

Introduction to Java Programming

Dr. Lixin Tao
<http://csis.pace.edu/lixin>
Computer Science Department
Pace University

September 9, 2015

Table of Contents

1	Computer Systems and Programming Languages	1
1.1	Problem vs. problem instances	1
1.2	Structure of a personal computer	1
1.3	CPU	2
1.4	Memory	3
1.5	Disk file system	5
1.6	Display	8
1.7	Keyboard	9
1.8	Computer Software	9
1.9	Java programming language	10
2	Software Installation	13
2.1	Installation of Java SE JDK	13
2.2	<i>Windows explorer and text editor</i>	15
2.3	Typical process for creating and running a Java program	16
3	Java Basics	18
3.1	Basic Hello World program	18
3.2	Code style and command-line arguments	19
3.3	Java package	21
3.4	Best practice for organizing multiple class projects	24
3.5	Local variables, expressions, and assignment	26
3.6	Basic built-in data types and type casting	29
3.7	Command-line arguments and loops	33
3.8	Calculator with if-else statements	36
3.9	Calculator with switch statement	38
3.10	Calculator with exception processing	41
3.11	Conditionally interrupting loops	44
3.12	Arrays	45
3.13	Method declaration and invocation	47
3.14	Java documentation comments	51
3.15	Scopes of local variables	54

-
- Copyright Dr. Lixin Tao, Pace University, ltao@pace.edu. This document should not be copied partially or in full, or published in any form, without the permission of the author.
 - This is an on-going project. The tutorial contents will be updated and expanded with time.

3.16	Formatting output.....	55
3.17	Timing the evaluation of π	58
3.18	Matrix multiplication	60
3.19	Interactive command-line data input	62
3.20	Window-based data input/output	64
3.21	Text file input/output	65
3.22	Recursive vs. iterative methods: factorial	69
3.23	Recursive vs. iterative methods: Fibonacci numbers.....	73
3.24	Algorithm and Towers of Hanoi	76
3.25	Bubble sort	81
3.26	Binary search	83
4	Object Oriented Programming.....	88
4.1	A simple <i>accumulator</i> class	89
4.2	A simple class for circles	90
4.3	Method overloading	92
4.4	Class inheritance	94
4.5	Abstract methods	97
4.6	Java interfaces	99
4.7	Java data field/method accessibility.....	101
4.8	Java collections	101
4.9	String tokenizers	105
4.10	Exception handling	107
5	Multi-threading	110
5.1	Controlling multi-threads in action.....	111
5.2	Two ways of creating threads	111
5.3	Resolving critical region problem with thread synchronization	114
6	Networking	117
6.1	A client-server program for calculating areas of circles	117
7	Database Programming	124

1 Computer Systems and Programming Languages

1.1 Problem vs. problem instances

There are many real-world problems that need be solved efficiently. Examples include weather forecast, mail sorting, chess playing, and the real-time control of a satellite. A computer system is used to help people solve complex problems efficiently.

A *problem* is a type of challenge. A problem is made up of many *problem instances*. For example, sorting (rearranging) integers into a non-decreasing sequence is a problem, and sorting integers in set $\{2, 5, 3, 1\}$ into sequence “1, 2, 3, 5” is a particular instance of the sorting problem.

A computer is designed for solving complex real-world problems fast. It is a tool. It is the human being that will instruct the computer how to solve problems. A *program* is basically a sequence of such instructions. A computer program is written to solve a particular problem. When we feed data (problem instances) to our running program, we solve particular problem instances. For example, we may want to write a program to address the “sorting integer” problem. Such a program is supposed to be able to solve any instances of the “sorting integer” problem. If a program can only solve a particular instance like “sorting $\{2, 1, 3\}$ ”, this program is of little use because it can only solve a *special case*. When we say that a program is *correct*, it means the program can solve all instances of the problem for which the program is designed for, as long as the input data (problem instances) are meaningful (as an example, feeding $\{a, 2, b\}$ to this program is not meaningful). In addition, if the program can also behave gracefully and not to give wrong answers when it is fed wrong input data, we also say that the program is *robust* (as an example, fed with $\{a, 2, b\}$, a robust integer sorting program should refuse to print out any sequence of the input values, and it should print out a message that “the input data are not all integers”).

1.2 Structure of a personal computer

A useful personal *computer* includes hardware, software, and data. The computer hardware can understand and execute a small set of low-level (simple) *machine-language instructions* like storing an integer in a memory location, summing two floating-point numbers (like $1.2 + 2.4$), or printing a letter on the screen. The computer software is made up of computer *programs*, which are sequences of machine-language instructions for the computer hardware to execute and solve problems. The data represent problem instances and the solutions to them. While the computer hardware is normally designed for general purpose, a program that runs on it is designed to solve a particular problem. Given input data for any instance of that problem, the program will use the hardware resources to compute the solution to the problem instance.



Figure 1 Computer as a data transformer

Computer *hardware* is typically made up of one *CPU* (central processing unit) for computing, one main memory for temporally storing programs and data that are processed by the CPU, a *hard disk* for storing programs and data for longer time (surviving power shutdown), a *keyboard* for getting user input, and a *display* for displaying computer output. All data processing is conducted by the CPU. When you shutdown your computer, all data in the CPU and main memory will be lost. But programs and data saved on a hard disk will not be affected by power shutdown. While hard disk is safe for storing large amount of programs and data, it is too slow relative to the CPU's working speed. The main memory functions like a high-speed copy of the *active* (those currently processed by the CPU) programs and data for improving the CPU's performance.

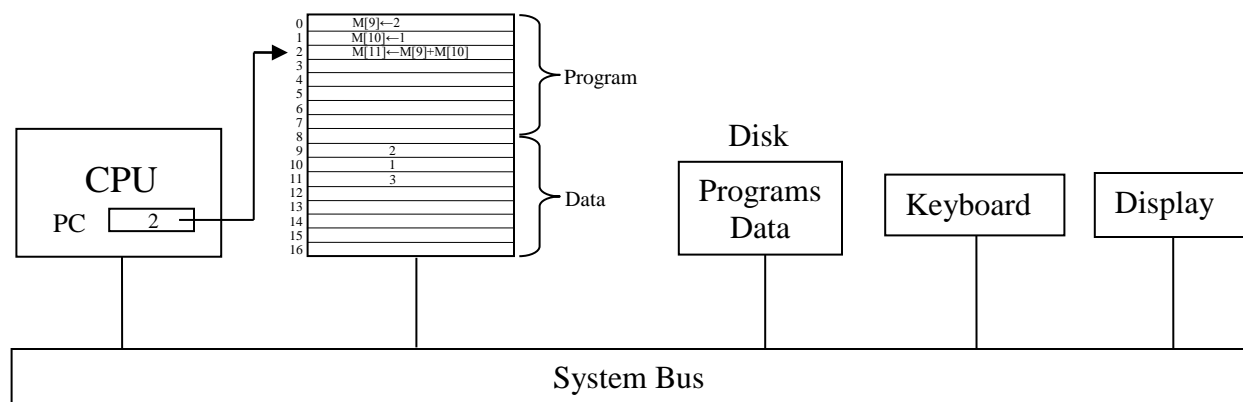


Figure 2 Computer organization

The CPU, the (main) memory, the (hard) disk, the keyboard, and the display are all devices connected through a *system bus*, as shown in Figure 2. At any instance, only one device can broadcast data on the system bus, but every device on the system bus can listen to the data broadcast.

1.3 CPU

All computing is conducted by the CPU. CPU is like the brain or control center of a computer.

A CPU is designed to understand a small set of machine-language instructions. The CPU is responsible for reading successive instructions, as well as data needed by the instructions, of a program stored in the main memory and carrying them out. All CPU activities are synchronized by the cycles of a digital clock. This clock repeats CPU cycles, and the number of CPU cycles repeated during each second is called the number of *Hz* indicating the speed of the CPU.

A CPU uses a *program counter (PC)* to remember the address for the next program instruction in the memory to be executed. Just before the execution of a program, the program counter is set the memory address of the first (in logical sense, not necessarily the physical first instruction; for example, a program may start its execution from instruction 5) instruction of the program to be executed. After the execution of each machine instruction, the value of the program counter is updated for the address of the next instruction to be executed. Since normally the instructions of a program are stored in successive memory locations and are supposed to be executed in that

order, the new value of the program counter is typically 1 plus its old value. If a program needs to jump out of order (say jumping back by ten instructions) to execute an instruction at a particular memory address (because some condition or event happened), the particular memory address will be set in the program counter for replacing the default successive address.

A CPU uses a few (from 8 to around 100) registers for speeding up its work. A register is a small data storage just large enough to hold a memory word. Registers are really scratch paper for storing the instruction under execution and the intermediate results of instruction execution. For example, to execute instruction “ $M[11] \leftarrow M[9] + M[10]$ ”, this instruction will first be copied in an *Instruction Register* (IR), two *general-purpose registers* will be used to hold value copies of memory words $M[9]$ and $M[10]$, another general-purpose register will be used to hold the summation of the values in the previous two registers, and resulting value of the last register will be then copied into memory word $M[11]$. Since the registers are part of the CPU chip, they work at the same speed as the CPU cycle, which is much faster than the memory cycles (the heartbeats used to synchronize memory accesses).

The execution of an instruction by the CPU is conducted in a few phases synchronized by the CPU clock cycles, as described below, and these phases repeat to execute a program.

1. The program counter's value is used as a memory address to read out the instruction from the memory location with that address, and the instruction is saved in the CPU's Instruction Register (IR).
2. The program counter's value is set to the memory address for the next instruction to be executed.
3. The value of IR is decoded, and the data needed for the execution of the instruction is read from memory and copied into some general-purpose registers.
4. The operation of the instruction is carried out and the result data is saved in a general-purpose register.
5. The result value is copied from the general-purpose register to its destination memory location.
6. Go back to step 1.

The speed of a CPU is determined by number of CPU cycles per second, and a CPU cycle is the basic time unit for the CPU to do some work. Each machine instruction takes one to ten cycles to get executed (to make our examples simpler, we assume each machine instruction takes one CPU cycle to get executed). Today's PC CPUs normally have a cycle of 1.5 GHz to 3.5 GHz, where a GHz is one giga Hertz, or 2^{30} cycles per second.

1.4 Memory

The basic data storage unit of a computer is a *bit*. A bit can store either 0 or 1, which may be used to represent false and true, or off and on of an electrical switch, respectively.

Eight consecutive bits make a *byte*. A byte can hold an integer of values between 0 and 255. The value of a byte can represent a key that we type on the keyboard.

A computer *word* consists of two bytes (16 bits, used by CPUs of 1970s), four bytes (32 bits, typical of today's PC that will soon be outdated), eight bytes (64 bits, for the coming generation

of PCs and for business computing; great effect for dynamic computer games), or sixteen bytes (128 bits, for supercomputers that need extremely high computing precision), depending on the hardware CPU architecture and bus architecture.

The computer's main memory is a one-dimensional array (sequence) of memory words. Each word is assigned a unique address, starting from zero (0). There are machine instructions for storing an integer into a memory word, or reading out the value of a memory word. A memory word is just a cell for storing program or data. The most important properties of a memory are:

- When a new value is stored into a word, the word's old value is erased and replaced with the new value.
- When the current value of a word is read, the value is not modified by the reading process.

Other forms of data, including alphabetic data, floating-point numbers and machine instructions, are encoded into integers and stored in one or multiple consecutive words. If a piece of data takes four words to store, say at addresses 200, 201, 202, and 203, then we say that the piece of data is stored at memory address 200, the smallest word address used for storing that piece of data. A computer memory is for storing both computer programs and data (to be manipulated by the programs).

We normally use symbol **M** to denote the main memory, and use **M[i]** to denote the i-th word of memory M. Here "i" is called an index of M, or the address of the memory word. Therefore, the main memory is made up of a long sequence of computer words represented as

$$M[0], M[1], M[2], M[3], M[4], \dots$$

PCs' main memory typically contains 256 MB (mega bytes, which is the same as 2^{20} bytes), 512 MB, 1 GB (giga bytes, which is the same as 2^{30} bytes), or 4 GB. A useful but smaller memory unit is one KB (k bytes), which is the same as $2^{10} = 1024$ bytes. For exaggerating the actual memory or disk size, the computer memory and hard disk manufacturing companies usually use 10^3 to denote a K, 10^6 an M (mega), and 10^9 a G (giga), which are smaller than our original definitions.

When a computer's power is shut off, the data in the memory will be lost. When a computer is turned on, each memory word will have random data that we cannot predict. One important task of a program is to initialize computer words that it uses before it starts to use those words in its computation.

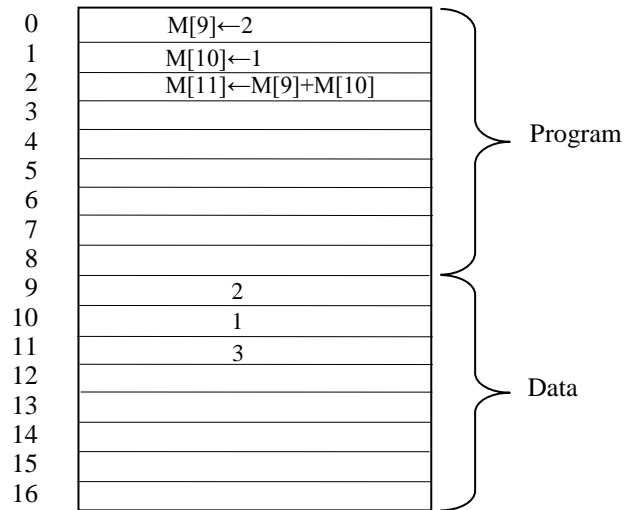


Figure 3 Memory with program and data

Figure 3 shows an example main memory with 17 words. The first 9 words, at addresses 0 through 8, are storing a program. The last 8 words, at addresses 9 through 16, are storing data. At the very beginning, the program counter PC has value 0, and machine instruction “ $M[9] \leftarrow 2$ ” in memory word $M[0]$ is executed, which stores integer 2 into memory word $M[9]$, the memory word with address 9. Then the program counter PC automatically increases its own value by 1 to become 1 and points to the next machine instruction. In the next CPU cycle, machine instruction “ $M[10] \leftarrow 1$ ” in $M[1]$ is executed, which stores integer 1 in memory word $M[10]$, the memory word with address 10. The program counter PC then automatically increases its own value by 1 to become 2 and points to the next machine instruction. In the next CPU cycle, machine instruction “ $M[11] \leftarrow M[9] + M[10]$ ” in $M[2]$ is executed, which retrieves values in memory words $M[9]$ and $M[10]$, adds them together, and stores the summation result into memory word $M[11]$, the memory word at address 11.

1.5 Disk file system

Hard disk is used for storing programs and data on long-term basis. Programs and data stored on a hard disk can survive power shutdowns. Today’s PC hard disks have capacities from 10 GB to 2 TB, where GB means giga bytes, or 2^{30} bytes; and TB means tera bytes, or 2^{40} bytes. Today’s computers normally boot up from its primary hard disk, where a reserved disk space is for storing a special program for loading the core of an operating system into the main memory (system bootup).

The programs and data are stored on hard disks in the unit of files. A *file* is a named unit of binary or alphabetic (text) data of any length. The name of a file typically has two parts separated by a period: the file name *stem*, and the file name *extension*. For example, a Java source code file may have name “Welcome.java”, where “Welcome” is the file name stem, which is supposed to be a concise name characterizing the purpose of the file; and “java” is the file name extension, which indicates the type of data that is stored in this file. By convention, “java” as a file extension is reserved by Java source code files. The file name extension tells a computer how to interpret the data stored under a particular file name.

Since we are encouraged to choose meaningful file names, it is possible that we incidentally create two files with the same name, leading to file name conflicts (the data of the second instance replaces that of the first instance). This leads to the introduction of the concept of file system directories. A file system *directory* can contain multiple files as well as other nested directories. A file system directory is a special file containing a table of contents for all files under its control. Two different files can have the same name as long as they belong to two different directories. File system directories are also referred to as file system *folders*. When a file directory is deleted, all of its contents are deleted too.

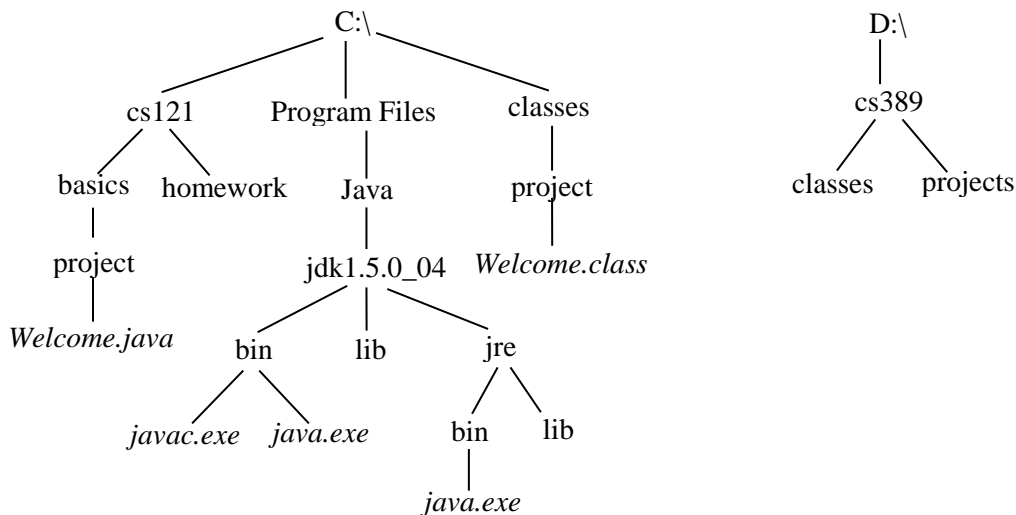


Figure 4 A sample file system

The file directories of a PC constitute a forest of directory trees, one tree for each hard disk partition (a *disk partition* is a logical disk; a hard disk can contain multiple disk partitions). In computer science, the root of a tree is drawn at the top, and its sons and descendants below the root. All the entities in a tree are called *nodes*. In our case, a node is either a file or a directory. There is a line connecting a father node and a son node in the trees. Figure 4 is a simplified file system for a particular PC. We assume that the PC has two hard disk partitions “C:\” and “D:\”, and they are the roots of their own directory trees respectively. Those nodes of the trees with names in *Italic font* are files, and the others are directories. The *absolute path* of a file or directory is the concatenation of all the node names on the path (connected lines) from the tree root to the file or directory, separated by “\” in Windows and “/” in Unix/Linux. For example, the absolute path of file “Welcome.java” in our example is “C:\cs121\basics\project\Welcome.java”. When we work in a *Command Prompt* window (*terminal* window for Linux), the *Command Prompt* window will work in a particular file system directory called the current *working directory*, for each directory tree. Inside a working directory, you can use *relative path* to denote a file or directory under the sub-tree rooted at the current working directory by concatenating all the node names on the path from the current working directory to that specific file or directory, excluding the current working directory, separated by “\” in Windows and “/” in Unix/Linux. As an example, if the *Command Prompt* window is currently working in directory “C:\cs121\basics”, then the relative path of file “Welcome.java” from the current working directory is “project\Welcome.java”; and if the *Command Prompt* window is currently working in directory “C:\cs121\basics\project”, then the relative path of file “Welcome.java” from the current working directory is “Welcome.java”. Inside a *Command Prompt* window, we can change the working directory by using the “cd” (change directory) command. There are two

special symbols used for representing two relative directories: “.” for the current directory, and “..” for the parent directory. Let us assume that the *Command Prompt* window is now working in directory “C:\cs121”. The following are some examples for changing the working directory:

Command	Working Directory	Comments
	C:\cs121	Starting working directory
cd \	C:\	Go to the directory tree root
cd cs121\basics	C:\cs121\basics	Use relative path “cs121\basics”
cd ..	C:\cs121	“..” is the parent directory
D:	D:\	Change directory tree
cd cs389	D:\cs389	
cd .	D:\cs389	“.” is the current directory; no change

If you are using a Linux operating system, the root of the file system is represented by “/”, the home folder of the current user is represented by “~”, the current working folder is represented by “.”, and the parent folder is represented by “..”. The following table lists the most useful commands for you to use on a Linux system. You don’t need to understand them all now, and can use the table as a reference when necessary.

Action	Command Example
Launch terminal window	Applications Accessories Terminal
Change working folder to “~/www”	cd ~/www
Go back to home folder	cd
Go back to home folder	cd ~
Go up one level	cd ..
Find current path	pwd
List files	ls
List all files with properties	ls -alg
Copy file or folder <i>old</i> to <i>new</i>	cp old new
Move or rename file/folder <i>old</i> to <i>new</i>	mv old new
Browse contents of file <i>test.txt</i>	more test.txt
Edit file <i>test.txt</i>	gedit test.txt
Delete file <i>test.txt</i>	rm test.txt
Create folder <i>test</i>	mkdir test
Delete folder <i>test</i> (no question)	rm -rf test
Change password of <i>user</i>	sudo passwd user
Edit file /etc/sudoers	visudo
Run script file <i>shell.sh</i> in current folder	./shell.sh
Display value of environment variable PATH	echo \$PATH
Process shell file “.bashrc”	source ~/.bashrc
Extract contents from <i>file.tar.gz</i> in current folder	tar xvzf file.tar.gz -C .
Install module <i>apache2</i>	sudo apt-get install apache2
Update apt-get source info	Sudo apt-get update
Create symbolic link “~/www” for folder “/var/www”	ln -s /var/www ~/www
Restart Apache web server	sudo apache2ctl restart
Change owner of all files in /var/www to <i>user</i>	sudo chown -R user /var/www

Make <i>file</i> executable by everyone	<code>chmod a+x file</code>
Make <i>file</i> not executable by anyone	<code>chmod a-x file</code>
Make <i>file</i> writable by everyone	<code>chmod a+w file</code>
Make <i>file</i> not writable by anyone	<code>chmod a-w file</code>
Make all files under folder <i>bin</i> writable by anyone	<code>chmod -R a+w bin</code>
Log in and work as root	<code>su -</code>
Run a command as root	<code>sudo command</code>
Download a web file at http://url to the current folder	<code>wget http://url</code>
Find the computer's IP address	<code>ifconfig</code>
Clear terminal screen	<code>clear</code>
Generate WAR file for a Java web application <i>test</i>	<code>jar cvf test.war * [in project folder test]</code>
Review contents of text file <i>test</i>	<code>more test</code>

1.6 Display

A *display* is a screen used to display program outputs and graphic user interfaces (GUI) for your programs.

Data printed by a program will go to one of the windows on the computer display. Inside your program, you will choose which window will be used to display your data.

If you are printing characters, each window in text mode maintains a display *cursor* indicating where printing will happen. The display cursor will be set initially to the first printable position of a window. In a *Command Prompt* window, the display cursor is sometimes represented by a blinking underscore. The next character to be printed will be printed at the current location of the cursor, and the cursor will then move to the right by one space. After a *New Line* character is printed, the cursor will move to the beginning of the next line.

If you are drawing graphic objects like a dot, a line, or a circle, the basic drawing unit is a *pixel* (picture element), and the drawing window in graphics mode will use a 2-D coordinate system as show in Figure 5, where the origin is at the left upper corner, the x-axis goes rightward, and the y-axis goes downward. Figure 5 also shows a pixel located at point (3, 2). Each pixel can take on one of many possible colors.

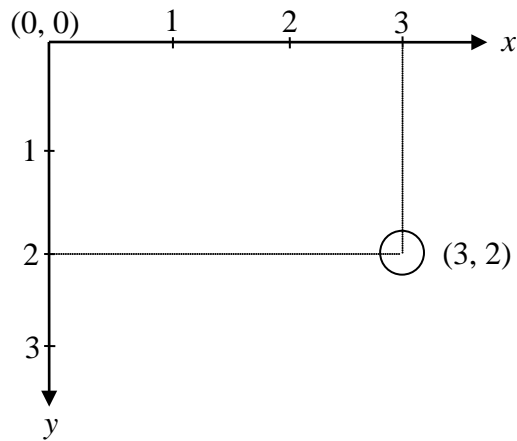


Figure 5 Display coordinate system with a pixel at (3, 2)

A typical VGA display can support an array (matrix) of 640 x 480 pixels (640 rows, each row having 480 pixels), where each pixel can display various colors. Most of today's displays have much higher *screen resolution*, which means the number of pixels per unit length in each row or column of the screen.

1.7 Keyboard

A keyboard is used to enter input data to a program running on a computer. A computer may run several programs at the same time, but only one program is the *active program* for receiving input data from the keyboard. Some programs run in windows. To make the program behind a window active, you just click anywhere in the window.

The keyboard is used to enter various characters including alphabetic characters like “a, b, A, B”; numeric characters like “1, 2, 3”; punctuation keys like “. , :”; and special function characters like PageUp, PageDown, Ctrl (Control), Alt (Alternative), and Esc (Escape). While most key strikes generate visible characters, some do not, as Ctrl and Alt.

The *space* key (the long key at the bottom middle of the keyboard), the *Enter* key (*new line* key), and the *Tab* key can be used to generate *white spaces*. A single space key strike will insert a single space character at the current cursor location of the display. A single *Enter* key strike will insert a *new line* character. A single *Tab* key strike will insert a single *Tab* character, or a predefined number of space characters. We say that the *space* key, the *Enter* key, and the *Tab* key all generate *white spaces*.

1.8 Computer Software

Computer software is computer programs. There are two categories of software: *system software* and *application software*. System software is for managing computer system hardware and software for their more efficient usage, and providing proper abstraction of the computer system for easing the work of application developers and end users. The most important example for system software is operating systems like Windows, Unix, and Linux. Java compiler and Java virtual machine, as well as C/C++ compilers, are also system software. Application software is

software for solving problems in a particular application domain, like word processing, games, engineering computation, and bank transaction processing.

Computer software is implemented with programming languages. The computer hardware can only understand very primitive machine instructions like adding two integers. While in theory we can implement all software in machine instructions, in practice it will be too complex and we may not be able to write correct programs when they are large. High-level programming languages like C, C++, and Java allow programmers to write readable programs with powerful statements (similar to instructions or commands), and let a compiler or an interpreter to convert the programmer-friendly programs into machine instructions before they are executed (run). Java is one of the most important high-level programming languages.

1.9 Java programming language

Java is a high-level *programming language* designed for *problem solving*. Unlike natural languages that use very flexible set of vocabulary and flexible syntax, a programming language uses a small set of *key or reserved words* and arranges these reserved words in very strict patterns to compose statements.

The computer hardware can only understand a set of low-level instructions of the so-called *machine language*. Different computer hardware architectures define their unique machine languages. Instructions at this level can only deal with basic operations like reading a key strike on the keyboard, saving a number at a particular memory address, reading value from a particular memory address, and adding two numbers. To solve a problem, a programmer needs to write a very long sequence of such machine language instructions called a machine-language program. Given the program and input data, a computer can compute the output.

While a machine language is native to the computer hardware and therefore can be executed most efficiently, the complexity of machine-language programs is overwhelming. This motivated the creation of various high-level programming languages. Java is one of the latest high-level programming languages. In a high-level language, the instructions are called *statements*. Each high-level statement is the abstraction of a sequence of machine instructions. As a result, a high-level program is much shorter than its machine-instruction counter-part, and it is much easier to write and *maintain* (understand and modify).

But a computer cannot understand a high-level language program directly. A high-level language program is called *source code*, which needs be translated into machine instructions, also called *executable code*, before computer hardware can understand and execute it.

A *compiler* is a computer program that transforms a high-level language source code into machine-language *executable code*, and saves the executable code in a computer *file* (*executable file*) for later execution. The translation process takes time. For the same source code, the compiler needs only to translate it once. Because the compiler can carefully analyze the complete source code, a compiler can generate very efficient machine-language code in the sense that the latter is shorter and executes faster on the computer hardware. Given a new set of input data, we only need to *run* (execute) the machine-language code on it (feed the data to the executable code) to get computation results. Languages aiming at high execution efficiency, including C, C++, COBOL, FORTRAN, and Pascal, usually use this compilation approach.

An *interpreter* is a computer program that reads the source code one-line at a time, translates the source code line into a sequence of equivalent machine instructions, executes them, throws them away, and repeats on the next source code line. The final execution result of an interpreted program should be the same as that of a compiled program. But in the interpreted case, we don't need the separate compilation and translation phases. Since we don't save the machine code resulted from the interpretation in a computer file, we need to transform the source code lines again every time we use the program to solve a new problem instance. Since the source code transformation is repeated each time we run it, an interpreted language usually runs slower than its compilation-based counter-parts. *Basic* and *Visual Basic* are examples of interpreted programming languages.

Java, as well as C#, uses a combination of the above two processes. Fundamentally, Java is an interpreted language, since its main objective is not runtime efficiency, but to fulfill its value proposition of “write once and run everywhere”. A Java source code file is first compiled into a so-called *bytecode* file, also called a *class file*, which is a standardized binary representation of the source code. We use Java command “**javac**” to compile a Java source code file. When we need to run this Java program, we use the Java interpreter, also called the *Java virtual machine*, to transform each bytecode statement of the program into a sequence of equivalent machine instructions for execution on the computer hardware. We use Java command “**java**” to interpret or execute a Java bytecode file. This two-step mechanism has a few advantages:

1. The bytecode is hardware and operating system independent. For each Java source code file, we only need to compile it into its bytecode form once. We can then distribute the same bytecode file to any computer on the Internet that needs to run the program.
2. While Java source code is human or character-oriented, a bytecode file is machine-oriented. The interpretation of bytecode statements into machine code is much more efficient than doing that directly from Java source code.
3. Before computer hardware finally interprets each bytecode statement for execution, it has a last chance to examine the bytecode statement to see whether it could be dangerous to the computer system. If it does, the interpreter could stop its execution and report problems to the user.

Java was developed by a team lead by James Gosling at Sun Microsystems in 1995. Java is a general-purpose programming language, and is based on the object-oriented computing paradigm. Java is especially suitable for developing web applications. Much of Java's interesting functions do not have counter-parts in C or C++.

At a lower level, a Java program is made up of a sequence of identifiers organized in a pattern satisfying the syntax requirement of a valid Java program. An *identifier* is a sequence of letters, digits, underscores (`_`), and dollar signs (`$`); and the first character of an identifier cannot be a digit. Example identifiers include “class”, “public”, “x”, “value”, “Hello1”, “HelloWorld”, and “squareRoot”. Identifiers are case sensitive, so “x” is different from “X”. Java identifiers should be concise but meaningful, indicating their intended usage. An identifier is often the concatenation of a few words, with the first letter of each word capitalized and the others in lower-case (the first letter of the first word will be capitalized if the identifier is the name of a class, in lower-case otherwise). A few dozens of identifiers are called *reserved words* or *keywords* of Java, and they have predefined meanings in Java and they cannot be used for other

purposes. Identifiers “class”, “public”, “static”, “void”, and “package” are examples of reserved words. Reserved words are all in lower-case. Identifiers can be of any length.

At a higher level, a Java program is made up of one or more classes, and each class contains one or more methods. A method is a sequence of operations to be executed by the CPU. A class is a container for holding related methods. Each class has a name with the first letter capitalized. Each method has a name with the first letter in lower-case. The classes of a program may be distributed in a few Java source code files. Java also uses variables for holding intermediate values. A variable’s name is an identifier with the first letter in lower-case.

2 Software Installation

2.1 Installation of Java SE JDK

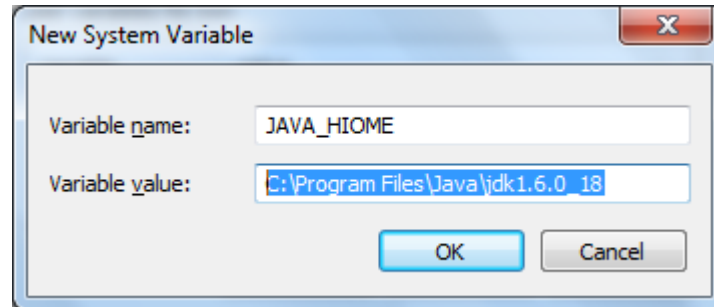
If your computer has *VMware Player* for Windows or *VirtualBox* for Mac installed, you can download my virtual machine (VM) at <http://community.seidenberg.pace.edu/files/sici2011/ubuntu2.exe>. Refer to Sections 1 and 2 of my *Seidenberg Institute of Computing Innovation 2011 lecture and lab manual* at <http://community.seidenberg.pace.edu/files/sici2011/SICI-2011-lab-manual.pdf>. JDK has already been installed in the VM so you can skip this step. All of my course VMs also have JDK installed.

I assume you are using a PC running Windows. You may need to adjust the instruction based on the version of your Windows.

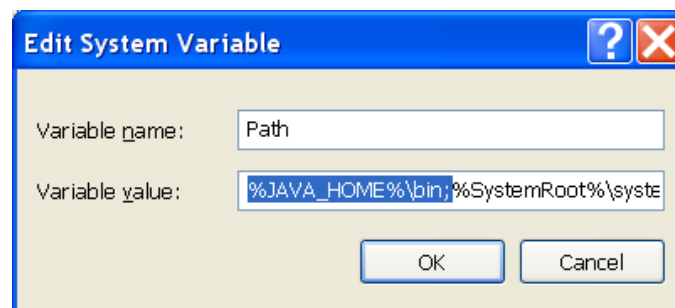
1. Visit “Start|Settings|Control Panel|Add/Remove Programs” to remove any of existing installations of Java SE JDK or JRE (Java runtime environment). Make sure that you reboot the system before you install any new software. After rebooting, delete the installation directories for those removed installations. Let a PC have multiple Java installations may lead to confusion.
2. Visit <http://java.sun.com/javase/downloads/index.jsp> to download the latest version of Java SE JDK (instead JRE) for your operating system.
3. In a *Windows Explorer*, click on the downloaded file to start its installation. Accept the license agreement, and take all default values. By default, your Java SE JDK installation directory will be “C:\Program Files\Java\jdk1.6.0_18” (the JDK folder name changes with JDK version number, which may be different for you). The installation of Java SE JDK will take about 318 MB of disk space. The installation includes the installation of an embedded JRE (Java Runtime Environment). After the JDK is installed, you will be prompted for installing an external version of JRE into “C:\Program Files\Java\jre1.6.0_18” (JRE folder name may change). This external JRE’s installation is important to you if you need to enable Java support in your web browsers.
4. To run Java compiler **javac** and interpreter **java** on the command-line, you need to add the *bin* directory of your JDK in the value of the environment variable PATH. An environment variable is a memory location where the operating system keeps some values for programs to access. When you type a command “javac” and then *Enter* in a *Command Prompt* window, for example, the operating system will try to find whether there is an executable file named “javac.exe”, “java.bat”, or “javac.com” with the value of your PATH. The value of PATH is a sequence of directories separated by semicolon “;” in Windows and colon “:” in Unix/Linux. Period (.) is used to represent the current working directory (the directory in which the *Command Prompt* window is working). The operating system will sequentially try to find an executable file for “javac” in the directories listed in the value of PATH, from left to right. The first executable file with file name stem “javac” will be executed. If the operating system completed the search in all directories on the PATH (specified by the PATH’s value) and none of the executable files for “javac” could be found, the operating system will declare that “`javac` is not recognized as an internal or external command, operable program or batch file.”

Right-click on “My Computer” and choose “Properties” from the popup menu. Click link “Advanced system settings” in the left-upper corner. In the “System Properties” window, click on the “Advanced” tab. Click on the “Environment Variables” button. In the bottom

“System variables” area of the “Environment Variables” pane, click on button *New* to launch the “New System Variable” pane. Enter value “JAVA_HOME” for variable name, and value “C:\Program Files\Java\jdk1.6.0_18” (the JDK folder name needs be adjusted for your installation) for “Variable value”, as shown below. (If you use a Java SDK installation at another directory, you should update the variable value accordingly.)



Click on the *OK* button to complete the definition. Now click to highlight the line for “Path” in the “System variables” area, and then click on the “Edit” button. In the “Variable value” textbox, move the cursor to the left end, and insert
%JAVA_HOME%\bin;
at the beginning, as shown below. Here “%JAVA_HOME%” is to be replaced with the current value of environment variable “JAVA_HOME” by the operating system.



Click on the *OK* button to complete the modification.

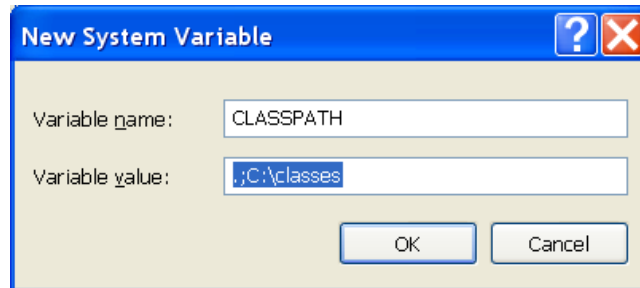
- Now you also need to set up the value of environment variable CLASSPATH, which is used by “java” and “javac” to find needed Java classes. When you type “java Hello”, the command “java.exe” will follow the value of environment variable CLASSPATH, from left to right, to search for the first file “Hello.class” to execute. The value of CLASSPATH is a sequence of directories for searching for Java class files. Period (.) represents the current working directory in which you typed “java Hello”. If there are multiple instances of file “Hello.class” in different directories listed by the value of CLASSPATH, only the first one will be executed. If there is no file “Hello.class” in any of the directories listed by the value of CLASSPATH, command “java” will declare that “Exception in thread “main” java.lang.NoClassDefFoundError: Hello”. **If you don’t set the value of CLASSPATH, by default CLASSPATH has period (.) for the current working directory as its value.**

In *Windows Explorer*, create a new directory “C:\classes”. This will be the directory where we keep all Java class files. In the bottom “System variables” area of the

“Environment Variables” pane, click on button *New* to launch the “New System Variable” pane. Enter value “CLASSPATH” for variable name, and value

.;C:\classes

for “Variable value”, as shown below. The value of CLASSPATH is a list of semicolon-separated directories. The first period (.) represents the current working directory.



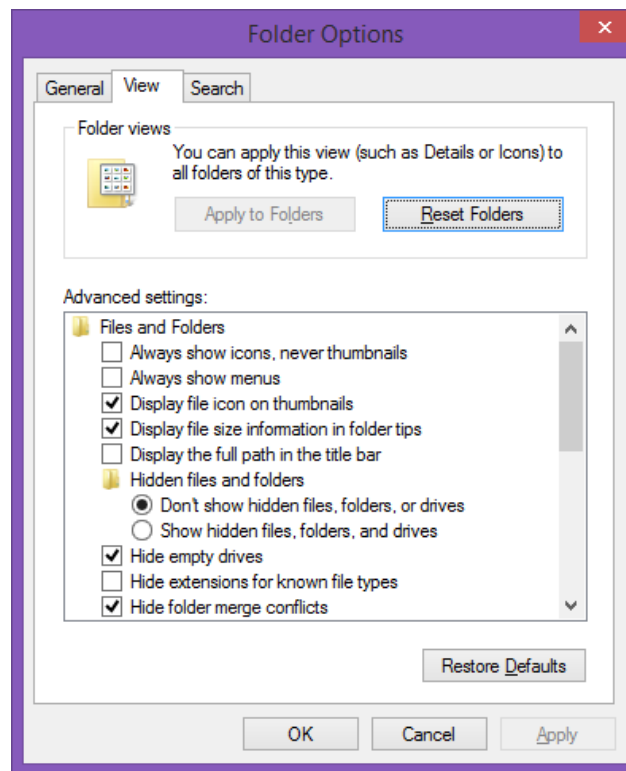
Click on the *OK* button to complete the definition. Click on the *OK* button of the “Environment Variables” pane to shut it down. Click on the *OK* button of the “System Properties” window to shut it down.

6. To test whether you have done correctly, click on “Start|Run...”, type “cmd” in the Open box, and click on the *OK* button to start a *Command Prompt* window. Type “javac” and the *Enter* key in the window. If you see usage information of “javac”, you have completed this step successfully. Otherwise you need to carefully redo this step.

2.2 Windows explorer and text editor

For software developers, *Windows Explorer* should show detailed view of files, and file extensions of known types should be displayed all the time. It will also be convenient if we can easily launch a new *Command Prompt* window from a *Windows Explorer* and change directory automatically to the directory that we choose in the *Windows Explorer*. This section introduces some useful skills and customization to your *Windows Explorer* for better supporting Java programming.

1. Type the Microsoft logo key and key “E” together to launch an instance of *Windows Explorer*.
2. To open a terminal window in a folder in *Windows Explorer*, push down the “Shift” key, right-click the folder, and then choose command “Open terminal window here” on the pop-up menu.
3. To view file name extensions, open Control Panel (one way is to push the Microsoft logo key and key “X” together, and then choose “Control Panel”), open “Folder Options”, click the “View” tab, then uncheck “Hide extensions for known file types” in the “Advanced settings” section.



4. Please don't use Microsoft *Word* to edit Java programs. You should install a text editor to edit Java programs. *EditPad Lite* at <http://www.editpadpro.com/editpadlite.html> an example free text editor for you to download and install.

If you are new to Java programming, I suggest that you refrain from using Integrated Development Environments (IDE) like *Eclipse*. You need to learn additional concepts for using them, and they will hide you from some very important concepts and skills that you can use in any Java environment. Of course Java IDEs will be very valuable for improving your productivity after you learn the basic concepts in this tutorial.

2.3 Typical process for creating and running a Java program

1. Assume you have a folder "C:\cs121" for working out examples in this tutorial. Use *EditPad* to create a Java source file named "Hello.java" in directory "C:\cs121". If you don't have directory "C:\cs121" yet, use a *Windows Explorer* to create it. Use your text editor to create a file "Hello.java" with the following contents:

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

2. Open a *Command Prompt* window in “C:\cs121”. You could achieve this by pushing down the shift key and right-clicking on folder “C:\cs121” in a *Window Explorer*, and then clicking on the *Open command window here* item in the popup menu.
3. In the *Command Prompt* window, type
javac -d C:\classes Hello.java
to compile the source file “Hello.java” into a Java class file “Hello.class” in folder “C:\classes”. The command-line switch “-d” of “javac” command is used to specify the destination of the class files resulting from the compilation. Learn to use switch “-d” will save your time when you need to declare Java classes in Java packages, as we will explain in the next section.
4. In any *Command Prompt* window, type
java Hello
to run file “Hello.class”. Since file “Hello.class” is in “C:\classes” and the latter is on our CLASSPATH, the “java” command can find it and run it.

3 Java Basics

This section assumes that you are using a Linux VM of mine, and my tutorial folder *JavaLabs* is in your home folder. If you are using a Windows PC, you can replace “~/JavaLabs” with “C:\JavaLabs”, and replace forward slashes (/) with backward slashes (\).

3.1 Basic Hello World program

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “Hello1.java” in directory “~/JavaLabs/basics” and the file has contents below:

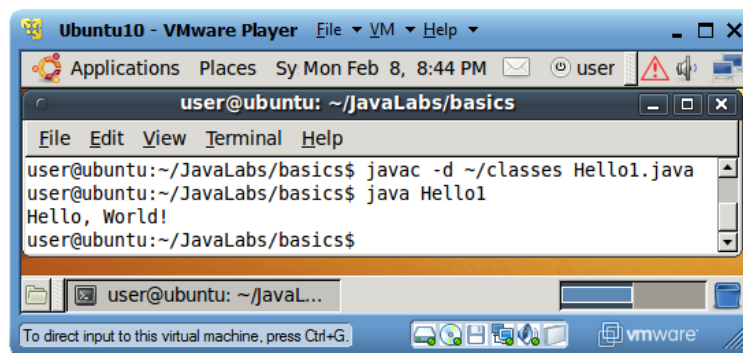
```
1 public class Hello1 {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

Explanation

- a. A simple Java program is a single Java *class*, which is the top-level container for Java source code. Each class has a name, in this case, “Hello1”. In this example, the first line identifies a class named “Hello1”. It is a public class, meaning that other classes can access it with no limitation (we will see this after we have a program containing two classes). A *public class* must be saved in a file whose name uses the class name as the file name stem and “java” as file name extension. By convention, a class name is an identifier with the first letter capitalized. The class name is followed by a *class body* enclosed in a pair of curly braces, as in “public class ClassName { ... }”.
- b. A class contains one or more methods. A method is the basic unit for performing some operations by a class. A method has a name, in our example “main”; a parameter list for passing input data, in our example “(String[] args)” meaning that input is an array (sequence) of character strings (we will have better understanding after we discuss arrays and variable declaration); a return data type for the type of data to be returned by the method to its invoker (caller), in our example “void” meaning no return value; and a *method body* that is a sequence of statements enclosed in a pair of curly braces. A method qualified by method modifier “*public*” is a method that can be accessed by any classes that have access to this class. A method is qualified by method modifier “*static*” if that method can be invoked against its containing class’s name without creating an object of that class first. For example, any class can invoke our method *main()* by expression “Hello1.main(null)”. More explanation about public and static methods will be provided after we have learned more about Java classes and objects in Section 4.
- c. When the Java interpreter runs a Java class, it will first try to run its method with signature (pattern) exactly as “public static void main(String[] args) { ... }”, where the method body can contain any statements. Please make effort to remember this pattern since we cannot fully explain this class yet (we have the chicken-egg dilemma).
- d. Each statement in the method body must be terminated by a semicolon (;).

- e. To print a message to the display and then move the display cursor to the beginning of the next line, use “`System.out.println("message body")`” (“`println`” means “print the argument followed by a *new line* character”).
3. Now it is time to compile the Java source code file “`Hello1.java`” into its bytecode file, also called *class file*, “`Hello1.class`”. While the terminal window is at “`~/JavaLabs/basics`”, type command
`javac -d ~/classes Hello1.java`
With a file explorer, check that a new file “`~/JavaLabs/basics/Hello1.class`” is created.
4. To run file `Hello1.class`, just type
`java Hello1`
in any terminal window.

The following is a screen capture of my session for the above steps.



3.2 Code style and command-line arguments

1. Open a terminal window. Change directory to “`~/JavaLabs/basics`”.
2. Use a text editor to open and review Java source code file “`Hello2.java`” in directory “`~/JavaLabs/basics`” and the file has contents below:

```
1 // This is a demo program
2 // Please provide a name as command-line argument
3 public class Hello2
4 {
5     // Entrance method to run first
6     // args[] is an array of command-line arguments
7     public static void main(String[] args)
8     {
9         System.out.print("Hello, ");
10        System.out.print(args[0]);
11        System.out.println("!");
12        System.out.println("Hello, " + args[0] + "!");
13    }
14 }
```

Explanation

- a. On any source code line, “`//`” signifies the start of a Java comment, which ends at the end of the line. A Java comment is ignored by the Java compiler as if it is a white space. Comments are used for programmers to write notes about the code for him/her self or for other programmers.

- b. To improve the readability of a program, the program authors are encouraged to insert enough comments to explain the purpose, function, and usage of each class and each method.
 - c. Java source code can be formatted for different display styles based on a few basic principles:
 - i. In a Java source code file, all the three types of white space characters are exchangeable, except in character strings like "Hello world!"
 - ii. A single white space character is equivalent to any number of consecutive white space characters, except in character strings.
 - d. Source code files "Hello1.java" and "Hello2.java" represent two styles of Java source code presentation, different in the location of the opening curly braces. The style used by "Hello1.java" puts an opening curly brace at the end of the first line of the class or method declaration and is thus more compact. The style used by "Hello2.java" puts the opening curly brace on a new line and aligns it with the first character of the class or method declaration. In either styles, the matching closing curly brace must be aligned with the first character of the class or method declaration, and the class or method body inside the pair of curly braces must have an indentation of two (or more, but be consistent in a project) space characters. You can use either of the two styles, but you should stick to one style in the same project. Never mix up the styles in the same source code file.
 - e. The above Java source code style specification is critical for the readability and maintainability of Java programs, and it will be strictly enforced in our courses. Deviation from the suggested styles will lead to losing grade points even if your code works correctly.
 - f. To print a message to the display and leave the cursor immediately to the right of the last character in the message, use "System.out.print("message body");"
 - g. The strings after "java Classname" are called command-line arguments. Normally a command-line argument cannot contain white space characters. If you want an argument to contain white space characters, you must put the argument value in a pair of double quotes, as in "Lixin Tao".
 - h. The first command-line argument is saved in args[0], the second in args[1], etc. Array args[] is a sequence of strings. Each string in this array is accessed through an integer index value. In Java, C, C++, and C#, array indexes always have zero (0) as their first (smallest) index value (instead of the more intuitive 1).
 - i. Operator "+" can be used to concatenate a few strings into a single string. Therefore,

"Lixin " + "Tao"

 is the same as "Lixin Tao".
 - j. If this program runs without command-line arguments, array args[] will be empty containing zero or no cells for strings, and the code for accessing args[0] will lead to an "array index out of bound" error at index value 0.
3. Make sure your working directory is now "~/JavaLabs/basics". To compile the source code, type

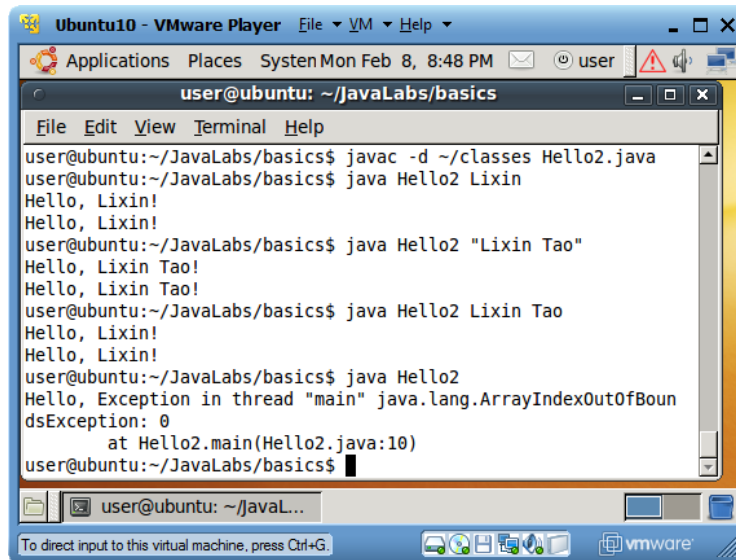
javac -d ~/classes Hello2.java

 You can use a file explorer to check that a new file "~/JavaLabs/basics/Hello2.class" has been created.
 4. To run the new class, type commands like the following two:

java Hello2 Lixin

```
java Hello2 "Lixin Tao"
```

The following is a screen capture for my test run session:



```
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes Hello2.java
user@ubuntu:~/JavaLabs/basics$ java Hello2 Lixin
Hello, Lixin!
user@ubuntu:~/JavaLabs/basics$ java Hello2 "Lixin Tao"
Hello, Lixin Tao!
user@ubuntu:~/JavaLabs/basics$ java Hello2 Lixin Tao
Hello, Lixin!
user@ubuntu:~/JavaLabs/basics$ java Hello2
Hello, Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at Hello2.main(Hello2.java:10)
user@ubuntu:~/JavaLabs/basics$
```

3.3 Java package

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “edu/pace/csis/Hello3.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 package edu.pace.csis;
2
3 public class Hello3 {
4     public static void main(String[] args) {
5         System.out.println("Hello, World!");
6     }
7 }
```

Explanation:

- a. A project usually contains classes from several departments, project teams, or companies. Since we are supposed to use meaningful class names, it is easy for classes from different authors to have the same name. This is called *naming conflicts*. If a project has more than one class with a particular class name, only one of them will actually be deployed and used. Java packages are introduced to reduce the possibility of naming conflicts.
- b. A class can be declared as part of a particular *package*. A package can contain another package. Package names are identifiers all in lower case. In this example, the top level package is “edu”, which contains a nested package “pace”, which in turn contains a nested package “csis”. String “edu.pace.csis”, which is a sequence of names of nested packages separated with period (.), is called a *package path*. “package edu.pace.csis;” declares that the following class “Hello3” is a member of package “csis”, which in turn is a member of package “pace”, which in turn is a member of package “edu”. For avoiding naming conflicts, it is common practice

to use the reverse of an organization's web site URL as the Java package path base for this organization's Java classes. Package declaration must be the first non-comment line of a source code file.

- c. After the compilation of a class contained in packages, each package of the class's package path will be implemented as a file system directory. To run a class, we must always qualify the class name with its full package path. In our example, to run class Hello3, we must use "java edu.pace.csis.Hello3". If you don't qualify the class name with its package path, the Java interpreter will declare that the class cannot be found.
 - d. The *best practice for storing a Java source class file* is to store the Java source class file inside a directory path corresponding to the class's package path, relative to the working directory in which you plan to compile and run the class. For our example here, we plan to use "~/JavaLabs/basics" as our working directory for compiling all of our Java classes in this section. Therefore "Hello3.java" should be stored in a directory path "~/JavaLabs/basics/edu/pace/csis", where the last three directories correspond to the package path for class Hello3.
 - e. To compile a class contained in packages, you should use the "-d" switch, as shown in the following test runs.
3. Make sure your working directory is now "~/JavaLabs/basics". To compile the source code, type
javac -d ~/classes edu/pace/csis/Hello3.java
You can use a terminal explorer to check that a new file "~/classes/edu/pace/csis/Hello3.class" has been created.
 4. To run the new class, type commands like the following two:
java edu.pace.csis.Hello3
java Hello3 (Error message that class Hello3 cannot be found)
 5. Use file explorer to delete file "~/classes/edu/pace/csis/Hello3.class". Recompile the class by typing
javac -d . edu/pace/csis/Hello3.java
You will notice that a new file "~/JavaLabs/basics/edu/pace/csis/Hello3.class" has been generated.
 6. Repeat step 5 to run class Hello3.
 7. Use file explorer to delete file "~/JavaLabs/basics/edu/pace/csis/Hello3.class". Recompile the class by typing
javac edu/pace/csis/Hello3.java
You will notice that a new file "~/JavaLabs/basics/edu/pace/csis/Hello3.class" has been generated. The effect of this step is the same as that of step 6. If you don't use command-line switch "-d", the class file will by default be generated next to the source file.
 8. Repeat step 5 to run class Hello3.
 9. Use file explorer to delete file "~/JavaLabs/basics/edu/pace/csis/Hello3.class".
 10. Change working directory to "~/JavaLabs/basics/edu/pace/csis" by typing
cd edu/pace/csis
Now you can use "ls" ("dir" for Windows) to see that source file "Hello3.java" is in the current working directory.
 11. Recompile the class by typing
javac Hello3.java

You will notice that a new file “~/JavaLabs/basics/edu/pace/csis/Hello3.class” has been created in the current working directory.

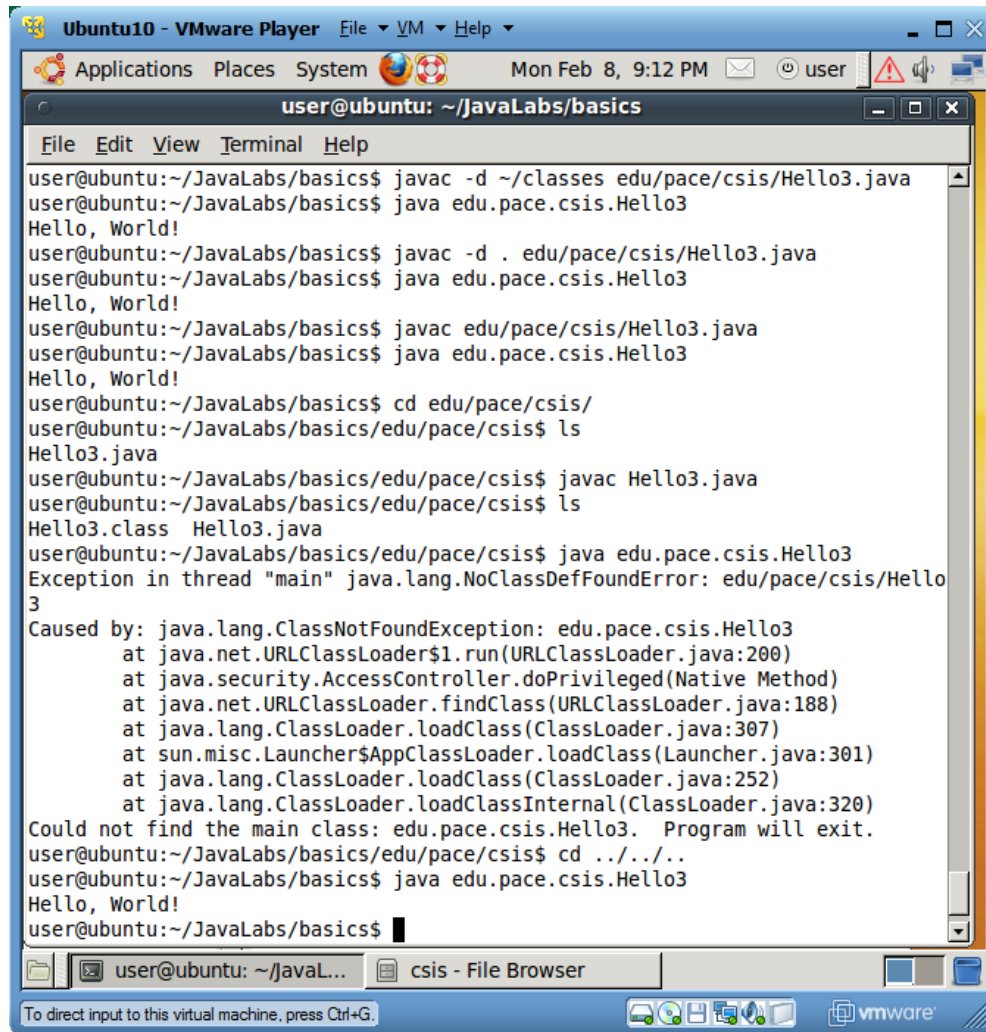
12. Repeat step 5 to run class Hello3. You will find either approach fails.

13. Now you change working directory back to “~/JavaLabs/basics” by typing

cd ../../..

14. Repeat step 5 to run class Hello3.

The following is a screen capture for part of my test run session:



```
Ubuntu10 - VMware Player  File  VM  Help
Applications  Places  System  Mon Feb 8, 9:12 PM  user
user@ubuntu: ~/JavaLabs/basics
File  Edit  View  Terminal  Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes edu/pace/csis/Hello3.java
user@ubuntu:~/JavaLabs/basics$ java edu.pace.csis.Hello3
Hello, World!
user@ubuntu:~/JavaLabs/basics$ javac -d . edu/pace/csis/Hello3.java
user@ubuntu:~/JavaLabs/basics$ java edu.pace.csis.Hello3
Hello, World!
user@ubuntu:~/JavaLabs/basics$ javac edu/pace/csis/Hello3.java
user@ubuntu:~/JavaLabs/basics$ java edu.pace.csis.Hello3
Hello, World!
user@ubuntu:~/JavaLabs/basics$ cd edu/pace/csis/
user@ubuntu:~/JavaLabs/basics/edu/pace/csis$ ls
Hello3.java
user@ubuntu:~/JavaLabs/basics/edu/pace/csis$ javac Hello3.java
user@ubuntu:~/JavaLabs/basics/edu/pace/csis$ ls
Hello3.class  Hello3.java
user@ubuntu:~/JavaLabs/basics/edu/pace/csis$ java edu.pace.csis.Hello3
Exception in thread "main" java.lang.NoClassDefFoundError: edu/pace/csis/Hello3
Caused by: java.lang.ClassNotFoundException: edu.pace.csis.Hello3
    at java.net.URLClassLoader$1.run(URLClassLoader.java:200)
    at java.security.AccessController.doPrivileged(Native Method)
    at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:307)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:301)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:252)
    at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:320)
Could not find the main class: edu.pace.csis.Hello3.  Program will exit.
user@ubuntu:~/JavaLabs/basics/edu/pace/csis$ cd ../../..
user@ubuntu:~/JavaLabs/basics$ java edu.pace.csis.Hello3
Hello, World!
user@ubuntu:~/JavaLabs/basics$
```

By now you should understand why we asked you to use “javac -d ~/classes ClassName.java” or “javac -d . ClassName.java” to compile a Java source file “ClassName.java”, while most textbooks just let you use “javac ClassName.java” to compile the same Java source code file. First, it is always recommended to separate Java source code files from their class files (bytecode files), since when we do code backup, we only need to backup the Java source code files. Second, for Java classes contained in Java packages, if we use “javac ClassName.java”, the class file “JavaName.class” will be created in the current working directory, not in directories corresponding to the Java packages to which the class belongs; we have to manually create a directory path corresponding to the package path for the class and move file “ClassName.class” to the end directory of that directory path before we can run the class.

3.4 Best practice for organizing multiple class projects

A software project usually contains a few functional units, and each functional unit contains a few Java classes.

Classes in the same functional unit are normally declared inside its own Java package. If the functional units are implemented by different developers, such an arrangement can avoid naming conflicts or constant negotiations for class names.

For more efficient project backup in terms of both disk space and backup time, it is a good practice to separate Java source code and *Java class* (bytecode) *files*. One *source base directory* will be chosen as the base of the Java class source files, and one *class base directory*, say “~/classes” or “.”, will be chosen as the base of the class files. The class base directory must be on the CLASSPATH. Java class source code should be created inside directory paths, parallel to the Java package paths that contain the classes, relative to the *source base directory*. Java class files should be created inside directory paths, parallel to the Java package paths that contain the classes, relative to the *class base directory*. For example, let us use “~/JavaLabs/basics” as the *source base directory* or *working directory*, “~/classes” as the *class base directory*, and we have a class named *Class1* that belongs to Java package *project*. Then the Java source class file “Class1.java” should be created inside relative directory path “project”, or absolute directory path “~/JavaLabs/basics/project”, and its corresponding class file “Class1.class” should also be created inside relative directory path “project”, but relative to the class base directory “~/classes”. Therefore the absolute directory path for file “Class1.class” is “~/classes/project/Class1.class”.

When we compile a Java class, all the other classes that are referenced from that class directly or indirectly will be compiled too if their class files are not available on the CLASSPATH yet. Normally (for all projects in this course), you only need to compile the main class that contains the entry method *main()* actually called by the Java virtual machine first. Compile the project by compiling the main class, say “MainClass” belonging to package “edu.pace.csis”, in the project’s working directory with a command line like the following one:

```
javac -d ~/classes edu/pace/csis/MainClass.java
```

To run the project, type

```
java edu.pace.csis.MainClass
```

in any *Command Prompt* window with any working directory. By this organization, the source code files and the class files will be in parallel directory structures relative to the *working directory* and the *class base directory* respectively. If we choose not to separate the source code and the class files and use command-line switch “-d .” for command *javac*, then the class files will stay in the same directories as their corresponding source files.

Demo project:

1. Open a *Command Prompt* window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “project/Class1.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1 package project;
2
3 public class Class1 {
4     public static String name = "Class1";
5
6     public static void main(String[] args) {
7         System.out.println("Hello from Class1!");
8         Class2.main(null);
9     }
10 }

```

3. Use a text editor to open and review Java source code file “project/Class2.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1 package project;
2
3 public class Class2 {
4     public static String name = "Class2";
5
6     public static void main(String[] args) {
7         System.out.println("Hello from Class2 called from " + Class1.name);
8     }
9 }

```

Explanation:

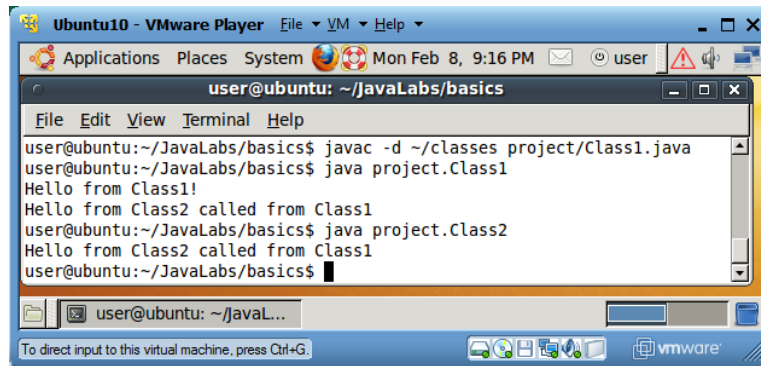
- a. A class contains the declarations of methods and variables. A variable at the class level (inside the body of a class but outside the body of a method) can be accessed by any methods in the class. A public class-level variable can be accessed by any classes that have access to the class. A static class-level variable is called a *class variable* and can be accessed from the class name, in addition to from an object as we will see in Section 4. A variable at the class level can be initialized on its declaration line. Without initialization, class-level variables will have initial values of either 0 or 0.0 for numerical types, *false* for boolean type, and *null* for class types.
 - b. Reserved word *null* represents a special value indicating that the variable holding it is not referring to any object. For parameter *args*, it will have value *null* if there are no command-line arguments to be passed to it.
 - c. To retrieve the current value of a static variable *name* of class *Class1*, we can use expression “Class1.name”.
 - d. To invoke the static method *main()* of class *Class2* without passing any arguments to it, we can use “Class2.main(null);” Upon the invocation of method *main()* of class *Class2*, the execution of the body of class *Class1*’s *main()* method is suspended, the body of the called method is executed, and then the control goes back to the body of method *main()* of *Class1* to resume its execution from where it left.
4. Make sure your working directory is now “~/JavaLabs/basics”. To compile the source code, type
javac -d ~/classes project/Class1.java
 You can use a file explorer to check that new files “~/classes/project/Class1.class” and “~/classes/project/Class2.class” have been created.
 5. To run the project, type the following command:

java project.Class1

6. You can also run *Class2* directly, as we do below. This shows that a Java project can have several entry points.

java project.Class2

The following is a screen capture for my test run session:



```
user@ubuntu: ~/JavaLabs/basics
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes project/Class1.java
user@ubuntu:~/JavaLabs/basics$ java project.Class1
Hello from Class1!
Hello from Class2 called from Class1
user@ubuntu:~/JavaLabs/basics$ java project.Class2
Hello from Class2 called from Class1
user@ubuntu:~/JavaLabs/basics$
```

3.5 Local variables, expressions, and assignment

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “LocalVariable.java” in directory “~/JavaLabs/basics” and the file has contents below:

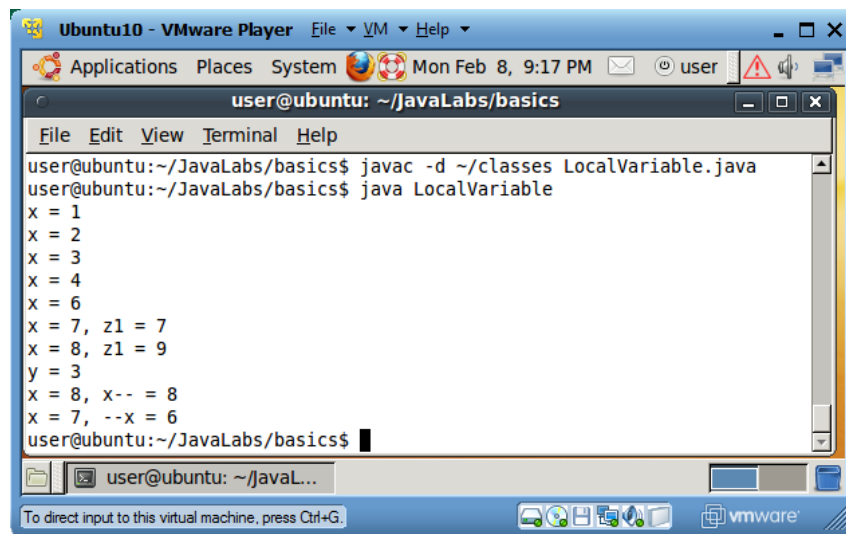
```
1 public class LocalVariable {
2     public static void main(String[] args) {
3         int x;
4         int y = 0;
5         int z1, z2 = 1, z3 = 2;
6         x = 1;
7         System.out.println("x = " + x);
8         x = x + 1;
9         System.out.println("x = " + x);
10        x++;
11        System.out.println("x = " + x);
12        ++x;
13        System.out.println("x = " + x);
14        x += 2;
15        System.out.println("x = " + x);
16        z1 = 1 + x++;
17        System.out.println("x = " + x + ", z1 = " + z1);
18        z1 = 1 + ++x;
19        System.out.println("x = " + x + ", z1 = " + z1);
20        y = z2 + z3;
21        System.out.println("y = " + y);
22        System.out.println("x = " + x + ", x-- = " + x--);
23        System.out.println("x = " + x + ", --x = " + --x);
24    }
25 }
```

Explanation

- a. A *local variable* is a named memory location for storing some values. A local variable is *declared* inside a method, and it is available from the point of its declaration to the closing curly brace matching the closest open curly brace before the local variable declaration.
- b. At any time, a variable can only hold one value. Reading from a variable will not change the variable's value. Writing a value into a variable will replace the old value in the variable with the new value.
- c. Each variable has a data type. In this section, we focus on primitive data types *int* (32-bit integer, like 5), *double* (64-bit floating-point number, like 3.14), *boolean* (true or false), *char* (character, like 'c'), as well as the predefined data-type *String* (sequence of characters, like "Hello world"). A variable declared of *int* type cannot be used to store the value of another type, like *double* type. For this reason, Java is called a *strongly typed language*. The stronger typing of variables reduces the chances of storing data in wrong variables.
- d. The name of a variable is an identifier with the first character in lower-case.
- e. "int x;" declares that "x" is the name of an integer-typed variable. We can think of "x" as the name of a memory cell in which we can only store an integer value that can be represented by 32 bits. This declaration will only create the memory cell named "x", and the *memory cell "x" will be filled up with random value*. A variable should not be read until it has been initialized with some value.
- f. "int y = 0;" combines two steps: declaring that "y" is an integer-typed variable, and initializing variable "y" with value "0". Here "y = 0" means to initialize variable "y" with value 0, and it has nothing to do with equality between the values to the two sides of the = operator.
- g. Multiple variable declarations with the same data type can be combined into a single one, as in "int z1, z2 = 1, z3 = 2;". Here variables "z1", "z2", and "z3" are all declared to be of type integer, "z1" is not initialized, "z2" is initialized to 1, and "z3" is initialized to 2. In such a declaration, the values will be initialized from left to right. For example, "int z1, z2 = 1, z3 = z2 + 1;" will have the same effect. Here "z3 = z2 + 1" means that variable "z3" is initialized with the current value of "z2" plus 1.
- h. A variable can only be declared once in a method. Multiple type declaration for a variable is an error.
- i. An *assignment* has the form of "variableName = value" or "variableName = expression". In either case, the left side of the = operator is a variable, and the right side of the = operator is a value. An assignment instructs the CPU to first evaluate the current value of the right side expression of the = operator, and then store the value in the variable to the left of the = operator. Operator = is an action, not a relationship for equality. Therefore, we never put a value to the left side of the = operator. For example, "1 = x;" doesn't make sense, because "1" is not a variable and it cannot store the current value of variable "x".
- j. In a Java program, constants like "1", "3.14", "true", "Hello", and 'c' are called *integer literal*, *double literal*, *Boolean literal*, *string literal*, and *character literal* respectively.
- k. When CPU executes assignment statement "x = x + 1;", it will first find out the value of the expression to the right side of operator =, "x + 1", which is the current value of variable "x" plus 1, and then store the result back into variable "x". Put it another way, assignment statement "x = x + 1;" increases the current value of variable "x" by 1.

- l. Expression “++x” is called a *preincrement expression*. It finds out the current value of variable “x”, increases its current value by 1, and then returns the new value of variable “x” as the value of expression “++x”.
 - m. Expression “x++” is called a *postincrement expression*. It finds out the current value of variable “x”, increases its current value by 1, and then returns the old value of variable “x” as the value of expression “x++”.
 - n. Expression “--x” is called a *predecrement expression*. It finds out the current value of variable “x”, decreases its current value by 1, and then returns the new value of variable “x” as the value of expression “--x”.
 - o. Expression “x--” is called a *postdecrement expression*. It finds out the current value of variable “x”, decreases its current value by 1, and then returns the old value of variable “x” as the value of expression “x--”.
 - p. When a string value is combined with a value of different type with the + operator, the value of different type will first be converted into a new string, and then concatenated with the old string value. For example, if variable *x* currently holds value 1, then “x = ” + x” will be converted into string “x = 1”.
 - q. Assignments like “x = 1” are expressions and “=” here is a binary operator. The return value of an assignment is the value assigned to the variable to the left of the “=” operator. Therefore, the value of expression “x = 1” is 1. We can chain several assignments together as in “x = y = 1”, which is executed as “x = (y = 1)”. The assignment operator “=” is right-associative (the evaluation for a sequence of chained assignment operations is from right to left). The above example chained assignment assigns value 1 into both variables x and y.
 - r. Assignments like “x = 1” and “x++”, as well as traditional expressions like “x > 1”, are all called expressions in Java. Mathematical expressions’ evaluation will return a value and not modify the value of variables in the expressions. But expressions like “x = 1” and “x++” modify the value of variables in the expression as well as return a value for the expression. The latter expressions are called *expressions with side effects*. Expressions with side effects can be used as statements. Expressions with side effects make writing correct programs much harder.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile the source code, type
javac -d ~/classes LocalVariable.java
 You can use a file explorer to check that a new file “~/classes/LocalVariable.class” has been created.
 4. To run the new class, type the following command:
java LocalVariable
 5. Carefully read the explanations, and make sure that you understand how the source code generates the printout.

The following is a screen capture for my test run session:



3.6 Basic built-in data types and type casting

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “BuiltinTypes.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1 public class BuiltinTypes {
2     public static void main(String[] args) {
3         // Variable declaration and initialization
4         int i = 0;
5         char c = 'c';
6         String s = "Hello";
7         double d = 1.414;
8         boolean isOK = true;
9         // Convert int string into int
10        i = Integer.parseInt("12");
11        System.out.println("String \"12\" is converted into int " + i);
12        // Convert double string into double
13        d = Double.parseDouble("3.14");
14        System.out.println("String \"3.14\" is converted into double " + d);
15        // Get a char from a string
16        c = s.charAt(0);
17        System.out.println("The first character of \" " + s + "\" is '" + c
18                           + "'");
19        // Convert a boolean string into boolean
20        isOK = Boolean.parseBoolean("false");
21        System.out.println("isOK now has value " + isOK);
22        // int remainder operator %
23        System.out.println("3 % 5 = " + (3 % 5));
24        System.out.println("5 % 2 = " + (5 % 2));
25        System.out.println("10 % 2 = " + (10 % 2));
26        // Integer division and implicit type promotion
27        System.out.println("5/2 = " + (5/2));
28        System.out.println("5.0/2 = " + (5.0/2));
29        // Convert int into double, implicitly
30        i = 12;
31        d = i;
32        System.out.println("d has now value " + d);

```

```

33 // Convert double into int, explicitly with type casting
34 i = (int)d;
35 System.out.println("i has now value " + i);
36 // Conversion between int and char
37 c = 'a';
38 System.out.println("Unicode value for 'a' is " + (int)c);
39 c = (char)(c + 1);
40 System.out.println("c now has value '" + c + "'");
41 // Relationship expressions and boolean
42 i = 5;
43 isOK = (i == 5); // equal to
44 System.out.println("(1) isOK = " + isOK);
45 isOK = (i <= 5); // less than or equal to
46 System.out.println("(2) isOK = " + isOK);
47 isOK = (i >= 6); // greater than or equal to
48 System.out.println("(3) isOK = " + isOK);
49 isOK = (i != 6); // not equal to
50 System.out.println("(4) isOK = " + isOK);
51 isOK = !isOK; // negation
52 System.out.println("(5) isOK = " + isOK);
53 isOK = (i >= 0) && (i <= 6); // Are two conditions both true?
54 System.out.println("(6) isOK = " + isOK);
55 isOK = (i < 2) || (i > 6); // At least one of the conditions is
56 // true?
57 System.out.println("(7) isOK = " + isOK);
58 // Common Math functions
59 System.out.println("abs(-2) = " + Math.abs(-2));
60 System.out.println("ceil(1.5) = " + Math.ceil(1.5));
61 System.out.println("ceil(2) = " + Math.ceil(2));
62 System.out.println("ceil(-1.5) = " + Math.ceil(-1.5));
63 System.out.println("floor(1.5) = " + Math.floor(1.5));
64 System.out.println("floor(2) = " + Math.floor(2));
65 System.out.println("floor(-1.5) = " + Math.floor(-1.5));
66 System.out.println("exp(1.0) = " + Math.exp(1.0));
67 System.out.println("log(10.0) = " + Math.log(10.0));
68 System.out.println("max(1, 2) = " + Math.max(1, 2));
69 System.out.println("min(1, 2) = " + Math.min(1, 2));
70 System.out.println("pow(2.0, 3.0) = " + Math.pow(2.0, 3.0));
71 System.out.println("sqrt(2.0) = " + Math.sqrt(2.0));
72 System.out.println("random() = " + Math.random());
73 }
74 }

```

Explanation

- Java uses exactly 32 bits to store an *int* value, hardware and OS platforms independent.
- Java uses exactly 64 bits to store a *double* (floating-point or real) value, hardware and OS platforms independent.
- If a numerical constant contains no decimal point, it is interpreted as of type *int*; otherwise as of type *double*.
- For *int* type, the supported operators include “+”, “-”, “*”, “/”, and “%”, representing addition, subtraction, multiplication, division, and remainder.
- Assume that variables x and y are both integers. Expression “x % y” will return the remainder when dividing the value of y into the value of x. If y = 2, “x % 2” can be used to check whether value x is an even number.

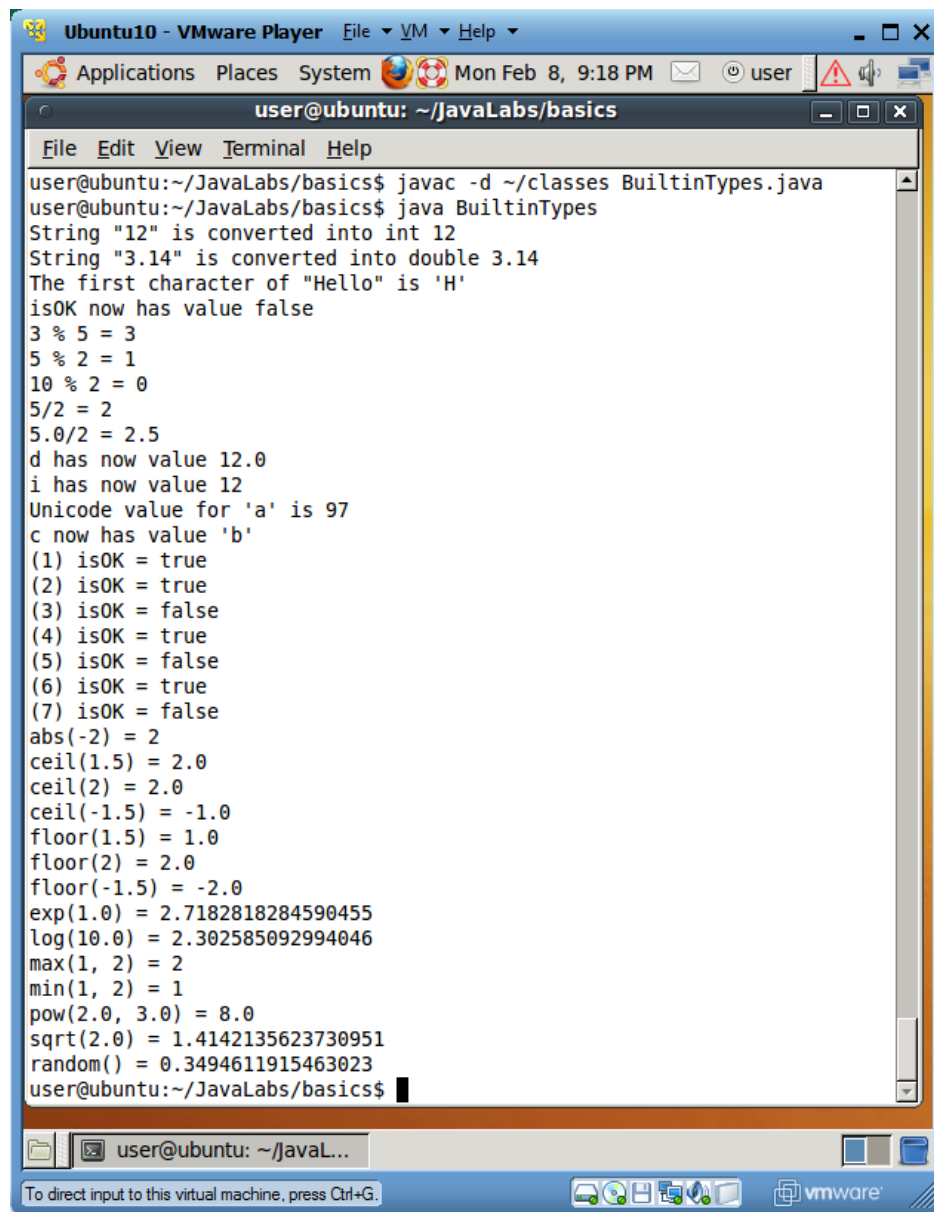
- f. If both operands for operator “/” are of type *int*, “/” is the integer division operator, and the return value is the largest integer less than or equal to the division result. For example, $5/2 = 2$.
- g. If two operands of a binary operator have different data types, the operand with a less precise type (the type that takes less bits for its value storage) will be implicitly *promoted* to the type of the another operand before the operator is applied to the operands. This is called automatic *type promotion*. Therefore, $(5.0 / 2)$ is first promoted to $(5.0 / 2.0)$ before the floating-point division “/” is applied to 5.0 and 2.0.
- h. In an arithmetic expression, parentheses () can be used recursively (nested) to override operator precedence, and square braces and curly braces are not allowed to be used. For example, “ $2 * ((x + 1)/y + 1)$ ” means that “ $x + 1$ ” will first be evaluated, its result will then be divided by the value of y , and the new result of the division will then be increased by 1, and the result of the summation will then be doubled.
- i. Java uses 16 bits to store a *char* value, which is the *Unicode* of a character. A *char* constant must be enclosed in straight single quotes, as in 'c'.
- j. A value of a lower-precision type can be assigned into a variable of a higher-precision type, and a data type conversion (promotion) will happen implicitly. As an example, “`double d = 1;`” will initialize double variable d with value 1.0. Here integer 1 is implicitly promoted to a more precise *double* type value 1.0 before it is used to initialize variable d . This kind of data type promotion is also called *implicit data type casting*.
- k. For preventing the loss of data precision, Java normally doesn’t allow a value of higher-precision to be assigned into a variable that is of a type with lower precision. For example, a *double* value is normally not allowed to be stored into an *int* variable since *int* only has 32 bits for data storage while a *double* value may use all of the allocated 64 bits for a *double* variable to store its value. But if a programmer is sure that this is what (s)he wants, (s)he can use the *explicit data type casting* mechanism to cast the more precise value into a less precise one before assigning it into a variable of a lower-precision type. For converting a value to a different type, precede the value with the destination type in parentheses. If the original value cannot be represented in the new type, data truncation may happen. For example, “`int i = (int)3.14;`” will first convert 3.14 into *int* value 3 before 3 is assigned into variable i . Here the fraction part of the original value is truncated.
- l. A *boolean* type variable can take on only one of two possible values: *true* and *false*. The value of a *boolean* variable is normally determined by the value of a *relationship expression*, which is two values compared by a binary relational operator. The popular relational operators include “==” for equal to, “!=” for not equal to, “<” for less than, “<=” for less than or equal to, “>” for greater than, and “>=” for greater than or equal to. **Attention:** “=” is for assignment, “==” is for relationship.
- m. The three most important operators on boolean values are “&&”, “||”, and “!”, called Boolean *and* (conjunction), Boolean *or* (disjunction), and Boolean *negation* respectively. Given any Boolean expression a and b , $(a \&\& b)$ is true if and only if expressions a and b are both true; $(a \parallel b)$ is false if and only if both a and b have value false; and $(!a)$ is true if and only if a has value false.
- n. To convert string “12” into int value, use “`Integer.parseInt("12")`”.

- o. To convert string “3.14” into double value, use “Double.parseDouble(“3.14”)”.
- p. To convert string “true” into a Boolean value, use “Boolean.parseBoolean(“false”)”.
- q. To retrieve the first character in string “cs121”, use “cs121”.charAt(0).
- r. Internally *char* values are stored as Unicode, a standard for encoding any international characters in 16 bits. Given a *char* literal 'a', we can get its Unicode value by casting it to an *int* value, as in “(int) 'a'”. All lower-case letters have larger Unicode values than their upper-case counterparts. All upper-case letters are assigned consecutive Unicode values, so are all lower-case letters. “((int) 'Z') + 1” is the Unicode value for 'a'. Let us assume that *c* is a *char* variable holding an upper-case letter. “System.out.println((char)(((int)c) + 26));” will print out the lower-case counterpart of the value stored in *c*.
- s. Java class *java.lang.Math* contains the following useful mathematical functions. Since all classes in package *java.lang* are always available to all classes, and all the following functions are static methods, they are supposed to be called against class name *Math*. Never try to create an object of class *Math* because that is not allowed. In the following table, i1 and i2 represent any int values, d1 and d2 represent any double values, and x1 and x2 represent either int or double values.

Method signature	Meaning
int Math.abs(x1)	Return the absolute value of the argument
int Math.ceil(d1)	Return the smallest integer that is larger than or equal to d1
int Math.floor(d1)	Return the largest integer that is smaller than or equal to d1
double Math.exp(d1)	Return the e up to d1
double Math.log(d1)	Return the natural logarithm of d1
Math.max(x1, x2)	Return the larger of x1 and x2, in the type with higher precision
Math.min(x1, x2)	Return the smaller of x1 and x2, in the type with higher precision
double Math.pow(d1, d2)	Return d1 up to the power of d2
double Math.sqrt(d1)	Return the square root of d1
double Math.random()	Return a random floating point number between 0 (inclusive) and 1 (exclusive)

- 3. Make sure your working directory is now “~/JavaLabs/basics”. To compile the source code, type
javac -d ~/classes BuiltinTypes.java
You can use a file explorer to check that a new file “~/classes/BuiltinTypes.class” has been created.
- 4. To run the new class, type:
java BuiltinTypes

The following is a screen capture for my test run session. Since *random()* returns random floating-point numbers between 0 and 1, you will get a different value from *random()* every time you run the code.



```
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes BuiltInTypes.java
user@ubuntu:~/JavaLabs/basics$ java BuiltInTypes
String "12" is converted into int 12
String "3.14" is converted into double 3.14
The first character of "Hello" is 'H'
isOK now has value false
3 % 5 = 3
5 % 2 = 1
10 % 2 = 0
5/2 = 2
5.0/2 = 2.5
d has now value 12.0
i has now value 12
Unicode value for 'a' is 97
c now has value 'b'
(1) isOK = true
(2) isOK = true
(3) isOK = false
(4) isOK = true
(5) isOK = false
(6) isOK = true
(7) isOK = false
abs(-2) = 2
ceil(1.5) = 2.0
ceil(2) = 2.0
ceil(-1.5) = -1.0
floor(1.5) = 1.0
floor(2) = 2.0
floor(-1.5) = -2.0
exp(1.0) = 2.7182818284590455
log(10.0) = 2.302585092994046
max(1, 2) = 2
min(1, 2) = 1
pow(2.0, 3.0) = 8.0
sqrt(2.0) = 1.4142135623730951
random() = 0.3494611915463023
user@ubuntu:~/JavaLabs/basics$
```

3.7 Command-line arguments and loops

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “CommandLineArguments.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 public class CommandLineArguments {
2     public static void main(String[] args) {
3         for (int i = 0; i < args.length; i++) {
4             System.out.println(args[i]);
5         }
6         // Sorting strings in args[]
7         java.util.Arrays.sort(args);
```

```

8      int j = 0;
9      while (j < args.length) {
10         System.out.println(args[j]);
11         j++;
12     }
13 }
14 }

```

Explanation:

- a. When you run a Java class, the strings after the class name are command-line arguments.
- b. If a command-line argument's value contains white space characters, the value must be enclosed in straight double quotes ("..."), where the double quotes are not part of the argument value.
- c. Method *main()* has a parameter *args* whose data type is an array (sequence) of Strings. The number of cells in this array is equal to the number of command-line arguments that are actually provided for the current run, and this number is also available as the value of *args.length*, which is an *instance variable* of data type *String[]*. If no command-line argument is provided for a run, *args.length* will have value 0, and there will be no cells in array *args[]*.
- d. Command-line arguments are sequentially stored in *args[0]*, *args[1]*, *args[2]*, ... For a particular run, if the user provides 3 command-line arguments, *args.length* will have value 3, and the argument values will be stored in *args[0]*, *args[1]*, and *args[2]*.
- e. *Arrays* is a class with package path *java.util*, and class *Arrays* has a public static method *sort()* that can be used to sort an array of strings or an array of numbers. Statement "java.util.Arrays.sort(args);" sorts the string values in array *args[]* so they will be rearranged in the lexicographic (dictionary) ordering.
- f. A *compound statement* is a sequence of statements enclosed in a pair of curly braces, like {...}. The body of a method is a compound statement. You can declare local variables anywhere inside a compound statement. Variables declared inside a compound statement will only be accessible from the point of the declaration to the end of the terminating curly brace for the compound statement. No methods can be declared inside a compound statement. Therefore, the body of a class declaration is not a compound statement. A compound statement can be used to replace any statement in a program without violating the syntax specification for Java.
- g. A loop is a sequence of statements that will be executed by the CPU repetitively based on some Boolean (true or false) conditions. The most popular loops are the *while* loop and the *for* loop.
- h. The while loop is of syntax "**while (booleanExpression) {...}**", where "while" is the reserved word for *while* loop, *booleanExpression* is any expression that can be evaluated to a boolean value *true* or *false*, and "{...}" is any compound statement. If the compound statement contains only one statement, the compound statement can be replaced with the single statement, and the while loop will then look like "while (booleanExpression) statement;". The execution of a while loop is repeating the following sequence of actions until *booleanExpression* is false:
 - a. Evaluate the Boolean expression *booleanExpression*. If the evaluation result is false, terminate this loop and execution resumes immediately after the loop body.

- b. Execute the body of the loop's compound statement.
 - c. Go to step a.
 - i. A loop normally uses a *loop variable* to control its execution. In our example, the loop variable is *j*. The boolean expression for a loop will check the value of the loop variable to see whether the loop should terminate. The loop body will modify the value of the loop variable so the loop has a chance to terminate eventually.
 - j. A *for* loop is of syntax "**for (loopVarInitialize; booleanExpression; changeLoopVariable) {...}**", where "for" is the reserved word for *for* loops, loopVarInitialize is either a variable assignment or a local variable declaration with value initialization, booleanExpression is any Boolean expression for checking whether the loop should terminate, changeLoopVariable is for modifying the loop variable, and "{...}" is any compound statement. "**(loopVarInitialize; booleanExpression; changeLoopVariable)**" is called the *control block* of a *for* loop. If a component of the control block is not needed, it can be empty, but the two semicolons for separating the three control block components must always be present. In the simplest form, the *for* loop can be of form "**for (;) {...}**", which has no loop variable initialization, the boolean expression is always true, and the loop variable is not modified in the loop's control block (but most likely the loop variable is modified in the compound statement). If the compound statement contains only one statement, the compound statement can be replaced with the single statement, and the *for* loop will then look like "**for (loopVarInitialize; booleanExpression; changeLoopVariable) statement;**". The execution of a *for* loop is repeating the following sequence of actions until booleanExpression is false:
 - a. Execute loop variable initialization loopVarInitialize.
 - b. Evaluate the boolean expression boolExpression to see whether its value is true. If the evaluation value is false, the loop terminates, and the program execution resumes immediately after this loop body.
 - c. Execute the loop body.
 - d. Update the loop variable value by executing statement changeLoopVariable.
 - e. Go to step b.
 - k. If a loop never terminates, it is called an *infinite loop*. An infinite loop can only be terminated by operating system operations like Ctrl+C or Ctrl+Z. Any program that may run into an infinite loop is not a correct program.
 - l. Each execution of a loop's body is called one *iteration* of the loop. The execution of a loop is made of many iterations of the loop.
 - m. If a loop variable is declared in the control block of a *for* loop, the loop variable is only accessible inside the loop's body. It is not accessible after the *for* loop.
5. Make sure your working directory is now "`~/JavaLabs/basics`". To compile the source code, type
- ```
javac -d ~/classes CommandLineArguments.java
```
- You can use a file explorer to check that a new file "`~/classes/CommandLineArguments.class`" has been created.
6. To run the new class, type like the following line, and replace the command-line arguments with yours:
- ```
java CommandLineArguments Java is cool "Hello world" 21 1
```

The following is a screen capture for my test run session. This program first prints the command-line arguments in the same order that they are listed on the command-line, and then prints them in the lexicographic order. Note that in lexicographic order, upper-case letters precede lower-case letters, numbers precede letters, and short strings precede longer strings that use the short strings as the prefixes.

```

user@ubuntu: ~/JavaLabs/basics
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes CommandLineArguments.java
user@ubuntu:~/JavaLabs/basics$ java CommandLineArguments Java is cool "Hello world" 21 1
Java
is
cool
Hello world
21
1
1
21
Hello world
Java
cool
is
user@ubuntu:~/JavaLabs/basics$

```

3.8 Calculator with if-else statements

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “Calculator1.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1 public class Calculator1 {
2     public static void main(String[] args) {
3         //for (int i = 0; i < args.length; i++)
4         //    System.out.println(args[i]);
5         if (args.length != 3) {
6             System.out.println("Usage: java    Calculator1    double    op    double\n"
+
7                 "where op can be +, -, x, or /");
8             System.exit(-1);
9         }
10        double d1 = Double.parseDouble(args[0]);
11        double d2 = Double.parseDouble(args[2]);
12        char op = args[1].charAt(0);
13        double result = 0;
14        if (op == '+')
15            result = d1 + d2;
16        else if (op == '-')
17            result = d1 - d2;
18        else if (op == 'x')
19            result = d1 * d2;
20        else if (op == '/')
21            result = d1 / d2;
22        else {
23            System.out.println("Error: accepted operators are +, -, x, and /");
24            System.exit(-1);
25        }

```

```

26     System.out.println(d1 + " " + op + " " + d2 + " = " + result);
27 }
28 }

```

Explanation

- Each array has an instance variable named “length” that can be used to find the length of the array (number of cells in the array). An *instance variable* is a variable that is defined inside a class but outside of a method and that is *not static*. If a class-level variable is static, it is called a *class variable*.
- Inside a String literal (constant), “\n” is used to introduce a *New Line* character, and “\t” is used to introduce a *Tab* character.
- An *if* statement has syntax

if (BooleanExpression) Statement;

where “if” is a Java reserved word, BooleanExpression is any expression that has a value either true or false (normally relational expressions), and the statement can be a compound statement, which is a sequence of statements enclosed in a pair of curly braces. During execution of this *if* statement, first the Boolean expression is evaluated. If the Boolean expression has value *true*, the (compound) statement is executed. Otherwise the (compound) statement is skipped, and the CPU executes the following statement.

- Reserved word “else” can be used to chain a list of *if* statements together into an *if-else* statement. For example, when statement

```

if (BooleanExpression1)
    statement1;
else if (BooleanExpression2)
    statement2;
else
    statement3;

```

is executed, *BooleanExpression1* is first evaluated. If it is true, *statement1* is executed, and execution then goes to the statement after *statement3*. If *BooleanExpression1* is false, then *BooleanExpression2* is evaluated. If it is true, *statement2* is executed, and execution then goes to the statement after *statement3*. If *BooleanExpression2* is false, then *statement3* will be executed, and execution continues to the next statement. In this *if-else* statement, exactly one of *statement1*, *statement2* and *statement3* is executed.

- The “else statement3;” part of the above if-else statement is optional. Without it, if both *BooleanExpression1* and *BooleanExpression2* have value false, the execution just continues from the following statement.
- There is no limitation on how many if-else statements can be chained together.
- Library class “System” has a static method “exit(int)” that can be used to terminate the execution of the current program. Method *exit(int)* takes one integer argument. You can pass any positive or negative integer to method *exit(int)*. As a convention, we use “System.exit(0);” to terminate a program normally, and we pass a non-zero integer to method *exit(int)* to indicate that a particular type of error happened before the program execution’s termination. At the operating system level (more often in OS batch or shell scripts), we can check this termination status value of a program. This value can be used to decide which program should run next. For simple programs you don’t need to terminate a program with “System.exit(int);”. When execution comes to the ending curly

brace of the body of method *main()*, the program will be terminated implicitly. You can use “System.exit(int);” when you need to terminate a program before executing some other statements in sequence. When you run a Java program that has graphic user interfaces, you should call “System.exit(int);” to explicitly terminate a program since such programs normally involve multiple concurrent threads (covered in Section 5) waiting indefinitely for user inputs, and they will not terminate by themselves.

- h. This program uses lower-case ‘x’ as the multiplication operator. It cannot use ‘*’ for command-line operator since on command-line “*” represents all file and directory names in the current working directory.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile the new class, type

```
javac -d ~/classes Calculator1.java
```

4. To run this new class, type lines like:

```
java Calculator1 1 + 2
```

```
java Calculator1 3 x 4
```

5. Insert the following loop at the beginning of method *main()*:

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

```
    System.out.println(args[i]);
```

Recompile the source code. Run the following line:

```
java Calculator1 2 * 3
```

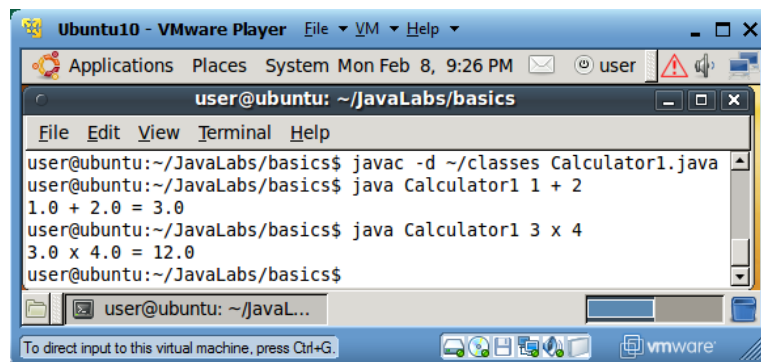
You will see that “*” is replaced with all file and directory names in the working directory, and the program fails to run due to wrong number of command-line arguments.

6. Comment off “else {...}” from the source code, recompile the code, then run the following line:

```
java Calculator1 7 % 2
```

In this test, “%” is an operator not supported by the program. Since no *if* statements’ Boolean expressions can be true and there is no “else {...}” statement at the end of the *if* statements, the program will not detect the illegal operator and it will let the last statement print garbage.

The following is a screen capture for part of my test run session.



3.9 Calculator with switch statement

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.

2. Use a text editor to open and review Java source code file “Calculator2.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 public class Calculator2 {
2     public static void main(String[] args) {
3         if (args.length != 3) {
4             System.out.println("Usage: java Calculator2 double op double\n"
5                                 + "where op can be +, -, x, or /");
6             System.exit(-1);
7         }
8         double d1 = Double.parseDouble(args[0]);
9         double d2 = Double.parseDouble(args[2]);
10        char op = args[1].charAt(0);
11        double result = 0;
12        switch (op) {
13            case '+':
14                result = d1 + d2;
15                break;
16            case '-':
17                result = d1 - d2;
18                break;
19            case 'x':
20                result = d1 * d2;
21                break;
22            case '/':
23                result = d1 / d2;
24                break;
25            default:
26                System.out.println(
27                    "Error: accepted operators are +, -, x, and /");
28                System.exit(-1);
29        }
30        System.out.println(d1 + " " + op + " " + d2 + " = " + result);
31    }
32 }
```

Explanation

- a. *Calculator2* functions the same as *Calculator1*. Their only difference is that *Calculator2* uses a single *switch* statement to replace *Calculator1*'s *if-else* statements.

- b. Java *switch* statement has the following syntax:

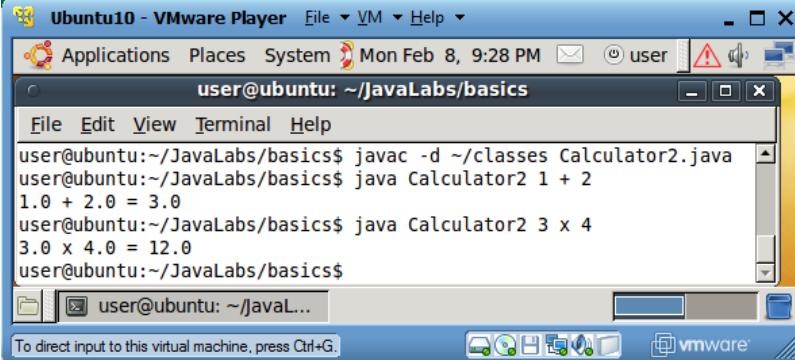
```
switch (switch-expression) {
    case value1: statements;
                break;
    case value2: statements;
                break;
    .....
    default:    statements;
                break;
}
```

where “switch”, “case”, “default”, and “break” are Java reserved words; switch-expression is normally a variable of type *int*, *char*, or *enum* (introduced in Section 5), and values after reserved word “case” must be literal (constant) values of the same data type as switch-expression. “**case** value:” is called a case label, in

which the *value* is called a *case value*. Different *case* labels must have different case values. The body of a switch statement is a compound statement in which some statements are preceded with case labels. Each case label is a possible execution start point for the switch statement body.

- c. During the execution of a switch statement, the switch expression is first evaluated. If its value is equal to the value of any of the case labels, execution resumes from the statement qualified by that case label. The execution will continue, ignoring case labels, until it meets a *break* statement, which will stop the execution of the switch statement, and the CPU will resume execution at the first statement after the body of this switch statement. If the value of the switch expression does not match any of the case values, then the execution starts from the statement qualified by the *default label*.
 - d. The *default* case of a *switch* statement is optional. If the switch expression value does not match any case values and there is no default case specified, the execution will just continue at the first statement after the body of this switch statement. Be careful: unless the default case is the last case, or it terminates with a *break* statement or a statement that will terminate the execution of the program, like *System.exit(int)*, the execution will continue to the following cases.
 - e. As long as the statements for each switch case are terminated by a *break* statement, the order of the cases in a switch statement is not important.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *Calculator2*, type
javac -d ~/classes Calculator2.java
 4. To run *Calculator2*, type lines like
java Calculator2 1 + 2
java Calculator2 3 x 4
 5. Reverse the order of the cases in the *switch* statement, recompile the class, run it to see whether any behavior of *Calculator2* changed. You should notice that the program behavior has not changed.
 6. Delete the *default* case, recompile the class, run it with line
java Calculator2 7 % 2
You should observe the same error as happened to *Calculator1* for the similar situation.

The following is the screen capture of my sample run of *Calculator2* for steps 3 and 4.



```
Ubuntu10 - VMware Player  File  VM  Help
Applications  Places  System  Mon Feb 8, 9:28 PM  user
user@ubuntu: ~/JavaLabs/basics
File  Edit  View  Terminal  Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes Calculator2.java
user@ubuntu:~/JavaLabs/basics$ java Calculator2 1 + 2
1.0 + 2.0 = 3.0
user@ubuntu:~/JavaLabs/basics$ java Calculator2 3 x 4
3.0 x 4.0 = 12.0
user@ubuntu:~/JavaLabs/basics$
user@ubuntu: ~/JavaL...
```

3.10 Calculator with exception processing

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “Calculator3.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 public class Calculator3 {
2     public static void main(String[] args) {
3         if (args.length != 3) {
4             System.out.println("Usage: java Calculator3 double op double\n"
5                               + "where op can be +, -, x, or /");
6             System.exit(-1);
7         }
8         double d1 = 0.0, d2 = 0.0;
9         try {
10            d1 = Double.parseDouble(args[0]);
11            d2 = Double.parseDouble(args[2]);
12        }
13        catch (Exception e) {
14            // System.out.println(e);
15            System.out.println(
16                "Error: at least one of the operands is not a number");
17            System.exit(-2);
18        }
19        /*
20        finally {
21            System.out.println("Finally clause executed. ");
22        }
23        */
24        char op = args[1].charAt(0);
25        double result = 0;
26        switch (op) {
27            default:
28                System.out.println(
29                    "Error: accepted operators are +, -, x, and /");
30                System.exit(-3);
31            case '+':
32                result = d1 + d2;
33                break;
34            case '-':
35                result = d1 - d2;
36                break;
37            case 'x':
38            case 'X':
39                result = d1 * d2;
40                break;
41            case '/':
42                result = d1 / d2;
43                break;
44        }
45        System.out.println(d1 + " " + op + " " + d2 + " = " + result);
46    }
47 }
```

Explanation

- a. *Calculator3* functions almost the same as *Calculator2*. But *Calculator3* supports both ‘x’ and ‘X’ as multiplication operators, and it uses the Java exception

handling mechanism to print more meaningful error messages when one of the operands is not a number.

- b. This example reinforces the point that the default case can be anywhere in the body of a switch statement, as long as its statements terminate the execution of the switch statement.
- c. This example also reinforces the point that case labels are just labels for some statements in the body of a switch statement. These labels indicate potential execution start points. In this example, both case labels “case 'x':” and “case 'X':” qualify statement “result = d1 * d2;” Therefore, if *op* has value either 'x' or 'X', execution will start from statement “result = d1 * d2;”
- d. Java has another form of comments delimited by “/*” and “*/”. This form of comments starts with “/*”, and ends with “*/”. Such comments can include a few lines, while the “//” form of comments must terminate at the end of the current line. The Java compiler will ignore both forms of comments. In addition to using comments as a form of Java source code documentation, we can also use them for temporarily deleting some statements from the source code so we can recover them easily later,
- e. During the execution of a method, some illegal events may happen. For example, suppose that a method contains an assignment statement “y = 5/x;”. At code compilation time, Java compiler has no idea what will be the value of variable “x” when this assignment is executed. But during the execution of the expression “5/x” at runtime, the value of “x” happens to be zero (0). It is illegal for zero to divide into any number, and the Java runtime system will *throw* a Divide-By-Zero *exception* to the method. As another example, when *Calculate2*’s method *main()* uses “Integer.parseDouble(args[0])” to convert the string form of a number into a double value, if the command-line argument args[0] is not representing a number, then method *parseDouble()* will also throw an exception to method *main()* informing it that something is seriously wrong. It is common for a method to throw an exception object back to its invoker when it finds something seriously wrong.
- f. If a method invocation statement receives an exception and it does not catch that exception, the current method containing the method invocation statement will be terminated, and the exception, like a ball, will be thrown to the invocation statement for the current method. This chain reaction will continue until the current method is *main()*, which may lead to the termination of the program with low-level exception messages printed to the *Command Prompt* window.
- g. Java provides an *Exception Handling* mechanism for a Java program to print meaningful error messages or recover from an exception when an exception happens. If some statements may cause exceptions at runtime, enclose them in a *try-catch block* as below:

```
try { // try clause
    statements that may cause exceptions
}
catch (Exception e) { // catch clause
    statements executed when exceptions happen
}
finally { // finally clause
    statements that will be executed no matter exceptions
    happen or not; normally for recovering resources allocated
```

in the try block.

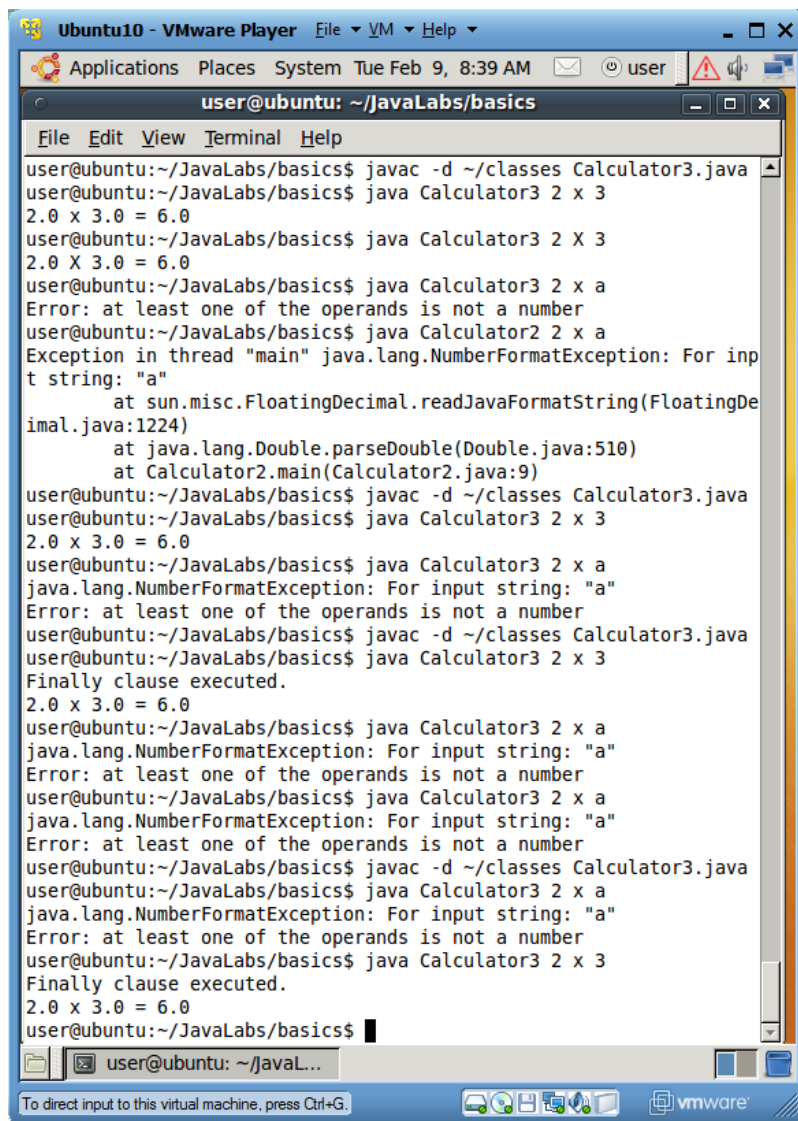
}

where “try”, “catch”, and “finally” are Java reserved words, the finally clause is optional, and *Exception* is the root type (class) for all Java exception types. When a try-catch block is executed, the execution will start from the first statement of the try clause. If no exception is thrown during the execution of the statements in the try clause, the body of the finally clause will be executed, and execution resumes at the first statement after the try-catch block. If any statement in the try clause causes an exception, the following statements of the try clause will be skipped, and execution continues at the first statement of the exception clause. After the statements of the catch clause have been executed, or before the current method terminates, the statement of the finally clause will be executed (if the catch clause terminates the program, then the finally clause will not be executed). The finally clause is used to execute some statements that must be executed no matter the exceptions happen or not.

- h. A try-catch block can contain a list of catch clauses between the try clause and the finally clause, each for catching a particular type of exceptions. A catch clause has a parameter with an exception type. In our example, the parameter name is “e”, and its type is Java library class *Exception*. In the body of this catch clause, we can access all information about this particular exception. We can simply print the value of parameter “e”. Later we will learn how to declare our own exception types, which are Java classes. The *Exception* class is the root class of all types of exceptions, so our example catch clause can be used to catch all types of exceptions.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *Calculator3*, type
javac -d ~/classes Calculator3.java
4. To run class *Calculator3*, type lines like
java Calculator3 2 x 3
java Calculator3 2 X 3
java Calculator3 2 x a (Error message expected)
java Calculator2 2 x a (Compare outputs for *Calculator2* and *Calculator3*)
5. Insert statement “System.out.println(e);” as the first statement of the catch clause of class *Calculator3*, recompile class *Calculator3*, and redo step 4.
6. Add the following finally clause immediately after the catch clause of class *Calculator3*. Recompile class *Calculator3*, and redo step 4.

```
finally {  
    System.out.println("Finally clause executed. ");  
}
```

The following is the screen capture of my test run for *Calculator3*.



The finally clause was not executed for the last step “java Calculator3 2 x a” because the “System.exit(-2);” statement in the catch clause terminated the program.

3.11 Conditionally interrupting loops

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “LoopInterruption.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 public class LoopInterruption {
2     public static void main(String args[]) {
3         int i = 0;
4         for (; i < 10; i++) {
5             System.out.println("Enter loop body: i = " + i);
6             if ((i % 2) == 0) continue;
7             if (i == 7) break;
8             System.out.println("Leave loop body: i = " + i);
9         }
10    }
```

```

10     System.out.println("After loop: i = " + i);
11 }
12 }

```

Explanation:

- a. If a *break* statement is executed inside a loop, the loop execution will be terminated, and execution will resume at the first statement after the loop containing the break statement.
 - b. If a *continue* statement is executed inside a loop, the current execution of the loop body stops, and the loop starts to prepare for its next iteration. If the loop is a *for* loop, a *continue* statement will stop the current loop body execution, the loop variable modification component of the *for* loop control block will be executed, and then the Boolean expression in the *for* loop control block will be evaluated to determine whether the loop needs the next iteration. If the loop is a *while* loop, a *continue* statement will stop the current loop body execution, and the while loop's Boolean expression will be evaluated to determine whether the loop needs the next iteration.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *LoopInterruption*, type
javac -d ~/classes LoopInterruption.java
 4. To run class *LoopInterruption*, type
java LoopInterruption

The following is the screen capture of my test run.

```

user@ubuntu: ~/JavaLabs/basics
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes LoopInterruption.java
user@ubuntu:~/JavaLabs/basics$ java LoopInterruption
Enter loop body: i = 0
Enter loop body: i = 1
Leave loop body: i = 1
Enter loop body: i = 2
Enter loop body: i = 3
Leave loop body: i = 3
Enter loop body: i = 4
Enter loop body: i = 5
Leave loop body: i = 5
Enter loop body: i = 6
Enter loop body: i = 7
After loop: i = 7
user@ubuntu:~/JavaLabs/basics$

```

3.12 Arrays

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “Arrays.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1 public class Arrays {
2     public static void main(String[] args) {
3         // 1-D arrays
4         int[] a1 = new int[6];
5         for (int i = 0; i < a1.length; i++)
6             a1[i] = i;
7         for (int i = 0; i < a1.length; i++)
8             System.out.print("a1[" + i + "] = " + a1[i] + " ");
9         System.out.println(); // add "\n"
10        int total = 0;
11        for (int i = 0; i < a1.length; i++)
12            total += a1[i];
13        System.out.println("Array a1 has total value " + total);
14        int[] a2 = {24, 12, 18, 45, 14, 5, 8};
15        for (int i = 0; i < a2.length; i++)
16            System.out.print("a2[" + i + "] = " + a2[i] + " ");
17        System.out.println(); // add "\n"
18        // 2-D arrays
19        int[][] a3 = new int[4][5];
20        for (int i = 0; i < a3.length; i++) {
21            for (int j = 0; j < a3[i].length; j++)
22                a3[i][j] = i + j;
23        }
24        System.out.println("Contents of array a3:");
25        for (int i = 0; i < a3.length; i++) {
26            for (int j = 0; j < a3[i].length; j++)
27                System.out.print(a3[i][j] + " ");
28            System.out.println(); // Add a "\n"
29        }
30    }
31 }

```

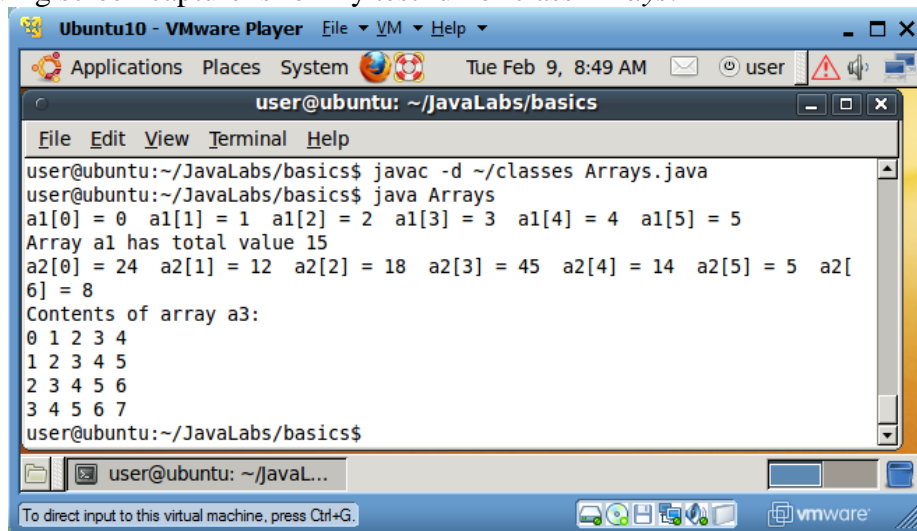
Explanation

- a. A one-dimensional array is a sequence of cells each with consecutive index values. The smallest index value is 0. All cells of an array must have the same data type.
- b. Array variable declaration “int[] a1;” declares that “a1” is a variable whose data type is an array of integers. Array variable “a1” only takes the fixed 32 bits for storing the starting address for the first cell of the array. This array variable declaration does not allocate space for the array cells. It does not limit the length of the array either.
- c. The array cells can be allocated with expressions like “a1 = new int[6];” Expression “new int[6]” will allocate space for 6 cells, each with 32 bits for an *int* value, and return the starting address of the first cell, which is then saved in array variable “a1”. After this array space allocation, the array cells will have random *int* values.
- d. Each cell of a one-dimensional array is accessed through expressions like “a1[i]”, where “a1” is an array variable, and “i” is a valid integer index value for array “a1”. For this particular case, index “i” can only take on values from 0 to 5 inclusive.
- e. If we know that a one-dimensional array needs just to hold a list of constant values, we can combine the variable declaration and array space allocation in a form like “int[] a2 = {24, 12, 18, 45, 14, 5, 8};”, which declares that “a2” is an array variable of type array of integers, allocates space for an array of 7 cells

(since we listed 7 values in the pair of curly braces), sets the first cell's address in array variable "a2", and initializes the array cells with the values given in the curly braces in the same order: $a2[0] = 24$, $a2[1] = 12$, ..., $a2[6] = 8$.

- f. Similarly, we can use "int[][] a3 = new int[4][5];" to declare that "a3" is an array variable for a two-dimensional array of integers, and we use expression "new int[4][5]" to allocate space for a two-dimensional array of integers with 4 rows and 5 columns. To access a cell in array "a3", we use expression "a3[i][j]", where i is an index value between 0 and 3 inclusive, for the row; and j is an index value between 0 and 4 inclusive, for the column.
 - g. To find the length of a particular row, say row a3[i], we use expression "a3[i].length". A two-dimensional array can be viewed as a one-dimensional array with each cell holding another one-dimensional array for the row.
 - h. We can generalize the above pattern to declare 3-D arrays. "int[][][] z3 = new int[4][5][6];" declares a 3-D array made up of 4 planes, each plane being a 2-D array containing 5 rows and 6 columns. To access the cell on plane 2, row 3 and column 4, use expression z3[2][3][4].
3. Make sure your working directory is now "~/JavaLabs/basics". To compile class *Arrays*, type
javac -d ~/classes Arrays.java
 4. To run class *Arrays*, type
java Arrays

The following screen capture is for my test run of class *Arrays*.



```
user@ubuntu: ~/JavaLabs/basics
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes Arrays.java
user@ubuntu:~/JavaLabs/basics$ java Arrays
a1[0] = 0  a1[1] = 1  a1[2] = 2  a1[3] = 3  a1[4] = 4  a1[5] = 5
Array a1 has total value 15
a2[0] = 24  a2[1] = 12  a2[2] = 18  a2[3] = 45  a2[4] = 14  a2[5] = 5  a2[6] = 8
Contents of array a3:
0 1 2 3 4
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
user@ubuntu:~/JavaLabs/basics$
```

3.13 Method declaration and invocation

1. Open a terminal window. Change directory to "~/JavaLabs/basics".
2. Use a text editor to open and review Java source code file "Method.java" in directory "~/JavaLabs/basics" and the file has contents below:

```
1 public class Method {
2     public static void main(String[] args) {
3         int x = 0;
```

```

4     x = square(2);
5     print(x);
6     x = max(x, 1, 5);
7     print(x);
8     print(min(5, 2, 8));
9 }
10
11 static int square(int x) {
12     return x*x;
13 }
14
15 static int max(int x1, int x2, int x3) {
16     if ((x1 > x2) && (x1 > x3))
17         return x1;
18     else if (x2 > x3)
19         return x2;
20     else
21         return x3;
22 }
23
24 static int min(int x1, int x2, int x3) {
25     int smaller = ((x1 < x2) ? x1 : x2);
26     return smaller < x3 ? smaller : x3;
27 }
28
29 static void print(int x) {
30     System.out.println("Output: " + x);
31 }
32 }

```

Explanation

- a. A program may contain some computation that needs be executed multiple times. Instead of repeating the source code for the common computation again and again, we put them in a *method* for multiple invocations.
- b. A method needs be declared before it can be invoked. Method declaration is contained in a Java class. A method cannot be declared inside a method. A method declaration includes a method return data type (at the end of method execution, what type of value will be sent back to the method invoker), method name, which is an identifier with the first letter in lower-case, a parameter list specified between a pair of parentheses (input values to the method), and a compound statement. We also say that a method declaration includes a *method signature* and a *method body*, where the method signature includes method return type, method name, and parameter list; and method body is the compound statement.
- c. The parameter list is a list of “type name” pairs separated by comma (,), where name is a parameter variable, and type is the variable’s data type. A parameter variable (from now on we just call it a parameter) is part of the method body’s local variable, and can be accesses anywhere in the method body, the compound statement. The name of a parameter can be changed without affecting the function of the method as long as all occurrences of the parameter in the method body are changed accordingly too. The order of parameter declarations in the parameter list is important and must be consistent with the way of method invocations.
- d. Some methods have “void” as return type. These methods will not return any value to the method invoker through the *method return value mechanism* (it may

use other mechanisms to pass some value back to the invoker, as we will see in Section 4 when we discuss class-typed parameters). When the execution of such method body comes to the end, or a return statement of form “**return;**” is executed, the execution resumes in the invoker method at the statement immediately after the method invocation statement.

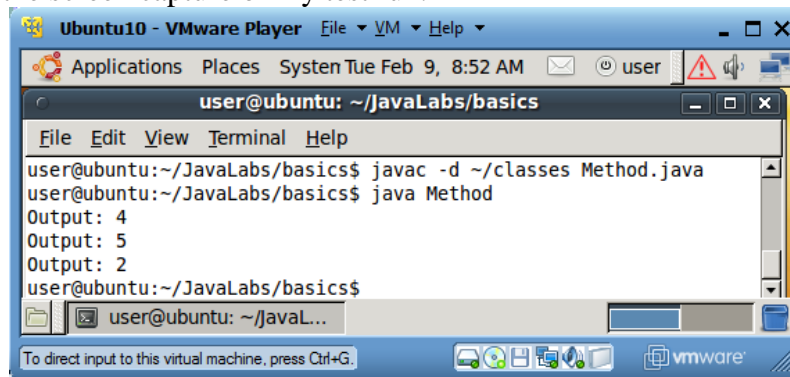
- e. For a method that has a non-void return type, every possible execution path of its method body must end with a *return* statement of form “**return expression;**” where *expression* is any expression of the same data type as the method’s return data type. When execution comes to such a *return* statement, the value of the expression in the *return* statement is evaluated and labeled as *method return value*, the execution resumes in the invoker method at the expression that contains the method invocation, the method invocation is replaced with the method return value, and execution resumes with the evaluation of the resulting expression.
- f. To invoke a method, we use the method invocation expression like
`methodName(argument1, argument2, ..., argumentN)`
 where N is the number of parameters listed in the parameter list of method *methodName*’s declaration, each argument is an expression that has the same data type as its corresponding parameter in the method declaration (relative to order of the argument list and the parameter list). Upon evaluation of this method invocation expression, each argument expression is evaluated and the resulting value is copied into its corresponding parameter of the method, the method containing the method invocation suspends its execution, then the invoked method’s body is executed with the initialized parameters treated as local variables. Upon the completion of the method execution, either by executing a return statement or coming to the end of statements of the method body, the invoked method’s execution is terminated, the method invocation expression in the invoking method is replaced with the method return value if there is any, and the CPU resumes its execution at the resulting expression or statement (remember, expression is a special case of a statement).
- g. Method declaration order in a class is not important. A method’s body can invoke methods declared after this invoking method.
- h. The statements in a static method can only invoke static methods. Since method *main()* is static by Java specification, all methods invoked from method *main()* must be declared static too. A static method can be invoked from another class with syntax “*ClassName.staticMethod()*”, where *staticMethod* is a static method declared in class *ClassName*. All methods in Java library class *java.lang.Math* are static methods.
- i. A *conditional expression* can return the value of one of two different sub-expressions depending on whether a Boolean expression is true or false. A conditional expression has the following syntax:

BooleanExpression ? sub-expression1 : sub-expression2

When this expression is evaluated, the Boolean expression is first evaluated. If the evaluation result is *true*, sub-expression1 is evaluated, and its evaluation result will be returned as the evaluation value of this entire conditional expression. If the Boolean expression evaluates to value *false*, sub-expression2 is evaluated, and its evaluation result will be returned as the evaluation value of this entire conditional expression. Therefore, expression (1 > 2 ? 3 : 4) will return value 4 since Boolean expression (1 > 2) is false, and expression (1 < 2 ? 3 : 4) will return value 3 since Boolean expression (1 < 2) is true.

- j. In class *Method*, we declared four methods. Method “int square(int x)” is for squaring the integer in parameter x. Method “int max(int x1, int x2, int x3)” is for finding the maximum value among parameter values. Method “int min(int x1, int x2, int x3)” is for finding the minimum value among parameter values. Method “void print(int x)” is for printing parameter value in a special way.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *Method*, type
- ```
javac -d ~/classes Method.java
```
4. To run class *Method*, type
- ```
java Method
```

The following is the screen capture of my test run.



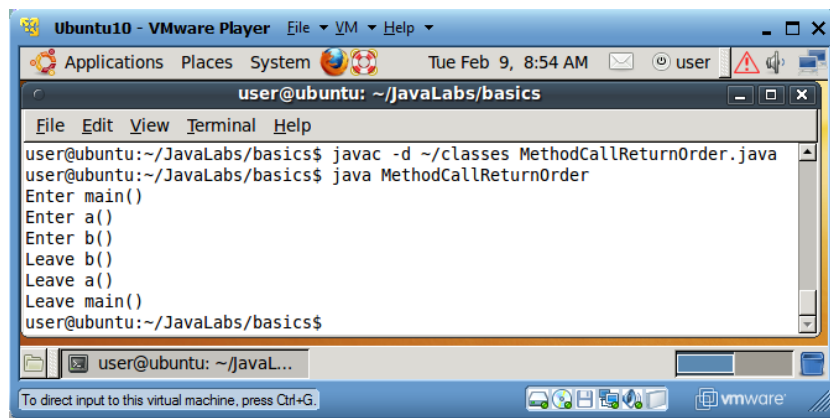
The screenshot shows a terminal window titled "user@ubuntu: ~/JavaLabs/basics". The terminal output is as follows:

```
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes Method.java
user@ubuntu:~/JavaLabs/basics$ java Method
Output: 4
Output: 5
Output: 2
user@ubuntu:~/JavaLabs/basics$
```

To further understand the *Last-Call-First-Return* nature of method invocations, please compile and run the following class *MethodCallReturnOrder*:

```
1 public class MethodCallReturnOrder {
2     public static void main(String[] args) {
3         System.out.println("Enter main()");
4         a();
5         System.out.println("Leave main()");
6     }
7
8     static void a() {
9         System.out.println("Enter a()");
10        b();
11        System.out.println("Leave a()");
12    }
13
14    static void b() {
15        System.out.println("Enter b()");
16        System.out.println("Leave b()");
17    }
18 }
```

The following is the screen capture for my compilation and run session:



3.14 Java documentation comments

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “JavaDocDemo.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1  /**
2  * Class <code>JavaDocDemo</code> shows example method
3  * declarations and invocations.
4  * @author Dr. Lixin Tao
5  * @version 1.0
6  */
7  public class JavaDocDemo {
8      /**
9       * Method <code>main()</code> is the entrance method to be executed.
10      * @param args the array of command line arguments.
11      */
12      public static void main(String[] args) {
13          int x = 0;
14          x = square(2);
15          print(x);
16          x = max(x, 1, 5);
17          print(x);
18          print(min(5, 2, 8));
19      }
20
21      /**
22      * Returns the squared value of parameter x.
23      * @param x the value to be squared.
24      * @return the squared value of parameter x.
25      */
26      static int square(int x) {
27          return x*x;
28      }
29
30      /**
31      * Finds the maximum of the input values.
32      * @param x1 the first input value.
33      * @param x2 the second input value.
34      * @param x3 the third input value.
35      * @return the maximum of the input values.
36      */

```

```

37 static int max(int x1, int x2, int x3) {
38     if ((x1 > x2) && (x1 > x3))
39         return x1;
40     else if (x2 > x3)
41         return x2;
42     else
43         return x3;
44 }
45
46 /**
47  * Finds the minimum of the input values.
48  * @param x1 the first input value.
49  * @param x2 the second input value.
50  * @param x3 the third input value.
51  * @return the minimum of the input values.
52  */
53 static int min(int x1, int x2, int x3) {
54     int smaller = ((x1 < x2) ? x1 : x2);
55     return smaller < x3 ? smaller : x3;
56 }
57
58 /**
59  * Prints input value in a special format.
60  * @param x value to be printed.
61  */
62 static void print(int x) {
63     System.out.println("Output: " + x);
64 }
65 }

```

Explanation

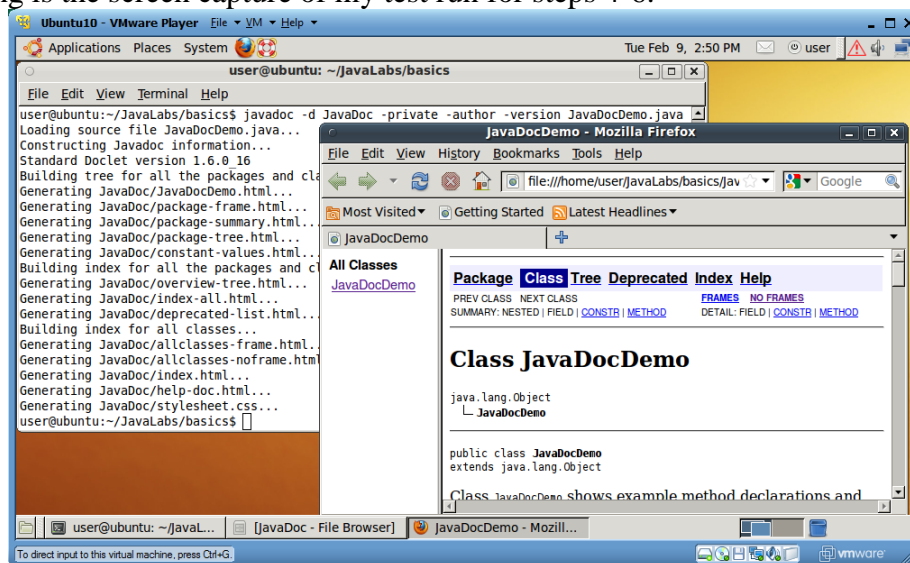
- a. Java class *JavaDocDemo* is basically the same as class *Method* except that the class name is changed and Java *doc (documentation) comments* are added before the declaration of the class and each method. Doc comments are for generating project documentation with a tool named *javadoc.exe*.
- b. A *doc comment* includes any string enclosed between a pair of “/**” and “*/” strings. The contents of a doc comment can have multiple lines, and each line can optionally start with a leading “*” which is not considered as part of the doc comment. A doc comment should immediately precede any class or method declarations, as well as any class or instance variable declarations (both will be introduced officially in Section 4). Doc comments inside a method declaration will be ignored.
- c. A *doc comment* should start with a concise sentence summarizing the function of a class or a method and end with an optional *tag section*. The tag section contains a list of tag elements each starting with a tag like “@tagName” and ending with contents for that tag. The most popular tag names include
 - i. @author, for specifying program authors
 - ii. @version, for specifying version number of the program
 - iii. @param, for specifying a parameter for a method
 - iv. @return, for specifying the return value of a method

The contents of a doc comment can use inline tag “<code> ... </code>” to include a small piece of code, or tag “<pre> ...</pre>” to include multiple lines of code aligned as it is. You can also use other HTML tags in a doc comment. But test the

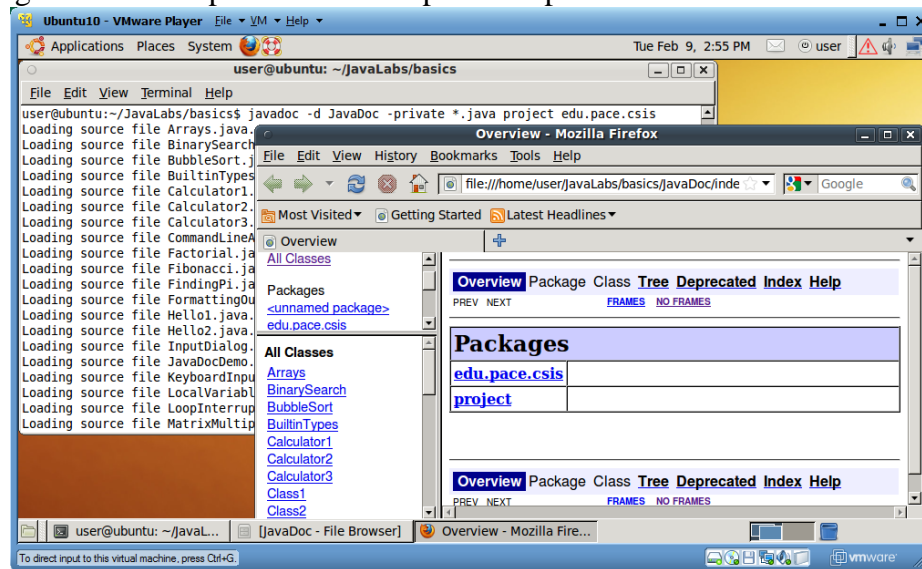
other tags before you actually use them so you know their impact on the generated Java documentation HTML files.

3. Create a directory named “JavaDoc” in the current working directory by typing
mkdir JavaDoc
4. Make sure your working directory is now “~/JavaLabs/basics”. To generate HTML documentation for class *JavaDocDemo*, including class author and version number, type
javadoc -d JavaDoc -private -author -version JavaDocDemo.java
where command-line flag “-d” is used to specify the location of the generated documentation files; flag “-private” is for generating documentation for all classes and methods, no matter they are public or private; flag “-author” is for including author information in the generated documentation; and flag “-version” is for including program version information in the generated documentation. If flag “-d” is not used, it is equivalent to using flag “-d .”, which generates the documentation files in the current working directory.
5. Start a file explorer and use it to open the generated file “~/JavaLabs/basics/JavaDoc/index.html” in a web browser (normally by double-clicking on the file). Now you can review your automatically generated program documentation.
6. To generate HTML documentation for class *JavaDocDemo* without including class author and version number, type
javadoc -d JavaDoc -private JavaDocDemo.java
Repeat step 6 to see the missing of author and version information in the generated documentation.
7. To generate HTML documentation for all Java source code in the current working directory, excluding class author and version number, type
javadoc -d JavaDoc -private *.java project edu.pace.csis
Repeat step 6 to review the generated documentation. In general, the arguments of command “javadoc” can be a list of Java source files and package names currently accessible by command “javac”. A package name in the argument list means that documentation needs be generated for all classes in that package.

The following is the screen capture of my test run for steps 4-6.



The following is a screen capture after I completed step 8.



3.15 Scopes of local variables

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “VariableScope.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
66 public class VariableScope {
67     public static void main(String[] args) {
68         int x1 = 0;
69         for (int i = 0; i < 5; i++) {
70             // i and square are only available in this for loop
71             int square = i*i;
72             x1 += square;
73             System.out.println("i = " + i + ", x1 = " + x1);
74         }
75         // The following line could cause error
76         // System.out.println("i = " + i + " , square = " + square);
77         System.out.println("x1 = " + x1);
78         if (x1 > 15) {
79             int x2 = 2 * x1;
80             System.out.println("x2 = " + x2);
81         }
82         // System.out.println("x2 = " + x2); // Cause error
83         System.out.println("x1 = " + x1);
84     }
85 }
```

Explanation

- a. Parameters of a method are equivalent to initialized local variables declared at the beginning of the method body. Therefore the parameters can be accessed anywhere in the method body.
- b. A local variable is accessible from the point it is declared to the end of the compound statement in which it is declared (including in the bodies of nested compound statements).

- c. A local variable is not accessible outside the compound statement in which it is declared.
 - d. If the loop variable of a *for* loop is declared in the control block of the *for* loop, the loop variable is only accessible inside the body of the *for* loop.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *VariableScope*, type
javac -d ~/classes VariableScope.java
 4. To run class *VariableScope*, type
java VariableScope
 5. Uncomment the line for “System.out.println(“i = ” + i + ” , square = ” + square);” Recompile the class to observe error messages. Comment away this line again.
 6. Uncomment the line for “System.out.println(“x2 = ” + x2);” Recompile the class to observe error message. Comment away this line again.

The following is the screen capture of my test run.

```

user@ubuntu: ~/JavaLabs/basics
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes VariableScope.java
user@ubuntu:~/JavaLabs/basics$ java VariableScope
i = 0, x1 = 0
i = 1, x1 = 1
i = 2, x1 = 5
i = 3, x1 = 14
i = 4, x1 = 30
x1 = 30
x2 = 60
x1 = 30
user@ubuntu:~/JavaLabs/basics$ # Try to access i and square out of the loop
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes VariableScope.java
VariableScope.java:10: cannot find symbol
symbol : variable i
location: class VariableScope
    System.out.println("i = " + i + " , square = " + square); // Cause error
                                ^
VariableScope.java:10: cannot find symbol
symbol : variable square
location: class VariableScope
    System.out.println("i = " + i + " , square = " + square); // Cause error
                                ^
2 errors
user@ubuntu:~/JavaLabs/basics$ # Try to access x2 out of the if statement
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes VariableScope.java
VariableScope.java:16: cannot find symbol
symbol : variable x2
location: class VariableScope
    System.out.println("x2 = " + x2); // Cause error
                                ^
1 error
user@ubuntu:~/JavaLabs/basics$

```

3.16 Formatting output

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “FormattingOutput.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1 import java.util.Random;
2
3 public class FormattingOutput {

```

```

4   public static void main(String[] args) {
5       int[][] a = new int[5][5];
6       double[][] d = new double[5][5];
7       Random r = new Random(); // Create a Random object
8       for (int i = 0; i < 5; i++)
9           for (int j = 0; j < 5; j++) {
10              a[i][j] = r.nextInt(10);
11              d[i][j] = r.nextDouble()*10.0;
12          }
13       System.out.println("Contents of matrix A:");
14       for (int i = 0; i < 5; i++) {
15           for (int j = 0; j < 5; j++)
16               System.out.printf("%6d", a[i][j]);
17           System.out.println();
18       }
19       System.out.println("Contents of matrix D:");
20       for (int i = 0; i < 5; i++) {
21           for (int j = 0; j < 5; j++)
22               System.out.printf("%6.2f", d[i][j]);
23           System.out.println();
24       }
25       double x = d[0][0];
26       System.out.println("Without formatting, x = " + x);
27       System.out.printf("With formatting, x = %4.2f", x);
28   }
29 }

```

Explanation

- Since this class uses a library class “java.util.Random” that belong to a different package than class *FormattingOutput*, we need to import the *Random* class to make it available to class *FormattingOutput*. If we delete the import statement “import java.util.Random;”, the program will still work if we replace all occurrences of “Random” with “java.util.Random”. This shows that importing classes from other packages is not absolutely necessary but it will allow us to avoid qualifying those classes with their complete package paths in the source code.
- A class declaration is like a stamp or stencil. An object is like a particular pattern generated from a stamp or stencil. A class can be used to create (instantiate) any number of objects of the same type as the class. A class declaration does not take much memory space. But each object of that class needs separate memory space allocated for holding the *instance* (non-static) *variables* of the class.
- A class is a kind of data type. A class’s name can be used to *declare reference variables* that can be use to store references (addresses) of objects of that class type. Each reference variable, no matter it is for references of which type, takes a fixed 32 bits of memory space.
- Let *ClassName* be the name of a class. Operator *new* can be used to create an object of type *ClassName*, with an expression like “new *ClassName*()”, which will allocate memory space and create a new object of type *ClassName*, and return a reference (address) of the new object. “*ClassName* x;” will declare that x is a reference variable of type *ClassName*. A reference variable is used to store the reference of an object, while a normal variable, like those declared with primitive data types *int*, *double* and *char*, are used directly to store values of those types.

- e. Statement “`Random r = new Random();`” declares that *r* is a reference variable for type *Random*, instantiates an object of type *Random*, and stores the object’s reference in reference variable *r*. The memory structure for the reference variable *r* and the *Random* object is described in Figure 6 (for contrast, we also included memory structure for “`int x = 2;`” in the same figure):

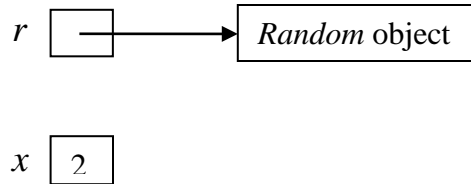


Figure 6 Reference variable vs. primitive variable

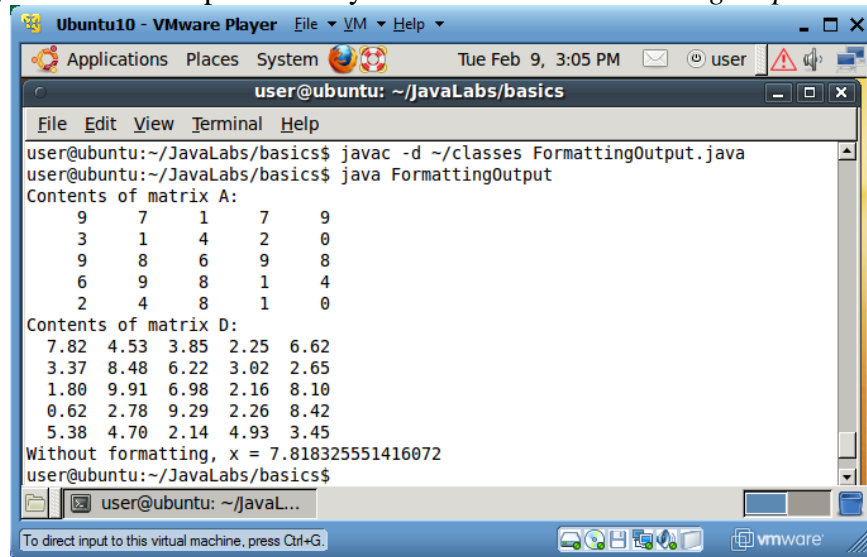
Here an arrow from memory box for *r* to the *Random* object means that the memory box for *r* holds the reference (address) of the *Random* object.

- f. Class *Random* has a (non-static) method *nextInt(int)*, which returns a random integer between 0 and the integer argument’s value minus 1, both sides inclusive. Therefore, “*r.nextInt(10)*” will return random integers between 0 and 9.
- g. Class *Random* has a (non-static) method *nextDouble()*, which returns a random double value between 0.0 (inclusive) and 1.0 (exclusive). Therefore, expression “*r.nextDouble()*10.0*” returns random double values between 0.0 (inclusive) and 10.0 (exclusive).
- h. Method *System.out.printf()* can be used for better formatting data outputs. Method *System.out.printf()* accepts variable number of arguments. The first argument must be a string, called the *formatting string*. The formatting string may contain *formatting substrings* of form like “%6d” or “%6.2f”, which are placeholders for integers and *double* values respectively. String “%6d” indicates that an integer will be printed at this location with a width of 6 characters. If the integer contains less than 6 digits, space characters will be added to the beginning of the string form of the integer to make it a string of 6 characters. If the integer contains more than 6 digits, the integer will be printed with its actual length. Here 6 is just an example width for printing integers, and you can change it to any positive integer. Similarly, string “%6.2f” indicates that a *double* value will be printed at this location with a width of 6 characters, including both the integer part, the decimal point, and the fraction part, and the fraction part will take exactly 2 characters. Therefore, a *double* value will be printed with 3 digits before the decimal point (padding space at the beginning if necessary), and 2 digits after the decimal point (padding 0s at the end if necessary). Here 6 and 2 are just example values, and you can change them to any positive integers that you need. If the *double* value has more fraction digits than allowed to print, the fraction part will be properly rounded, instead of truncated. After the formatting string argument, there must be exactly as many extra arguments to method *printf()* as the number of formatting substrings of forms like “%6d” or “%6.2f” in the formatting string, and the values of these extra arguments must be of the same type as their corresponding formatting substrings relative to the order from left to right (the type of the second argument must match the type of the first formatting substring, the type of the third argument must match the type of the second formatting substring, etc.), and

the values of these arguments will be printed in formats specified by their corresponding formatting substrings.

3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *FormattingOutput*, type
javac -d ~/classes FormattingOutput.java
4. To run class *FormattingOutput*, type
java FormattingOutput

The following is the screen capture for my test run of class *FormattingOutput*.



```
user@ubuntu: ~/JavaLabs/basics
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes FormattingOutput.java
user@ubuntu:~/JavaLabs/basics$ java FormattingOutput
Contents of matrix A:
 9  7  1  7  9
 3  1  4  2  0
 9  8  6  9  8
 6  9  8  1  4
 2  4  8  1  0
Contents of matrix D:
7.82 4.53 3.85 2.25 6.62
3.37 8.48 6.22 3.02 2.65
1.80 9.91 6.98 2.16 8.10
0.62 2.78 9.29 2.26 8.42
5.38 4.70 2.14 4.93 3.45
Without formatting, x = 7.818325551416072
user@ubuntu:~/JavaLabs/basics$
```

3.17 Timing the evaluation of π

Constant π is very important in mathematics. Its value can be approximated with the equation

$$\pi = 4 \times \left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right]$$

in which the terms in the square brackets should follow the pattern to go on forever. In reality we only need approximate values of π , say the value of π that is precise up to 6 digits in the fraction part. If we need more precise approximation of π , we can add more terms in the square brackets. When we increase the number of terms in the square brackets, the approximate value of π will approach its accurate version. The impact of the terms on the value of π will diminish gradually as we scan through the terms of $\pi/4$ from left to right.

In this sub-section we write a program to find the approximate value of π with a simple loop, and we report the number of milliseconds used for the program execution.

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “FindingPi.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 public class FindingPi {
2     public static void main(String[] args) {
```

```

3    long startTime = System.currentTimeMillis(); // record start time
4    double current = 1.0; // current approximate value of Pi/4
5    double last = 0.0;    // last approximate value of Pi/4
6    double sign = 1.0;    // last term's sign
7    double denominator = 1.0; // last term's denominator
8    double tolerance = 0.000000001; // allowed tolerance
9    while (Math.abs(current - last) > tolerance) {
10       sign *= -1.0;        // negate the sign for the sign of the next term
11       denominator += 2.0; // get the denominator for the next term
12       last = current;      // current approximate becomes the last one
13       // next approximate becomes the current one
14       current += sign/denominator;
15    }
16    System.out.printf("Pi = %9.7f\n", 4.0*current);
17    long stopTime = System.currentTimeMillis(); // record stop time
18    System.out.println("This program has spent " +
19                       (stopTime-startTime) + " msecs running");
20 }
21 }

```

Explanation

- Java has another primitive data type *long* for representing large integers. While an *int* value only takes 32 bits for its storage, a *long* value takes 64 bits for its storage.
- Java static method *System.currentTimeMillis()* returns the number of milliseconds that has passed since zero clock of January 1, 1970. Since this is a large number, we need to use *long* type variables to hold such counts. For finding out how long a program runs, we can get the current time at the beginning of the method *main()* and at the end of method *main()*, and then report the difference between these two times.
- We use a loop to approximate the value of π . Before the loop is executed, variable *current* holds the sum of the first one term of $\pi/4$, or 1.0; variable *last* holds the sum of the first zero terms of $\pi/4$, or 0.0; variable *sign* holds the sign of the last term in sum *current*, which is +1; and variable *denominator* holds the value of the denominator of the last term of sum *current*, which is 1.0 (the first term is 1.0/1.0). During each iteration, we add the following term to the sum *current*. The sign of the following term is the negation of that of the last term. The denominator of the following term is that of the last term plus 2.0. We let *last* hold the value of *current*, and update *current* by adding the following term to it. Therefore, after the first loop iteration, *current* holds the sum of the first two terms of $\pi/4$; *last* holds the sum of the first one term of $\pi/4$, *sign* is the sign of the second term, which is -1; and *denominator* holds the denominator for the second term of $\pi/4$. The loop will repeat this process until the difference between *current* and *last* is smaller than a preset tolerance level. A smaller tolerance will significantly increase the program running time.
- Many floating-point (real) numbers need infinite number of digits to hold their values. For example, $1.0/3.0 = 0.3333\dots$ in which the fraction part has infinite number of digit 3. But a Java double value has only 64 bits to store such values. Therefore, Java double values are only approximations of real world floating-point numbers. As a consequence, when we need to test whether a double variable holds a specific value, we should not use equal operator `==` or not-equal operator `!=`, but to use the `<` or `>` operators to check whether the variable's value and the

specific value are close enough. For example, suppose we use a loop to modify the value of a double variable until the variable has value 3.0, the following loop may lead to infinite loop if the loop variable only takes on values 2.999999999999 or 3.000000000001, both are reasonable approximations of 3.0, instead of the exact value 3.0.

```
double d = 0.0;
while (d != 3.0) {
    // update value of d
}
```

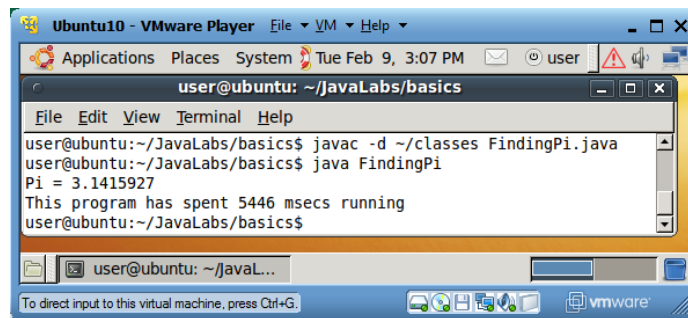
A better version for the above loop is

```
double d = 0.0;
double tolerance = 0.0000001;
while (Math.abs(3.0 - d) > tolerance) {
    // update value of d
}
```

where the value of *tolerance* can be adjusted based on the need of the program's application.

3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *FindingPi*, type
javac -d ~/classes FindingPi.java
4. To run class *FindingPi*, type
java FindingPi

The following is the screen capture of my test run of class *FindingPi*.



3.18 Matrix multiplication

This sub-section uses matrix multiplication to practice our skills on nested loops, arrays, formatted output, and methods.

Let a , b and c be all $n \times n$ matrices with both row and column indices running from 0 through $n-1$. Assume matrices a and b have both been initialized with values. Let $c = a \times b$. For all combinations of $0 \leq i < n$ and $0 \leq j < n$, $c[i][j]$ can be calculated with the following formula:

$$c[i][j] = \sum_{k=0}^{n-1} a[i][k] \times b[k][j].$$

Informally, $c[i][j] = a[i][0] \times b[0][j] + a[i][1] \times b[1][j] + \dots + a[i][n-1] \times b[n-1][j]$.

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “MatrixMultiplication.java” in directory “~/JavaLabs/basics” and the file has contents below:

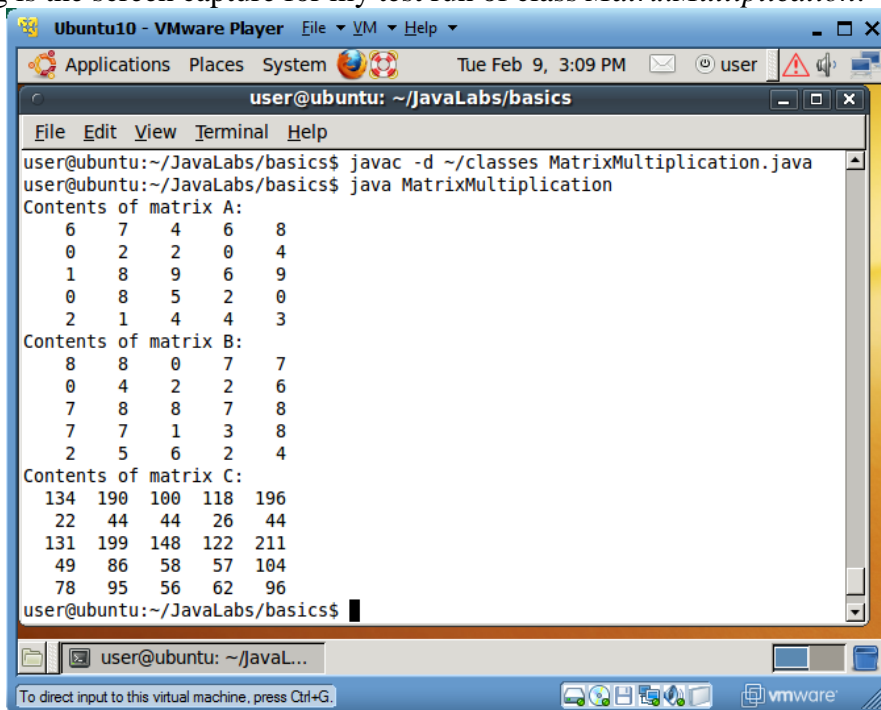
```
1 import java.util.Random;
2
3 public class MatrixMultiplication {
4     public static void main(String[] args) {
5         int[][] a = new int[5][5];
6         int[][] b = new int[5][5];
7         int[][] c = new int[5][5];
8         Random r = new Random(); // Create a Random object
9         // r.setSeed(1);
10        // Initialize arrays a and b with random integers, 0..9
11        for (int i = 0; i < 5; i++)
12            for (int j = 0; j < 5; j++) {
13                a[i][j] = r.nextInt(10);
14                b[i][j] = r.nextInt(10);
15            }
16        // Calculate C = A x B
17        c = multiply(a, b);
18        // Print arrays
19        printMatrix(a, "Contents of matrix A:", "%5d");
20        printMatrix(b, "Contents of matrix B:", "%5d");
21        printMatrix(c, "Contents of matrix C:", "%5d");
22    }
23
24    static void printMatrix(int[][] x, String title, String format) {
25        System.out.println(title);
26        for (int i = 0; i < x.length; i++) {
27            for (int j = 0; j < x[0].length; j++)
28                System.out.printf(format, x[i][j]);
29            System.out.println();
30        }
31    }
32
33    // Calculate C = A x B
34    static int[][] multiply(int[][] a, int[][] b) {
35        // c[][] has the same size as a[][]
36        int[][] c = new int[a.length][a[0].length];
37        for (int i = 0; i < c.length; i++)
38            for (int j = 0; j < c[0].length; j++) {
39                int sum = 0;
40                for (int k = 0; k < c[0].length; k++)
41                    sum += a[i][k] * b[k][j];
42                c[i][j] = sum;
43            }
44        return c;
45    }
46 }
```

Explanation

- a. Class *java.util.Random* has a method *nextInt(int)* that can return a random integer between 0 (inclusive) and the argument integer (exclusive).

- b. Normally, every time you run a program that uses a *Random* object, the *Random* object will generate different series of random values. But there are times when we need to repeat the same sequence of random numbers so we can debug the program. Class *java.util.Random* has a method *setSeed(int)* that can be used to initialize the *Random* object with any integer, which is called a *seed for the random generator*. As long as you use the same random seed integer as the argument for method *setSeed(int)*, the *Random* object will generate the same sequence of random numbers.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *MatrixMultiplication*, type
javac -d ~/classes MatrixMultiplication.java
4. To run class *MatrixMultiplication*, type
java MatrixMultiplication
5. Uncomment the “r.setSeed(1);” statement and redo steps 3 and 4.
6. Change the constant 1 in the “r.setSeed(1);” statement to other integers and redo steps 3 and 4.

The following is the screen capture for my test run of class *MatrixMultiplication*.



```
user@ubuntu: ~/JavaLabs/basics
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes MatrixMultiplication.java
user@ubuntu:~/JavaLabs/basics$ java MatrixMultiplication
Contents of matrix A:
6 7 4 6 8
0 2 2 0 4
1 8 9 6 9
0 8 5 2 0
2 1 4 4 3
Contents of matrix B:
8 8 0 7 7
0 4 2 2 6
7 8 8 7 8
7 7 1 3 8
2 5 6 2 4
Contents of matrix C:
134 190 100 118 196
22 44 44 26 44
131 199 148 122 211
49 86 58 57 104
78 95 56 62 96
user@ubuntu:~/JavaLabs/basics$
```

3.19 Interactive command-line data input

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “KeyboardInput.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 import java.util.*;
2
3 public class KeyboardInput {
```



```

4   public static void main(String[] args) {
5       Scanner keyboard = new Scanner(System.in);
6       System.out.print("Please enter an integer: ");
7       int n1 = keyboard.nextInt();
8       System.out.println("You have just entered integer " + n1);
9       System.out.print("Please enter a floating-point number: ");
10      double d = keyboard.nextDouble();
11      System.out.println("You have just entered floating-point number "
12                          + d);
13      System.out.print("Please enter a string containing no spaces: ");
14      String s = keyboard.next();
15      System.out.println("You have just entered a string \"" + s + "\"");
16      s = keyboard.nextLine(); // consume the "\n" on the current line
17      System.out.print("Please enter a few strings: ");
18      s = keyboard.nextLine();
19      System.out.println("You have just entered line \"" + s + "\"");
20      System.out.print("Please enter two integers on the same line: ");
21      n1 = keyboard.nextInt();
22      int n2 = keyboard.nextInt();
23      System.out.println("You have just entered integers " + n1 + " and "
24                          + n2);
25  }
26 }

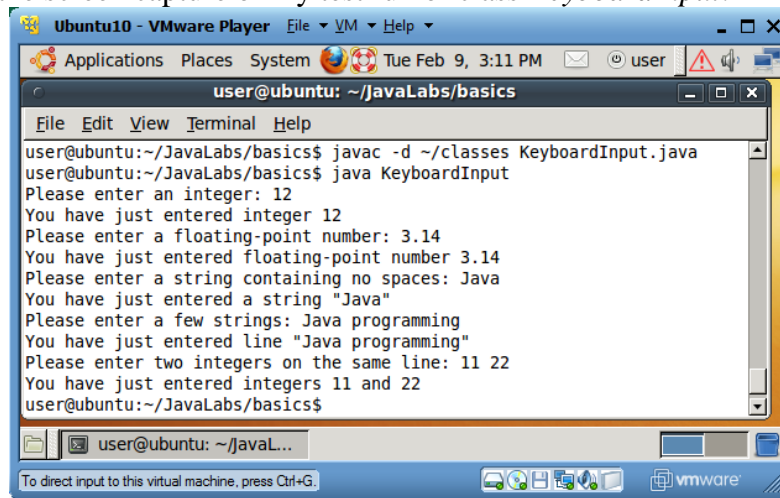
```

Explanation

- a. Java *System.in* is an object representing the primitive keyboard *input data stream* working at character level, which is not convenient for getting *int*, *double*, or *String* data from keyboard. JDK 5 introduced a new helper class *java.util.Scanner* to make interactive keyboard data input easier.
- b. Statement “*Scanner keyboard = new Scanner(System.in);*” instantiates a *Scanner* object that will get input from *System.in* or keyboard, and assigns the reference of this object in *Scanner* reference variable *keyboard*.
- c. When a program is ready to accept some data input from a program user, a prompt string should be printed first to remind the user what kind of data should be typed next.
- d. Class *Scanner* has a method *nextInt()* for returning the next integer that the user types in the *Command Prompt* window running this program.
- e. Class *Scanner* has a method *nextDouble()* for returning the next floating-point value that the user types in the *Command Prompt* window running this program.
- f. Class *Scanner* has a method *next()* for returning the next string (containing no space in the string) that the user types in the *Command Prompt* window running this program.
- g. Class *Scanner* has a method *nextLine()* for returning the strings from the current keyboard cursor position up to the end of the current line that the user types in the *Command Prompt* window running this program. The *new line* character at the end of the line is consumed (used up) by the *nextLine()* method, but not returned as part of the return string of *nextLine()*.
- h. The first “*s = keyboard.nextLine();*” statement in this class is called when the cursor is still on the previous line and the *new line* character for the line has not been used up (read) yet. This statement is used to consume up the previous line from the current keyboard cursor position up to the end of that line, including the *new line* character, so that the next call to *nextLine()* will return what the user types on the next line.

- i. Except for *nextLine()*, other methods of *Scanner* will return the next value starting from the current keyboard cursor position, using white space characters (which are *space*, *Tab* and *new line* characters) as delimiters. Therefore you can type several input values on the same line (separating each other with space). The input strings cannot be accessed by the *Scanner* object until the *Enter* key is typed.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *KeyboardInput*, type
javac -d ~/classes KeyboardInput.java
4. To run class *KeyboardInput*, type
java KeyboardInput

The following is the screen capture of my test run of class *KeyboardInput*.



```
user@ubuntu: ~/JavaLabs/basics
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes KeyboardInput.java
user@ubuntu:~/JavaLabs/basics$ java KeyboardInput
Please enter an integer: 12
You have just entered integer 12
Please enter a floating-point number: 3.14
You have just entered floating-point number 3.14
Please enter a string containing no spaces: Java
You have just entered a string "Java"
Please enter a few strings: Java programming
You have just entered line "Java programming"
Please enter two integers on the same line: 11 22
You have just entered integers 11 and 22
user@ubuntu:~/JavaLabs/basics$
```

3.20 Window-based data input/output

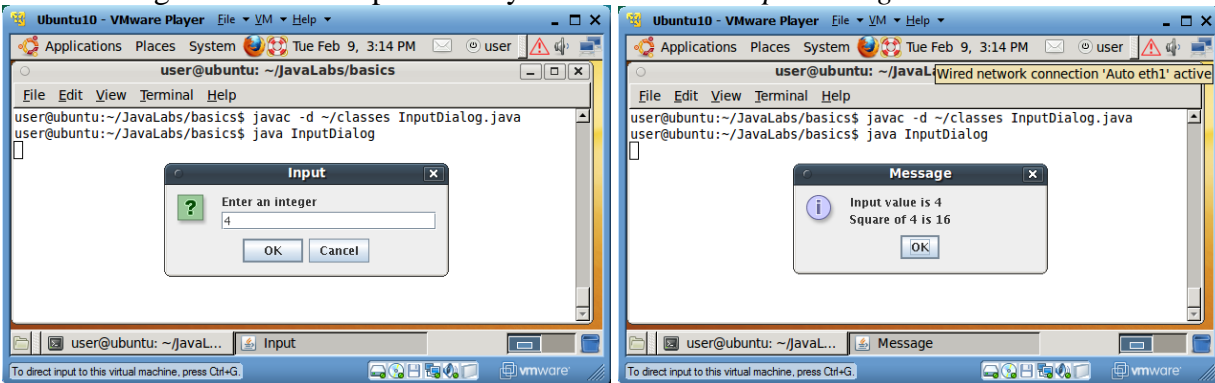
1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “InputDialog.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 import javax.swing.JOptionPane;
2
3 public class InputDialog {
4     public static void main(String[] args) {
5         String input =
6             JOptionPane.showInputDialog(null, "Enter an integer");
7         int n = Integer.parseInt(input);
8         String message = "Input value is " + n + "\n"
9             + "Square of " + n + " is " + n*n;
10        JOptionPane.showMessageDialog(null, message);
11        System.exit(0);
12    }
13 }
```

Explanation:

- a. Java library class *javax.swing.JOptionPane* supports two static methods for getting user inputs and posting messages to users through graphic windows.
 - b. For getting user input, we can use method *JOptionPane.showInputDialog(null, prompt)*, where *prompt* is a string explaining what input is expected, and the return value of the method is the string the user typed in the input window popped up by this method.
 - c. For posting a message to the user, we can use method *JOptionPane.showMessageDialog(null, message)*, where *message* is the string for being displayed to the user in a new message window.
 - d. A Java program with graphic user interfaces normally contains multiple threads of computing (like having multiple CPUs running different pieces of code at the same time, some are waiting for user input, and some are doing computing at the background), and the completion of the execution of method *main()* may not terminate all the threads thus may not terminate the program's execution. This is also a platform dependent issue. To make sure that our programs always terminate properly, Java programs with graphic user interfaces should terminate program execution explicitly with the *System.exit(0)* statement.
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *InputDialog*, type
javac -d ~/classes InputDialog.java
 4. To run class *InputDialog*, type
java InputDialog

The following is the screen capture of my test run of class *InputDialog*.



3.21 Text file input/output

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “TextFileIO.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1 import java.io.*;
2 import java.util.Random;
3
4 public class TextFileIO {
5     public static void main(String[] args) throws IOException {
6         File file = new File("temp.txt");

```

```

7      if (!file.exists()) {
8          // if file temp.txt does not exist, create an output stream to it
9          PrintWriter output = new PrintWriter(new FileWriter(file));
10
11         // generate 100 random integers
12         Random r = new Random();
13         for (int i = 0; i < 10; i++) {
14             for (int j = 0; j < 10; j++)
15                 output.printf("%4d", r.nextInt(100));
16             output.println();
17         }
18
19         // close the output stream
20         output.close();
21     }
22
23     // open an input stream from file temp.txt
24     BufferedReader input =
25         new BufferedReader(new FileReader("temp.txt"));
26
27     int total = 0;
28     String line = null;
29     while ((line = input.readLine()) != null) {
30         String[] token = line.trim().split("\\s+");
31         for (int i = 0; i < token.length; i++)
32             total += Integer.parseInt(token[i]);
33     }
34     System.out.println("Total is " + total);
35
36     // close input stream
37     input.close();
38 }
39 }

```

Explanation

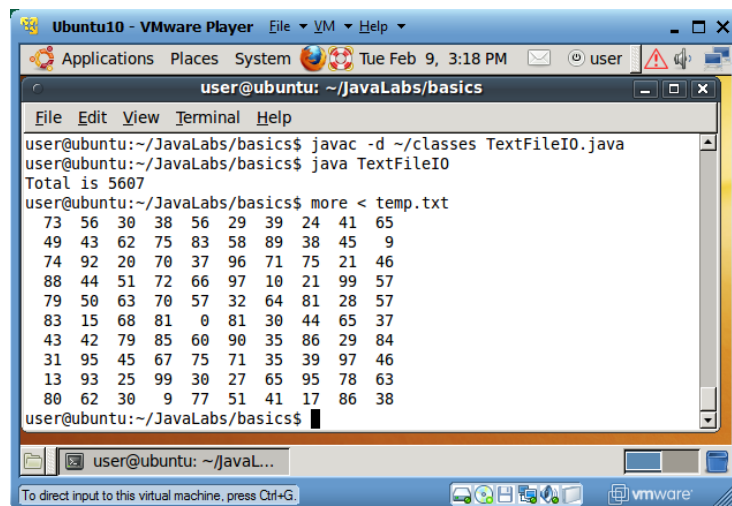
- a. Java programs use files to store persistent data. Most Java classes supporting file input/output are in package *java.io*. Since we need to import multiple classes from package *java.io*, we use “import java.io.*;” to import all classes in this package. Importing classes do not take runtime memory space. If a class is imported, we can use the class’s name in the current class source file without qualifying the class name with its complete package path. Therefore class importing is only for more concise source code.
- b. Most file input/output operations may throw *IOException*, a kind of pre-defined Java exception for file or network input/output. We need to enclose these operations in try-catch blocks before compiling the source code containing these file operations. A lazier approach is to add a “throws *IOException*” clause at the end of the containing method’s signature. It declares that *IOException* may be thrown during the execution of its body, and if it does happen, the current method will terminate its execution, and the *IOException* will be thrown back to the current method’s invoker. Since our method is *main()*, the program execution will terminate if *IOException* is thrown, with the message of the *IOException* printed in the current *Command Prompt* window. We use this lazier approach here so the main logic of the method body can be easier to read.

- c. Java has a class *java.io.File* that can instantiate objects representing all meta data (attributes, not contents) about a file or directory. Statement “File file = new File(“temp.txt”);” declares that variable *file* is a reference variable for class *File*, instantiates a *File* object for a file named “temp.txt”, and assigns the reference of the *File* object to the reference variable *file*.
- d. Class *java.io.File* has a Boolean method *exists()* that returns *true* if the file exists in the file system, or *false* otherwise. This program checks whether the file named “temp.txt” exists in the current working directory. If it is not in existence, then it will be created with 100 random integers between 0 and 99, ten integers per row.
- e. Both file input and output use a *file cursor*. When a file is opened or created, its file cursor is at the first character position of the file. Each file read or write operation will start from the file cursor position, and advance the file cursor forward or to the next line, as the *Command Prompt* window’s cursor does when we print to or read data from the *Command Prompt* window.
- f. Class *java.io.FileWriter* can support basic text write functions to the text file specified as the argument of its constructor (a *constructor* of a class has syntax of the class name followed by a parameter list; it is executed when a new object of that class type is instantiated). Class *java.io.PrintWriter* can wrap up a *FileWriter* object to provide higher-level operations like *print()*, *println()*, and *printf()*. Declaration “PrintWriter output = new PrintWriter(new FileWriter(file));” declares that *output* is a reference variable of type *PrintWriter*, creates a new *FileWriter* object connected to the *File* object specified by variable *file*, wraps up the *FileWriter* object inside a *PrintWriter* object, and then assigns the reference of the *PrintWriter* object to reference variable *output*. We also say that this declaration *opens* file “temp.txt” for text output. You can print data to variable *output* as you print data to *System.out*. The difference is that the former prints to the file “temp.txt”, while the latter prints to the computer display.
- g. A *FileWriter* or *PrintWriter* object takes significant amount of operating system resources for data buffering in the main memory. An operating system can only open a limited number of files for input or output a time. Therefore it is a good practice to close a file when its I/O (input/output) is completed. Class *FileWriter* has a method *close()* for this purpose. Statement “output.close();” releases all operating system resources allocated for file “temp.txt”. By the same token, statement “input.close();” also releases all operating system resources allocated for file “temp.txt” after we finish reading its contents.
- h. Java class *java.io.FileReader* opens a text file for low-level reading operations. Java class *java.io.BufferedReader* can wrap up a *FileReader* object to support higher-level buffered operations like *readLine()*. Declaration “BufferedReader input = new BufferedReader(new FileReader(“temp.txt”));” declares that *input* is a reference variable for *BufferedReader* objects, instantiates a *FileReader* object for text file “temp.txt”, wraps the *FileReader* object inside a *BufferedReader* object, and then assigns the reference of the *BufferedReader* object in reference variable *input*.
- i. Class *BufferedReader* has a method *readLine()*, which returns the next line in its underlying file, starting from the current position of the file cursor to the end of the current line. If the file cursor has come to the end of file, method *readLine()* will return *null*. Boolean expression “(line = input.readLine()) != null” first evaluates the assignment sub-expression to the left of the *!=* operator. The value of an assignment expression is the value assigned. If *input.readLine()* returns *null*,

null will be assigned in variable *line*, and *null* is also the value of the assignment expression “(line = input.readLine())”. This will lead to value *false* for the Boolean expression of the while loop, thus terminating the while loop. To put it simple, the while loop will read one line from file “temp.txt” during its each iteration. If there is no line to be read, the while loop terminates.

- j. Type *String* is actually class *java.lang.String*. Since Java automatically imports all classes in package “java.lang”, we never need to explicitly import “java.lang.*”. Class *String* has a method *trim()*, which strips off any leading and trailing white space characters of the string that invokes method *trim()*, and then returns the resulting string. Class *String* also has a method *split("\\s+")* that can use white space characters as delimiters to split the string that invokes this *split("\\s+")* method into a list of sub-strings containing no white space characters, and return the sub-strings in an array of *String*. This process is also called *parsing* the invoking string into a list a *tokens* separated by white space characters. The size of the returned array is equal to the number of sub-strings found.
 - k. In Java, method invocations can be chained if the return value of the previous method invocation is of type that implements the current method invocation. Declaration “String[] token = line.trim().split("\\s+");” declares that *token* is a reference variable for *array of Strings*; replaces “line.trim()” with the return value of method *trim()*, which is a new string obtained by removing leading and trailing white space characters in the value of *line*; invokes method *split("\\s+")* against the resulting string to split it into an array of sub-strings containing no white space characters; and finally assigns the reference for this array into reference variable *token*.
- 3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *TextIO*, type
javac -d ~/classes TextIO.java
 - 4. To run class *TextIO*, type
java TextIO

The following is the screen capture of my test run of class *TextIO*. Remember that the new text file “temp.txt” will be generated only if it is already in existence. You can force its generation by first deleting it.



```
user@ubuntu: ~/JavaLabs/basics
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes TextFileIO.java
user@ubuntu:~/JavaLabs/basics$ java TextFileIO
Total is 5607
user@ubuntu:~/JavaLabs/basics$ more < temp.txt
73 56 30 38 56 29 39 24 41 65
49 43 62 75 83 58 89 38 45 9
74 92 20 70 37 96 71 75 21 46
88 44 51 72 66 97 10 21 99 57
79 50 63 70 57 32 64 81 28 57
83 15 68 81 0 81 30 44 65 37
43 42 79 85 60 90 35 86 29 84
31 95 45 67 75 71 35 39 97 46
13 93 25 99 30 27 65 95 78 63
80 62 30 9 77 51 41 17 86 38
user@ubuntu:~/JavaLabs/basics$
```

3.22 Recursive vs. iterative methods: factorial

A method can call (invoke) another method. As a special case, a method can call itself. If a method can call itself, the method is called a *recursive method*. On the other hand, if a method doesn't contain calls to itself, directly or indirectly (like method *a()* calls method *b()* and method *b()* calls method *a()*), then it is called an *iterative method* because its significant running time must be spent in some loops. In theory, recursion and loops are equally powerful. Any problem that can be solved with loops only can also be solved with recursive methods only, and any problem that can be solved with recursive methods only can also be solved with loops. Many problems are defined recursively in nature and it is much easier to design recursive solutions for them. But in general recursive solutions take much more memory space and running time than their loop-based counter parts. Therefore, if possible, we should avoid using recursive solutions.

When a method invokes another method, the Java runtime system will allocate a chunk of main memory, called an *activation record*, for storing the values of the invoked method's parameters and other local variables, as well as for storing the return address for the machine instruction of the invoking method that is to be executed when the method invocation is over. When the method invocation is over, this activation record will be returned to the Java runtime system for memory space reuse, the return value will be passed to a fixed location in the invoking method's activation record, and the return address saved in the invoked method's activation record will be used to resume the execution of the invoking method. Figure 7 shows how method *main()* invokes method *m1()*, and method *m1()* invokes method *m2()*, and then method *m1()* resumes execution upon the completion of method *m2()*'s execution, and method *main()* resumes execution upon the completion of method *m1()*'s execution. The lower half of Figure 7 shows how an activation record is created for each method invocation, and how an activation record is deleted when its corresponding method completes its execution.

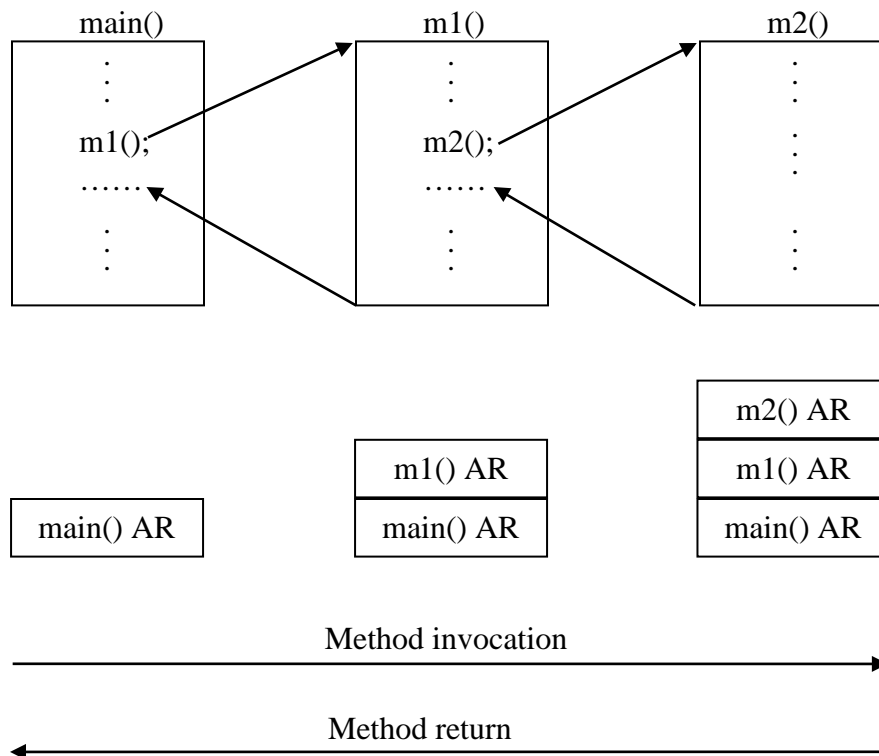


Figure 7 Chained method invocations and their activation records

Therefore, even if a method calls itself, different invocations to the same method will use different activation records to store its parameter values, local variable values, and return addresses, and these values for different invocations will not be mixed up.

As an example of recursive methods, this sub-section introduces both recursive and iterative solutions to the evaluation of the factorial of a non-negative integer. Given a non-negative integer n , its factorial $n!$ can be defined recursively as below:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

Therefore,

$$\begin{aligned} 3! &= 3 \times 2! \\ &= 3 \times 2 \times 1! \\ &= 3 \times 2 \times 1 \times 0! \\ &= 3 \times 2 \times 1 \times 1 \\ &= 6 \end{aligned}$$

But the factorial of n can also be defined as

$$n! = 1 \times 2 \times \cdots \times n$$

which obviously leads to a solution based on a simple loop. This definition tells us that $3! = 1 \times 2 \times 3 = 6$.

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “Factorial.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 public class Factorial {
2     public static void main(String[] args) {
3         int n = 0; // input integer
4         try {
5             n = Integer.parseInt(args[0]);
6             if (n < 0)
7                 throw new Exception("Input value is negative");
8             if (n > 20)
9                 throw new Exception("Input value is too large");
10        }
11        catch (Exception e) {
12            System.out.println(e);
13            System.out.println(
14                "Usage: java Factorial [non-negative-integer]");
15            System.exit(-1);
16        }
17        long answer = 0;
18        answer = recursiveFactorial(n);
19        System.out.println("Recursive Factorial gets answer " + answer);
20        answer = iterativeFactorial(n);
21        System.out.println("Iterative Factorial gets answer " + answer);
22    }
23
24    static long recursiveFactorial(long n) {
25        if (n == 0)
26            return 1; // 0! = 1
27        else
28            return n*recursiveFactorial(n-1);
29    }
30
31    static long iterativeFactorial(int n) {
32        long answer = 1;
33        for (int i = 1; i <= n; i++)
34            answer *= i;
35        return answer;
36    }
37 }
```

Explanation

- a. Factorial of an integer grows very fast, so we use data type *long* to declare variables for holding factorial results. While an *int* variable takes 32 bits of memory for its value storage, a *long* variable takes 64 bits of memory for its value storage. But even a variable of type *long* can only hold the factorial of up to 20. That is why when you run this program, you need to make sure that the input integer is between 0 and 20.
- b. If we detect that something is wrong, we can *throw* a new *Exception* object, and we can pass a message to the constructor of class *Exception* to record the reason for the exception. Expression “new *Exception*(“reason for this exception”)” instantiates a new *Exception* object initialized with the reason for this exception, and Java reserved word “throw” is used to throw out this exception object.

- c. In our try-catch block, there are four reasons that can lead to the throw of an exception object. If there is no command-line argument, `args[0]` will throw a `java.lan.ArrayIndexOutOfBoundsException` object. If the command-line argument is not representing an integer, `Integer.parseInt(args[0])` will throw a `java.lang.NumberFormatException` object. If the input integer is less than zero or larger than 20, we throw a new `Exception` object respectively. These exception objects will all be caught by the catch clause, and the parameter *e* of the catch clause will be initialized to the reference of the thrown exception object by the Java runtime system. In the catch clause, we print out the exception object, which will actually print out the reason for the exception; print out a usage message for how to run the program; and then terminate the program's execution.
- d. The following Figure 8 shows how the activation records work when we call `recursiveFactorial(3)`. Each invocation to this method creates a new activation record for holding its local variables, return value and return address (return address is omitted here). The new activation record will always be put on top of the existing ones. The CPU will always act on the top activation record at any time, and the activation records below the top one are temporarily hidden away. The life cycle of activation records actually implements a popular data structure called *stack*, for which the data are organized so the *first in* (stored into the stack) will be the *last out* (removed from the stack). In this case, the first activation record created will be the last one to be destroyed (recycled).

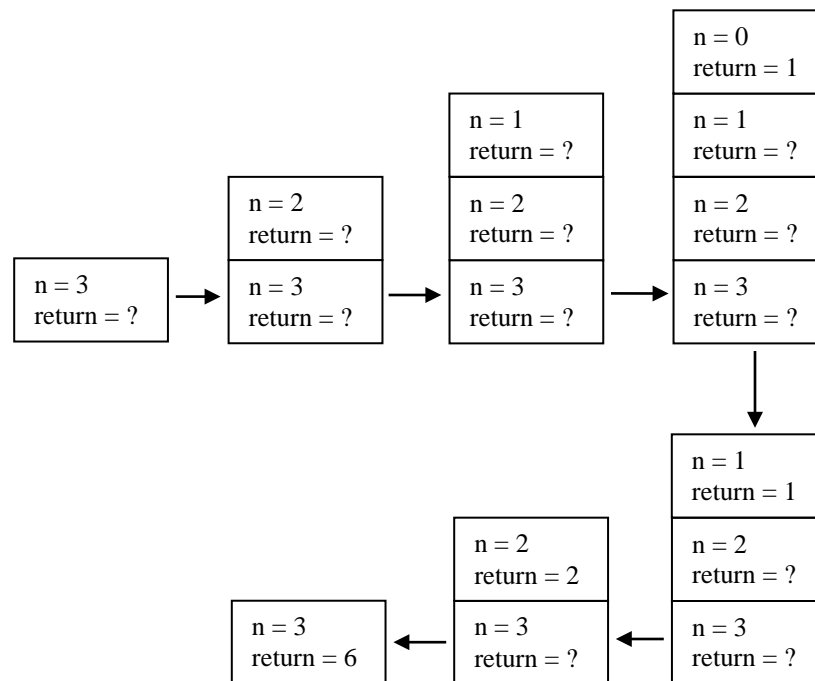


Figure 8 Evolution of activation records for recursiveFactorial(3)

- e. Since data type *long* can only hold around 20 digits, we cannot see the running time difference between the recursive and iterative solutions. But method invocation incurs penalty in terms of both memory space for the activation records and running time. Therefore the iterative version is more efficient. The difference in running time will be more significant in the *Fibonacci* number example in the following sub-section since the recursive solution for *Fibonacci*

number evaluation introduces many redundant method calls (calls to the same method with the same arguments repeatedly).

3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *Factorial*, type

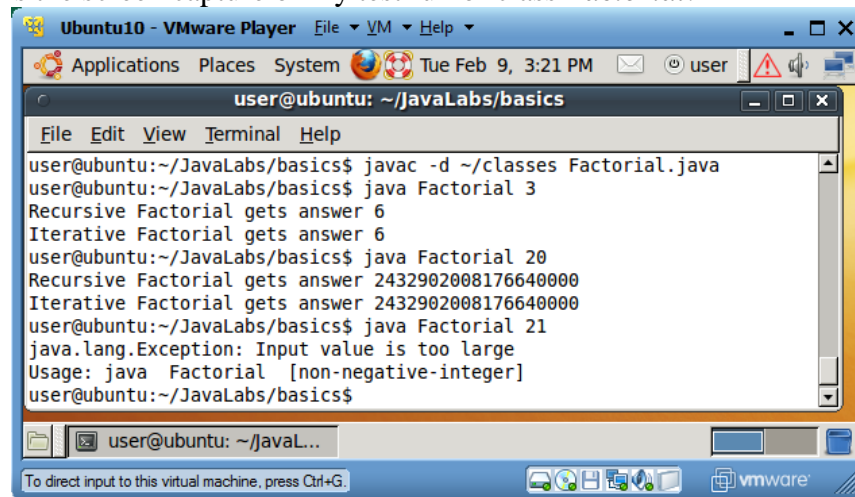
```
javac -d ~/classes Factorial.java
```

4. To run class *Factorial*, type

```
java Factorial 3
java Factorial // error
java Factorial a // error
java Factorial -2 // error
java Factorial 21 // error
```

You can replace the above 3 with any integer between 0 and 20.

The following is the screen capture of my test run of class *Factorial*.



```
Ubuntu10 - VMware Player  File  VM  Help
Applications  Places  System  Tue Feb 9, 3:21 PM  user
user@ubuntu: ~/JavaLabs/basics
File  Edit  View  Terminal  Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes Factorial.java
user@ubuntu:~/JavaLabs/basics$ java Factorial 3
Recursive Factorial gets answer 6
Iterative Factorial gets answer 6
user@ubuntu:~/JavaLabs/basics$ java Factorial 20
Recursive Factorial gets answer 2432902008176640000
Iterative Factorial gets answer 2432902008176640000
user@ubuntu:~/JavaLabs/basics$ java Factorial 21
java.lang.Exception: Input value is too large
Usage: java Factorial [non-negative-integer]
user@ubuntu:~/JavaLabs/basics$
```

3.23 Recursive vs. iterative methods: Fibonacci numbers

This sub-section starts with a story on *Fibonacci* numbers, and then shows how to evaluate *Fibonacci* numbers with both recursive and iterative approaches. The story begins in the early 13th century, when the great Italian mathematician *Fibonacci* posed the following simple problem. A man puts a pair of baby rabbits into an enclosed garden. Assuming that each pair of rabbits in the garden bears a new pair every month, which from the second month on itself becomes productive, how many pairs of rabbits will there be in the garden after one year? Like most mathematics problems, you are supposed to ignore such realistic happenings as death, escape, impotence, or whatever. It is not hard to see that the number of pairs of rabbits in the garden in each month is given by the numbers in the sequence 1, 1, 2, 3, 5, 8, 13, etc. This sequence of numbers is called the *Fibonacci* sequence. The general rule that produces it is that each number after the second one is equal to the sum of the two previous numbers. (So $1+1 = 2$, $1+2 = 3$, $2+3 = 5$, etc.) This corresponds to the fact that each month, the new rabbit births consists of one pair to each of the newly adult pairs plus one pair for each of the earlier adult pairs. Once you have the sequence, you can simply read off that after one year there will be 377 pairs.

Most popular expositions of mathematics mention the *Fibonacci* sequence, usually observing that the *Fibonacci* numbers arise frequently in nature. For example, if you count the number of petals in various flowers you will find that the answer is often a *Fibonacci* number (much more frequently than you would get by chance). For instance, an iris has 3 petals, a primrose 5, a delphinium 8, ragwort 13, an aster 21, a daisy 34, and michaelmas daisies 55 or 89 petals – all Fibonacci numbers.

Fibonacci numbers can be naturally defined recursively as

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2) + fib(n-1) & n > 1 \end{cases}$$

Now we develop a Java class to evaluate *Fibonacci* numbers for any value of index n .

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “Fibonacci.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1 public class Fibonacci {
2     public static void main(String[] args) {
3         int n = 0; // input integer
4         try {
5             n = Integer.parseInt(args[0]);
6             if (n < 0)
7                 throw new Exception("Input value is negative");
8             if (n > 50)
9                 throw new Exception("Input value is too large");
10        }
11        catch (Exception e) {
12            System.out.println(e);
13            System.out.println(
14                "Usage: java Fibonacci [non-negative-integer]");
15            System.exit(-1);
16        }
17        long answer = 0;
18        long startTime = System.currentTimeMillis(); // record start time
19        answer = recursiveFibonacci(n);
20        long stopTime = System.currentTimeMillis(); // record stop time
21        System.out.println("Recursive Fibonacci gets answer " + answer
22            + " in " + (stopTime - startTime)
23            + " milliseconds");
24        startTime = System.currentTimeMillis(); // record start time
25        answer = iterativeFibonacci(n);
26        stopTime = System.currentTimeMillis(); // record stop time
27        System.out.println("Iterative Fibonacci gets answer " + answer
28            + " in " + (stopTime - startTime)
29            + " milliseconds");
30    }
31
32    static long recursiveFibonacci(long n) {
33        if (n == 0 || n == 1)
34            return n; // fib(0) = 0, fib(1) = 1

```

```

35     else
36         return recursiveFibonacci(n-2) + recursiveFibonacci(n-1);
37     }
38
39     static long iterativeFibonacci(int n) {
40         if (n == 0 || n == 1)
41             return n; // fib(0) = 0, fib(1) = 1
42         long previous = 0; // fib(0)
43         long current = 1;  // fib(1)
44         long next = 0;      // fib(2), fib(3), ... fib(n)
45         for (int i = 2; i <= n; i++) {
46             // evaluate next = fib(i), and make previous = old-current,
47             // current = next
48             next = previous + current;
49             previous = current;
50             current = next;
51         }
52         return next;
53     }
54 }

```

Explanation

- We limit the value of the input integer to be no more than 50. Otherwise the recursive version of the solution takes too long time to complete.
- The iterative version is much more complex than the recursive version. If the input value n is 0 or 1, the answer is returned right away. Otherwise the loop will sequentially find $\text{fib}(2)$, $\text{fib}(3)$, ..., $\text{fib}(n)$. At the beginning of iteration i of the loop body, variables *previous* and *current* hold values of the last two *Fibonacci* values $\text{fib}(i-2)$ and $\text{fib}(i-1)$, and variable *next* will be used to store the next *Fibonacci* number $\text{fib}(i)$, which is the summation of the previous two *Fibonacci* numbers $\text{fib}(i-2)$ and $\text{fib}(i-1)$ stored in variables *previous* and *current* respectively. Then the loop body prepares for the next loop iteration. From the point of view of the next iteration $i+1$, the *current* value becomes the *previous* value, and the *next* value becomes the current value.
- The recursive version runs much slower than the iterative one because the former conducts many redundant method calls. Let us look at the evaluation process of $\text{recursiveFibonacci}(4)$. The following Figure 9 shows the method invocations spawned by $\text{recursiveFibonacci}(4)$.

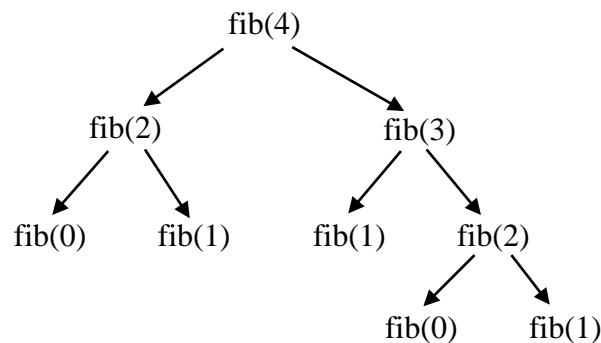


Figure 9 $\text{Fib}(4)$ method invocation spawn tree

We can see from this method spawn tree that $\text{fib}(2)$ is called 2 times, $\text{fib}(1)$ is called 3 times, and $\text{fib}(0)$ is called 2 times. During my test run with $n = 40$, the

iterative version took less than one millisecond to complete, while the recursive version took 9203 milliseconds. The difference in running times between the two versions grow very fast when n grows in its value. This confirms our suggestion that if possible, try to avoid using recursive methods.

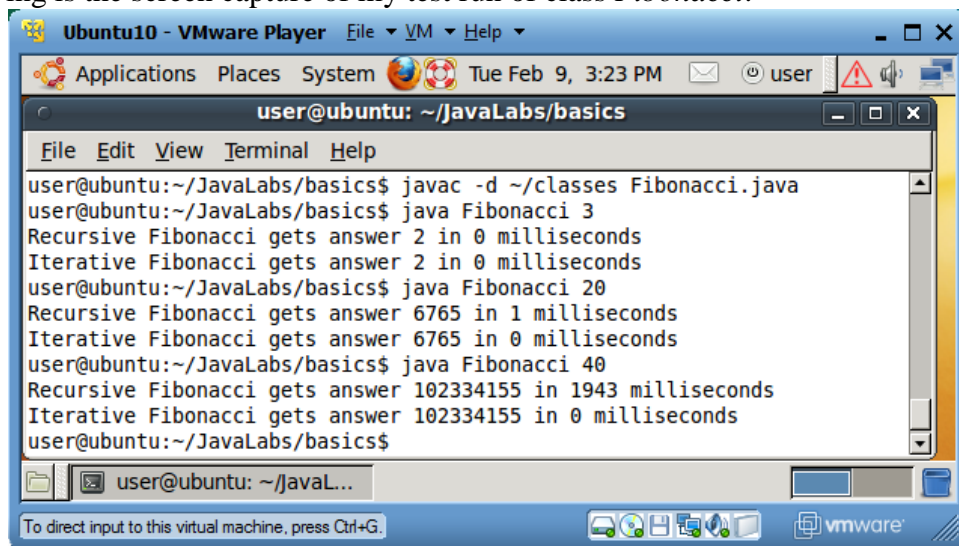
3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *Fibonacci*, type

```
javac -d ~/classes Fibonacci.java
```

4. To run class *Fibonacci*, type

```
java Fibonacci 3  
java Fibonacci 10  
java Fibonacci 20  
java Fibonacci 30  
java Fibonacci 40
```

The following is the screen capture of my test run of class *Fibonacci*.



```
Ubuntu10 - VMware Player  File  VM  Help  ▾  
Applications  Places  System  Tue Feb 9, 3:23 PM  user  [Warning Icon] [Volume Icon] [Network Icon]  
user@ubuntu: ~/JavaLabs/basics  
File  Edit  View  Terminal  Help  
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes Fibonacci.java  
user@ubuntu:~/JavaLabs/basics$ java Fibonacci 3  
Recursive Fibonacci gets answer 2 in 0 milliseconds  
Iterative Fibonacci gets answer 2 in 0 milliseconds  
user@ubuntu:~/JavaLabs/basics$ java Fibonacci 20  
Recursive Fibonacci gets answer 6765 in 1 milliseconds  
Iterative Fibonacci gets answer 6765 in 0 milliseconds  
user@ubuntu:~/JavaLabs/basics$ java Fibonacci 40  
Recursive Fibonacci gets answer 102334155 in 1943 milliseconds  
Iterative Fibonacci gets answer 102334155 in 0 milliseconds  
user@ubuntu:~/JavaLabs/basics$  
user@ubuntu: ~/javaL...  
To direct input to this virtual machine, press Ctrl+G.
```

3.24 Algorithm and Towers of Hanoi

An *algorithm* is a well-defined procedure for solving a problem satisfying two conditions:

1. Each step of an algorithm is well-specified so it can be precisely carried out with no ambiguity;
2. The execution of an algorithm can always terminate with correct answer for any input data valid for the problem.

Cooking recipe is a simple example of algorithm. A recipe tells us detailed steps in cooking. An algorithm is the essential idea for solving a problem, and it is normally described in natural languages like English. An algorithm itself is independent of programming languages, but computing algorithms are normally implemented in programming languages so they can execute on computers. A program that can sometimes run into some infinite loops cannot implement algorithms.

Over the last half century we have identified many important computing tasks and designed various algorithms for their solutions. Learning good algorithms for solving important problems can prevent us from reinventing wheels. Therefore one important topic for computer science study is the study of algorithms.

For a given problem, there are many ideas thus algorithms for solving it. Some are more straightforward, and some are more complicated or subtle. Some take more main memory space for their execution, and some take more CPU time for their execution. For practical reasons, computer scientists evaluate the quality of an algorithm based on two major criteria: its execution time efficiency and the amount of memory space it takes to run. The former is called the *time complexity* of an algorithm, and the latter is called the *space complexity* of an algorithm.

The running time of a program indicates the time complexity of the algorithm implemented by the program. But running time is not the same as time complexity of an algorithm. Time complexity is about the quality of an algorithm only, while the running time of a program depends on computer speed, amount of available main memory, the programming language used, and the quality of the compiler or interpreter that transforms the program source code to machine code. In this course we will learn some basic but important computer science algorithms, and learn to be conscientious about program running time or time efficiency.

There are many general categories for problem-solving, and recursion is one of them. In this subsection we will see how recursion can be a powerful problem-solving strategy, and how we use it to design an algorithm to solve a hard problem named Towers of Hanoi.

In the Towers of Hanoi problem, there are $n \geq 1$ disks of different diameters and three pegs named A, B, and C. Initially all the n disks are on peg A, sorted from small to large, with the largest at the bottom; and pegs B and C are empty. The objective is to move all the disks from peg A to peg C with the help of peg B. The only allowed operation is to move a top disk from one peg to the top of another peg. At any time, no disk can be on top of a smaller one on any of the three pegs. Figure 10 (a) shows the initial configuration for $n = 3$, and Figure 10 (d) shows the final configuration for its solution.

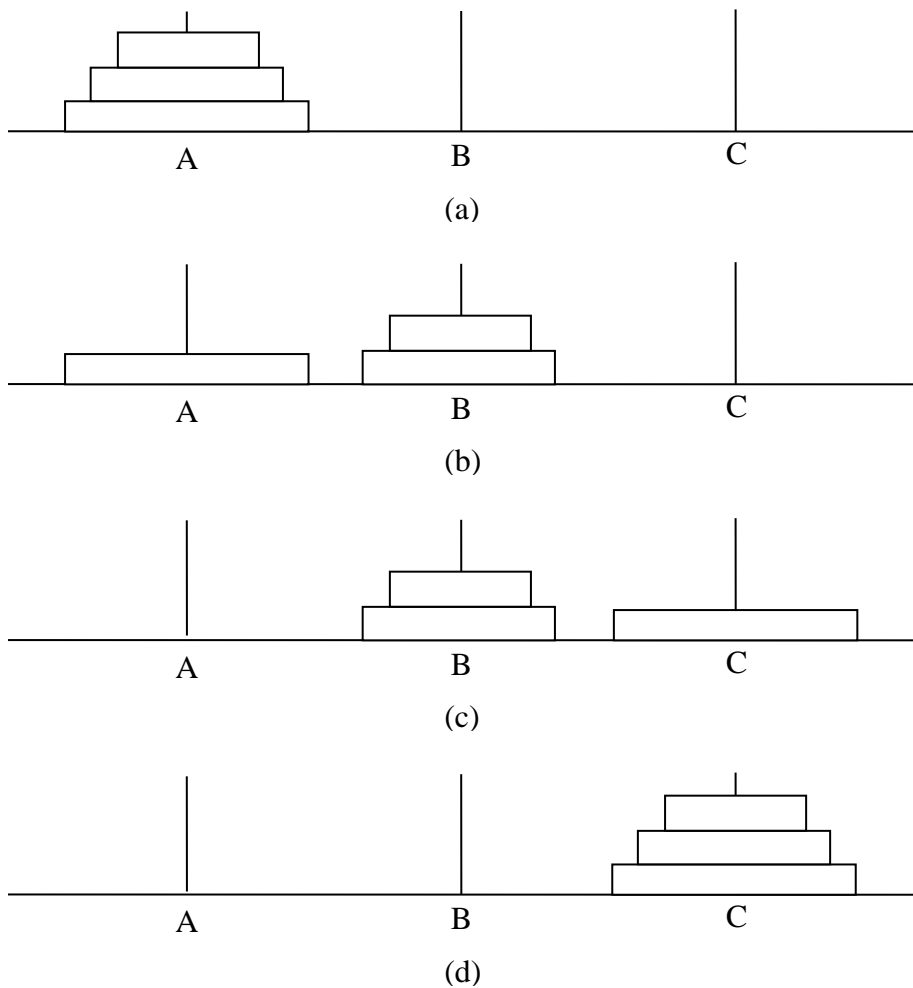


Figure 10 Tower of Hanoi problem and its solution strategy

Now we use recursion as the approach to design an algorithm for solving this problem. The idea is very similar to the mathematical proof process by induction. We first have the following two observations:

- The designation of the starting peg and the destination peg is not significant. For example, if we know how to move n disks from peg A to peg B with the help of peg C, we can certainly use a similar approach to move n disks from peg A to peg C with the help of peg B.
- If some larger disks are at the bottom of a peg, we can ignore them and treat the peg as an empty one when we move disks smaller than them. This is because the larger disks at the bottom of the peg will not cause violations to the constraint that no disk can be on top of a smaller one.

Now let us describe our solution strategy based on induction or recursion.

- If $n = 1$, we know how to solve the problem: we just move the only disk from peg A to peg C.
- Let us assume that we already know how to solve the problem when the number of disks is less than n . Put it another way, if the number of disks is less than n , we already know how to move the disks from any starting peg to any destination peg.

- Now let us show how we can move n disks from peg A to peg C with the help of peg B. We solve this problem in three steps:
 - We first move $n-1$ disks from peg A to peg B with the help of peg C. Since the largest disk at the bottom of peg A is not causing problem and thus can be ignored, and by our induction hypothesis in the previous step that we know how to move $n-1$ disks from any starting peg to any destination peg, we know how to complete this step. Figure 10 (b) shows the result of this step when $n = 3$.
 - Since peg A now only has the largest disk and peg C is empty, we can move the largest disk from peg A to peg C. Figure 10 (c) shows the result of this step when $n = 3$.
 - Since the largest disk on peg C can be safely ignored, what we need to do now to complete the solution is to move the $n-1$ smaller disks from peg B to peg C with the help of peg A. This step can be done due to the same reasons as for the first step. Figure 10 (d) shows the final disk placement for $n = 3$.
- Therefore we can conclude that for any number of disks, we can solve the Towers of Hanoi problem.

Now we develop a Java class to print out the disk move steps for solving the Towers of Hanoi problem for any positive integer n .

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “Tower.java” in directory “~/JavaLabs/basics” and the file has contents below:

```

1 public class Tower {
2     public static void main(String[] args) {
3         int n = 0; // input integer
4         try {
5             n = Integer.parseInt(args[0]);
6             if (n < 1)
7                 throw new Exception("Input value is not positive");
8             if (n > 15)
9                 throw new Exception("Input value is too large");
10        }
11        catch (Exception e) {
12            System.out.println(e);
13            System.out.println("Usage: java Tower [positive-integer]");
14            System.exit(-1);
15        }
16        long startTime = System.currentTimeMillis(); // record start time
17        tower(n, 'A', 'C', 'B');
18        long stopTime = System.currentTimeMillis(); // record stop time
19        System.out.println("Problem is solved in "
20            + (stopTime - startTime) + " milliseconds");
21    }
22
23    /**
24     * Move <code>n</code> disks from peg <code>fromPeg</code>
25     * to peg <code>toPeg</code> with the help of peg <code>helpPeg</code>.
26     * @param n number of disks
27     * @param fromPeg starting peg
28     * @param toPeg destination peg
29     * @param helpPeg auxiliary peg
30     */

```

```

31 static void tower(int n, char fromPeg, char toPeg, char helpPeg) {
32     if (n <= 0)
33         return;
34     if (n > 1)
35         tower(n-1, fromPeg, helpPeg, toPeg);
36     System.out.println("Move top disk from peg " + fromPeg + " to peg "
37                       + toPeg);
38     if (n > 1)
39         tower(n-1, helpPeg, toPeg, fromPeg);
40 }
41 }

```

Explanation

- a. We limit the value of the input integer to be between 1 and 15. Otherwise the recursive solution takes too long time to complete. The memory space taken by the activation records for the recursive invocation to method `tower()` grows very fast as number of disks grow, so your PC may crash when the number of disks exceeds 15. You should incrementally increase the number of disks when you run this program.
- b. Method `tower()` implements the algorithm we outlined before. It is basically a simple paraphrase in Java statements for our informal algorithm description. It is a more concise version equivalent to the following more intuitive one:

```

static void tower(int n, char fromPeg, char toPeg, char helpPeg) {
    if (n <= 0)
        return;
    if (n == 1) {
        System.out.println("Move disk from peg " + fromPeg
                          + " to peg " + toPeg);
        return;
    }
    tower(n-1, fromPeg, helpPeg, toPeg);
    System.out.println("Move disk from peg " + fromPeg
                      + " to peg " + toPeg);
    tower(n-1, helpPeg, toPeg, fromPeg);
}

```

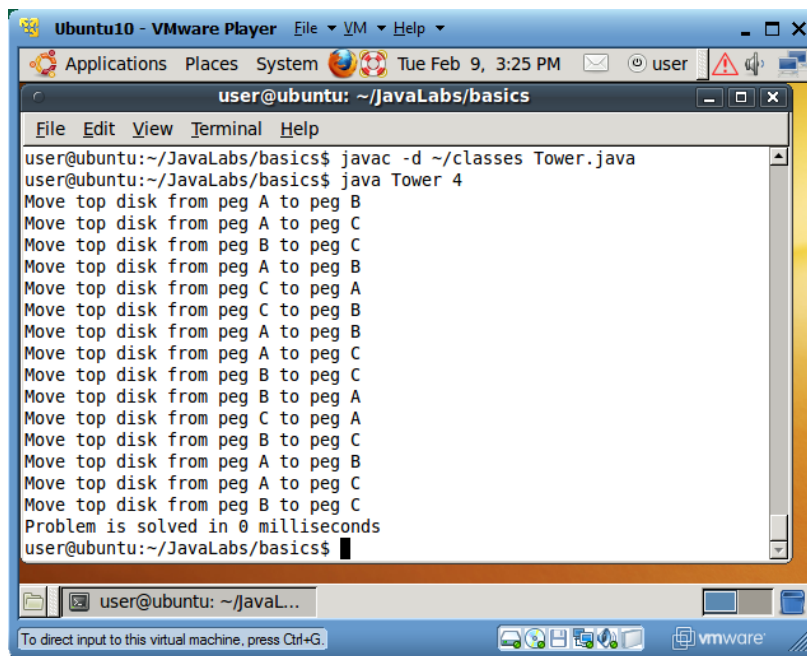
The version that we adopted avoided the duplicate print statement.

3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *Tower*, type


```
javac -d ~/classes Tower.java
```
4. To run class *Tower*, type


```
java Tower 4
```

The following is the screen capture of my test run of class *Tower*.



```
Ubuntu10 - VMware Player  File  VM  Help
Applications  Places  System  Tue Feb 9, 3:25 PM  user
user@ubuntu: ~/JavaLabs/basics
File  Edit  View  Terminal  Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes Tower.java
user@ubuntu:~/JavaLabs/basics$ java Tower 4
Move top disk from peg A to peg B
Move top disk from peg A to peg C
Move top disk from peg B to peg C
Move top disk from peg A to peg B
Move top disk from peg C to peg A
Move top disk from peg C to peg B
Move top disk from peg A to peg B
Move top disk from peg A to peg C
Move top disk from peg B to peg C
Move top disk from peg B to peg A
Move top disk from peg C to peg A
Move top disk from peg B to peg C
Move top disk from peg B to peg A
Move top disk from peg A to peg B
Move top disk from peg A to peg C
Move top disk from peg B to peg C
Problem is solved in 0 milliseconds
user@ubuntu:~/JavaLabs/basics$
```

3.25 Bubble sort

Sorting data means to rearrange data in either increasing or decreasing order. Sorting is one of the most important building blocks of many real-life applications. In this sub-section we learn one of the simplest algorithms for data sorting named *bubble sort*.

In bubble sort, data are initially stored in a one-dimensional array. Assume that there are n numbers in the array to be sorted. The algorithm goes through $n-1$ loop iterations. During each iteration, we scan the data from left to right, compare successive numbers, and keep swapping the largest value that we have seen so far in this iteration, called the *bubble*, to the right. At the end of such an iteration, the largest value (bubble) found in this iteration is at its final destination and doesn't need be considered in the following loop iterations.

Now we are ready to develop a Java class for bubble sort.

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “BubbleSort.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 import java.util.Random;
2
3 public class BubbleSort {
4     public static void main(String[] args) {
5         int[] d = new int[10];
6         Random r = new Random(); // create a Random object
7         for (int i = 0; i < d.length; i++)
8             d[i] = r.nextInt(50); //generate random numbers between 0 and 49
9         System.out.print("Data before sorting: ");
10        printArray(d);
```

```

11     bubbleSort(d);
12     System.out.print("Data after sorting:  ");
13     printArray(d);
14 }
15
16 static void bubbleSort(int[] a) {
17     for (int limit = a.length - 1; limit > 0; limit--) {
18         for (int j = 0; j < limit; j++) {
19             if (a[j] > a[j+1]) {
20                 // swap values in a[j] and a[j+1]
21                 int temp = a[j];
22                 a[j] = a[j+1];
23                 a[j+1] = temp;
24             }
25         }
26     }
27 }
28
29 static void printArray(int[] a) {
30     for (int i = 0; i < a.length; i++)
31         System.out.print(a[i] + " ");
32     System.out.println();
33 }
34 }

```

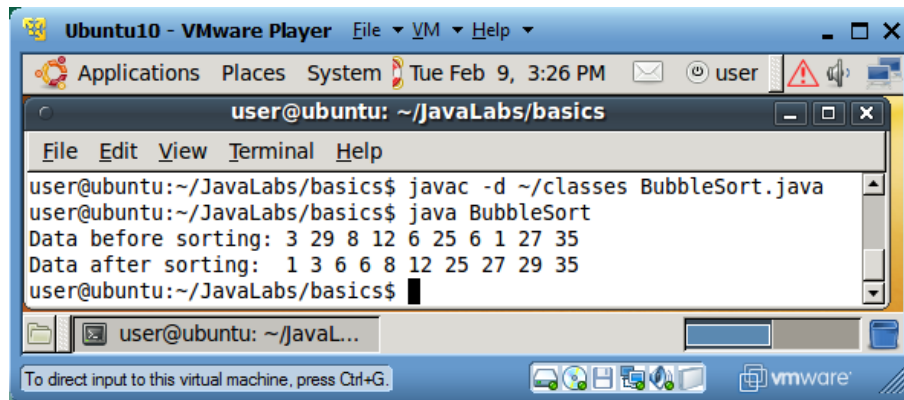
Explanation

- a. Method `bubbleSort(int[])` consists of two nested loops. The outer loop uses *limit* as its loop variable. Variable *limit* is initialized to point to the last cell of the array `a[]`. Each iteration of the outer loop finds the largest value in `a[0..limit]`, and moves it to `a[limit]`. Initially *limit* holds the index of the last array cell, so we find the largest value in the entire array and then leave it in the last array cell. In the next iteration we reduce the value of *limit* by 1 so we exclude the previous largest value from further consideration. We repeat this process until *limit* has value 1, and we make sure that the larger value in `a[0..1]` will be in `a[1]`.
- b. The nested loop uses *j* as its loop variable. During each iteration, we call the largest value that we have met so far a *bubble*. We view the right end of the array as the water surface, and we let the bubble go up to the water surface. Initially loop variable *j* has value 0 and the value in `a[0]` is the bubble or the largest value we have seen in this iteration. For each iteration of the nested loop, we compare the current value in `a[j]` with the next value in `a[j+1]`, and swap them if `a[j] > a[j+1]`. Therefore, after the *j*-th iteration of the nested loop, `a[j+1]` has the largest value in `a[0..j+1]`. The largest value for loop variable *j* to take on is *limit* - 1, so at the end of one complete execution of the nested loop, `a[limit]` has the largest value in `a[0..limit]`.
- c. We should always avoid writing repeated code. We can use methods to implement the common functions, and call these methods whenever we need their functions. Method `printArray(int[])` is one of our examples here. The main objective of a method is to encapsulate some functions in it so it can be used at multiple places of a program for performing similar tasks. The programmers who invoke a method don't need to understand the implementation of the method's functions. In case the method implementer finds a more efficient way for the method's implementation, he/she has the full freedom to update the body of the method as long as he/she doesn't change the method name, method parameters,

and method semantics. Because similar functions are abstracted into a single method, we avoid the nasty scenario in which we need to modify function implementations at multiple source code locations if some bug needs be fixed.

3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *BubbleSort*, type
javac -d ~/classes BubbleSort.java
4. To run class *BubbleSort*, type
java BubbleSort

The following is the screen capture of my test run of class *BubbleSort*. Because we sort random numbers, every run will generate different number sequences.



```
user@ubuntu: ~/JavaLabs/basics
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes BubbleSort.java
user@ubuntu:~/JavaLabs/basics$ java BubbleSort
Data before sorting: 3 29 8 12 6 25 6 1 27 35
Data after sorting:  1 3 6 6 8 12 25 27 29 35
user@ubuntu:~/JavaLabs/basics$
```

3.26 Binary search

Many applications need to maintain a large set of data records and then check whether a particular data record is in the data set. Each *data record* consists of a few *data fields*. For a human resource personnel record, a data record normally includes data fields like social security number, last name, first name, date of birth, date of joining the company, current monthly salary, etc. Some times these data records have a special *key field* that has a unique value for each record. For example, human resource offices normally use social security number as the key field for each employer’s personnel record. Given the value of a (key) field, we are often asked to search for any data record (or all data records) that have the same value for that field. When the data set is very large, the efficiency of data search can be significant.

To simplify the problem so we can focus on the fundamental idea of the *data search problem*, let us use a set of integers to represent the data sets, and store the integers in a one-dimensional array. Given a key integer value, we need to report the index of that key value in the array, or -1 if the key value is not in the array. If there are multiple occurrences of the key value in the array, we can return the index for any occurrence of that key value in the array.

The most natural approach to solve this data search problem is to use *sequential search*. Let us assume that the integers are stored in array *d[]*, and the key value is stored in variable *key*. We can use the following loop to sequentially check each array cell to see whether the key value is stored there. If the answer is true, we return the current index value. Otherwise if we have not come to the end of the array, we repeat the check in the next array cell.

```

int sequentialSearch(int[] d, int key) {
    for (int i = 0; i < d.length; i++)
        if (d[i] == key)
            return i;
    return -1;
}

```

This sequential search works, but its *time efficiency* is a problem. Let us assume that array $d[]$ has n integers. The search time for a key value in $d[]$ is in general proportional to n . But in real life we normally do better than sequential search. A White Book for telephone numbers stores telephone entries according to their owners' names in lexicographical order. When we look up a friend's telephone entry in a White Book, we don't start from page one and do a sequential search. Instead we make a guess based on the name that we are looking for, and we go and check a page normally in the middle of the book. If we are lucky, we find our friend's entry on that page and we are done. Otherwise, if the page contains names smaller than the one that we are looking for in lexicographical order, we know that we can safely ignore the first half of the book and we repeat the above process in the second half of the book. On the other hand, if the page contains names larger than the one that we are looking for in lexicographical order, we know that we can safely ignore the second half of the book and we repeat the above process in the first half of the book. This is the fundamental idea of *binary search*. If we always start the search at the middle of the data range that we have not checked yet, each iteration will allow us to halve the size of data set that we still need to look into, and the worst-case running time for a binary search is therefore $\log_2 n$.

Binary search is much more efficient than sequential search. But before we can conduct binary search, we have to sort the data set in a particular order. This is the one-time overhead. If the problem at hand needs to maintain a stable large set of data and search particular data items repeatedly, it pays off to sort the stable data set once and then we can enjoy the faster binary search from then on. On the other hand, if the data set is very dynamic or the data set is very small, sequential search may prove to be a better solution.

It is good time to introduce the concepts of *best-case time complexity*, *worst-case time complexity*, and *average-case time complexity* of an algorithm. The execution time of an algorithm first depends on the size of the problem instance on which we run the algorithm. For the data search problem, the *size of a problem instance* can be naturally defined as the number of values in the array that we need to search into. Let us use n to represent this problem instance size. It is easy to see that solving a larger problem instance in general takes more CPU time than solving a smaller problem instance. If the execution time of an algorithm is independent of problem instance sizes, we say that the time complexity of the algorithm is $O(1)$, meaning that the running time is a constant. If the execution time of an algorithm is proportional to the problem instance size, we say that the time complexity of the algorithm is $O(n)$, meaning that the running time is proportional to the size of the problem instance. The constant running time $O(1)$ is always better than $O(n)$ when the size of the problem instances grow large.

But even for the same problem instance size, the execution time of an algorithm is in general problem instance dependent. Let us think about the sequential search algorithm first. If we are lucky, the key value is in the first array cell. In this case we say that the sequential search algorithm has a best-case time complexity of *constant time* (the running time is independent of n ,

the number of data records in the array), or $O(1)$. If we are not lucky, the key value is not in the array at all, but we don't know this fact until we have checked into each cell of the array. Therefore the worst-case time complexity of sequential search is proportional to n , and we use symbol $O(n)$ to represent "proportional to n ". Suppose the key takes on any value in the array with the same probability. Then the average-case time complexity of the sequential search algorithm is roughly half of the array size, or $n/2$, which is still proportional to n , or $O(n)$.

Now let us study the case of running binary search for a key value in a data set with n values. If we are lucky, the value at the middle of the data array is holding the key value. Therefore the best-case time complexity for binary search is $O(1)$. If we are not lucky and the key value is not in the data array, we have to halve the size of the array section to be searched into repeatedly until the array section contains only one value, and this process takes time proportional to $\log_2 n$ ($2^{\log_2 n} = n$). (If $n = 8$, the array section to be searched into will be halved from 8 to 4, then to 2, then to 1) We use symbol $O(\log_2 n)$ to mean that the running time is proportional to $\log_2 n$. Therefore the worst-case time complexity of binary search is $O(\log_2 n)$, which grows much slower than $O(n)$. Further study shows that the average-case time complexity of binary search is also $O(\log_2 n)$.

When we talk about algorithm time complexity, unless we specify otherwise, we assume the worst-case time complexity.

Now we are ready to develop a Java class for binary search.

1. Open a terminal window. Change directory to "`~/JavaLabs/basics`".
2. Use a text editor to open and review Java source code file "`BinarySearch.java`" in directory "`~/JavaLabs/basics`" and the file has contents below:

```

1  import java.util.*;
2
3  public class BinarySearch {
4      public static void main(String[] args) {
5          int[] d = new int[10];
6          Random r = new Random(); // create a Random object
7          for (int i = 0; i < d.length; i++)
8              d[i] = r.nextInt(50); //generate random numbers between 0 and 49
9          BubbleSort.bubbleSort(d);
10         System.out.print("Data to be searched into: ");
11         BubbleSort.printArray(d);
12         System.out.print("Please enter an integer to search for: ");
13         // creat a Scanner object for keyboard
14         Scanner keyboard = new Scanner(System.in);
15         int v = keyboard.nextInt();
16         int index1 = binarySearch(d, v);
17         // search by library method
18         int index2 = java.util.Arrays.binarySearch(d, v);
19         if (index1 == -1) {
20             System.out.println("Value " + v + " is not in array d[]");
21             System.exit(0);
22         }
23         System.out.println("Our implementation:      Value "
24                             + v + " is in d[" + index1 + "]");
25         System.out.println("Library implementation: Value "
```

```

26         + v + " is in d[" + index2 + "]);
27     }
28
29     /*
30     * Returns the index of key in array a[] if key is in a[],
31     * -1 otherwise.
32     * @param a the array of data to be searched in.
33     * @param key the value to be searched for
34     */
35     static int binarySearch(int[] a, int key) {
36         int leftLimit = 0;
37         int rightLimit = a.length - 1;
38         while (leftLimit <= rightLimit) {
39             int middle = (leftLimit + rightLimit)/2;
40             if (a[middle] == key)
41                 return middle;
42             if (a[middle] < key)
43                 // key can only be in a[middle+1..rightLimit]
44                 leftLimit = middle + 1;
45             else //a[middle] > key
46                 // key can only be in a[leftLimit..middle-1]
47                 rightLimit = middle - 1;
48         }
49         return -1;
50     }
51 }

```

Explanation

- a. We used the static method *bubbleSort()* of our class *BubbleSort* to sort the random integers in array *d[]*. We could also have used *java.util.Arrays.sort()* to do the same sorting. This is an example of code reuse.
 - b. We also used the static method *printArray()* of our class *BubbleSort* to print out the data in an array. This is also an example of code reuse.
 - c. Java class *java.util.Arrays* also supports a static method *binarySearch()* with the same signature as ours. We invoke both implementations of binary search to show that they provide the same answers.
 - d. During each iteration of the loop of the binary search method, the key value is searched in array section *a[leftLimit..rightLimit]*. Initially this array section is *a[0..a.length-1]*, or the whole array. The middle index for the array section is then evaluated. If *a[middle]* is equal to the key value, we return the value of the middle index. If *a[middle]* is smaller than the key value, we repeat the same process in the following loop iteration in array section *a[middle+1..rightLimit]*. Otherwise if *a[middle]* is larger than the key value, we repeat the same process in the following loop iteration in array section *a[leftLimit..middle-1]*.
3. Make sure your working directory now is “~/JavaLabs/basics”. To compile class *BinarySearch*, type
javac -d ~/classes BinarySearch.java
 4. To run class *BinarySearch*, type
java BinarySearch

The following is the screen capture of my test run of class *BinarySearch*. Because we use random numbers, every run will generate a different number sequence.


```
Ubuntu10 - VMware Player  File VM Help
Applications Places System Tue Feb 9, 3:29 PM user
user@ubuntu: ~/JavaLabs/basics
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/basics$ javac -d ~/classes BinarySearch.java
user@ubuntu:~/JavaLabs/basics$ java BinarySearch
Data to be searched into: 1 14 16 19 24 30 30 32 36 37
Please enter an integer to search for: 24
Our implementation: Value 24 is in d[4]
Library implementation: Value 24 is in d[4]
user@ubuntu:~/JavaLabs/basics$ java BinarySearch
Data to be searched into: 6 10 17 20 26 28 30 32 38 48
Please enter an integer to search for: 22
Value 22 is not in array d[]
user@ubuntu:~/JavaLabs/basics$
```

4 Object Oriented Programming

A class is an abstraction of some physical or conceptual objects. A class declaration includes reserved word **class**, a name and a class body delimited by a pair of curly braces, as “class ClassName { ... }”. The class body contains a list of declarations of data fields (variables global to methods in the class) and methods that normally work on the data fields of the class. The order of the data field and method declarations is not important. The methods’ bodies can access any of the data fields and methods of the class, independent of the order of the declarations.

Normally, data fields are declared as *private* ones so they can only be accessed in the body of methods of the same class. This is called *data encapsulation*. The users of a class are only interested in the public methods of the class, and data fields are part of a class’s internal implementation. By making the data fields private, we can modify field names or structure later as long as we don’t change the class’s public methods’ names. To enable other objects to get the current value of a private field *value* of type *int*, we declare a public *getter* method “public int getValue()”. To enable other objects to get the current value of a private field *found* of type boolean, we declare a public getter method “public boolean isFound()”. To enable other objects to set a new value in a private field *value* of type *int*, we declare a public *setter* method “public void setValue(int v)” (the parameter can be of any name). Getter methods are also called *accessors*, and setter methods are also called *mutators*. Not all private data fields need setters and getters. If you don’t want users to modify the value of a private fields, you can just remove its setter method.

Each class has one or more *constructors*, which have signatures similar to a method, normally public. A constructor has no return type, and its name must be the same as that of the containing class. A constructor can have any number of parameters of any types. If a constructor has no parameter, it is called a *default constructor*. If we don’t declare any constructor for a class, the class will have a default constructor with an empty body. But if we declare any constructors for a class, the class will not have a default constructor unless we explicitly declare it. Class constructors are used to conduct any computation at the beginning of an object’s life, normally for initializing some variables.

A class declaration basically introduces a new data type with which we can create objects of that class type. A class declaration is like a stamp or stencil, and an object is like a print. From a stamp we can make any number of copies of prints. With a class declaration, we can instantiate (create) any number of objects of that class type. Each object will take a chunk of memory space for storing its data fields. No memory space is needed for methods in each object. For example, if a class contains two *int*-type fields, then each object of that class will take $2 \times 4 = 8$ bytes or 64 bits to store two integers.

We create a new object by using the “new” operator followed by invoking one of the class’s constructors. First the memory space for the class’s data fields will be allocated. Then the body of the constructor will be executed to initialize the object’s memory space. Finally the starting address of the object’s memory space will be returned as the return value of the “new” expression.

If a data field is declared without qualifier “static” inside a class, it is called an *instance variable* because each instance (object) of the class will have its own memory space for that data field

(variable) and the values of an instance variable belonging to two different objects of the same class type will not interfere with each other. On the other hand, if a data field is declared with qualifier “static” inside a class, it is called a *class variable*, since all instances (objects) of that class will share the same memory space for class variables. Class variables are global or shared variables among all objects of the same class type.

Every class of Java implicitly or explicitly declares a method of signature “public String toString()”. When we print an object, we implicitly call the “toString()” method of the class to generate a string for the object. The default implementation of “toString()” only prints the class name and a reference number for that object, which are normally not what we need. We can override the default implementation of the “toString()” method by explicitly declaring it.

4.1 A simple *accumulator* class

1. Open a terminal window. Change directory to “~/JavaLabs/oop”.
2. Use a text editor to open and review Java source code file “Accumulator.java” in directory “~/JavaLabs/oop” and the file has contents below:

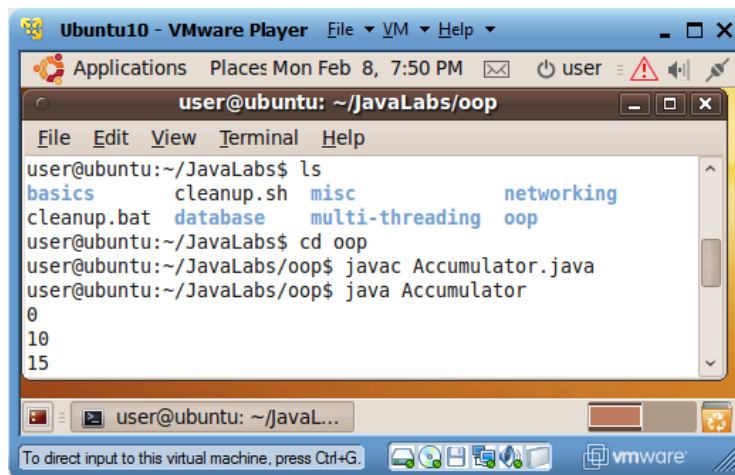
```
1 public class Accumulator {
2     private int value = 0;
3
4     public Accumulator() {} // optional
5
6     public void add(int v) {
7         value = value + v;
8     }
9
10    public int getValue() {
11        return value;
12    }
13
14    public void setValue(int value) {
15        this.value = value;
16    }
17
18    public static void main(String[] args) {
19        Accumulator a = new Accumulator();
20        System.out.println(a.getValue());
21        a.setValue(10);
22        System.out.println(a.getValue());
23        a.add(5);
24        System.out.println(a.getValue());
25    }
26 }
```

Explanation

- a. Class *Accumulator* has declarations for one private instance variable *value*, one default constructor (not necessary since it would be provided by Java compiler by

- default), a public method “add(int v)”, a public getter method “getValue()”, a public setter method “setValue(int value)”, and the public “main()” method.
- An object of type *Accumulator* takes four bytes to hold the value of instance variable *value*, which has initial value zero.
 - Since there is no declaration of parameter or local variable “value” inside method “add(int v)”, variable “value” referenced on line 7 is the instance variable “value”.
 - Since the parameter of “setValue(int value)” uses same name as the instance variable, on line 15 we need to use “this.value” to explicitly reference the instance variable “value” instead of the parameter or local variable “value”. Java keyword “this” means the current object, and “this.value” means the data field “value” instead of the local variable “value”.
 - The execution of a Java object starts with the body of its method “main()”. On line 19 an *Accumulator* object is created with the “new” operator. Line 20 invokes the getter method to retrieve the current value of instance variable “value”, which is zero, and prints it out. Line 21 invokes the setter method to set 10 in the instance variable. Line 22 prints out the current value of the instance variable. Line 23 adds five to the instance variable. Finally line 24 prints out the current value of the instance variable.
- Make sure your working directory is now “~/JavaLabs/oop”. To compile class *Accumulator*, type
javac Accumulator.java
 - To run class *Accumulator*, type
java Accumulator

The following is the screen capture of my test run of class *Accumulator*.



```
user@ubuntu: ~/JavaLabs/oop
File Edit View Terminal Help
user@ubuntu:~/JavaLabs$ ls
basics      cleanup.sh  misc          networking
cleanup.bat database    multi-threading oop
user@ubuntu:~/JavaLabs$ cd oop
user@ubuntu:~/JavaLabs/oop$ javac Accumulator.java
user@ubuntu:~/JavaLabs/oop$ java Accumulator
0
10
15
```

4.2 A simple class for circles

- Open a terminal window. Change directory to “~/JavaLabs/oop”.
- Use a text editor to open and review Java source code file “TestCircle.java” in directory “~/JavaLabs/oop” and the file has contents below:

```
1 public class TestCircle {
2     public static void main(String[] args) {
```

```

3     Circle x = new Circle(1.0);
4     System.out.println(x);
5     Circle y = new Circle();
6     System.out.println(y);
7     y.setRadius(2.0);
8     System.out.println(y + " has area " + y.area());
9 }
10 }
11
12 class Circle {
13     private double radius;
14     // define PI as nickname for constant 3.14
15     private final double PI = 3.14;
16
17     // Constructor 1
18     public Circle(double radius) {
19         System.out.println("Enter constructor Circle(" + radius + ")");
20         this.radius = radius; // this = the current object
21     }
22
23     // Constructor 2: the default constructor
24     public Circle() {
25         this(0.0); // call constructor 1; this = class name
26     }
27
28     // Setter method
29     public void setRadius(double radius) {
30         this.radius = radius;
31     }
32
33     // Getter method
34     public double getRadius() {
35         return radius;
36     }
37
38     // Calculate area of the circle
39     public double area() {
40         return PI * radius * radius;
41     }
42
43     // Define how to print Circle objects
44     public String toString() {
45         return "Circle(" + radius + ")";
46     }
47 }

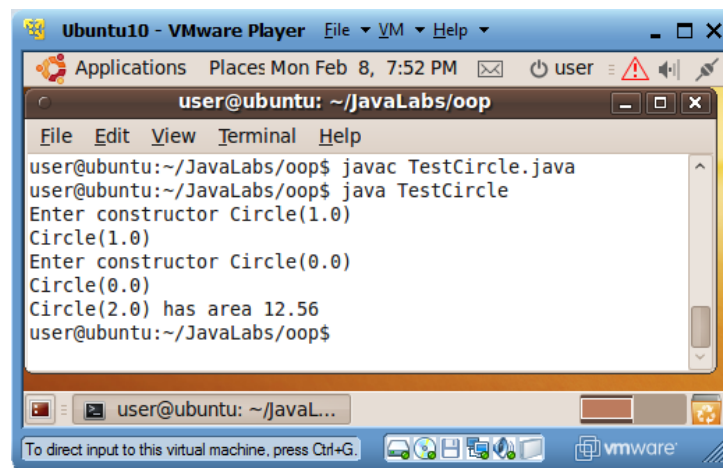
```

Explanation

- File “TestCircle.java” contains two class declarations. A Java source code file can hold at most one public class, and the file name stem must be the same as the name of the public class.
- If a variable declaration is qualified by “final”, as on line 15, then we actually define a constant.
- This class has two constructors. The one on line 18 has a parameter to specify the radius of the circle. The one on line 24 has no parameters, and it creates a circle with a radius of zero. Instead of repeating the code in the body of the first constructor, line

- 25 indirectly invokes the first constructor and passing zero as the argument value. While keyword “this” in “this.variable” refers to a variable declared in the object (not in the method), expression “this(…)” is for invoking the corresponding constructor of the class for the current object.
- d. The method “toString()” on line 43-46 specifies how to convert a *Circle* object into a meaningful string.
 - e. When an object is printed, the string generated by the object’s “toString()” method is printed.
3. Make sure your working directory is now “~/JavaLabs/oop”. To compile class *TestCircle*, type
javac TestCircle.java
 4. To run class *TestCircle*, type
java TestCircle

The following is the screen capture of my test run of class *TestCircle*.



```
user@ubuntu: ~/JavaLabs/oop
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/oop$ javac TestCircle.java
user@ubuntu:~/JavaLabs/oop$ java TestCircle
Enter constructor Circle(1.0)
Circle(1.0)
Enter constructor Circle(0.0)
Circle(0.0)
Circle(2.0) has area 12.56
user@ubuntu:~/JavaLabs/oop$
```

4.3 Method overloading

1. Open a terminal window. Change directory to “~/JavaLabs/oop”.
2. Use a text editor to open and review Java source code file “MethodOverLoading.java” in directory “~/JavaLabs/oop” and the file has contents below:

```
1 public class MethodOverloading {
2     public static void main(String[] args) {
3         Utility u = new Utility();
4         u.print();
5         u.print(5);
6         u.print(5.0);
7         u.print(2, 4);
8     }
9 }
10
11 class Utility {
12     public void print() {
13         System.out.println("null");
14     }
```

```

15 public void print(int i) {
16     System.out.println("Integer " + i);
17 }
18
19 public void print(double d) {
20     System.out.println("Double " + d);
21 }
22
23 public void print(int i, int j) {
24     System.out.println("Integers " + i + " and " + j);
25 }
26 }

```

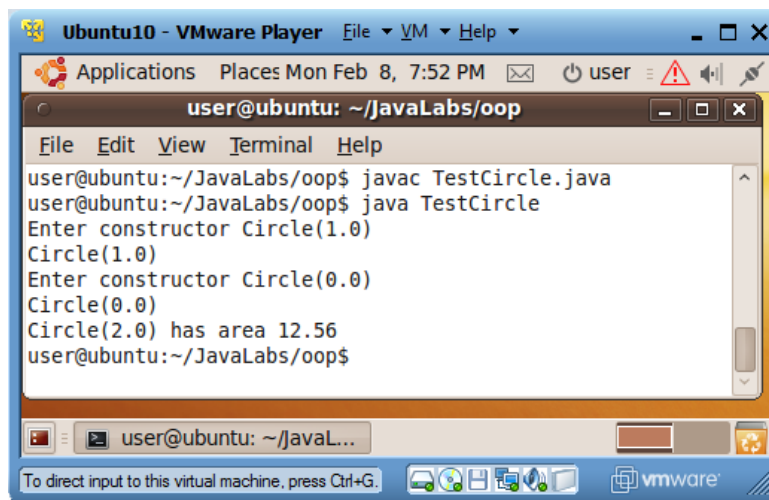
Explanation

- a. The signature of a method includes the name of the method and the list of data types of the parameters of the method (number of parameters, and the type of each parameter; if there are multiple parameter types, then the type order is also important). For example, the method on line 23 of class *Utility* has signature “print(int, int)”, and the parameter names and return type “void” are not part of a method signature, neither is the method modifier “public/private/protected” or “static”.
 - b. If two methods have the same name but different signatures (differ in the list of parameter types), they can co-exist in a class. When such methods are invoked, the method implementation with the closest list of types of the parameters and the arguments will be executed. This mechanism is called *method overloading*.
 - c. In class *Utility*, method *print* is overloaded five times.
 - d. Note: If two methods have the same signature but differ by their return value types, they cannot co-exist in the same class to overload. For example, class *Utility* cannot have another method that reads “public int print() {...}”.
3. Make sure your working directory is now “~/JavaLabs/oop”. To compile class *MethodOverLoading*, type


```
javac MethodOverLoading.java
```
 4. To run class *MethodOverLoading*, type


```
java MethodOverLoading
```

The following is the screen capture of my test run of class *MethodOverLoading*.



4.4 Class inheritance

Class inheritance enables the creation of a new class (*subclass* or *derived class*) by the incremental extension of an existing class (*superclass* or *base class*). All public instance/class variables and methods of the superclass are inherited by the subclass, the subclass can declare methods with the same signatures as those in the superclass to override (hide) them, and the subclass can declare new data fields and methods to add new features and functions.

Class inheritance serves two purposes:

- Code reuse. When we need to have functions similar to those of an existing class, instead of creating the new class from scratch, we can get the new class by modifying (overriding) and extending the existing class.
- Object modeling. The subclass objects can represent special cases of the superclass objects. For example, a pet is an abstract concept, a cat or a dog is a special case of a pet, and a Persian cat is a special kind of cat. We could have classes Pet, Cat, Dog, and PersianCat, where classes Cat and Dog have all features and functions of a Pet as well as their extra features and functions to make them unique, and class PersianCat has all features and functions of a Cat as well as its extra features and functions to make it unique. Therefore class inheritance is a natural mechanism to model such object/concept classification.

1. Open a terminal window. Change directory to “~/JavaLabs/oop”.
2. Use a text editor to open and review Java source code file “Inheritance.java” in directory “~/JavaLabs/oop” and the file has contents below:

```

1 public class Inheritance {
2     public static void main(String[] args) {
3         Cat c = new Cat("Cathy");
4         Dog d = new Dog("Champion");
5         c.talk();
6         d.talk();
7         System.out.println("I am " + c + ", my name is " + c.getName());
8         System.out.println("I am " + d + ", my name is " + d.getName());
9         // polymorphism demo: the version of toString() used is the one
10        // declared in the subclasses
11        Pet[] p = new Pet[2];

```



```

12     p[0] = c;
13     p[1] = d;
14     for (int i = 0; i < p.length; i++)
15         System.out.println("I am " + p[i] + ", my name is "
16                             + p[i].getName());
17     // p[0].talk(); // this will fail, since Pet has no method talk()
18     for (int i = 0; i < p.length; i++) {
19         if (p[i] instanceof Cat) // Is p[i] an instance of Cat?
20             ((Cat)p[i]).talk();    // Cast p[i] back into a reference for Cat
21     }
22 }
23 }
24
25 class Pet {
26     private String name;
27
28     public Pet(String n) {
29         this.name = n;
30         System.out.println("Pet(" + n + ")");
31     }
32
33     public String getName() {
34         return name;
35     }
36
37     public String toString() {
38         return "Pet(" + name + ")";
39     }
40 }
41
42 class Cat extends Pet {
43     public Cat(String name) {
44         super(name); // call superclass constructor
45         System.out.println(this + ", I am also " + super.toString());
46     }
47
48     public void talk() {
49         System.out.println("I like fish.");
50     }
51
52     public String toString() { // override toString() method in superclass
53         return "Cat(" + getName() + ")";
54     }
55 }
56
57 class Dog extends Pet {
58     public Dog(String name) {
59         super(name); // call superclass constructor
60         System.out.println(this + ", I am also " + super.toString());
61     }
62
63     public void talk() {
64         System.out.println("I like sports.");
65     }
66
67     public String toString() { // override toString() method in superclass
68         return "Dog(" + getName() + ")";
69     }
70 }

```

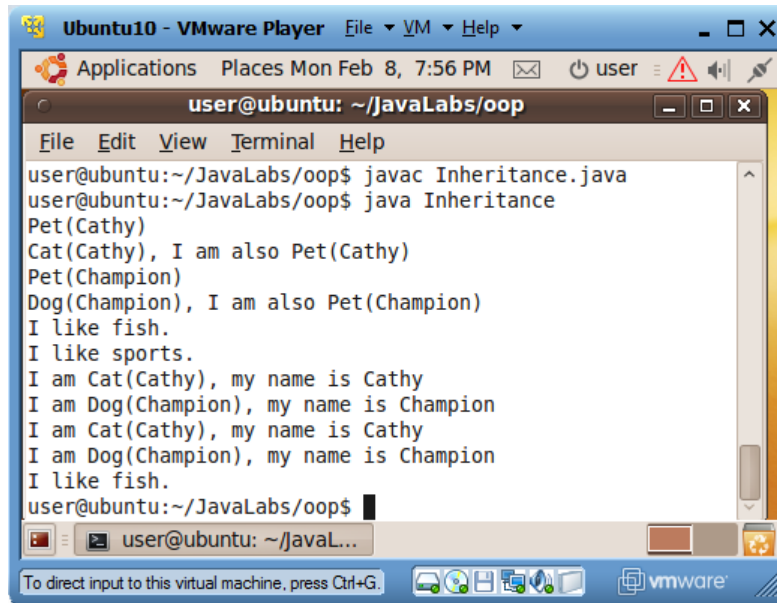
Explanation

- a. Lines 25-40 declare a class *Pet* that remembers the name of a pet.
 - b. On lines 42 and 57, classes *Cat* and *Dog* extend or inherit the base class *Pet* with syntax “class SubClass extends SuperClass {...}”, and the classes *Cat* and *Dog* both inherit the public method “getName()” of class *Pet*.
 - c. Classes *Cat* and *Dog* also inherit the public method “toString()” of class *Pet*, but it is overridden by the re-declaration of the method inside classes *Cat* and *Dog*.
 - d. Subclasses *Cat* and *Dog* also respectively add their own version of a new method “talk()”.
 - e. When the constructor for a subclass is executed, the constructor for the superclass is first executed before the body of the subclass’ constructor is executed. If the subclass needs to invoke a constructor of the superclass that has arguments, the first line of the superclass must be of form “super(...);” where “super” represents the name of the superclass. If the first line of the superclass is not of form “super(...);” then the default constructor of the superclass is executed. Java keywords “super” and “this” also represent the current superclass object and subclass objects respectively.
 - f. Class names are data types, and they can be used to declare *reference variables*.
 - g. Lines 3 and 4 instantiate a *Cat* object and a *Dog* object, and save their references (addresses) in variables *c* and *d* respectively.
 - h. Line 7 invokes method “toString()” of class *Cat* to convert the *Cat* object in variable *c* into a string, and invokes *Cat*’s method “getName()”, inherited from class *Pet*, to get the name of the *Cat* object.
 - i. A superclass reference variable can be used to hold the reference of a subclass object, and that is why on lines 12 and 13 we can copy references for the *Cat* and *Dog* objects into “p[0]” and “p[1]” both of type *Pet*.
 - j. When a superclass reference variable holds the reference of a subclass object and both the superclass and the subclass implement the same method with potentially different functions, an invocation of the method against the superclass reference variable will execute the method implementation in the subclass. This is called *polymorphism*: the actual method invoked against a superclass reference variable depends on at runtime what object is saved in the reference variable. As an example, on line 15, *Pet* reference variable “p[0]” holds the reference of a *Cat* object, and classes *Pet* and *Cat* both implement method “toString()”. Due to polymorphism, “p[0].toString()” (indirectly invoked by converting “p[0]” to a string) invokes the *Cat* class implementation of method “toString()”.
 - k. When a superclass reference variable holds the reference of a subclass object, we can only use it to access public data fields and methods declared in the superclass. That is why “p[0].talk();” on line 17 would abort the execution if it is uncommented because method “talk()” is not declared in the superclass *Pet*. To make “p[0].talk();” work, we need first cast the reference “p[0]” to a *Pet* object to a reference to a *Cat* object with the type casting expression “(Cat)p[0]”, as shown on line 20.
 - l. On line 19 we use expression “referenceVariable **instanceof** ClassName” to check whether the reference variable is an instance of a particular class.
3. Make sure your working directory is now “~/JavaLabs/oop”. To compile class *Inheritance*, type

javac Inheritance.java

4. To run class *Inheritance*, type
java Inheritance

The following is the screen capture of my test run of class *Inheritance*.



```
user@ubuntu: ~/JavaLabs/oop
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/oop$ javac Inheritance.java
user@ubuntu:~/JavaLabs/oop$ java Inheritance
Pet(Cathy)
Cat(Cathy), I am also Pet(Cathy)
Pet(Champion)
Dog(Champion), I am also Pet(Champion)
I like fish.
I like sports.
I am Cat(Cathy), my name is Cathy
I am Dog(Champion), my name is Champion
I am Cat(Cathy), my name is Cathy
I am Dog(Champion), my name is Champion
I like fish.
user@ubuntu:~/JavaLabs/oop$
```

4.5 Abstract methods

Sometimes we know a class will serve as the superclass for many classes that have not yet been designed, and we want to make sure that all of these subclasses must implement a particular method. There are two approaches to achieve this: making the superclass an abstract class, or using the Java interface mechanism introduced in the next section.

An abstract method's declaration has no method body, and it starts with modifier keyword "abstract" and ends with a semicolon, as in "abstract public void talk();". A class containing one or more abstract methods must be qualified by keyword "abstract" to become an abstract class.

An abstract class does not have all implementation details, so we cannot create objects for abstract classes. But the name of an abstract class can be used to declare reference variables that can hold references to objects based on any concrete (non-abstract) subclasses of the abstract class. To make an abstract class concrete, we can extend it into subclasses and implement (override) all abstract methods of the abstract class.

1. Open a terminal window. Change directory to "~/JavaLabs/oop".
2. Use a text editor to open and review Java source code file "AbstractMethods.java" in directory "~/JavaLabs/oop" and the file has contents below:

```
1 public class AbstractMethods {
2     public static void main(String arg[]) {
3         Pet pets[] = new Pet[2]; // creates an array of 3 Pet references
4         pets[0] = new Cat("Pussy"); // since Cat is a derived class of Pet
```

```

5     pets[1] = new Dog("Champion"); // since Dog is a derived class of Pet
6
7     for (int i = 0; i < pets.length; i++)
8         pets[i].talk();
9
10    Cat c = (Cat)pets[0]; // cast a Pet type object to a Cat object
11    c.hello();
12    // pets[0].hello(); // this line must be commented off; pets[0] has
13                        // type Pet,
14                        // only methods listed in class Pet can be
15                        // called from it.
16    c.helloFromParent(); // indirectly call a method of the base class
17 }
18 }
19
20 abstract class Pet {
21     abstract public void talk(); // all pets must implement method talk()
22     public void hello() {
23         System.out.println("Pet parent says hello to you.");
24     }
25 }
26
27 class Cat extends Pet {
28     String name;
29
30     public Cat(String name) {
31         this.name = name;
32     }
33
34     public void talk() { // this method is required by the abstract class
35                         // class Pet
36         System.out.println("I am cat " + name + ", meow.");
37     }
38
39     public void hello() { // this is a method not required by the abstract
40                         // class Pet
41         System.out.println("Cat " + name + " says hello to you.");
42     }
43
44     public void helloFromParent() {
45         super.hello(); // call hello() of the super (base) class
46     }
47 }
48
49 class Dog extends Pet {
50     String name;
51
52     public Dog(String name) {
53         this.name = name;
54     }
55
56     public void talk() { // this method is required by the abstract class
57                         // Pet
58         System.out.println("I am dog " + name + ", wow.");
59     }
60
61     public void hello() { // this is a method not required by the abstract
62                         // class Pet
63         System.out.println("Dog " + name + " says hello to you.");

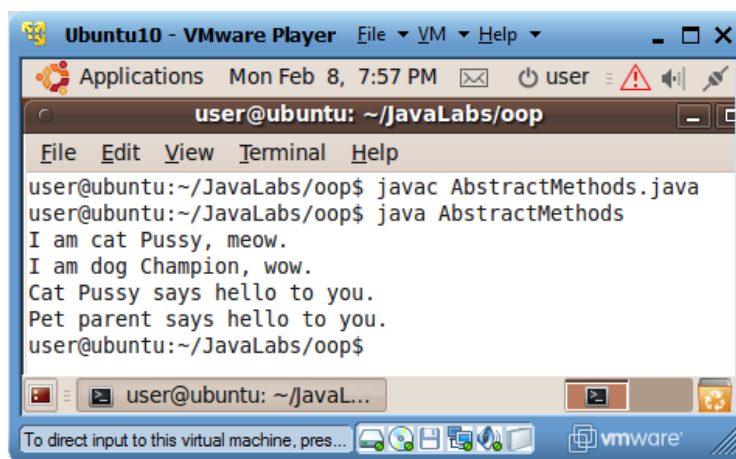
```

```
64 }
65 }
```

Explanation

- a. *Pet* is an abstract class that contains one abstract method “talk()”.
 - b. Classes *Cat* and *Dog* both extend abstract class *Pet* and implement the inherited abstract method “talk()”.
5. Make sure your working directory is now “~/JavaLabs/oop”. To compile class *AbstractMethods*, type
javac AbstractMethods.java
 6. To run class *AbstractMethods*, type
java AbstractMethods

The following is the screen capture of my test run of class *AbstractMethods*.



The screenshot shows a terminal window titled "user@ubuntu: ~/JavaLabs/oop". The terminal output is as follows:

```
user@ubuntu:~/JavaLabs/oop$ javac AbstractMethods.java
user@ubuntu:~/JavaLabs/oop$ java AbstractMethods
I am cat Pussy, meow.
I am dog Champion, wow.
Cat Pussy says hello to you.
Pet parent says hello to you.
user@ubuntu:~/JavaLabs/oop$
```

4.6 Java interfaces

1. Open a terminal window. Change directory to “~/JavaLabs/oop”.
2. Use a text editor to open and review Java source code file “InterfaceDemo.java” in directory “~/JavaLabs/oop” and the file has contents below:

```
1 public class InterfaceDemo {
2     public static void main(String arg[]) {
3         Pet pets[] = new Pet[2]; // creates an array of 3 Pet references
4         pets[0] = new Cat("Pussy"); // since Cat implements interface Pet
5         pets[1] = new Dog("Champion"); // since Dog implements interface Pet
6
7         for (int i = 0; i < pets.length; i++)
8             pets[i].talk();
9
10        Cat c = (Cat)pets[0]; // cast a Pet type object to a Cat object
11        c.hello();
12        // pets[0].hello(); // this line must be commented off; pets[0] has
13                           // type Pet,
14                           // only methods listed in interface Pet can be
15                           // called from it.
```

```

16     }
17 }
18
19 interface Pet {
20     void talk();    // all pets must implement method talk()
21 }
22
23 class Cat implements Pet {
24     String name;
25
26     public Cat(String name) {
27         this.name = name;
28     }
29
30     public void talk() {    // this method is required by the interface Pet
31         System.out.println("I am cat " + name + ", meow.");
32     }
33
34     public void hello() {    // this is a method not required by interface Pet
35         System.out.println("Cat " + name + " says hello to you.");
36     }
37 }
38
39 class Dog implements Pet {
40     String name;
41
42     public Dog(String name) {
43         this.name = name;
44     }
45
46     public void talk() {    // this method is required by the interface Pet
47         System.out.println("I am dog " + name + ", wow.");
48     }
49
50     public void hello() {    // this is a method not required by interface Pet
51         System.out.println("Dog " + name + " says hello to you.");
52     }
53 }

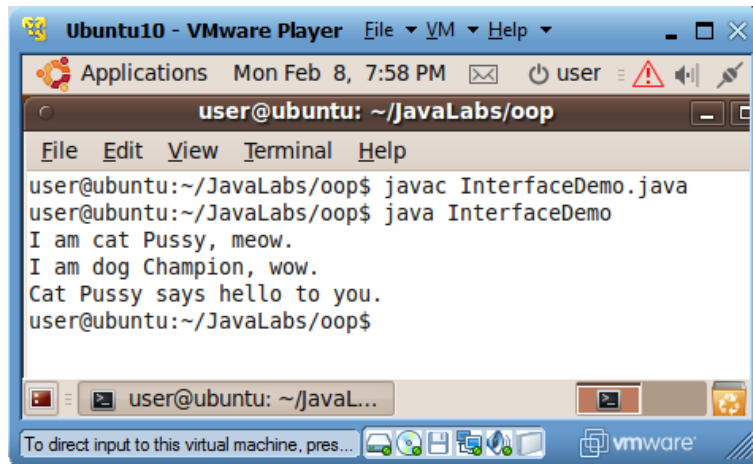
```

Explanation

- a. A Java interface is basically a list of method signatures (with return types). Lines 19-21 declare an interface that contains only one method signature. The signatures in an interface are by default public.
- b. A class implementing an interface promises that it would implement all methods in the interface. A class can implement multiple interfaces separated by commas.
- c. While an interface name cannot instantiate objects, it can be used as a data type to declare reference variables that can hold objects of any class that implements the interface. Only methods listed in the interface can be invoked against such a reference variable declared with the interface name. Line 3 declares an array of two reference variables `p[0]` and `p[1]` of type *Pet*. Line 4 can assign a *Cat* object in variable `p[0]` because class *Cat* implements interface *Pet*. Line 8 can call method “`talk()`” against variable `p[0]` because “`talk()`” is a method in interface *Pet*.
- d. Statement “`pet[0].hello()`” on line 12 cannot be executed because “`hello()`” is not a method listed in interface *Pet*.

7. Make sure your working directory is now “~/JavaLabs/oop”. To compile class *InterfaceDemo*, type
javac InterfaceDemo.java
8. To run class *InterfaceDemo*, type
java InterfaceDemo

The following is the screen capture of my test run of class *InterfaceDemo*.



```

user@ubuntu: ~/JavaLabs/oop
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/oop$ javac InterfaceDemo.java
user@ubuntu:~/JavaLabs/oop$ java InterfaceDemo
I am cat Pussy, meow.
I am dog Champion, wow.
Cat Pussy says hello to you.
user@ubuntu:~/JavaLabs/oop$

```

4.7 Java data field/method accessibility

[Use example code to explain accessibility qualifiers; to be completed later]

Type of data fields/methods to be accessed	Accessing data fields or methods from methods in				
	Same class	Unrelated class in the same package	Diff derived class in the same package	Unrelated class in diff package	Derived class in diff package
<i>public</i>	yes	yes	yes	yes	yes
<i>private</i>	yes	no	no	no	no
<i>protected</i>	yes	yes	yes	no	yes
<i>[package access]</i>	yes	yes	yes	no	no

4.8 Java collections

1. Open a terminal window. Change directory to “~/JavaLabs/misc”.
2. Use a text editor to open and review Java source code file “CollectionDemo.java” in directory “~/JavaLabs/misc” and the file has contents below:

```

1 import java.util.*;
2
3 public class CollectionDemo {

```

```

4
5 public static void main(String arg[]) {
6     CollectionDemo o = new CollectionDemo();
7     o.VectorDemo();
8     o.ArrayListDemo();
9     o.StackDemo();
10    o.HashtableDemo();
11 }
12
13 // Vector is an array of Objects with variable size.
14 // Class Vector is roughly equivalent to ArrayList, except that it is
15 // synchronized
16 // It should be used when the data could be accessed by multiple objects
17 // simultaneously
18 // Its basic operations include add(Object), add(int, Object),
19 // iterator(), remove(int), size(), toArray()
20 // Iterator is a Java interface supports methods hasNext() and next()
21 void VectorDemo() {
22     System.out.println("-----Vector demo-----");
23     // create an empty vector
24     Vector<String> vector = new Vector<String>();
25     for (int k = 0; k < 5; k++)
26         vector.add("Item " + k); // add String objects to the end of vector
27     // insert a value at position 3, values from position 3 move right
28     vector.add(3, "Inserted value");
29     Iterator i = vector.iterator(); // retrieve an Iterator for vector
30     // An Iterator is a list of references for the objects in a data
31     // structure, with a read pointer initialized to before the
32     // first value
33     while (i.hasNext()) // does i still have unvisited values?
34         System.out.println(i.next()); // if yes, retrieve the next one
35     // retrieve value at position 3
36     System.out.println("vector[3] = " + vector.get(3));
37     vector.remove(3); // remove value at position 3
38     vector.add("A"); // add "A" to the end for showing off sorting later
39     System.out.println("vector has now " + vector.size() + " elements");
40     // return list values as an array of Objects
41     Object[] array = vector.toArray();
42     Arrays.sort(array); // sort values in array
43     for (int k = 0; k < array.length; k++)
44         System.out.println("array[" + k + "] = " + array[k]);
45 }
46
47 // Class ArrayList is roughly equivalent to Vector, except that it is
48 // unsynchronized
49 // It should be used when only one object will access it a time, and
50 // efficiency is important
51 // Its basic operations include add(Object), add(int, Object),
52 // iterator(), remove(int), size(), toArray()
53 // Iterator is a Java interface supports methods hasNext() and next()
54 void ArrayListDemo() {
55     System.out.println("-----ArrayList demo-----");
56     // create an empty list
57     ArrayList<String> list = new ArrayList<String>();
58     for (int k = 0; k < 5; k++)
59         list.add("Item " + k); // add String objects to the end of the list
60     // insert a value at position 3, values from position 3 move right
61     list.add(3, "Inserted value");
62     Iterator i = list.iterator(); // retrieve an Iterator for the list

```



```

63 // An Iterator is a list of references for the objects in a data
64 // structure, with a read pointer initialized to before the first
65 // value
66 while (i.hasNext()) // does i still have unvisited values?
67     System.out.println(i.next()); // if yes, retrieve the next one
68 // retrieve value at position 3
69 System.out.println("list[3] = " + list.get(3));
70 list.remove(3); // remove value at position 3
71 list.add("A"); // add "A" to the end for showing off sorting later
72 System.out.println("list has now " + list.size() + " elements");
73 // return list values as an array of Objects
74 Object[] array = list.toArray();
75 Arrays.sort(array); // sort values in array
76 for (int k = 0; k < array.length; k++)
77     System.out.println("array[" + k + "] = " + array[k]);
78 }
79
80 // Stack is a First-In-Last-Out data structure for Objects
81 // Its most important methods are push(Object), pop(), and isEmpty()
82 void StackDemo() {
83     System.out.println("-----Stack demo-----");
84     Stack<String> s = new Stack<String>();
85     for (int i = 1; i < 5; i++) {
86         System.out.println("Pushing " + i);
87         s.push(""+i); // pushing String objects
88     }
89     while (!s.isEmpty())
90         System.out.println("popping " + s.pop());
91 }
92
93 // A hashtable maintains a set of values associated with their keys.
94 // Both keys and values must be objects of any type.
95 // To insert a (key, value) pair in the table h, use "h.put(key,
96 // value);"
97 // To retrieve value associated with key in table h, use "h.get(key)".
98 void HashtableDemo() {
99     System.out.println("-----Hashtable demo-----");
100     Hashtable<String, Integer> h = new Hashtable<String, Integer>();
101     // Count occurrence number for each string in words[]
102     // The words in words[] are used as keys, their associated values are
103     // their occurrences
104     String[] words = new String[]{"a", "b", "a", "c", "b", "c", "b"};
105     for (int i = 0; i < words.length; i++) {
106         // If a word is not in table yet, insert object 1 in the table: it
107         // has been seen once
108         // Retrieved value is always of type Object
109         Object o = h.get(words[i]);
110         if (o == null) {
111             h.put(words[i], new Integer(1));
112             continue; // resume from for loop header
113         }
114         // Otherwise, get the current count (value) for the word, increase
115         // it by 1, and save it as new value for the word
116         Integer iv = (Integer)o; // cast Object to Integer
117         int v = iv.intValue(); // get Integer's primitive int value
118         // increase value by 1, save it back in table
119         h.put(words[i], new Integer(v+1));
120     }
121     Integer iv = (Integer)h.get("b"); // retrieve count for "b"

```

```

122     System.out.println("\na\" occurred " + iv + " times");
123 }
124 }

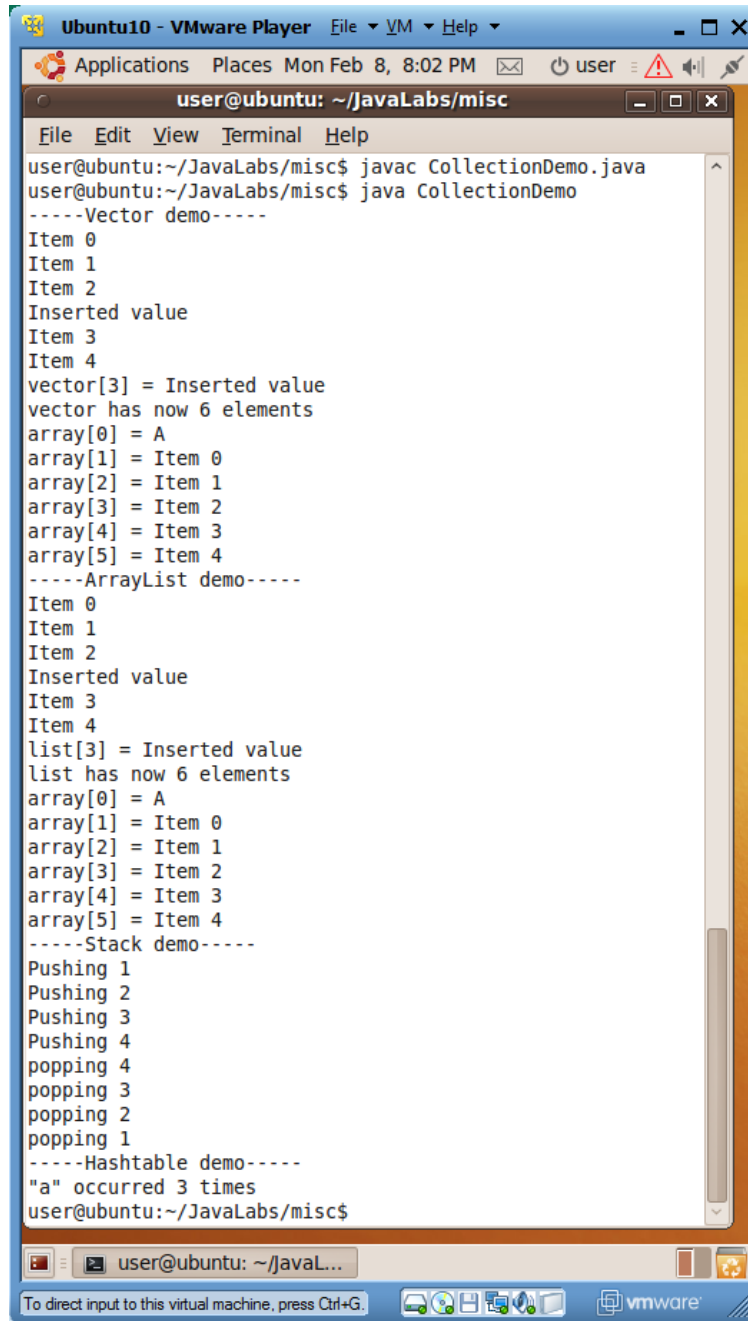
```

Explanation

- a. Line 6 creates an object of the class containing this “main()” method and save its reference in a reference variable for this class. Lines 7-10 each invokes a different method to test a different Java collection data structure.
 - b. A Java array cannot change its size during code execution. Sometime we don’t know how many data items need be maintained. A Java Vector is an array of objects with variable size. Line 24 creates an empty vector of strings. Lines 25-26 add five strings to the end of the vector. Line 28 inserts a new string as the fourth value of the vector and the old fourth value as well as all values to its right is moved right by one position to make room for the new string. An *Iterator* object maintains a pointer to the next object in a list (vector) to be accessed. This pointer is initialized to point to the first object in the list. If there is a next object to be returned, *Iterator* method “hasNext()” will return true. *Iterator* method “next()” returns the reference for the next object pointed to by the internal pointer, and moves the pointer to point to the next object in the list. Vector method “toArray()” returns a new array of objects in the vector object. Line 42 uses Java’s library method to sort the values in an array. At any instance only one vector value can be accessed. Therefore there can be no simultaneous read or write operations on a vector object. While this could reduce speed for simultaneous read operations, it avoids confusions due to simultaneous read and write operations.
 - c. Class *ArrayList* is similar to *Vector* except it allows simultaneous accesses to its values. It is more efficient for maintaining a stable list of objects.
 - d. A stack is a list of objects supporting three operations: “push(Object)”, “pop()” and “isEmpty()”. The first object pushed into the stack will be the last to be popped out.
 - e. A hash table is a table with two columns. The first column contains objects working as keys, and the second column contains objects working as values associated with the corresponding key values. Operation “put(key, value)” inserts a new pair of (key, value) into the hash table if the table does not have the key yet. Otherwise the new value will replace the old value associated with the specified key. Operation “get(key)” returns the value associated with “key” if the (key, value) pair is found in the hash table, or *null* (nothing) if the key cannot be found in the hash table.
 - f. Line 104 creates an array *words* of strings large enough to hold the string constants specified in the pair of curly braces, and initializes the array with those strings.
 - g. Lines 100-122 uses a hash table to find out the number of occurrence of each string in array *words*. When a string is not found in the hash table, the string and a new *Integer* object for 1 is inserted into the hash table as a (key, value) pair. If a string is found in the hash table, its associated integer value is increased by 1 and put back into the table as the new value associated with the string key.
3. Make sure your working directory is now “~/JavaLabs/misc”. To compile class *CollectionDemo*, type
javac CollectionDemo.java
 4. To run class *CollectionDemo*, type

java CollectionDemo

The following is the screen capture of my test run of class *CollectionDemo*.



```
user@ubuntu: ~/JavaLabs/misc
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/misc$ javac CollectionDemo.java
user@ubuntu:~/JavaLabs/misc$ java CollectionDemo
-----Vector demo-----
Item 0
Item 1
Item 2
Inserted value
Item 3
Item 4
vector[3] = Inserted value
vector has now 6 elements
array[0] = A
array[1] = Item 0
array[2] = Item 1
array[3] = Item 2
array[4] = Item 3
array[5] = Item 4
-----ArrayList demo-----
Item 0
Item 1
Item 2
Inserted value
Item 3
Item 4
list[3] = Inserted value
list has now 6 elements
array[0] = A
array[1] = Item 0
array[2] = Item 1
array[3] = Item 2
array[4] = Item 3
array[5] = Item 4
-----Stack demo-----
Pushing 1
Pushing 2
Pushing 3
Pushing 4
popping 4
popping 3
popping 2
popping 1
-----Hashtable demo-----
"a" occurred 3 times
user@ubuntu:~/JavaLabs/misc$
```

4.9 String tokenizers

Many applications need to split a string into tokens separated by some special delimiters which are normally white-space characters (Tab, space and newline characters). The following program shows two approaches for splitting a string into tokens defined as the longest substrings that don't contain white-space characters.

1. Open a terminal window. Change directory to “~/JavaLabs/misc”.

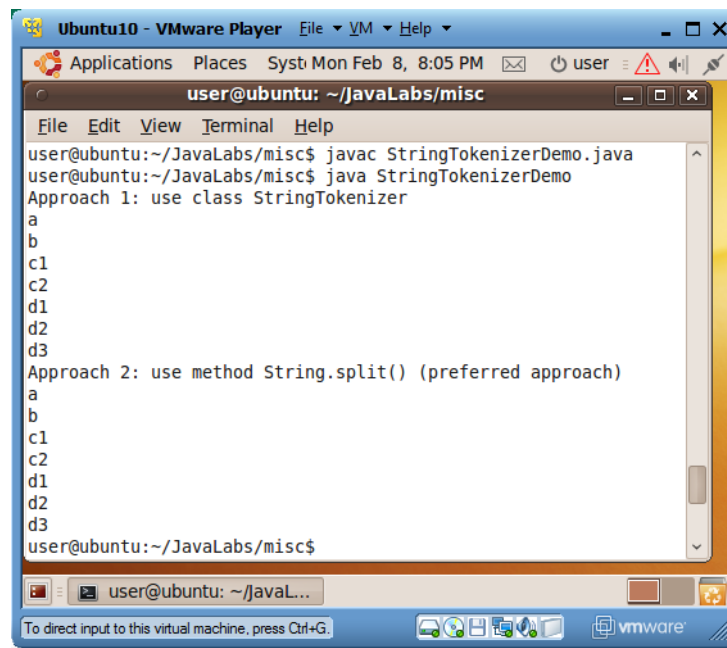
2. Use a text editor to open and review Java source code file “StringTokenizerDemo.java” in directory “~/JavaLabs/misc” and the file has contents below:

```
1 import java.util.StringTokenizer;
2
3 public class StringTokenizerDemo {
4
5     public static void main(String args[]) {
6         String words = "a b c1 c2 d1 d2 d3";
7         System.out.println("Approach 1: use class StringTokenizer");
8         StringTokenizer st = new StringTokenizer(words);
9         // A StringTokenizer allows you read tokens from left to right, one a
10        // time
11        while (st.hasMoreTokens()) { // Do we have more tokens to return?
12            // return and print the next token in the string
13            System.out.println(st.nextToken());
14        }
15        System.out.println("Approach 2: use method String.split() (preferred
16        approach)");
17        // "\s" is the regular expression for white space; "\\s" is used
18        // because "\" is
19        // an escape character.
20        // change "\\s" to change the separation character between tokens
21        String[] token = words.split("\\s+");
22        for (int i = 0; i < token.length; i++)
23            System.out.println(token[i]);
24    }
```

Explanation

- a. Line 6 defines a string *words* containing seven tokens defined by white-space characters.
 - b. Line 8 creates a new *StringTokenizer* object for string *words*, which has an internal pointer to the next token to be returned. This pointer is initialized to point to the first token. *StringTokenizer*’s method “hasMoreTokens()” returns true if and only if there are more tokens to return. *StringTokenizer*’s method “nextToken()” returns the next token and moves the internal pointer to point to the next token.
 - c. Line 20 is a much simpler approach to tokenize a string. Class *String* has a method “split(*delimiter*)” to return an array of token strings defined by the value of the *delimiter* argument. Since “\s+” represents any sequence of white-space characters, “words.split(“\s+”)” returns an array of tokens delimited by white-space characters.
3. Make sure your working directory is now “~/JavaLabs/misc”. To compile class **StringTokenizerDemo**, type
javac StringTokenizerDemo.java
 4. To run class *StringTokenizerDemo*, type
java StringTokenizerDemo

The following is the screen capture of my test run of class *StringTokenizerDemo*.



4.10 Exception handling

1. Open a terminal window. Change directory to “~/JavaLabs/basics”.
2. Use a text editor to open and review Java source code file “ShowException.java” in directory “~/JavaLabs/basics” and the file has contents below:

```
1 class Exception1 extends Exception {}
2 class Exception2 extends Exception {}
3 class Exception3 extends Exception {}
4
5 public class ShowException {
6     public static void main(String[] args) {
7         for (int i = 1; i<=3; i++) {
8             try {
9                 f(i);
10            }
11            catch (Exception3 e) {
12                System.out.println("main() caught " + e);
13            }
14        }
15    }
16
17    static void f(int i) throws Exception3 {
18        try {
19            if (i == 1)
20                throw new Exception1();
21            else if (i == 2)
22                throw new Exception2();
23            else if (i == 3)
24                throw new Exception3();
25        }
26        catch (Exception1 e) {
27            System.out.println("f(" + i + ") caught " + e);
28        }
29        catch (Exception2 e) {
30            System.out.println("f(" + i + ") caught " + e);
31        }
32    }
33 }
```

```

31     throw new Exception3();
32 }
33 finally {
34     System.out.println("f(" + i + ") executes finally block");
35 }
36 }
37 }

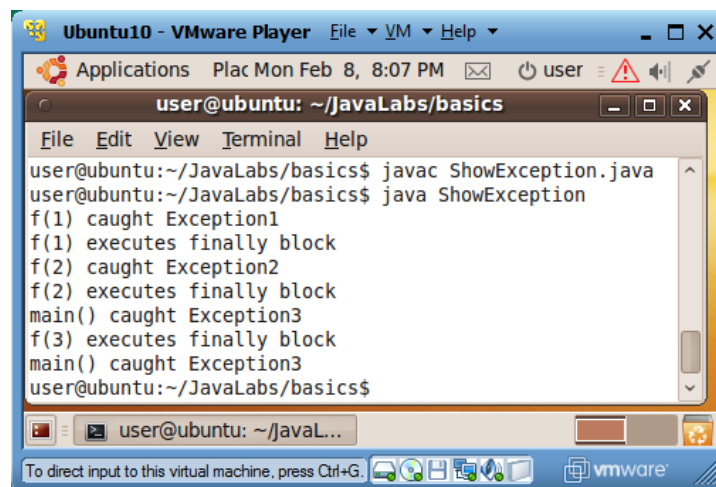
```

Explanation

- a. Lines 1-2 declare three subclasses for class *Exception*. Even though they don't declare any data members or methods, their names can pass information on what caused the exceptions.
- b. Lines 20, 22, 24 and 31 explicitly create exception objects and throw them back to the method "main()" that invoked method f() which contains these throw statements. Many Java operations can implicitly throw exception objects to their invokers. Examples for such implicitly thrown exception objects include divide-by-zero, file-not-found, and network-connection-fail.
- c. Lines 18-35 provide an example for a try-catch-finally statement. Statements that could throw exceptions are put in a *try block* (lines 18-25). If there is no exceptions during the execution of the try block, the code in the finally block will be executed, and then the execution resumes after the finally block. If an exception happens during the execution of this try block, the rest code lines in the try block will not be further executed, and the execution goes to the first catch block (lines 26-28). If the exception object thrown has type *Exception1*, its reference will be copied into parameter *e*, and the body of the catch block (line 27) will be executed, then the body of the *finally block* (lines 33-35) will be executed, and then the execution resumes after the finally block. If the exception object thrown does not have type *Exception1*, it will be compared with *Exception2* in the next catch block. If there is a type match, the code in the second catch block (line 27) will be executed, then the finally block will be executed, and then the execution resumes after the finally block. If the thrown exception does not match any of the exception types in the catch blocks, then the finally block will be executed, the current method "f()" will terminate, and the exception object will be thrown back to the line of code that called this method.
- d. If a method received an exception object without catching it, the method execution will terminate and the exception object will be further thrown back to the method that invoked this method. If the "main()" method receives an exception object without catching it, the program execution will terminate with error messages.
- e. The finally block is for specifying code that will be executed no matter whether the exceptions happen or not. The finally block is generally used for releasing resources claimed in the try block.
- f. Lines 29-32 shows an example in which an exception is partially processed and then the control goes back to the invoker of the method for further exception processing.
- g. If an exception object may be thrown from a method and there is no catch block for catching that type of exception in the method, the method signature needs to add a "throws" clause, as you see on line 17, to acknowledge that you are aware of this situation. The "throws Exception3" clause is needed for method "f(i)" because *Exception3* objects could be thrown from the method to its invoker and the method has no catch block to catch it.

- h. During the first iteration of the *for* loop of method “main()”, f(1) is invoked in the try block (line 9). Due to code on lines 19-20, an *Exception1* object is thrown. This object is caught by the catch block on line 26-28, and its body, line 27, is executed. Then the body of the finally block is executed, and the method call “f(1)” returns normally to line 10. The *for* loop then starts the next iteration.
 - i. During the second iteration of the *for* loop of method “main()”, f(2) is invoked in the try block (line 9). Due to code on lines 21-22, an *Exception2* object is thrown. This object is caught by the catch block on line 29-32. A message is printed by line 30, and a new *Exception3* object is created and thrown out by line 31. Since the *Exception3* object is not in a try block, it is not caught inside the f(i) method, and it needs be thrown back to the invoker of f(2). But before the execution leaves f(i), the finally block is executed, and then the method call to f(2) terminates, and the *Exception3* object is thrown in the try block on line 9. This object is caught by the catch block on lines 11-13, and a message is printed by line 12. Then the next iteration of the *for* loop starts.
 - j. During the third iteration of the *for* loop of method “main()”, f(3) is invoked in the try block (line 9). Due to code on lines 23-24, an *Exception3* object is thrown. Since this object is not caught in the f(i) method, the current method needs terminate its execution and the *Exception3* object needs be thrown back to line 9. But before the execution leaves f(3), the finally block is executed. Then the *Exception3* object is caught by the catch block on line 11-13, and a message is printed. Now the *for* loop terminates, the “main()” method terminates, and the program terminates.
- 3. Make sure your working directory is now “~/JavaLabs/basics”. To compile class *ShowException*, type
javac ShowException.java
 - 4. To run class *ShowException*, type
java ShowException

The following is the screen capture of my test run of class *ShowException*.



```
Ubuntu10 - VMware Player  File VM Help
Applications  Plac Mon Feb 8, 8:07 PM  user
user@ubuntu: ~/JavaLabs/basics
File Edit View Terminal Help
user@ubuntu:~/JavaLabs/basics$ javac ShowException.java
user@ubuntu:~/JavaLabs/basics$ java ShowException
f(1) caught Exception1
f(1) executes finally block
f(2) caught Exception2
f(2) executes finally block
main() caught Exception3
f(3) executes finally block
main() caught Exception3
user@ubuntu:~/JavaLabs/basics$
```


5 Multi-threading

A CPU has only one program counter PC for specifying next instruction (statement) to run so it can only run one program a time. But most of today's computer applications need to have multiple programs to run at the same time. For example, when you are editing a document, you may want to keep communicating with your friends with emails or social networks. If your computer has multiple CPUs, each of the CPUs could run a separate program and we say that the programs run in *parallel*. If your computer has less CPUs than the number of programs that need to run together, you could let multiple programs time-share a CPU and provide the illusion that they run at the same time. For example, if you need to run ten programs at the same time on a single CPU, you could let each program take turns (round robin) to run on the CPU for 1/10 of a second (a CPU time slice) so we get the illusion that each program runs on its own virtual CPU running at a speed 1/10 of the physical CPU. When parallelism is implemented by such emulation, we say that the programs run *concurrently*. Linux and Windows are both time-sharing operating systems supporting the concurrent execution of multiple programs.

The time slice size for each program to run (I used 1/10 of a second in the last example, actually it is much smaller) in round robin has important impacts on system efficiency and quality of concurrent execution. If the time slice is too small, there would be too much overhead for the CPU (program counter and general-purpose registers), memory and disk to swap code and data for different programs (context switching). This would also happen if you run too many programs at the same time: you may find your disk busy swapping (*thrashing*) data with the memory and the programs not responding. At this time you should stop some less important programs so the other programs could have more CPU time and resources to complete their execution. If the time slice is too large, you would notice that the programs were not running concurrently.

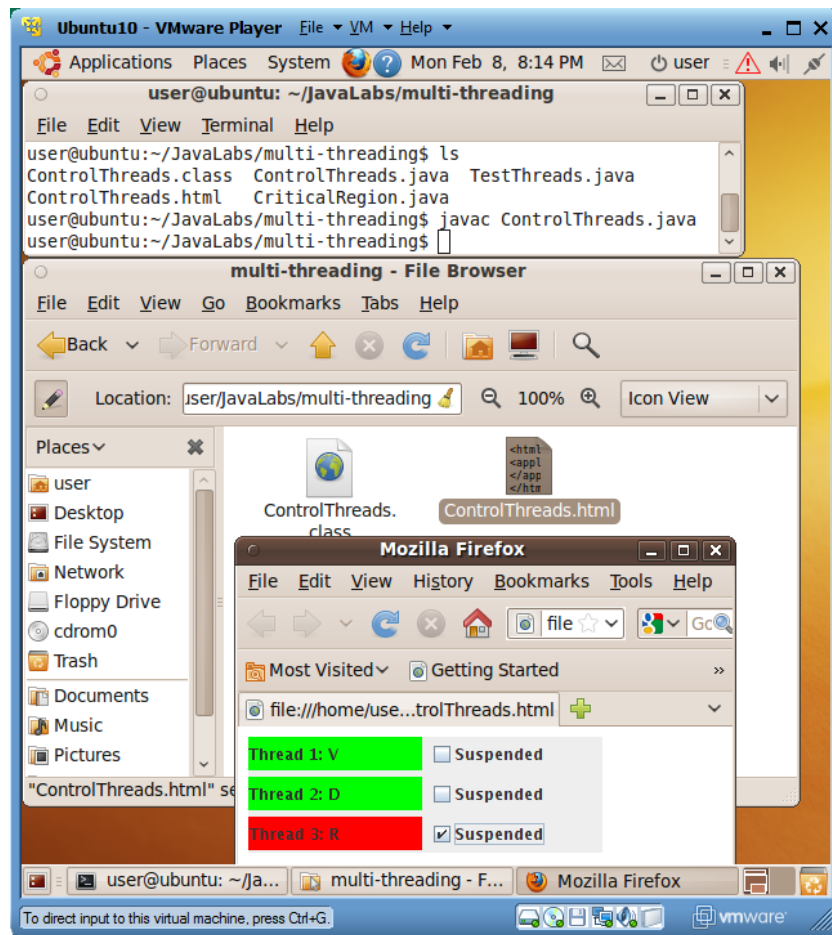
Modern operating systems organize computation tasks into processes and threads. Each program normally runs in its own process, which contains one or more threads. Thread is a unit for an OS to allocate CPU time and resources (like memory and disk space), while a thread is a program in execution. A thread is represented by its own program counter value for the next instruction that the thread should run next and well as a copy of the data in the general-purpose registers before the thread left the CPU. When a thread gets its turn to run on the shared CPU, its program counter value and register values would first be copied into the CPU PC and registers before its execution resumes. When the time slice of the thread is up, the current values of the PC and other registers would be copied back to the thread's memory table for its next turn.

When a Java program starts to run, a new process is generated with its associated system resources (at least memory and disk space for its code and data), and a new thread is then created inside the process to run the *main()* method of the program. In the body of the method, new threads could be spawned to run concurrently. All threads in the same process can share data in the process, while threads of different processes cannot share data.

The thread mechanism was first introduced in 1980s. Since then Windows, UNIX and Linux all support the threads under a process. The older web servers used CGI (Common Gateway Interface) technologies that uses separate processes to generate HTML files (web pages) for different client requests or visits to the servers, but the newer Java and C# based web server technologies use the faster threads to generate the HTML files.

5.1 Controlling multi-threads in action

1. Open a terminal window. Make directory “~/JavaLabs/multi-threading” to be the working directory.
2. Compile class *ControlThreads* by typing
javac ControlThreads.java
3. In a file explorer, double click on “~/JavaLabs/multi-threading/ControlThreads.html”. Accept any web browser warning for running Java applet or active code. You will now see a screen similar to the following one. There are three threads, each printing random letters. You can check/uncheck each thread to pause or resume its execution.



5.2 Two ways of creating threads

4. Open a terminal window. Create a new folder “~/JavaLabs/multi-threading” and change directory to it.
5. Use a text editor to open and review Java source code file “TestThreads.java” in directory “~/JavaLabs/ multi-threading” and the file has contents below:

```
1 public class TestThreads {  
2     public static void main(String[] args) {  
3         // Create threads
```

```

4     PrintChar printA = new PrintChar('a', 100);
5     PrintChar printB = new PrintChar('b', 100);
6     Thread print100 = new Thread(new PrintNum(100));
7     // Start threads
8     print100.start();
9     printA.start();
10    printB.start();
11 }
12 }
13
14 class PrintChar extends Thread {
15     private char charToPrint; // The character to print
16     private int times; // The times to repeat
17
18     public PrintChar(char c, int t) {
19         charToPrint = c;
20         times = t;
21     }
22
23     public void run() {
24         for (int i=0; i < times; i++) {
25             System.out.print(charToPrint);
26             try { Thread.sleep(5); } catch(Exception e){}
27         }
28     }
29 }
30
31 class PrintNum implements Runnable {
32     private int lastNum;
33
34     // Construct a thread for print 1, 2, ... i
35     public PrintNum(int n) {
36         lastNum = n;
37     }
38
39     public void run() {
40         for (int i=1; i <= lastNum; i++) {
41             System.out.print(" " + i);
42             try { Thread.sleep(5); } catch(Exception e){}
43         }
44     }
45 }

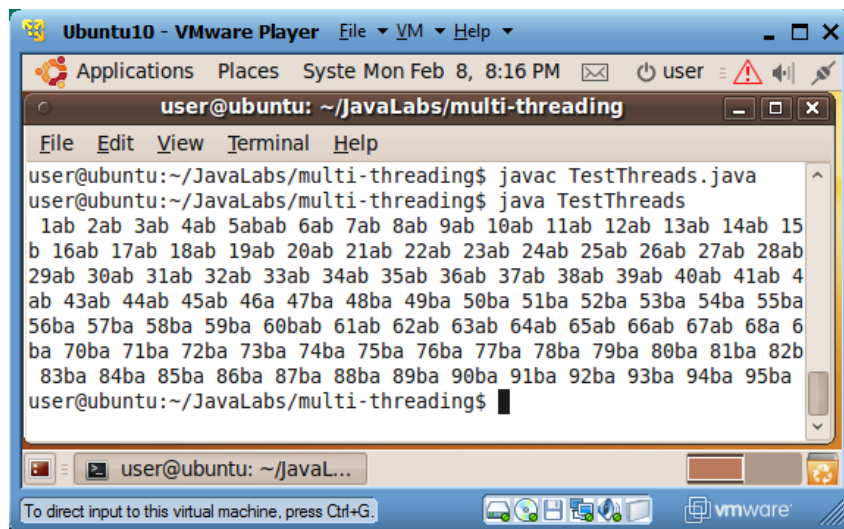
```

Explanation

- a. The purpose of this program is to create three new threads to run concurrently. The first thread prints “a” 100 times. The second thread prints “b” 100 times. The third thread prints numbers 1, 2, ..., 100.
- b. There are two approaches to specify code to run in new threads.
 - i. Create a class that extends Java library class *Thread*, specify the code to run in a method with signature “public void run()”, instantiate an object from the class, and invoke the object’s method “start()” to allocate resources for the new thread and execute the “run()” method of the object in the new thread. In class *TestThreads*, the thread for printing characters, *PrintChar*, was created using this approach.

- ii. Create a class that implements Java library interface *Runnable*, specify the code to run in a method with signature “public void run()”, instantiate an object, say *obj1*, of the new class, instantiate an object of class *Thread* with the last object *obj1* as argument to the class constructor, and invoke the *Thread* object’s method “start()” to allocate resources for the new thread and execute the “run()” method of the object in the new thread. In class *TestThreads*, the thread for printing numbers, *PrintNum*, was created using this approach.
 - c. Class *PrintNum* is used to print positive integers 1, 2, ... in a new thread, and the last number to be printed is specified by the argument of the class constructor. Code “Thread.sleep(5);” in the *for* loop of the *run()* method is for letting the thread sleep for 5 milliseconds after printing each number. This extra delay is for limiting the number of iterations of the loop that can be completed during a single time slice for the thread. You could change the argument 5 to a different number to see the effect of the delay.
 - d. Class *PrintChar* is used to print a character a fixed number of times, both specified by the class constructor’s arguments, in a new thread.
 - e. When this program executes, the first thread runs the *main()* method of class *TestThreads*. When the three objects of class *PrintChar* and *PrintNum* are created, there is still one thread running. The last three lines of method *main()* create and run the three new threads. Immediately after “printB.start();” is executed, there are four threads running at the same time.
6. Make sure your working directory is now “~/JavaLabs/multi-threading”. To compile class *TestThreads*, type
javac TestThreads.java
 7. To run class *TestThreads*, type
java TestThreads

The following is the screen capture of my test run of class *TestThreads*.



```
Ubuntu10 - VMware Player  File  VM  Help
Applications  Places  Sys... Mon Feb 8, 8:16 PM  user
user@ubuntu: ~/JavaLabs/multi-threading
File  Edit  View  Terminal  Help
user@ubuntu:~/JavaLabs/multi-threading$ javac TestThreads.java
user@ubuntu:~/JavaLabs/multi-threading$ java TestThreads
1ab 2ab 3ab 4ab 5abab 6ab 7ab 8ab 9ab 10ab 11ab 12ab 13ab 14ab 15
b 16ab 17ab 18ab 19ab 20ab 21ab 22ab 23ab 24ab 25ab 26ab 27ab 28ab
29ab 30ab 31ab 32ab 33ab 34ab 35ab 36ab 37ab 38ab 39ab 40ab 41ab 4
ab 43ab 44ab 45ab 46a 47ba 48ba 49ba 50ba 51ba 52ba 53ba 54ba 55ba
56ba 57ba 58ba 59ba 60bab 61ab 62ab 63ab 64ab 65ab 66ab 67ab 68a 6
ba 70ba 71ba 72ba 73ba 74ba 75ba 76ba 77ba 78ba 79ba 80ba 81ba 82b
83ba 84ba 85ba 86ba 87ba 88ba 89ba 90ba 91ba 92ba 93ba 94ba 95ba
user@ubuntu:~/JavaLabs/multi-threading$
```

5.3 Resolving critical region problem with thread synchronization

1. Open a terminal window. Change working directory to “~/JavaLabs/multi-threading”.
2. Use a text editor to open and review Java source code file “CriticalRegion.java” in directory “~/JavaLabs/multi-threading” and the file has contents below:

```
1 class BankAccount {
2     private int balance = 0;
3     public int getBalance() { return balance; }
4     public void setBalance(int balance) { this.balance = balance; }
5 }
6
7 public class CriticalRegion {
8     private BankAccount account = new BankAccount();
9     private Thread[] thread = new Thread[100];
10
11     public static void main(String[] args) {
12         CriticalRegion test = new CriticalRegion();
13         System.out.println("Final balance is " + test.account.getBalance());
14     }
15
16     public CriticalRegion() {
17         // generate a new thread group
18         ThreadGroup g = new ThreadGroup("group");
19         boolean done = false;
20
21         for (int i = 0; i < 100; i++) {
22             // create a new thread and add the thread to thread group g
23             thread[i] = new Thread(g, new AddADollarThread());
24             thread[i].start();
25         }
26         // wait for all the 100 new threads terminate
27         while (!done) {
28             if (g.activeCount() == 0) // if there are active threads, continue
29                 done = true;         // looping
30         }
31     }
32
33     /* synchronized */ void deposit(int sum) {
34         int newBalance = account.getBalance(); // retrieve the old balance
35         newBalance += sum; // add sum to the old balance
36
37         try {
38             Thread.sleep(1); // purposely add a delay before writing
39                             // back balance
40         }
41         catch (InterruptedException e) { // a sleeping thread could be
42             System.out.println(e);      // interrupted by another thread
43         }
44         account.setBalance(newBalance);
45     }
46
47     class AddADollarThread extends Thread {
48         public void run() {
49             deposit(1);
50         }
51     }
52 }
```

Explanation

- a. If a data region needs be written into and read from by multiple threads or processes concurrently, then there is the possibility that the data writes interfere with each other and the data reads cannot retrieve the correct updated values. This is the *critical region* problem important in operating system design and most multi-threading programs.
- b. This program lets 100 threads each deposit \$1 into an initially empty bank account. If we don't take action to address the critical region problem, the account could end up with a balance of 1-2 dollars. After we address the critical region problem with Java's *synchronized* modifier, the final bank balance would be the correct \$100.
- c. Lines 1-5 declare a simple bank account with the initial balance of zero.
- d. Line 8 creates a *BankAccount* object. Since 100 threads would deposit money into it concurrently, it is a critical region.
- e. Line 9 creates an array of 100 reference variables for *Thread* objects.
- f. Most of the computation is conducted in the constructor of class *CriticalRegion*.
- g. Class *AddADollarThread* is an inner class nested inside class *CriticalRegion*. It is justifiable because it is a utility class for the latter and is not supposed to be used by other classes. Since this class inherits class *Thread*, its method "run()" specifies what will run in a new thread based on this class: depositing \$1 in the bank account.
- h. In method "deposit(sum)", the current balance of the account is first retrieved and saved in variable *newBalance*; the amount to deposit, *sum*, is then added to variable *newBalance*. Line 44 saves the new balance back into the account. Think about the worst case. All 100 threads start to run lines 33-35 and pause because their time slices for using the CPU are up. Since no thread has executed line 44 to modify the account balance yet, all the threads will have retrieved account balance zero, and set the value of variable *newBalance* to one. When each of thread gets the CPU again and resume to execute line 44, each will write one into the account balance leading to the final balance of \$1 in the account. To increase the chance for such worst case happens, we inserted code lines 37-43 to let each thread to wait for one millisecond before it executes line 44. If there is no such a delay, the worst case could still happen but with a very small probability. You may try to resolve this problem by rewriting the code so retrieving account balance and writing the updated balance back into the account happen in a single Java statement. This will not work since the single line of Java code will be compiled into a long sequence of machine instructions and the operating systems schedule the computation at the machine instruction level.
- i. There are two solutions to the above critical region problem. One is to qualify method "deposit(sum)" with Java modifier "synchronized". This modifier will make sure that at any time no more than one thread can run inside the method. If the method is big and only few lines of the method body could cause the critical region problem, the above solution of synchronizing the entire method would unnecessarily serialize the execution and degrade system performance. To maximize concurrent execution for improved performance, we could use the approach 2: using the synchronized block to only synchronize the few lines of a method that could lead to the critical region problem. The syntax for the synchronized block is as follows:

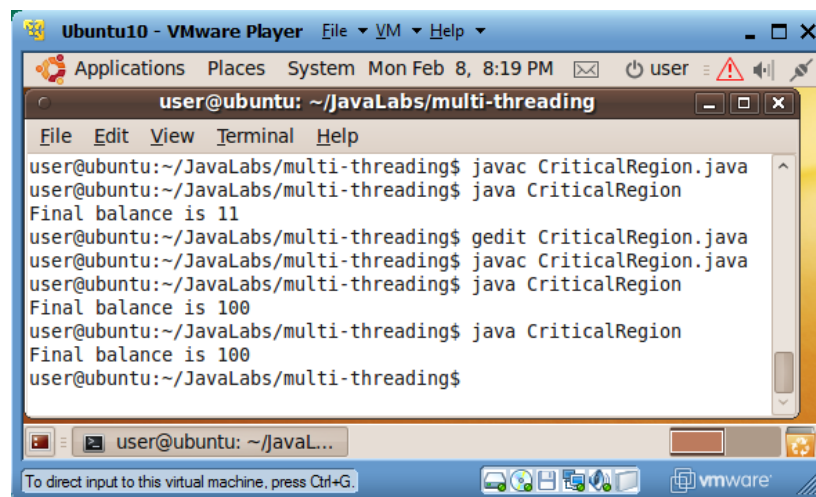
```
synchronized (this) {  
    // any statements that could cause the
```

```
// critical region problem
}
```

The “this” in the above structure could be any Java object, and we just used the current object containing this code.

- j. Line 18 creates a thread group with name “group”. When we create a thread, we can add the thread in a thread group, as we do on line 23. We add the 100 threads in the same thread group because class *ThreadGroup* supports the method “activeCount()” reporting how many threads in the group are still alive. On lines 27-30, we keep invoking method “activeCount()” and we terminate the execution of the class constructor when all the 100 *AddADollarThread* threads have terminated.
3. Make sure your working directory is now “~/JavaLabs/multi-threading”. To compile class *CriticalRegion*, type
javac CriticalRegion.java
4. To run class *CriticalRegion*, type
java CriticalRegion
You will find that the balance is a few dollars, much less than the total deposited \$100.
5. Use the text editor to uncomment “synchronized” on line 78 so it reads “synchronized void deposit(int sum) {”}. Repeat steps 3 and 4 and recompile and run the code. Because now more than one thread can deposit into the account a time, the account will display the correct balance of \$100.

The following is the screen capture of my test runs of class *CriticalRegion*.



```
Ubuntu10 - VMware Player  File  VM  Help
Applications  Places  System  Mon Feb 8, 8:19 PM  user
user@ubuntu: ~/JavaLabs/multi-threading
File  Edit  View  Terminal  Help
user@ubuntu:~/JavaLabs/multi-threading$ javac CriticalRegion.java
user@ubuntu:~/JavaLabs/multi-threading$ java CriticalRegion
Final balance is 11
user@ubuntu:~/JavaLabs/multi-threading$ gedit CriticalRegion.java
user@ubuntu:~/JavaLabs/multi-threading$ javac CriticalRegion.java
user@ubuntu:~/JavaLabs/multi-threading$ java CriticalRegion
Final balance is 100
user@ubuntu:~/JavaLabs/multi-threading$ java CriticalRegion
Final balance is 100
user@ubuntu:~/JavaLabs/multi-threading$
```

6 Networking

Each server computer has an IP address, like 198.105.44.27, as its unique identifier on the Internet. Domain names, like *www.pace.edu*, are used as user friendly identifications of server computers, and they are mapped to IP addresses by a Domain Name Server (DNS). There is a special domain name “localhost” that is normally defined as an alias of local IP address 127.0.0.1. Domain name “localhost” and IP address 127.0.0.1 are for addressing a local computer, very useful for testing server applications where the server application and client application are running on the same computer.

A server computer may run multiple server programs including web and database servers. Each server program on a server computer uses a port number, between 0 and 65535, unique on the server machine as its local identification (by default a web server uses port 80). Port numbers less than 1000 are mainly reserved for standard server applications.

Server sockets and *sockets* are the abstractions of TCP/IP communication endpoints. Each server runs a single server socket object to wait for client programs to connect to it. In Java, code

```
ServerSocket serverSocket = new ServerSocket(8000);  
Socket connectToClient = serverSocket.accept();
```

create a server socket object and keeps listening to any request for a client program to connect to it using a code line like

```
Socket connectToServer = new Socket(localhost, 8000);
```

where “localhost” should be replaced by the proper IP address or domain name for the server computer on which the server program is running (since we run both the server and client programs on the same computer, we will just use “localhost”), and 8000 should be the same port number that the *ServerSocket* constructor uses to listen to client connection requests. If there is no such a client connection request, “new ServerSocket(8000)” will wait and not continue its execution. Upon a client request, “new ServerSocket(8000)” will return a new *Socket* object that is connected to the new *Socket* object created by “new Socket(localhost, 8000)” running on the client computer. From then on the server and client applications communicate through their respective local *Socket* objects. Each of the communicating parties can derive an input stream and an output stream from its local *Socket* object. To send data to its communication partner, a program just needs to print the data to the output stream derived from its local *Socket* object. To receive input from its communicating partner, a program just needs to read data from the input stream derived from its local *Socket* object. From then on networking is similar to reading/writing data from/into data files.

6.1 A client-server program for calculating areas of circles

1. Open a terminal window. Change directory to “~/JavaLabs/networking”.
2. Use a text editor to open and review Java source code file “AreaServer.java” in directory “~/JavaLabs/networking” and the file has contents below:

```
1 import java.io.*;  
2 import java.net.*;
```



```

3 import java.util.*;
4
5 public class AreaServer {
6
7     // maximum number of clients the server can serve simultaneously
8     static final int MAX_THREAD_LIMIT = 10;
9
10    public static void main(String[] args) {
11        int portNbr = 8000; // default port number
12        if (args.length > 0) { // user provides port number; adopt it
13            try {
14                portNbr = Integer.parseInt(args[0]);
15            } catch (Exception e) {
16                System.out.println("Usage: java AreaServer [portNbr]");
17            }
18        }
19        try {
20            // Create a server socket at the specified port
21            ServerSocket serverSocket = new ServerSocket(portNbr);
22            System.out.println("Server is running at port number " + portNbr);
23            // Create a thread group to hold all active threads serving the
24            // clients
25            ThreadGroup g = new ThreadGroup("serverThreads");
26            // As long as there are less than MAX_THREAD_LIMIT active threads,
27            // wait for new client service requests;
28            // otherwise sleep for 1000 ms and check again
29            while (true) {
30                if (g.activeCount() >= MAX_THREAD_LIMIT) { // thread number limit
31                    try { // reached
32                        Thread.sleep(1000); // 1000 can be adjusted; sleep for 1000 ms
33                    } catch (InterruptedException e) {}
34                    // resume the while loop from beginning to check number of
35                    // active threads again
36                    continue;
37                }
38                // Start listening for connections on the server socket
39                Socket connectToClient = serverSocket.accept();
40                // If execution reaches here, accept() has returned by responding
41                // to a client request to create a socket to connect to this
42                // server
43                // connectToClient holds a new socket responsible for
44                // communicating with the corresponding socket on the client side
45                Thread thread = new Thread(g, new Server(connectToClient));
46                thread.start(); // launch a new thread dedicated to serve the new
47                             // client
48            }
49        } catch (Exception e) {
50            System.err.println("Server launch failed");
51            System.err.println(e);
52        }
53        System.exit(0);
54    }
55 }
56
57 // Each new thread running class Server serves one client
58 class Server extends Thread {
59     Socket connectToClient;
60
61

```



```

62 public Server(Socket connectToClient) {
63     this.connectToClient = connectToClient;
64 }
65
66 public void run() {
67     PrintWriter osToClient; // output stream for sending data to client
68     BufferedReader isFromClient; // input stream for data from client
69     try {
70         // Create a buffered reader stream to get data from the client
71         isFromClient = new BufferedReader(new
72             InputStreamReader(connectToClient.getInputStream()));
73         // Create a buffered writer stream to send data to the client
74         osToClient = new PrintWriter(connectToClient.getOutputStream(),
75             true);
76     } catch (IOException e) { // networking error
77         System.err.println(
78             "Networking error, shutting down serving thread");
79         return;
80     }
81     // Continuously read from the client and process it,
82     // and send result back to the client
83     while (true) {
84         String[] tokens;
85         // Read a line and split it into array of tokens
86         try {
87             tokens = isFromClient.readLine().split("\\s+");
88         } catch (IOException e) {
89             System.err.println(
90                 "Networking error, shutting down serving thread");
91             return;
92         }
93         // convert the first token into a double value
94         double radius;
95         try {
96             radius = Double.parseDouble(tokens[0]);
97         } catch (Exception e) { // Value from client is not a number
98             System.out.println("Client sent illegal value: " + tokens[0]);
99             return; // exit from this thread; stop serving this client
100         }
101         // Compute area
102         double area = radius*radius*Math.PI;
103         // Send the result to the client
104         osToClient.printf("%6.2f\n", area);
105         // Print radius and result to the console
106         System.out.printf(
107             "Radius (from client): %6.2f; Area found: %6.2f\n", radius, area);
108     }
109 }
110 }

```

Explanation

- a. Class *AreaServer* implements a server program that keep waiting for a radius value for a circle from a client program, calculating the area for the circle, and sending back the area value back to the client program.

- b. By default the server runs on port 8000. You can use command line argument to specify a different port number. If you do so, make sure you provide the same port number as command line argument when you run the *AreaClient* class.
 - c. Line 21 creates a *ServerSocket* object for the specified port number.
 - d. Line 25 creates a *ThreadGroup* object so later we can insert all threads serving clients in this thread group. We need this *ThreadGroup* object because on line 30 we need to know how many threads are serving the clients (“g.activeCount()”). We set an upper limit for how many clients that we can serve at the same time so the server would not run out of resources.
 - e. The server mainly runs a while loop (lines 29-48). Lines 30-37 make sure that the server will serve no more than ten clients a time. If the limit has reached, the current thread is put to sleep for one second and then check for the client number again. The execution of line 39 will be suspended until a client requests connection to this server application by creating a *Socket* object with the IP address or domain name for the server computer and the port number used by the *ServerSocket* object. When this request arrives, the client-side *Socket* objects is creates, and the server-side *Socket* object is also created as return value of “serverSocket.accept()”. These two sockets are now hooked up for later client-server communication.
 - f. Each client will be served by a new thread whose logics are specified by the “run()” method of class *Server*. Line 45 create a new *Server* object initialized with the local *Socket* object connecting to the current client, and add this object in the thread group “serverThreads”. Line 46 creates the new thread and uses it to run the “run()” method of class *Server*.
 - g. Lines 71-72 derive an input stream from the socket object connected to a client-side socket. The more primitive input stream objects are wrapped in more advanced input stream objects to access the more advanced data reading operations.
 - h. Lines 74-75 derive an output stream from the socket object connected to a client-side socket. The more primitive output stream objects are wrapped in more advanced output stream objects to access the more advanced data printing operations.
 - i. The while loop on lines 83-108 repeatedly serves its client’s requests for calculating the area of circles. During each loop iteration, the code tries to read a double radius value from the socket’s input stream. If there is no value from the client side, the “readline()” operation on line 87 will wait. The string returned by “readline()” is split into tokens delimited by white-space characters, and the first token is converted to a double value. Then the area for the received radius is calculated, and the value is printed into the output stream of the socket as well as to the screen for debugging purpose.
3. Use a text editor to open and review Java source code file “AreaClient.java” in directory “~/JavaLabs/networking” and the file has contents below:

```

111 import java.io.*;
112 import java.net.*;
113 import java.util.*;
114 import javax.swing.*;
115
116 // Usage: java AreaClient [serverURL [portNbr]]
117 // If your provide port number, you must also provide server URL
118 public class AreaClient {
119     public static void main(String[] args) {

```

```

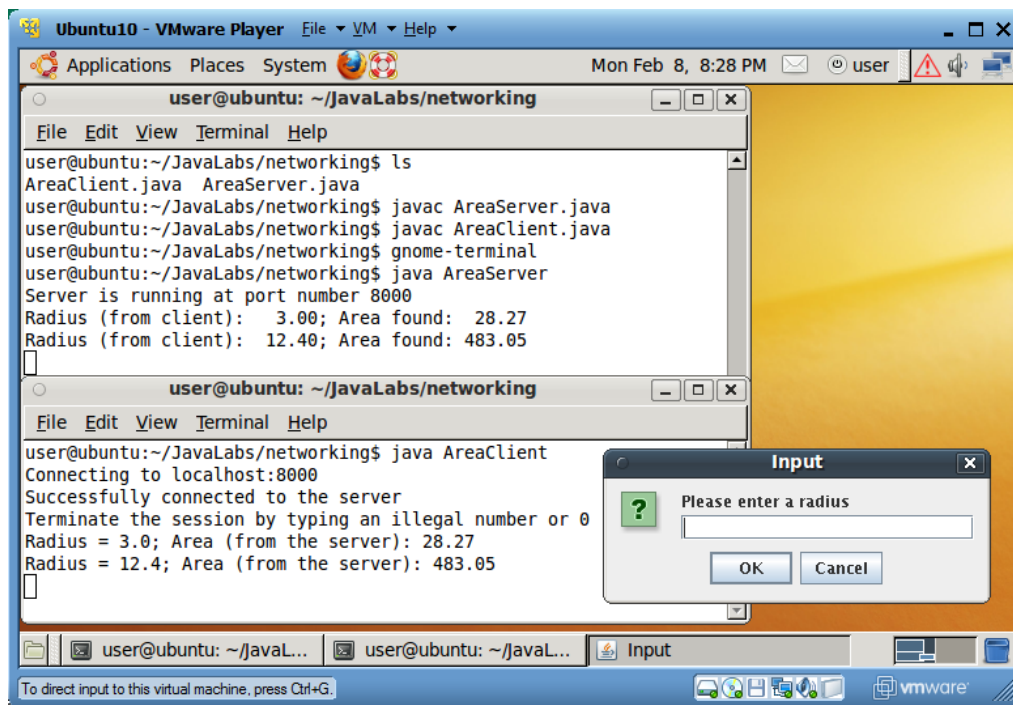
120 String serverURL = "localhost"; // default server URL
121 int portNbr = 8000;           // default server port number
122 if (args.length > 0)
123     serverURL = args[0];
124 if (args.length > 1) {
125     try {
126         portNbr = Integer.parseInt(args[1]);
127     } catch (Exception e) {
128         // ignore the wrong port number
129     }
130 }
131 try {
132     // Create a socket to connect to the server
133     System.out.println("Connecting to " + serverURL + ":" + portNbr);
134     Socket connectToServer = new Socket(serverURL, portNbr);
135     // Create a buffered input stream to receive data
136     // from the server
137     BufferedReader isFromServer = new BufferedReader(
138         new InputStreamReader(connectToServer.getInputStream()));
139     // Create a buffered output stream to send data to the server
140     PrintWriter osToServer =
141         new PrintWriter(connectToServer.getOutputStream(), true);
142     // Continuously send radius and receive area from the server
143     // Terminate the session by typing an illegal number or 0
144     System.out.println("Successfully connected to the server");
145     System.out.println(
146         "Terminate the session by typing an illegal number or 0");
147     while (true) {
148         // Read a radius from an input dialog
149         String s = JOptionPane.showInputDialog("Please enter a radius");
150         double radius;
151         try {
152             radius = Double.parseDouble(s.trim());
153         }
154         catch (Exception e) {
155             break;
156         }
157         if (radius == 0) // if radius is 0, terminate the session
158             break;
159         // Send the radius to the server
160         osToServer.println(radius);
161         // Get area from the server
162         String line = isFromServer.readLine().trim();
163         double area = Double.parseDouble(line.split("\\s+")[0]);
164         // Print area on the console
165         System.out.println(
166             "Radius = " + radius + "; Area (from the server): " + area);
167     }
168 }
169 catch (IOException e) {
170     System.err.println(e);
171 }
172 System.exit(0);
173 }
174 }

```

Explanation

- j. Class *AreaClient* implements the client program. By default it will try to connect to port 8000 on *localhost* (the same computer on which we run the server program). If we specified a different port number, say 9000, for launching the *AreaServer* program, we should run this *AreaClient* program with “java *AreaClient* localhost 9000”.
 - k. Line 134 requests connection to the server program by creating a *Socket* object with the specified server computer IP address or domain name as well as the port number to which the server program is listening.
 - l. Lines 137-138 derive an input stream from the socket object connected to the server-side socket. The more primitive input stream objects are wrapped in more advanced input stream objects to access the more advanced data reading operations.
 - m. Lines 140-141 derive an output stream from the socket object connected to the server-side socket. The more primitive output stream objects are wrapped in more advanced output stream objects to access the more advanced data printing operations.
 - n. The client program then uses the while loop (lines 147-167) to repeat getting a radius value from the client, printing the value to the output stream of the socket, and reading the return area value from the input stream of the socket. If the input is zero or non-numeric, the client program shuts down.
4. Make sure your working directory now is “~/JavaLabs/networking”. To compile classes *AreaServer* and *AreaClient*, type
- ```
javac AreaServer .java
javac AreaClient .java
```
5. To run class *AreaServer*, type
- ```
java AreaServer
```
6. Use terminal menu “File|Open Terminal” (or simply run “gnome-terminal”) to start a new terminal window in the same directory. To run class *AreaClient*, type
- ```
java AreaClient
```
- Type numbers for radius in the popup window, click on the OK button, and see the area calculated by the server. To terminal the client program, enter zero or non-numeric characters when prompted for numbers. To terminate the server, type Ctrl+c in the server’s terminal window, or just shut down the server’s terminal window.

The following is the screen capture of my test run of the two client-server classes.



## 7 Database Programming

[Coming soon]

### Useful MySql Commands

| Command                                         | Explanation                                           |
|-------------------------------------------------|-------------------------------------------------------|
| >mysql -u root -p123456<br>mysql>               | Login MySQL console                                   |
| mysql>help;                                     | List valid console commands                           |
| mysql>use test;                                 | Change to use database “test”                         |
| mysql>source sqlFileName.sql;                   | Execute an sql file in the current directory          |
| mysql>show databases;                           | Show list of existing databases                       |
| mysql>show tables;                              | Show list of tables in the current database           |
| mysql>drop table xx;                            | Delete table xx                                       |
| mysql>create database cs612;                    | Create a new database “cs612”                         |
| mysql>quit;                                     | Quit and get out of the MySQL console                 |
| >mysqldump -u root -p database > file.sql       | Dump all tables in “database” in sql file file.sql    |
| >mysqldump -u root -p database table > file.sql | Dump table “table” in “database” in sql file file.sql |