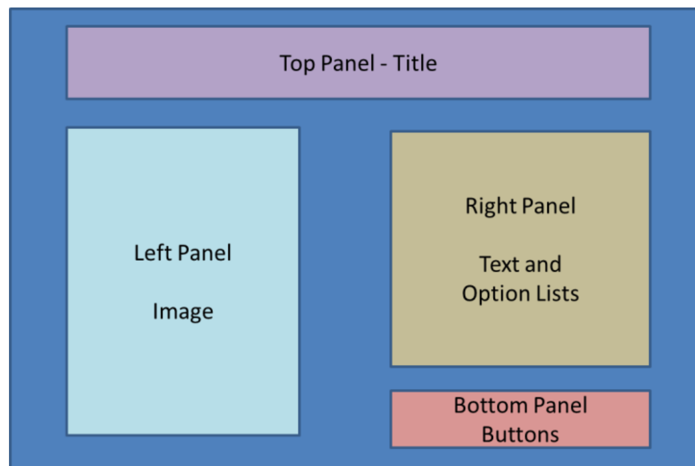


Note that at each step, running the program ensures that any errors introduced are corrected immediately. Frequent testing can save hours of debugging and fixing minor errors.

Appendix G

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 well 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 G

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.

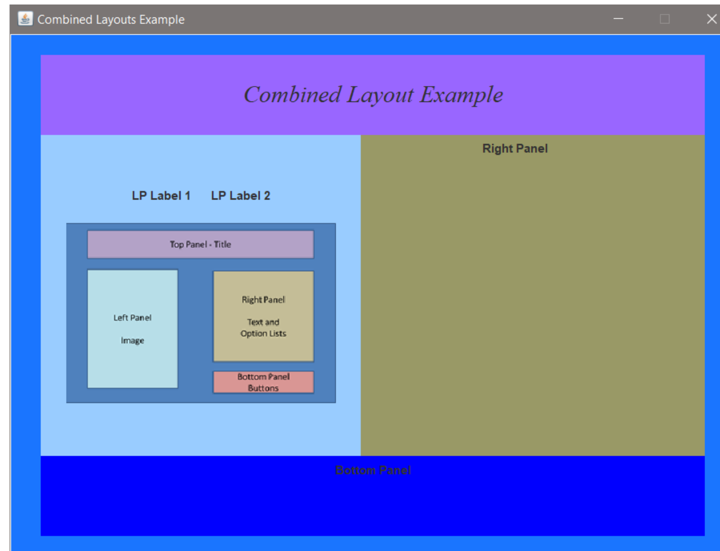
```
public void populateLeftPanel(JPanel leftPanel) {

    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 G



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 adjusted. 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);
```

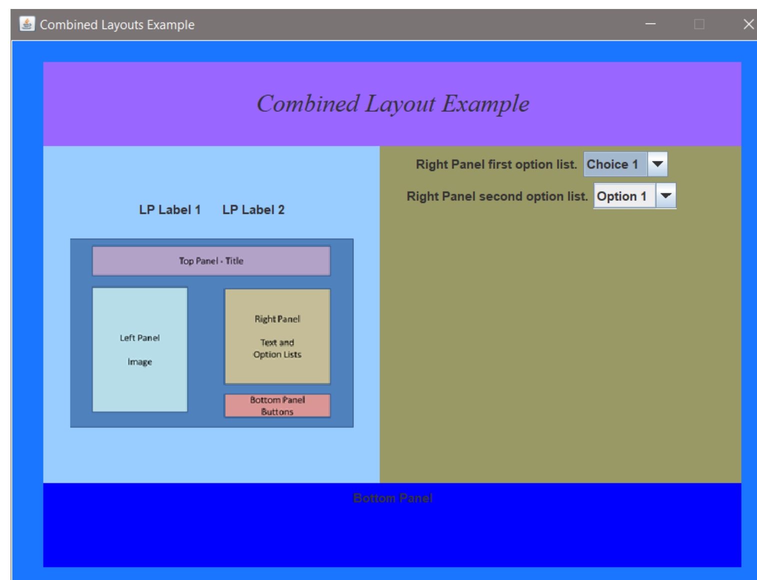
The choice of what layout to use for this or any other panel is subjective, and programmers tend to use the layouts that they are more familiar with. The right panel has text and option lists. This could be done top-down and a flow layout would work, or a grid. Left alignment would be appealing (a design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

```java
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);

}
```

A 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.



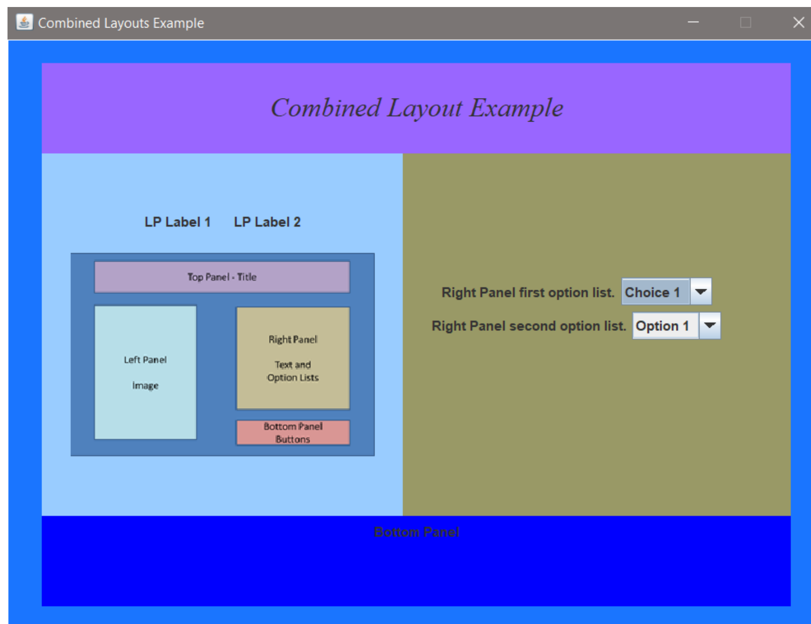
Java provides rigid areas and struts that can be used as spacers to position components. A rigid area can be a custom size and added to the panel to “push” other components around. It is invisible and acts as a spacer. A strut is similar but has no height dimension.

Appendix G

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);
```

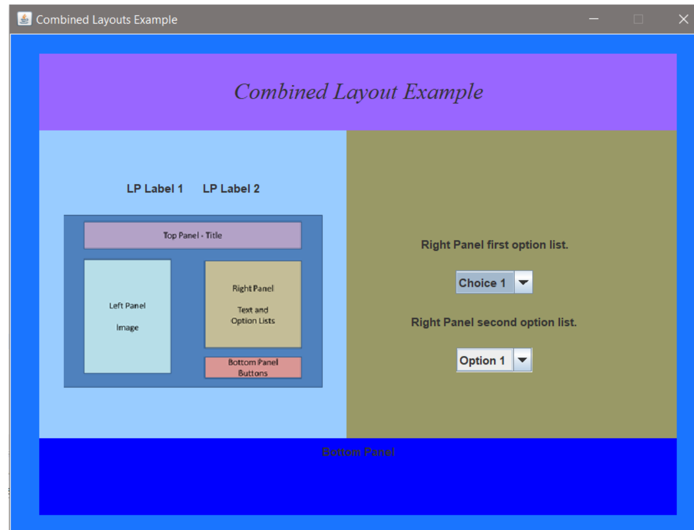


Additional rigid areas could be created and added between each of the components to force them to the next row, and they would be centered by default.

```
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 G

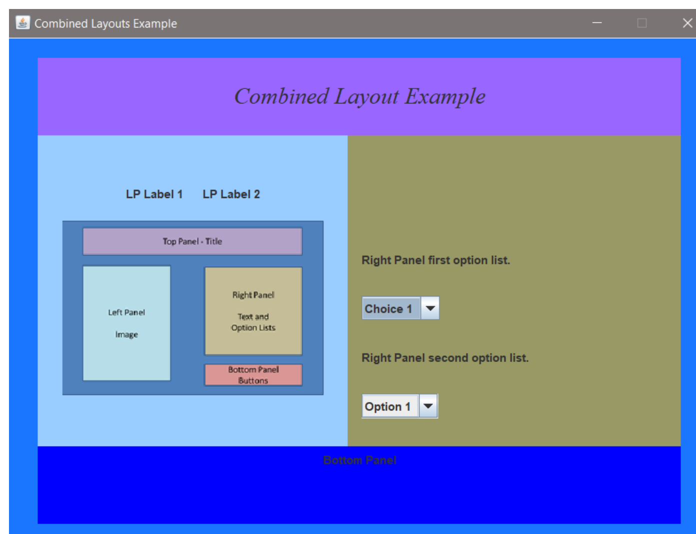


The horizontal alignment of the components is centered as a default. A grid layout would allow them to be aligned by row columns using constraints and allow placement within each column (EAST, WEST). However, the Flow Layout constructor can accept arguments for alignment that apply to all of the components 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 G

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.

A 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);
```

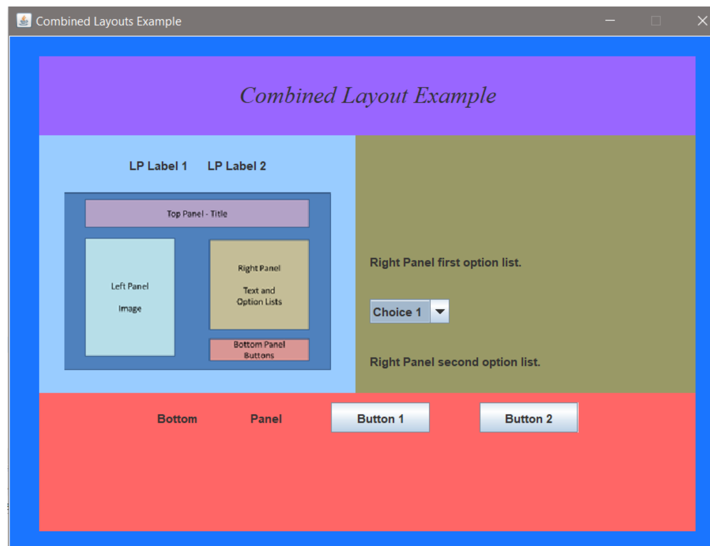
The components for the panel are now moved to the method, and each time something is moved the program is run to ensure that none of the changes introduces an issue.

```
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
```

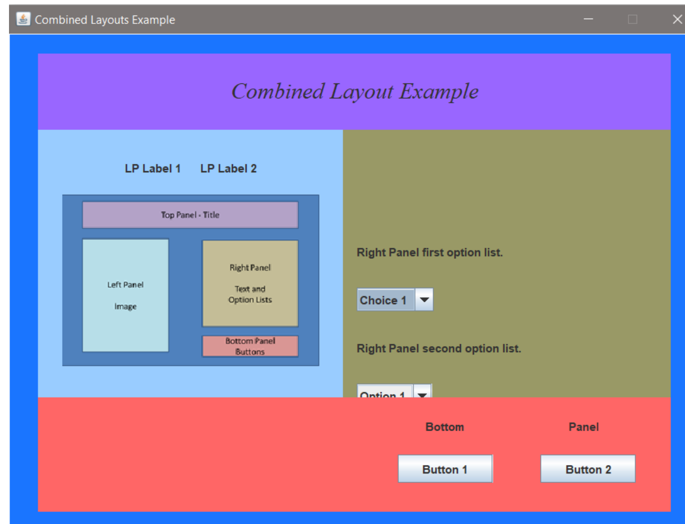
Notice in the code above that the button sizes can be set. This helps with consistency when the text on one button is shorter than the other, since the buttons will automatically size to fit the text. Also note that the flow layout (at least for now) uses centering, and the hGap and vGap are

Appendix G

guesses. The running program now produces the window below. Note that the right panel has been partially obscured and the right 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 G

```
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:

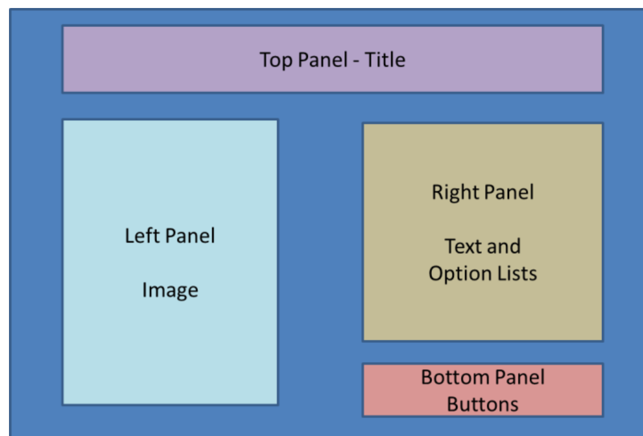
Panel	width	height	
Main	700	700	defaults to the frame size
Top	700	80	
Left	300	200	
Right	380	200	
Bottom	700	120	

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 G

Comparing the design sketch while making minor changes to various dimensions, makes things much easier and saves time.



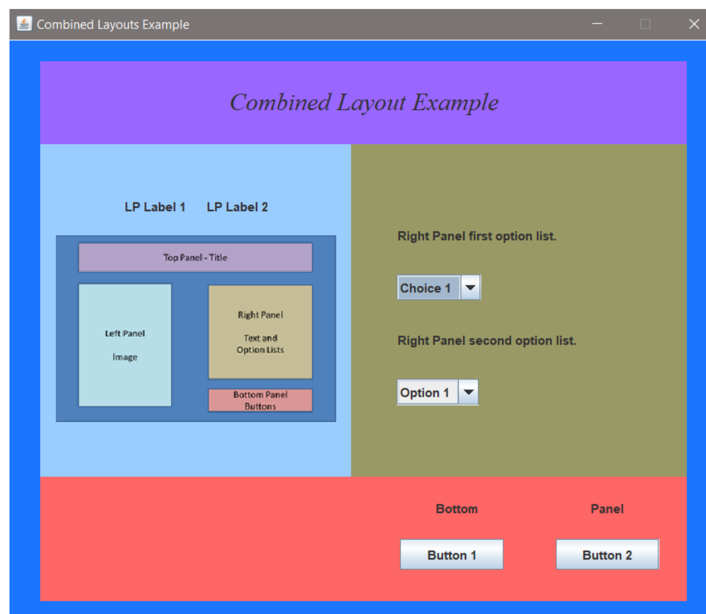
The first change made was to shorten the height of the main frame, and the left and right sides by 40 pixels. That forced an adjustment to the right panel spacing. The height of the topmost rigid area of the right panel was easily adjusted to 60 to realign things,

```
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.

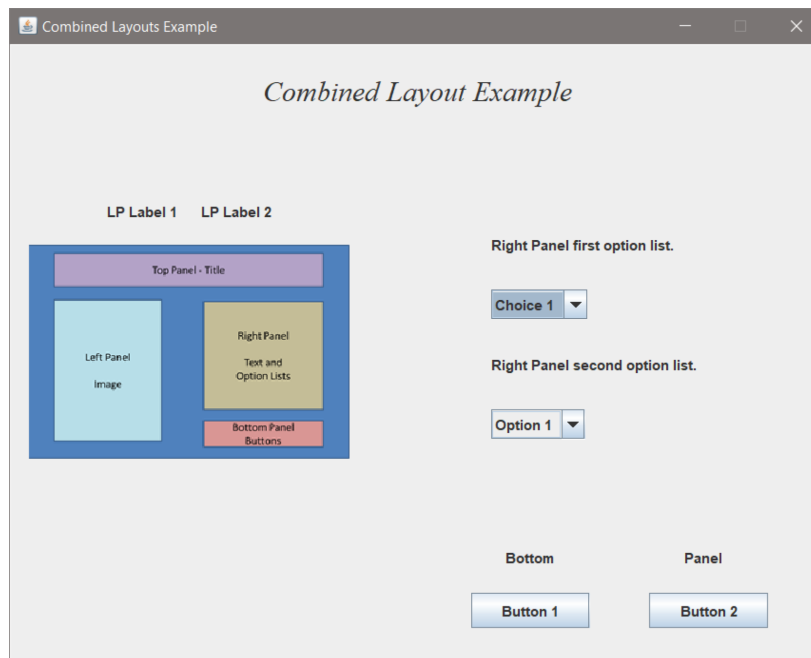


Appendix G

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 in the code below.

```
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 in the display that follows.

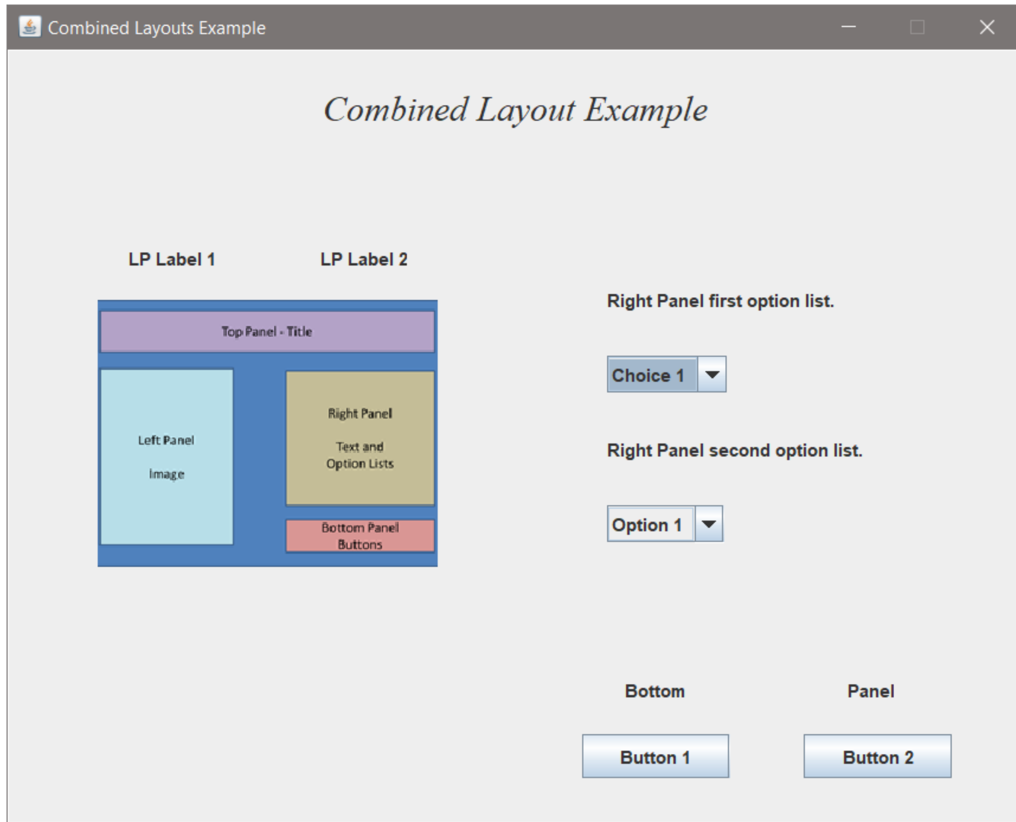
Appendix G

The interface is now complete in terms of 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. This organizes the development and enhances the quality and maintainability of the program. 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
```

The final User Interface is shown below

Appendix G



Index

A

acos(x) 34

Abstract Window Toolkit 254

ActionEvent 264

actionPerformed() 264

ActionListener, class 263

add(), method

  ArrayList 153

  ButtonGroup 270

  JFrame 260

  JMenu 296

  JMenuBar 296

  JPanel 260

addActionListener() 263

addWindowListener() 295

Agile Development 13

Agile Methodologies 13

Addition (+) operator

  defined 30

  concatenation 42

alignment, output 39

anchor Appendix G (7)

and operator && 66

Animation 310

appending data, files 177

append() 177

Arguments, passing 33

ArrayList 153

  declaring 153

  add() 153

  get() 154

  remove() 154

  set() 155

  size() 154

  sorting 157

asin(x) 34

Assignment operator 56

atan(x) 34

AudioInputStream 104

AudioSystem 307

  getClip() 308

AWT Abstract Window Toolkit 254

B

Backslash, displaying "\\" 40

bar chart 303

Boolean data type 70

  expressions 56

  logic 66

border

  JPanel Appendix G (5)

BorderFactory 281

BorderLayout 281

Bounds checking 160

BoxLayout 259

browse() 309

Browser, launch 309

BufferedImage 279

Button

  addActionListener() 264

  create 263

  labels 262

  listener 264

  radio 270

  setBackground() 262

  setForeground() 262

  text 263

Index

ButtonGroup 270
- *add()* 270

Byte 5

C

calling methods 111
casting 34
case-sensitive 27
Centering windows 257
Char data type 41
characters 41
- comparing 41
- escape 39
- finding in Strings 41
- indexing 41
- newline 39
- tab 39

Charts 299
- bar 303
- flow 11
- line 303
- Tools 303

check box 271
Classes 191
- wrapper 156

Class Name, method call 32
Clip, audio 307
close(), Scanner 35
Closing programs 294
Columnar data 279
Collections.sort() 157
Combo Box 269
Comments 22
Concatenation 42
Conditional statements 53
Constants 29

D

data types 25
Date 305
DateTimeFormatter 305
Decision structures 53
delimiter 178
Desktop 185
Design Phase 35
Development
- Agile 13
- cycle 13
- methodologies 14

Dialog boxes 126
- confirmation 127
- File, Save As 186
- *JFileChooser()* 184
- *showInputDialog()* 127
- *showMessageDialog()* 126

Dimension Appendix G (2)
display output 36
dispose() 295
Division 30
do-while loop 88
double 25
Double.parseDouble() 64
drawLine() 130
drawOval() 130
drawRect() 130
drawString() 130
Drop-down menus 295

Index

E

Eclipse, IDE 29
else condition 57
enhanced for loop 149
Email, launching 309
Equal sign, assignment 56
Equivalence operator 56
Errors 10
- cost by phase 10
- dialogs 126
- IDE notification 27
- StackTrace 174

Escape sequences 39
Event listener 264
exec() 307
Exceptions 167
- Checked 176
- FileNotFound 172
- Handlers 172
- IndexOutOfBounds 160
- NumberFormat 65
- Unchecked 176

exponentiation 34

F

File 167
- appending 177
- *close()* 171
- opening 169
- read 169
- read numeric data 177
- writing numeric data 177
- writing text 171

File selection dialog 184
File "Save As" dialog 298
FileNameExtensionFilter() 185
FileWriter class 177
fillOval() 130
fillRect() 130
final, key word 29
float data type 25
floating point division 29
Flowchart 11
FlowLayout 259
font
- create 279
- *setFont()* 279

for loop 86
for-each loop 149
format(), String 277
formatted output 37
format specifier 37
Frame component 128
- methods 257

G

get(i), ArrayList 154
getAbsoluteFile() 308
getAudioInputStream() 308
getClip() 308
getContentPane() 301
getDesktop() 309
getRunTime() 307
getSelectedFile() 185
getSelectedItem() 269
getSource() 271
getText() 266

Index

Global variables 30
Graphics object 129
GridBagLayout Appendix G (6)
GUI 253
- design 256
- positioning components 259
- Layout Managers 259

H

hasNext() 65
hasNextDouble() 65
hasNextInt() 65
hasNextLine() 65
Hello World 20
HTML 306
hypot(x) 34

I

if-else 57
images 279
ImageIcon() 279
imageIO.read() 280
import 34
- wildcard 128

indentation 44
indexes
- characters 41
- ArrayLists 154
- Strings 41

Information dialog box 126
Insets Appendix G (7)
instance 192
integer 25
Integer.parseInt() 64
Interface Design 254
IPO document 122
isDesktopSupported() *309*
isdigit() 100
isLetter() 100
isLowerCase() 100
isUpperCase() 100
isWhiteSpace() 100
Iterative Enhancement 125

J

Java 8
Java Foundation Classes 254
JavaFX 303
JButton 262
JComboBox 269
JComponent 129
JDK 19
JFileChooser 184
JFrame 257
JFreeChart 303
JLabel 260
JOptionPane 126
JPanel 260
JScrollPane 294
JTextField 261
JDK 19
JRE 19
JVM 19

K

keyboard input 34

Index

key words 22

L

Label component 260
Layouts
    multiple Appendix G
layout manager 259
length() 42
Line drawing 130
Line Chart, JavaFX 303
Line, *drawLine()* 130
line feed 39
LocalDateTime 305
log() 34
Logical operators 66
Loops 81

M

Main method 21
Mathematical operators 30
Math Methods 33
Math.PI 33
Math.pow() 34
Math.random() 101
Math.round() 33
Menu 295
Methods 111
    calling 113
    naming 119
Mixed-type expressions 32
Modulus (%) operator 30
Multi-file Programs 124
Multiplication (*) operator 30

N

New, keyword 193
Newline (\n) character 39
next() 35
nextDouble() 36
nextInt() 35
nextLine() 36
not operator "!" 66
Numbers
    floating point 6
    formatting 37
    random 101
NumbeFormatException 176

O

Objects 191
Object Behavior Diagram 206
Object Oriented 191
Object Sequence Diagram 206
Open file dialog 184
Operators
    logical 66
    mathematical 30
    precedence of 32
    relational 56
Option Lists 255
or operator || 66
Output
    displaying 24
    file 171
    formatting 37
Override 246

Index

Notation 295

P

Package, Java 20
Package Explorer 124
Package, import 34
paintComponent() 129
Panel 259
add() 260
setBackground() 262
setBorder() 281
setVisible() 128
Panels, Multiple Appendix G
Parameter 112
Passing arguments 114
Pie Chart 303
plotting 299
Precedence 32
Primitive data types 156
print function 36
printf() 37
print formatted 37
println() 36
PrintWriter 171
private 196
Process 9
Program design 93
protected 196
Pseudocode 10
public 196

Q

Quick Fix 172
quotes, displaying \” 39

R

Radio buttons 270
groups 270
random numbers 101
Read, file 169
Read, keyboard input 34
Relational operators 56
remove(), ArrayLists 153
repaint() 300
replace(), Strings 100
Requirements 9
return statements 113
rounding 33

S

“Save As” dialog 298
Scanner 34
Scene 304
Scroll bars 294
Scrum 13
Sequence Diagram 206
set(), ArrayList 155
setBackground() 262
setBorder() 281
setColor() 132
setCurrentDirectory() 185
setDefaultCloseOperation() 257
setFilter() 185
setFont() 279
setForeground() 262
setIcon() 280

Index

setJMenuBar() 296

setLocationRelativeTo() 274

setResizable() 259

setSize() 262

setText() 267

setTitle() 260

setVisible() 257

showInputDialog() 126

showMessageDialog() 126

sin(x) 34

Sprint 13

Software Development Process 13

Sound 307

SpringLayout 259

sqrt() 34

Stage 304

Standards Appendix F

static 121

String 40

- concatenate 42
- conversion 64
- comparing 64
- *StringBuilder()* 67
- *charAt()* 41
- *equals()* 63
- *format()* 277
- *isNumeric()* 177
- *length()* 42
- *replace()* 100
- *substring()* 43
- *toLowerCase()* 101
- *toUpperCase()* 101
- *split()* 67
- *trim()* 177

Subclass 227

Substring 43

Subtraction (-) operator 30

super 229

swing components 254

T

tab \t" 40

tan(x) 34

Text files 167

Time 305

Timer 305

- *start()* 306
- *stop()* 310

TimerListener 306

toLower() 43

toUpper() 43

Toolkit 254

Traceback 15

trim(), Strings 177

Truncation 31

try/catch 172

try-with-resources 175

Types, data 25

U

UML 205

Unified Modeling Language 205

UML Diagram 205

UML Superstructure 205

UnsupportedAudioFile 308

URI 309

Uniform Resource Identifier 309

Index

useDelimiter() 179
User interface 4

V

Validating Input 65
Variables 18
  assignment 28
  constants 29
  Naming conventions 26
  Types 25

W

W3C 26
W3Schools, link Appendix E
Web browser, launch 309
weightx, weighty Appendix G (7)
While loop 82
Wildcard import 128
Window
  JFrame 128
  image 279
  menu 295
  parent 126
  setDefaultCloseOperation() 294
  setLocationRelativeTo() 257
  setSize() 257
  setTitle() 136
  setVisible() 257
WindowAdapter() 295
WindowClosing() 294
WindowEvent 295
WindowListener 295
Workspace, Eclipse Appendix C (1)
World Wide Web Consortium 26
Wrapper Classes 156

X

x axis, line 301
x coordinate 130

Y

y axis, line 301
y coordinate 130