2024

Object Oriented Concepts & Programming - I (Core Java)

Dr. Babasaheb Ambedkar Open University

Object Oriented Concepts & Programming - I (Core Java)

Content Editor

Dr. Himanshu Patel
Assistant Professor
School of Computer Science
Dr. Babasaheb Ambedkar Open University, Ahmedabad

Content Reviewer

Prof. (Dr.) Nilesh Modi
Professor and Director
School of Computer Science
Dr. Babasaheb Ambedkar Open University, Ahmedabad

Printed and published by: Dr. Babasaheb Ambedkar Open University, Ahmedabad

Acknowledgement: The content in this block is modifications based on work created and shared by the David J. Eck, Department of Mathematics and Computer Science, Hobart and William Smith Colleges Geneva, NY 14456 for the book titled “Introduction to Programming Using Java” and is used according to terms described in Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

.

ISBN:

Dr. Babasaheb Ambedkar Open University

BCAR-204

Object Oriented Concepts & Programming - I (Core Java)

Block-1: Introduction to Programming

Block-2: Programming in the Large

Block-3: Data Structure

Block-1

Introduction to Programming

Unit 1: The Mental Landscape 1

Unit Structure

1.1 LEARNING OBJECTIVES

After studying this unit student should be able to understand:

- the basics of what computers are and how they work
- idea of what a computer program is and how one is created
- Fundamental Building Blocks of Programs
- Java Virtual Machine
- Java language in particular and
- about the modern computing environment for which Java is designed.

1.2 INTRODUCTION

When you begin a journey, it's a good idea to have a mental map of the terrain you'll be passing through. The same is true for an intellectual journey, such as learning to write computer programs. In this case, you'll need to know the basics of what computers are and how they work. You'll want to have some idea of what a computer program is and how one is created.

Since you will be writing programs in the Java programming language, you'll want to know something about that language in particular and about the modern computing environment for which Java is designed.

As you read this chapter, don't worry if you can't understand everything in detail. (In fact, it would be impossible for you to learn all the details from the brief expositions in this chapter.)

Concentrate on learning enough about the big ideas to orient yourself, in preparation for the rest of the book. Most of what is covered in this chapter will be covered in much greater detail later in the book.

1.3 The Fetch and Execute Cycle: Machine Language

A computer is a complex system consisting of many different components. But at the Heart or the brain, if you want of the computer is a single component that does the actual computing. This is the ***Central Processing Unit***, or ***CPU***. In a modern desktop computer, the CPU is a single "chip" on the order of one square inch in size. The job of the CPU is to execute programs.

A ***program*** is simply a list of unambiguous instructions meant to be followed mechanically by a computer. A computer is built to carry out instructions that are written in a very simple type of language called machine language. Each type of

computer has its own ***machine language***, and the computer can directly execute a program only if the program is expressed in that language. (It can execute programs written in other languages if they are first translated into machine language.)

When the CPU executes a program, that program is stored in the computer's ***main memory*** (also called the RAM or Random Access Memory). In addition to the program, memory can also hold data that is being used or processed by the program. Main memory consists of a sequence of ***locations***. These locations are numbered, and the sequence number of a location is called its ***address***. An address provides a way of picking out one particular piece of information from among the millions stored in memory. When the CPU needs to access the program instruction or data in a particular location, it sends the address of that information as a signal to the memory; the memory responds by sending back the value contained in the specified location. The CPU can also store information in memory by specifying the information to be stored and the address of the location where it is to be stored.

On the level of machine language, the operation of the CPU is fairly straightforward (although it is very complicated in detail). The CPU executes a program that is stored as a sequence of machine language instructions in main memory. It does this by repeatedly reading, or ***fetching***, an instruction from memory and then carrying out, or ***executing***, that instruction. This process fetches an instruction, execute it, fetch another instruction, execute it, and so on forever is called the ***fetch-and-execute cycle***. With one exception, which will be covered in the next section, this is all that the CPU ever does. (This is all really somewhat more complicated in modern computers. A typical processing chips these days contains several CPU "cores," which allows it to execute several instructions simultaneously. And access to main memory is speeded up by memory "caches," which can be more quickly accessed than main memory and which are meant to hold data and instructions that the CPU is likely to need soon. However, these complications don't change the basic operation.)

A CPU contains an ***Arithmetic Logic Unit***, or ***ALU***, which is the part of the processor that carries out operations such as addition and subtraction. It also holds a small number of registers, which are small memory units capable of holding a single number. A typical CPU might have 16 or 32 "general purpose" registers, which hold data values that are immediately accessible for processing, and many machine language instructions refer to these registers. For example, there might be an instruction that takes two numbers from two specified registers, adds those numbers (using the ALU), and stores the result back into a register. And there might be instructions for copying a data value from main memory into a register, or from a register into main memory.

The CPU also includes special purpose registers. The most important of these is the ***program counter***, or PC. The CPU uses the PC to keep track of where it is in

the program it is executing. The PC simply stores the memory address of the next instruction that the CPU should execute. At the beginning of each fetch-and-execute cycle, the CPU checks the PC to see which instruction it should fetch. During the course of the fetch-and-execute cycle, the number in the PC is updated to indicate the instruction that is to be executed in the next cycle. Usually, but not always, this is just the instruction that sequentially follows the current instruction in the program. Some machine language instructions modify the value that is stored in the PC. This makes it possible for the computer to "jump" from one point in the program to another point, which is essential for implementing the program features known as loops and branches that are discussed in Section 1.6.

A computer executes machine language programs mechanically that is without understanding them or thinking about them simply because of the way it is physically put together.

This is not an easy concept. A computer is a machine built of millions of tiny switches called ***transistors***, which have the property that they can be wired together in such a way that an output from one switch can turn another switch on or off. As a computer computes, these switches turn each other on or off in a pattern determined both by the way they are wired together and by the program that the computer is executing.

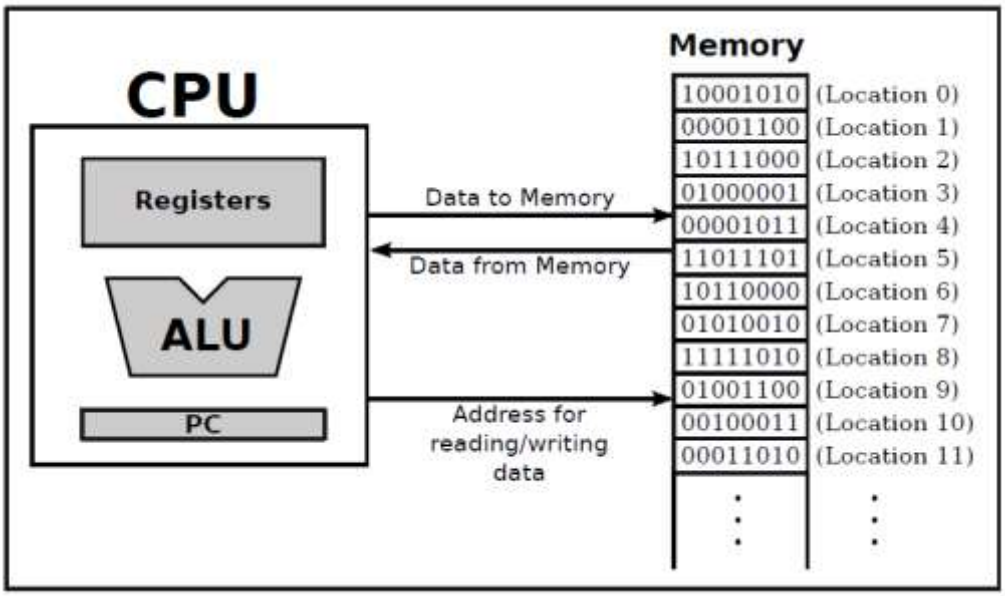
Machine language instructions are expressed as binary numbers. A binary number is made up of just two possible digits, zero and one. Each zero or one is called a ***bit***. So, a machine language instruction is just a sequence of zeros and ones. Each particular sequence encodes some particular instruction. The data that the computer manipulates is also encoded as binary numbers. In modern computers, each memory location holds a ***byte***, which is a sequence of eight bits. A machine language instruction or a piece of data generally consists of several bytes, stored in consecutive memory locations. For example, when a CPU reads an instruction from memory, it might actually read four or eight bytes from four or eight memory locations; the memory address of the instruction is the address of the first of those bytes.

A computer can work directly with binary numbers because switches can readily represent such numbers: Turn the switch on to represent a one; turn it off to represent a zero. Machine language instructions are stored in memory as patterns of switches turned on or off. When a machine language instruction is loaded into the CPU, all that happens is that certain switches are turned on or off in the pattern that encodes that instruction. The CPU is built to respond to this pattern by executing the instruction it encodes; it does this simply because of the way all the other switches in the CPU are wired together.

So, you should understand this much about how computers work: Main memory holds machine language programs and data. These are encoded as binary numbers.

The CPU fetches machine language instructions from memory one after another and executes them. Each instruction makes the CPU perform some very small tasks, such as adding two numbers or moving data to or from memory. The CPU does all this mechanically, without thinking about or understanding what it does and therefore the program it executes must be perfect, complete in all details, and unambiguous because the CPU can do nothing but execute it exactly as written.

Here is a schematic view of this first-stage understanding of the computer:



1.4 Asynchronous Events: Polling Loops and Interrupts

The CPU spends almost all of its time fetching instructions from memory and executing them. However, the CPU and main memory are only two out of many components in a real computer system. A complete system contains other devices such as:

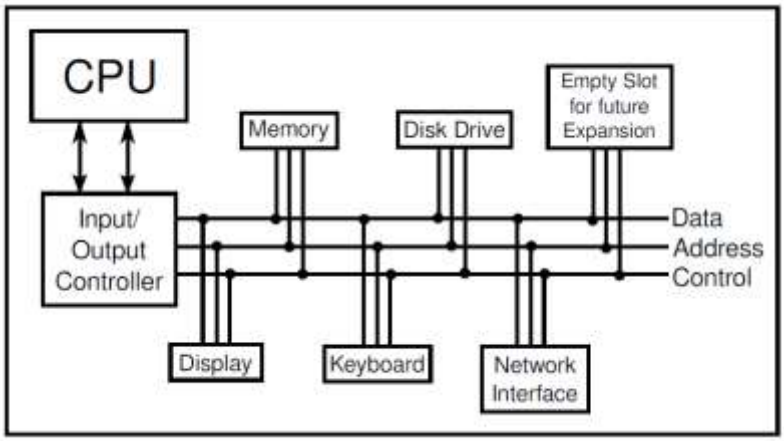
- A hard disk or solid-state drive for storing programs and data files. (Note that main memory holds only a comparatively small amount of information, and holds it only as long as the power is turned on. A hard disk or solid-state drive is used for permanent storage of larger amounts of information, but programs have to be loaded from there into main memory before they can actually be executed. A hard disk stores data on a spinning magnetic disk, while a solid-state drive is a purely electronic device with no moving parts.)
- A keyboard and mouse for user input.
- A monitor and printer which can be used to display the computer's output.
- An audio output device that allows the computer to play sounds.
- A network interface that allows the computer to communicate with other computers that are connected to it on a network, either wirelessly or by wire.

- A scanner that converts images into coded binary numbers that can be stored and manipulated on the computer.

The list of devices is entirely open ended, and computer systems are built so that they can easily be expanded by adding new devices. Somehow the CPU has to communicate with and control all these devices. The CPU can only do this by executing machine language instructions (which is all it can do, period). The way this works is that for each device in a system, there is a device driver, which consists of software that the CPU executes when it has to deal with the device. Installing a new device on a system generally has two steps: plugging the device physically into the computer, and installing the device driver software. Without the device driver, the actual physical device would be useless, since the CPU would not be able to communicate with it.

* * *

A computer system consisting of many devices is typically organized by connecting those devices to one or more busses. A bus is a set of wires that carry various sorts of information between the devices connected to those wires. The wires carry data, addresses, and control signals. An address directs the data to a particular device and perhaps to a particular register or location within that device. Control signals can be used, for example, by one device to alert another that data is available for it on the data bus. A fairly simple computer system might be organized like this:



Now, devices such as keyboard, mouse, and network interface can produce input that needs to be processed by the CPU. How does the CPU know that the data is there? One simple idea, which turns out to be not very satisfactory, is for the CPU to keep checking for incoming data over and over. Whenever it finds data, it processes it. This method is called polling, since the CPU polls the input devices continually to see whether they have any input data to report. Unfortunately, although polling is very simple, it is also very inefficient. The CPU can waste an awful lot of time just waiting for input.

To avoid this inefficiency, interrupts are generally used instead of polling. An interrupt is a signal sent by another device to the CPU. The CPU responds to an interrupt signal by putting aside whatever it is doing in order to respond to the interrupt.

Once it has handled the interrupt, it returns to what it was doing before the interrupt occurred. For example, when you press a key on your computer keyboard, a keyboard interrupt is sent to the CPU. The CPU responds to this signal by interrupting what it is doing, reading the key that you pressed, processing it, and then returning to the task it was performing before you pressed the key.

Again, you should understand that this is a purely mechanical process: A device signals an interrupt simply by turning on a wire. The CPU is built so that when that wire is turned on, the CPU saves enough information about what it is currently doing so that it can return to the same state later. This information consists of the contents of important internal registers such as the program counter. Then the CPU jumps to some predetermined memory location and begins executing the instructions stored there. Those instructions make up an interrupt handler that does the processing necessary to respond to the interrupt. (This interrupt handler is part of the device driver software for the device that signalled the interrupt.) At the end of the interrupt handler is an instruction that tells the CPU to jump back to what it was doing; it does that by restoring its previously saved state.

Interrupts allow the CPU to deal with asynchronous events. In the regular fetch-and-execute cycle, things happen in a predetermined order; everything that happens is “synchronized” with everything else. Interrupts make it possible for the CPU to deal efficiently with events that happen “asynchronously,” that is, at unpredictable times.

As another example of how interrupts are used, consider what happens when the CPU needs to access data that is stored on a hard disk. The CPU can access data directly only if it is in main memory. Data on the disk has to be copied into memory before it can be accessed.

Unfortunately, on the scale of speed at which the CPU operates, the disk drive is extremely slow. When the CPU needs data from the disk, it sends a signal to the disk drive telling it to locate the data and get it ready. (This signal is sent synchronously, under the control of a regular program.) Then, instead of just waiting the long and unpredictable amount of time that the disk drive will take to do this, the CPU goes on with some other tasks. When the disk drive has the data ready, it sends an interrupt signal to the CPU. The interrupt handler can then read the requested data.

* * *

Now, you might have noticed that all this only makes sense if the CPU actually has several tasks to perform. If it has nothing better to do, it might as well spend its time polling for input or waiting for disk drive operations to complete. All modern computers use multitasking to perform several tasks at once. Some computers can be used by several people at once. Since the CPU is so fast, it can quickly switch its attention from one user to another, devoting a fraction of a second to each user in turn. This application of multitasking is called timesharing. But a modern personal computer with just a single user also uses multitasking. For example, the user might be typing a paper while a clock is continuously displaying the time and a file is being

downloaded over the network.

Each of the individual tasks that the CPU is working on is called a thread. (Or a process; there are technical differences between threads and processes, but they are not important here, since it is threads that are used in Java.) Many CPUs can literally execute more than one thread simultaneously such CPUs contain multiple “cores,” each of which can run a thread but there is always a limit on the number of threads that can be executed at the same time.

Since there are often more threads than can be executed simultaneously, the computer has to be able switch its attention from one thread to another, just as a timesharing computer switches its attention from one user to another. In general, a thread that is being executed will continue to run until one of several things happens:

- The thread might voluntarily yield control, to give other threads a chance to run.
- The thread might have to wait for some asynchronous event to occur. For example, the thread might request some data from the disk drive, or it might wait for the user to press a key. While it is waiting, the thread is said to be blocked, and other threads, if any, have a chance to run. When the event occurs, an interrupt will “wake up” the thread so that it can continue running.
- The thread might use up its allotted slice of time and be suspended to allow other threads to run. Most computers can “forcibly” suspend a thread in this way; computers that can do that are said to use pre-emptive multitasking. To do pre-emptive multitasking, a computer needs a special timer device that generates an interrupt at regular intervals, such as 100 times per second. When a timer interrupt occurs, the CPU has a chance to switch from one thread to another, whether the thread that is currently running likes it or not. All modern desktop and laptop computers, and even typical smartphones and tablets, use pre-emptive multitasking.

Ordinary users, and indeed ordinary programmers, have no need to deal with interrupts and interrupt handlers. They can concentrate on the different tasks that they want the computer to perform; the details of how the computer manages to get all those tasks done are not important to them. In fact, most users, and many programmers, can ignore threads and multitasking altogether. However, threads have become increasingly important as computers have become more powerful and as they have begun to make more use of multitasking and multiprocessing.

In fact, the ability to work with threads is fast becoming an essential job skill for programmers.

Fortunately, Java has good support for threads, which are built into the Java programming language as a fundamental programming concept. Programming with threads will be covered in Block-3.

Just as important in Java and in modern programming in general is the basic concept of asynchronous events. While programmers don't actually deal with interrupts directly, they do often find themselves writing event handlers, which, like interrupt handlers, are called asynchronously when specific events occur. Such "event-driven programming" has a very different feel from the more traditional straight-through, synchronous programming. We will begin with the more traditional type of programming, which is still used for programming individual tasks, but we will return to threads and events later in the text, starting in Block-4

* * *

By the way, the software that does all the interrupt handling, handles communication with the user and with hardware devices, and controls which thread is allowed to run is called the operating system. The operating system is the basic, essential software without which a computer would not be able to function. Other programs, such as word processors and Web browsers, are dependent upon the operating system. Common desktop operating systems include Linux, various versions of Windows, and Mac OS. Operating systems for smartphones and tablets include Android and iOS.

1.5 The Java Virtual Machine

Machine language consists of very simple instructions that can be executed directly by the CPU of a computer. Almost all programs, though, are written in high-level programming languages such as Java, Python, or C++. A program written in a high-level language cannot be run directly on any computer. First, it has to be translated into machine language. This translation can be done by a program called a compiler. A compiler takes a high-level-language program and translates it into an executable machine-language program. Once the translation is done, the machine-language program can be run any number of times, but of course it can only be run on one type of computer (since each type of computer has its own individual machine language). If the program is to run on another type of computer it has to be re-translated, using a different compiler, into the appropriate machine language.

There is an alternative to compiling a high-level language program. Instead of using a compiler, which translates the program all at once, you can use an interpreter, which translates it instruction-by-instruction, as necessary. An interpreter is a program that acts much like a CPU, with a kind of fetch-and-execute cycle. In order to execute a program, the interpreter runs in a loop in which it repeatedly reads one instruction from the program, decides what is necessary to carry out that instruction, and then performs the appropriate machine-language commands to do so.

(A compiler is like a human translator who translates an entire book from one language to another, producing a new book in the second language. An interpreter is more like a human interpreter who translates a speech at the United Nations from one language to another at the same time that the speech is being given.)

One use of interpreters is to execute high-level language programs. For example, the programming language Lisp is usually executed by an interpreter rather than a compiler. However, interpreters have another purpose: They can let you use a machine-language program meant for one type of computer on a completely different type of computer. For example, one of the original home computers was the Commodore 64 or “C64”. While you might not find an actual C64, you can find programs that run on other computers or even in a web browser that “emulate” one. Such an emulator can run C64 programs by acting as an interpreter for the C64 machine language.

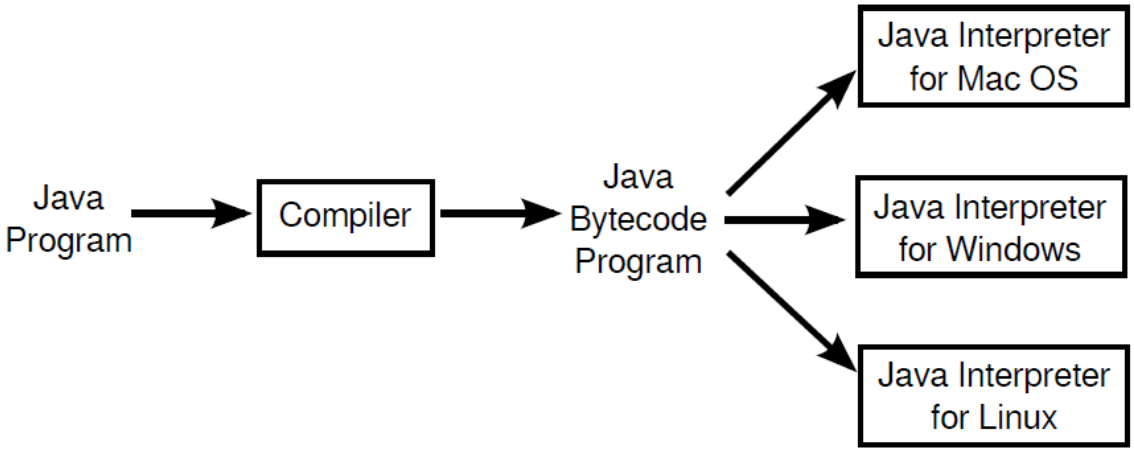
* * *

The designers of Java chose to use a combination of compiling and interpreting. Programs written in Java are compiled into machine language, but it is a machine language for a computer that doesn’t really exist. This so-called “virtual” computer is known as the Java Virtual Machine, or JVM. The machine language for the Java Virtual Machine is called Java bytecode.

There is no reason why Java bytecode couldn’t be used as the machine language of a real computer, rather than a virtual computer. But in fact the use of a virtual machine makes possible one of the main selling points of Java: the fact that it can actually be used on any computer.

All that the computer needs is an interpreter for Java bytecode. Such an interpreter simulates the JVM in the same way that a C64 emulator simulates a Commodore 64 computer. (The term JVM is also used for the Java bytecode interpreter program that does the simulation, so we say that a computer needs a JVM in order to run Java programs. Technically, it would be more correct to say that the interpreter implements the JVM than to say that it is a JVM.)

Of course, a different Java bytecode interpreter is needed for each type of computer, but once a computer has a Java bytecode interpreter, it can run any Java bytecode program, and the same program can be run on any computer that has such an interpreter. This is one of the essential features of Java: the same compiled program can be run on many different types of computers.



Why, you might wonder, use the intermediate Java bytecode at all? Why not just distribute the original Java program and let each person compile it into the machine language of whatever computer they want to run it on? There are several reasons. First of all, a compiler has to understand Java, a complex high-level language. The compiler is itself a complex program.

A Java bytecode interpreter, on the other hand, is a relatively small, simple program. This makes it easy to write a bytecode interpreter for a new type of computer; once that is done, that computer can run any compiled Java program. It would be much harder to write a Java compiler for the same computer.

Furthermore, some Java programs are meant to be downloaded over a network. This leads to obvious security concerns: you don't want to download and run a program that will damage your computer or your files. The bytecode interpreter acts as a buffer between you and the program you download. You are really running the interpreter, which runs the downloaded program indirectly. The interpreter can protect you from potentially dangerous actions on the part of that program.

When Java was still a new language, it was criticized for being slow: Since Java bytecode was executed by an interpreter, it seemed that Java bytecode programs could never run as quickly as programs compiled into native machine language (that is, the actual machine language of the computer on which the program is running). However, this problem has been largely overcome by the use of just-in-time compilers for executing Java bytecode. A just-in-time compiler translates Java bytecode into native machine language. It does this while it is executing the program. Just as for a normal interpreter, the input to a just-in-time compiler is a Java bytecode program, and its task is to execute that program. But as it is executing the program, it also translates parts of it into machine language. The translated parts of the program can then be executed much more quickly than they could be interpreted. Since a given part of a program is often executed many times as the program runs, a just-in-time compiler can significantly speed up the overall execution time.

I should note that there is no necessary connection between Java and Java bytecode. A program written in Java could certainly be compiled into the machine language of a real computer. And programs written in other languages can be compiled into Java bytecode. However, the combination of Java and Java bytecode is platform-independent, secure, and network-compatible while allowing you to program in a modern high-level object-oriented language.

In the past few years, it has become fairly common to create new programming languages, or versions of old languages, that compile into Java bytecode. The compiled bytecode programs can then be executed by a standard JVM. New languages that have been developed specifically for programming the JVM include Scala, Groovy, Clojure, and Processing. Jython and JRuby are versions of older languages, Python and Ruby, that target the JVM. These languages make it possible to enjoy many of the advantages of the JVM while avoiding some of the technicalities of the Java language. In fact, the use of other languages with the JVM has become important enough that several new features have been added to the JVM specifically to add better support for some of those languages. And this improvement to the JVM has in turn made possible some new features in Java.

* * *

I should also note that the really hard part of platform-independence is providing a “Graphical User Interface”—with windows, buttons, etc.—that will work on all the platforms that support Java.

1.6 Fundamental Building Blocks of Programs

There are two basic aspects of programming: data and instructions. To work with data, you need to understand variables and types; to work with instructions, you need to understand control structures and subroutines. You’ll spend a large part of the course becoming familiar with these concepts.

A variable is just a memory location (or several consecutive locations treated as a unit) that has been given a name so that it can be easily referred to and used in a program. The programmer only has to worry about the name; it is the compiler’s responsibility to keep track of the memory location. As a programmer, you just need to keep in mind that the name refers to a kind of “box” in memory that can hold data, even though you don’t have to know where in memory that box is located.

In Java and in many other programming languages, a variable has a type that indicates what sort of data it can hold. One type of variable might hold integers whole numbers such as 3, -7, and 0—while another hold floating point numbers with decimal points such as 3.14, -2.7, or 17.0. (Yes, the computer does make a distinction between the integer 17 and the floating-point number 17.0; they actually look quite different inside the computer.) There could also be types for individual characters (’A’, ’;’, etc.),

strings (“Hello”, “A string can include many characters”, etc.), and less common types such as dates, colors, sounds, or any other kind of data that a program might need to store.

Programming languages always have commands for getting data into and out of variables and for doing computations with data. For example, the following “assignment statement,” which might appear in a Java program, tells the computer to take the number stored in the variable named “principal”, multiply that number by 0.07, and then store the result in the variable named “interest”:

```
interest = principal * 0.07;
```

There are also “input commands” for getting data from the user or from files on the computer’s disks, and there are “output commands” for sending data in the other direction.

These basic commands—for moving data from place to place and for performing computations—are the building blocks for all programs. These building blocks are combined into complex programs using control structures and subroutines.

* * *

A program is a sequence of instructions. In the ordinary “flow of control,” the computer executes the instructions in the sequence in which they occur in the program, one after the other. However, this is obviously very limited: the computer would soon run out of instructions to execute. Control structures are special instructions that can change the flow of control.

There are two basic types of control structure: loops, which allow a sequence of instructions to be repeated over and over, and branches, which allow the computer to decide between two or more different courses of action by testing conditions that occur as the program is running.

For example, it might be that if the value of the variable “principal” is greater than 10000, then the “interest” should be computed by multiplying the principal by 0.05; if not, then the interest should be computed by multiplying the principal by 0.04. A program needs some way of expressing this type of decision. In Java, it could be expressed using the following “if statement”:

```
if (principal > 10000)
interest = principal * 0.05;
else
interest = principal * 0.04;
```

(Don’t worry about the details for now. Just remember that the computer can test a condition and decide what to do next on the basis of that test.)

Loops are used when the same task has to be performed more than once. For example, if you want to print out a mailing label for each name on a mailing list, you might say, “Get the first name and address and print the label; get the second name and address and print the label; get the third name and address and print the label...” But this quickly becomes ridiculous—and might not work at all if you don’t know in advance how many names there are.

What you would like to say is something like “While there are more names to process, get the next name and address, and print the label.” A loop can be used in a program to express such repetition.

* * *

Large programs are so complex that it would be almost impossible to write them if there were not some way to break them up into manageable “chunks.” Subroutines provide one way to do this. A subroutine consists of the instructions for performing some task, grouped together as a unit and given a name. That name can then be used as a substitute for the whole set of instructions. For example, suppose that one of the tasks that your program needs to perform is to draw a house on the screen. You can take the necessary instructions, make them into a subroutine, and give that subroutine some appropriate name—say, “drawHouse()”. Then anyplace in your program where you need to draw a house, you can do so with the single command:

```
drawHouse();
```

This will have the same effect as repeating all the house-drawing instructions in each place.

The advantage here is not just that you save typing. Organizing your program into subroutines also helps you organize your thinking and your program design effort. While writing the house-drawing subroutine, you can concentrate on the problem of drawing a house without worrying for the moment about the rest of the program. And once the subroutine is written, you can forget about the details of drawing houses—that problem is solved, since you have a subroutine to do it for you. A subroutine becomes just like a built-in part of the language which you can use without thinking about the details of what goes on “inside” the subroutine.

* * *

Variables, types, loops, branches, and subroutines are the basis of what might be called “traditional programming.” However, as programs become larger, additional structure is needed to help deal with their complexity. One of the most effective tools that has been found is object-oriented programming, which is discussed in the next section.

1.7 Objects and Object-oriented Programming

Programs must be designed. No one can just sit down at the computer and compose a program of any complexity. The discipline called software engineering is concerned with the construction of correct, working, well-written programs. The software engineer tries to use accepted and proven methods for analysing the problem to be solved and for designing a program to solve that problem.

During the 1970s and into the 80s, the primary software engineering methodology was structured programming. The structured programming approach to program design was based on the following advice: To solve a large problem, break the problem into several pieces and work on each piece separately; to solve each piece, treat it as a new problem which can itself be broken down into smaller problems; eventually, you will work your way down to problems that can be solved directly, without further decomposition. This approach is called top-down programming.

There is nothing wrong with top-down programming. It is a valuable and often-used approach to problem-solving. However, it is incomplete. For one thing, it deals almost entirely with producing the instructions necessary to solve a problem. But as time went on, people realized that the design of the data structures for a program was at least as important as the design of subroutines and control structures. Top-down programming doesn't give adequate consideration to the data that the program manipulates.

Another problem with strict top-down programming is that it makes it difficult to reuse work done for other projects. By starting with a particular problem and subdividing it into convenient pieces, top-down programming tends to produce a design that is unique to that problem. It is unlikely that you will be able to take a large chunk of programming from another program and fit it into your project, at least not without extensive modification. Producing high-quality programs is difficult and expensive, so programmers and the people who employ them are always eager to reuse past work.

* * *

So, in practice, top-down design is often combined with bottom-up design. In bottom-up design, the approach is to start "at the bottom," with problems that you already know how to solve (and for which you might already have a reusable software component at hand). From there, you can work upwards towards a solution to the overall problem.

The reusable components should be as "modular" as possible. A module is a component of a larger system that interacts with the rest of the system in a simple, well-defined, straightforward manner. The idea is that a module can be "plugged into" a system. The details of what goes on inside the module are not important to the

system as a whole, as long as the module fulfils its assigned role correctly. This is called information hiding, and it is one of the most important principles of software engineering.

One common format for software modules is to contain some data, along with some subroutines for manipulating that data. For example, a mailing-list module might contain a list of names and addresses along with a subroutine for adding a new name, a subroutine for printing mailing labels, and so forth. In such modules, the data itself is often hidden inside the module; a program that uses the module can then manipulate the data only indirectly, by calling the subroutines provided by the module. This protects the data, since it can only be manipulated in known, well-defined ways. And it makes it easier for programs to use the module, since they don't have to worry about the details of how the data is represented. Information about the representation of the data is hidden.

Modules that could support this kind of information-hiding became common in programming languages in the early 1980s. Since then, a more advanced form of the same idea has more or less taken over software engineering. This latest approach is called object-oriented programming, often abbreviated as OOP.

The central concept of object-oriented programming is the object , which is a kind of module containing data and subroutines. The point-of-view in OOP is that an object is a kind of self-sufficient entity that has an internal state (the data it contains) and that can respond to messages (calls to its subroutines). A mailing list object, for example, has a state consisting of a list of names and addresses. If you send it a message telling it to add a name, it will respond by modifying its state to reflect the change. If you send it a message telling it to print itself, it will respond by printing out its list of names and addresses.

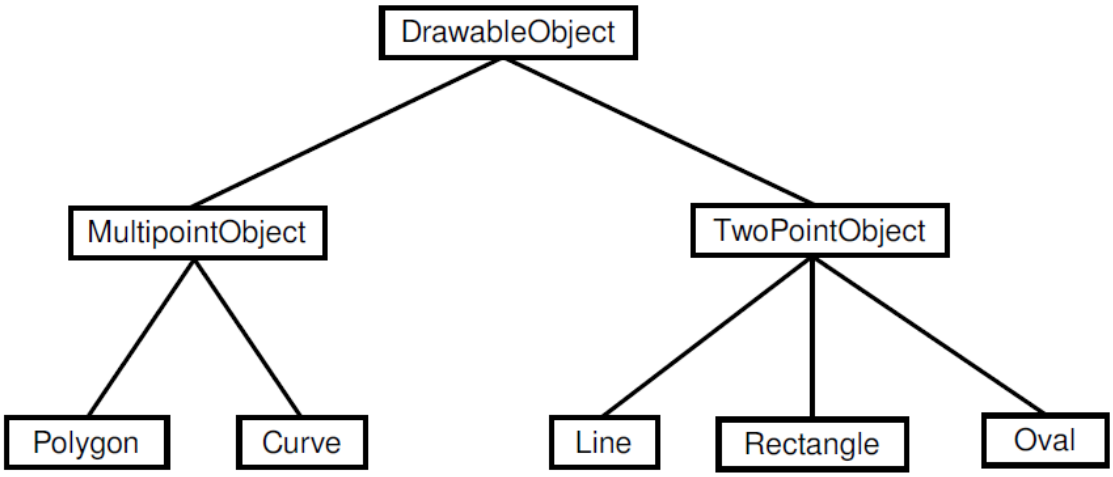
The OOP approach to software engineering is to start by identifying the objects involved in a problem and the messages that those objects should respond to. The program that results is a collection of objects, each with its own data and its own set of responsibilities. The objects interact by sending messages to each other. There is not much "top-down" in the large-scale design of such a program, and people used to more traditional programs can have a hard time getting used to OOP. However, people who use OOP would claim that object-oriented programs tend to be better models of the way the world itself works, and that they are therefore easier to write, easier to understand, and more likely to be correct.

* * *

You should think of objects as "knowing" how to respond to certain messages. Different objects might respond to the same message in different ways. For example, a "print" message would produce very different results, depending on the object it is sent to. This property of objects that different objects can respond to the same message in different ways is called polymorphism.

It is common for objects to bear a kind of “family resemblance” to one another. Objects that contain the same type of data and that respond to the same messages in the same way belong to the same class. (In actual programming, the class is primary; that is, a class is created and then one or more objects are created using that class as a template.) But objects can be similar without being in exactly the same class.

For example, consider a drawing program that lets the user draw lines, rectangles, ovals, polygons, and curves on the screen. In the program, each visible object on the screen could be represented by a software object in the program. There would be five classes of objects in the program, one for each type of visible object that can be drawn. All the lines would belong to one class, all the rectangles to another class, and so on. These classes are obviously related; all of them represent “drawable objects.” They would, for example, all presumably be able to respond to a “draw yourself” message. Another level of grouping, based on the data needed to represent each type of object, is less obvious, but would be very useful in a program: We can group polygons and curves together as “multipoint objects,” while lines, rectangles, and ovals are “two-point objects.” (A line is determined by its two endpoints, a rectangle by two of its corners, and an oval by two corners of the rectangle that contains it. The rectangles that I am talking about here have sides that are vertical and horizontal, so that they can be specified by just two points; this is the common meaning



of “rectangle” in drawing programs.) We could diagram these relationships as follows:

DrawableObject, MultipointObject, and TwoPointObject would be classes in the program. MultipointObject and TwoPointObject would be subclasses of DrawableObject. The class Line would be a subclass of TwoPointObject and (indirectly) of DrawableObject. A subclass of a class is said to inherit the properties of that class. The subclass can add to its inheritance and it can even “override” part of that inheritance (by defining a different response to some message). Nevertheless, lines, rectangles, and so on are drawable objects, and the class DrawableObject expresses this relationship.

Inheritance is a powerful means for organizing a program. It is also related to the problem of reusing software components. A class is the ultimate reusable component. Not only can it be reused directly if it fits exactly into a program you are trying to write, but if it just almost fits, you can still reuse it by defining a subclass and making only the small changes necessary to adapt it exactly to your needs.

So, OOP is meant to be both a superior program-development tool and a partial solution to the software reuse problem. Objects, classes, and object-oriented programming will be important themes throughout the rest of this text. You will start using objects that are built into the Java language in the next chapter, and in Chapter 5 you will begin creating your own classes and objects.

1.8 The Modern User Interface

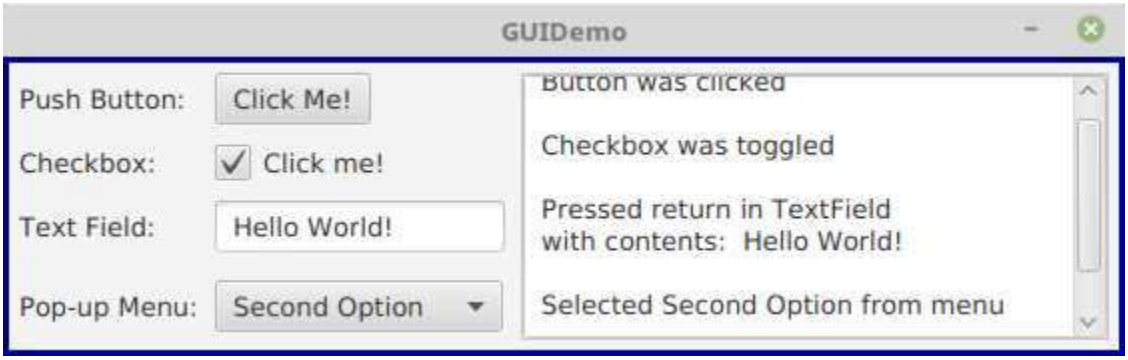
When computers were first introduced, ordinary people—including most programmers couldn't get near them. They were locked up in rooms with white-coated attendants who would take your programs and data, feed them to the computer, and return the computer's response sometime later. When timesharing where the computer switches its attention rapidly from one person to another—was invented in the 1960s, it became possible for several people to interact directly with the computer at the same time. On a timesharing system, users sit at "terminals" where they type commands to the computer, and the computer types back its response. Early personal computers also used typed commands and responses, except that there was only one person involved at a time. This type of interaction between a user and a computer is called a command-line interface.

Today, of course, most people interact with computers in a completely different way. They use a Graphical User Interface, or GUI. The computer draws interface components on the screen. The components include things like windows, scroll bars, menus, buttons, and icons. Usually, a mouse is used to manipulate such components or, on touchscreens," your fingers. Assuming that you have not just been teleported in from the 1970s, you are no doubt already familiar with the basics of graphical user interfaces!

A lot of GUI interface components have become fairly standard. That is, they have similar appearance and behavior on many different computer platforms including Mac OS, Windows, and Linux. Java programs, which are supposed to run on many different platforms without modification to the program, can use all the standard GUI components. They might vary a little in appearance from platform to platform, but their functionality should be identical on any computer on which the program runs.

Shown below is an image of a very simple Java program that demonstrates a few standard GUI interface components. When the program is run, a window similar

to the picture shown here will open on the computer screen. There are four components in the window with which the user can interact: a button, a checkbox, a text field, and a pop-up menu. These components are labeled. There are a few other components in the window. The labels themselves are components (even though you can't interact with them). The right half of the window is a text area component, which can display multiple lines of text. A scrollbar component appears alongside the text area when the number of lines of text becomes larger than will fit in the text area. And in fact, in Java terminology, the whole window is itself considered to be a "component".



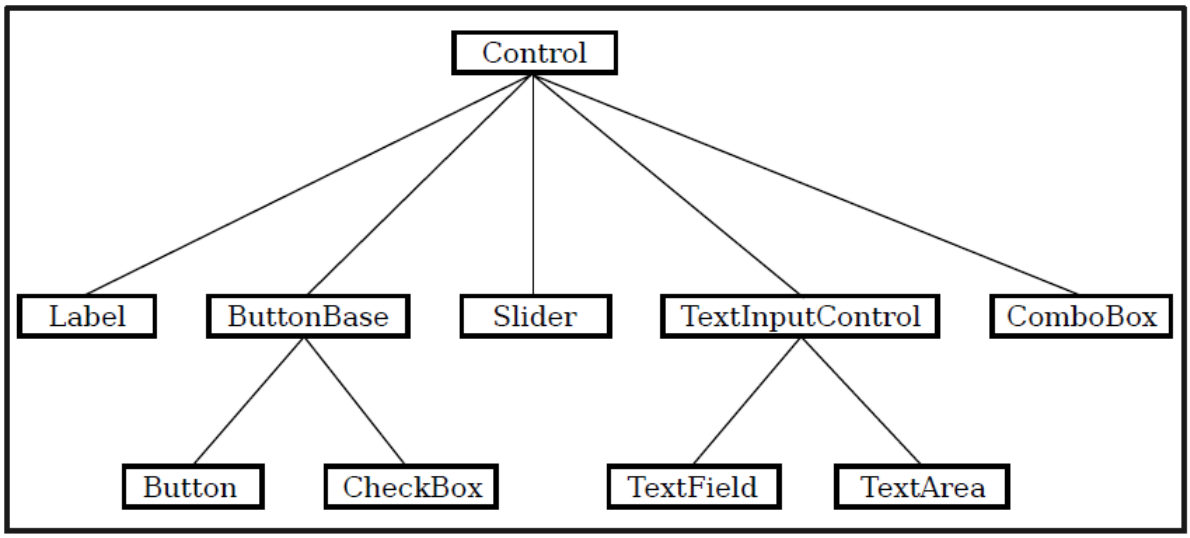
(If you would like to run this program, the source code, GUIDemo.java, is available on line. For more information on using this and other examples from this textbook, see Section 2.6.)

Now, Java actually has three complete sets of GUI components. One of these, the AWT or Abstract Windowing Toolkit, was available in the original version of Java. The second, which is known as Swing, was introduced in Java version 1.2, and was the standard GUI toolkit for many years. The third GUI toolkit, JavaFX, became a standard part of Java in Version 8 (but has recently been removed, so that it requires separate installation in some versions of Java). Although Swing, and even the AWT, can still be used, JavaFX is meant as a more modern way to write GUI applications. This textbook covers JavaFX exclusively. (If you need to learn Swing, you can take a look at the previous version of this book.)

When a user interacts with GUI components, "events" are generated. For example, clicking a push button generates an event, and pressing a key on the keyboard generates an event. Each time an event is generated, a message is sent to the program telling it that the event has occurred, and the program responds according to its program. In fact, a typical GUI program consists largely of "event handlers" that tell the program how to respond to various types of events. In the above example, the program has been programmed to respond to each event by displaying a message in the text area. In a more realistic example, the event handlers would have more to do.

The use of the term "message" here is deliberate. Messages, as you saw in the previous section, are sent to objects. In fact, Java GUI components are implemented as objects. Java includes many predefined classes that represent various types of GUI

components. Some of these classes are subclasses of others. Here is a diagram showing just a few of the JavaFX GUI classes and their relationships:



Don't worry about the details for now, but try to get some feel about how object-oriented programming and inheritance are used here. Note that all the GUI classes shown here are subclasses, directly or indirectly, of a class called Control, which represents general properties that are shared by many JavaFX components. In the diagram, two of the direct subclasses of Control themselves have subclasses. The classes TextField and TextArea, which have certain behaviors in common, are grouped together as subclasses of TextInputControl. Similarly, Button and CheckBox are subclasses of ButtonBase, which represents properties common to both buttons and checkboxes. (ComboBox, by the way, is the class that represents pop-up menus.)

Just from this brief discussion, perhaps you can see how GUI programming can make effective use of object-oriented design. In fact, GUIs, with their "visible objects," are probably a major factor contributing to the popularity of OOP.

1.9 The Internet and Beyond

Computers can be connected together on networks. A computer on a network can communicate with other computers on the same network by exchanging data and files or by sending and receiving messages. Computers on a network can even work together on a large computation.

Today, millions of computers throughout the world are connected to a single huge network called the Internet. New computers are being connected to the Internet every day, both by wireless communication and by physical connection using technologies such as DSL, cable modems, and Ethernet.

There are elaborate protocols for communication over the Internet. A protocol is simply a detailed specification of how communication is to proceed. For two computers to communicate at all, they must both be using the same protocols. The most basic protocols on the Internet are the Internet Protocol (IP), which specifies how data is to be physically transmitted from one computer to another, and the Transmission Control Protocol (TCP), which ensures that data sent using IP is received in its entirety and without error. These two protocols, which are referred to collectively as TCP/IP, provide a foundation for communication. Other protocols use TCP/IP to send specific types of information such as web pages, electronic mail, and data files.

All communication over the Internet is in the form of packets. A packet consists of some data being sent from one computer to another, along with addressing information that indicates where on the Internet that data is supposed to go. Think of a packet as an envelope with an address on the outside and a message on the inside. (The message is the data.) The packet also includes a “return address,” that is, the address of the sender. A packet can hold only a limited amount of data; longer messages must be divided among several packets, which are then sent individually over the Net and reassembled at their destination.

Every computer on the Internet has an IP address, a number that identifies it uniquely among all the computers on the Net. (Actually, the claim about uniqueness is not quite true, but the basic idea is valid, and the full truth is complicated.) The IP address is used for addressing packets. A computer can only send data to another computer on the Internet if it knows that computer’s IP address. Since people prefer to use names rather than numbers, most computers are also identified by names, called domain names. For example, the main computer of the Mathematics Department at Hobart and William Smith Colleges has the domain name math.hws.edu. (Domain names are just for convenience; your computer still needs to know IP addresses before it can communicate. There are computers on the Internet whose job it is to translate domain names to IP addresses. When you use a domain name, your computer sends a message to a domain name server to find out the corresponding IP address. Then, your computer uses the IP address, rather than the domain name, to communicate with the other computer.)

The Internet provides a number of services to the computers connected to it (and, of course, to the users of those computers). These services use TCP/IP to send various types of data over the Net. Among the most popular services are instant messaging, file sharing, electronic mail, and the World-Wide Web. Each service has its own protocols, which are used to control transmission of data over the network. Each service also has some sort of user interface, which allows the user to view, send, and receive data through the service.

For example, the email service uses a protocol known as SMTP (Simple Mail Transfer Protocol) to transfer email messages from one computer to another. Other protocols, such as POP and IMAP, are used to fetch messages from an email account so that the recipient can read them. A person who uses email, however, doesn't need to understand or even know about these protocols. Instead, they are used behind the scenes by computer programs to send and receive email messages. These programs provide the user with an easy-to-use user interface to the underlying network protocols.

The World-Wide Web is perhaps the most exciting of network services. The World Wide Web allows you to request pages of information that are stored on computers all over the Internet. A Web page can contain links to other pages on the same computer from which it was obtained or to other computers anywhere in the world. A computer that stores such pages of information is called a web server. The user interface to the Web is the type of program known as a web browser. Common web browsers include Microsoft Edge, Internet Explorer, Firefox, Chrome, and Safari. You use a Web browser to request a page of information.

The browser sends a request for that page to the computer on which the page is stored, and when a response is received from that computer, the web browser displays it to you in a neatly formatted form. A web browser is just a user interface to the Web. Behind the scenes, the web browser uses a protocol called HTTP (HyperText Transfer Protocol) to send each page request and to receive the response from the web server.

* * *

Now just what, you might be thinking, does all this have to do with Java? In fact, Java is intimately associated with the Internet and the World-Wide Web. When Java was first introduced, one of its big attractions was the ability to write applets. An applet is a small program that is transmitted over the Internet and that runs on a web page. Applets made it possible for a web page to perform complex tasks and have complex interactions with the user.

Alas, applets have suffered from a variety of problems, and they have fallen out of use. There are now other options for running programs on Web pages.

But applets were only one aspect of Java's relationship with the Internet. Java can be used to write complex, stand-alone applications that do not depend on a Web browser. Many of these programs are network-related. For example, many of the largest and most complex web sites use web server software that is written in Java. Java includes excellent support for network protocols, and its platform independence makes it possible to write network programs that work on many different types of computers.

Its support for networking is not Java's only advantage. But many good programming languages have been invented only to be soon forgotten. Java has had the good luck to ride on the coattails of the Internet's immense and increasing popularity.

* * *

As Java has matured, its applications have reached far beyond the Net. The standard version of Java already comes with support for many technologies, such as cryptography, data compression, sound processing, and three-dimensional graphics. And programmers have written Java libraries to provide additional capabilities. Complex, high-performance systems can be developed in Java. For example, Hadoop, a system for large scale data processing, is written in Java. Hadoop is used by Yahoo, Facebook, and other Web sites to process the huge amounts of data generated by their users.

Furthermore, Java is not restricted to use on traditional computers. Java can be used to write programs for many smartphones (though not for the iPhone). It is the primary development language for Android-based devices. (Android uses Google's own version of Java and does not use the same graphical user interface components as standard Java.) Java is also the programming language for the Amazon Kindle eBook reader and for interactive features on Blu-Ray video disks.

At this time, Java certainly ranks as one of the most widely used programming languages. It is a good choice for almost any programming project that is meant to run on more than one type of computing device, and is a reasonable choice even for many programs that will run on only one device. It is probably still the most widely taught language at Colleges and Universities. It is similar enough to other popular languages, such as C++, JavaScript, and Python, that knowing it will give you a good start on learning those languages as well. Overall, learning Java is a great starting point on the road to becoming an expert programmer. I hope you enjoy the journey!

1.10 Let Us Sum Up

In this unit we have discussed the basics of what computers are and how they work, idea of what a computer program is and how one is created, Fundamental Building Blocks of Programs, Java Virtual Machine, Java language in particular and about the modern computing environment for which Java is designed.

1.11 Further Reading

1. "Java 2: The Complete Reference" by Herbert Schildt, McGraw Hill Publications.
2. "Effective Java" by Joshua Bloch, Pearson Education.

1.12 Assignments

- One of the components of a computer is its CPU. What is a CPU and what role does it play in a computer?
- Explain what is meant by an “asynchronous event.” Give some examples.
- What is the difference between a “compiler” and an “interpreter”?
- Explain the difference between high-level languages and machine language.
- If you have the source code for a Java program, and you want to run that program, you will need both a compiler and an interpreter. What does the Java compiler do, and what does the Java interpreter do?
- What is a subroutine?
- Java is an object-oriented programming language. What is an object?
- What is a variable? (There are four different ideas associated with variables in Java. Try to mention all four aspects in your answer. Hint: One of the aspects is the variable’s name.)
- Java is a “platform-independent language.” What does this mean?
- What is the “Internet”? Give some examples of how it is used.

Unit 2: Programming in the Small I: Names and Things

2

Unit Structure

2.1 Learning Objectives

After studying this unit learner should be able to

- The Basic Java Application
- Variables and the Primitive Types
- Strings, Classes, Objects, and Subroutines
- Text Input and Output
- Details of Expressions
- Programming Environments

2.2 Introduction

On a basic level (the level of machine language), a computer can perform only very simple operations. A computer performs complex tasks by stringing together large numbers of such operations. Such tasks must be “scripted” in complete and perfect detail by programs. Creating complex programs will never be really easy, but the difficulty can be handled to some extent by giving the program a clear overall structure. The design of the overall structure of a program is what I call “programming in the large.”

Programming in the small, which is sometimes called coding, would then refer to filling in the details of that design. The details are the explicit, step-by-step instructions for performing fairly small-scale tasks. When you do coding, you are working “close to the machine,” with some of the same concepts that you might use in machine language: memory locations, arithmetic operations, loops and branches. In a high-level language such as Java, you get to work with these concepts on a level several steps above machine language. However, you still have to worry about getting all the details exactly right.

This chapter and the next examine the facilities for programming in the small in the Java programming language. Don’t be misled by the term “programming in the small” into thinking that this material is easy or unimportant. This material is an essential foundation for all types of programming. If you don’t understand it, you can’t write programs, no matter how good you get at designing their large-scale structure.

The last section of this chapter discusses programming environments. That section contains information about how to compile and run Java programs, and you should take a look at it before trying to write and use your own programs or trying to use the sample programs in this book.

2.3 The Basic Java Application

A program is a sequence of instructions that a computer can execute to perform some task. A simple enough idea, but for the computer to make any use of the instructions, they must be written in a form that the computer can use. This means that programs have to be written in programming languages. Programming languages differ from ordinary human languages in being completely unambiguous and very strict about what is and is not allowed in a program. The rules that determine what is allowed are called the syntax of the language. Syntax rules specify the basic vocabulary of the language and how programs can be constructed using things like loops, branches, and subroutines. A syntactically correct program is one that can be successfully compiled or interpreted; programs that have syntax errors will be rejected (hopefully with a useful error message that will help you fix the problem).

So, to be a successful programmer, you have to develop a detailed knowledge of the syntax of the programming language that you are using. However, syntax is only part of the story. It's not enough to write a program that will run—you want a program that will run and produce the correct result! That is, the meaning of the program has to be right. The meaning of a program is referred to as its semantics. More correctly, the semantics of a programming language is the set of rules that determine the meaning of a program written in that language. A semantically correct program is one that does what you want it to.

Furthermore, a program can be syntactically and semantically correct but still be a pretty bad program. Using the language correctly is not the same as using it well. For example, a good program has "style." It is written in a way that will make it easy for people to read and to understand. It follows conventions that will be familiar to other programmers. And it has an overall design that will make sense to human readers. The computer is completely oblivious to such things, but to a human reader, they are paramount. These aspects of programming are sometimes referred to as pragmatics. (I will often use the more common term style.)

When I introduce a new language feature, I will explain the syntax, the semantics, and some of the pragmatics of that feature. You should memorize the syntax; that's the easy part. Then you should get a feeling for the semantics by following the examples given, making sure that you understand how they work, and, ideally, writing short programs of your own to test your understanding. And you should try to appreciate and absorb the pragmatics his means learning how to use the language feature well, with style that will earn you the admiration of other programmers.

Of course, even when you've become familiar with all the individual features of the language, that doesn't make you a programmer. You still have to learn how to

construct complex programs to solve particular problems. For that, you'll need both experience and taste. You'll find hints about software development throughout this textbook.

* * *

We begin our exploration of Java with the problem that has become traditional for such beginnings: to write a program that displays the message "Hello World!". This might seem like a trivial problem, but getting a computer to do this is really a big first step in learning a new programming language (especially if it's your first programming language). It means that you understand the basic process of:

1. getting the program text into the computer,
2. compiling the program, and
3. running the compiled program.

The first time through, each of these steps will probably take you a few tries to get right. I won't go into the details here of how you do each of these steps; it depends on the particular computer and Java programming environment that you are using. See Section 2.6 for information about creating and running Java programs in specific programming environments. But in general, you will type the program using some sort of text editor and save the program in a file. Then, you will use some command to try to compile the file. You'll either get a message that the program contains syntax errors, or you'll get a compiled version of the program. In the case of Java, the program is compiled into Java bytecode, not into machine language. Finally, you can run the compiled program by giving some appropriate command. For Java, you will actually use an interpreter to execute the Java bytecode. Your programming environment might automate some of the steps for you—for example, the compilation step is often done automatically but you can be sure that the same three steps are being done in the background.

Here is a Java program to display the message "Hello World!". Don't expect to understand what's going on here just yet; some of it you won't really understand until a few chapters from now:

```
/** A program to display the message
* "Hello World!" on standard output.
*/
public class HelloWorld {
     public static void main(String[] args) {
          System.out.println("Hello World!");
     }
} // end of class HelloWorld
```

The command that actually displays the message is:

```
System.out.println("Hello World!");
```

This command is an example of a subroutine call statement . It uses a “built-in subroutine” named System.out.println to do the actual work. Recall that a subroutine consists of the instructions for performing some task, chunked together and given a name. That name can be used to “call” the subroutine whenever that task needs to be performed. A built-in subroutine is one that is already defined as part of the language and therefore automatically available for use in any program.

When you run this program, the message “Hello World!” (Without the quotes) will be displayed on standard output. Unfortunately, I can’t say exactly what that means! Java is meant to run on many different platforms, and standard output will mean different things on different platforms. However, you can expect the message to show up in some convenient or inconvenient place. (If you use a command-line interface, like that in Oracle’s Java Development Kit, you type in a command to tell the computer to run the program. The computer will type the output from the program, Hello World!, on the next line. In an integrated development environment such as Eclipse, the output might appear somewhere in one of the environment’s windows.)

You must be curious about all the other stuff in the above program. Part of it consists of comments. Comments in a program are entirely ignored by the computer; they are there for human readers only. This doesn’t mean that they are unimportant. Programs are meant to be read by people as well as by computers, and without comments, a program can be very difficult to understand. Java has two types of comments. The first type begins with // and extends to the end of a line. There is a comment of this form on the last line of the above program. The computer ignores the // and everything that follows it on the same line. The second type of comment starts with /* and ends with */, and it can extend over more than one line. The first three lines of the program are an example of this second type of comment. (A comment that actually begins with /**, like this one does, has special meaning; it is a “Javadoc” comment that can be used to produce documentation for the program.)

Everything else in the program is required by the rules of Java syntax. All programming in Java is done inside “classes.” The first line in the above program (not counting the comment) says that this is a class named HelloWorld. “HelloWorld,” the name of the class, also serves as the name of the program. Not every class is a program. In order to define a program, a class must include a subroutine named main, with a definition that takes the form:

```
public static void main(String[] args) {
     <statements>
}
```

When you tell the Java interpreter to run the program, the interpreter calls this main() subroutine, and the statements that it contains are executed. These statements make up the script that tells the computer exactly what to do when the program is executed. The main() routine can call other subroutines that are defined in the same

class or even in other classes, but it is the main() routine that determines how and in what order the other subroutines are used.

The word “public” in the first line of main() means that this routine can be called from out-side the program. This is essential because the main() routine is called by the Java interpreter, which is something external to the program itself. The remainder of the first line of the routine is harder to explain at the moment; for now, just think of it as part of the required syntax.

The definition of the subroutine that is, the instructions that say what it does—consists of the sequence of “statements” enclosed between braces, {and}. Here, I’ve used <statements> as a placeholder for the actual statements that make up the program. Throughout this textbook, I will always use a similar format: anything that you see in <this style of text> (italic in angle brackets) is a placeholder that describes something you need to type when you write an actual program.

As noted above, a subroutine can’t exist by itself. It has to be part of a “class”. A program is defined by a public class that takes the form:

```
<optional-package-declaration>
<optional-imports >
public class <program-name> {
<optional-variable-declarations-and-subroutines >
        public static void main(String[] args) {
                <statements >
        }
        <optional-variable-declarations-and-subroutines>
}
```

The first two lines have to do with using packages. A package is a group of classes.

The <program-name> in the line that begins “public class” is the name of the program, as well as the name of the class. (Remember, again, that <program-name> is a placeholder for the actual name!) If the name of the class is HelloWorld, then the class must be saved in a file called HelloWorld.java. When this file is compiled, another file named HelloWorld.class will be produced. This class file, HelloWorld.class, contains the translation of the program into Java bytecode, which can be executed by a Java interpreter. HelloWorld.java is called the source code for the program. To execute the program, you only need the compiled class file, not the source code.

The layout of the program on the page, such as the use of blank lines and indentation, is not part of the syntax or semantics of the language. The computer

doesn’t care about layout you could run the entire program together on one line as far as it is concerned. However, layout is important to human readers, and there are certain style guidelines for layout that are followed by most programmers.

Also note that according to the above syntax specification, a program can contain other subroutines besides main(), as well as things called “variable declarations.”

2.4 Variables and the Primitive Types

Names are fundamental to programming. In programs, names are used to refer to many different sorts of things. In order to use those things, a programmer must understand the rules for giving names to them and the rules for using the names to work with them. That is, the programmer must understand the syntax and the semantics of names.

According to the syntax rules of Java, the most basic names are identifiers. Identifiers can be used to name classes, variables, and subroutines. An identifier is a sequence of one or more characters. It must begin with a letter or underscore and must consist entirely of letters, digits, and underscores. (“Underscore” refers to the character ’ ’.) For example, here are some legal identifiers:

N n rate x15 quite_a_long_name HelloWorld

No spaces are allowed in identifiers; HelloWorld is a legal identifier, but “Hello World” is not. Upper case and lower case letters are considered to be different, so that HelloWorld, helloworld, HELLOWORLD, and hElloWorLD are all distinct names. Certain words are reserved for special uses in Java, and cannot be used as identifiers. These reserved words include:

class, public, static, if, else, while, and several dozen other words. (Remember that reserved words are not identifiers, since they can’t be used as names for things.)

Java is actually pretty liberal about what counts as a letter or a digit. Java uses the Unicode character set, which includes thousands of characters from many different languages and different alphabets, and many of these characters count as letters or digits. However, I will be sticking to what can be typed on a regular English keyboard.

The pragmatics of naming includes style guidelines about how to choose names for things. For example, it is customary for names of classes to begin with upper case letters, while names of variables and of subroutines begin with lower case letters; you can avoid a lot of confusion by following this standard convention in your own programs. Most Java programmers do not use underscores in names, although some do use them at the beginning of the names of certain kinds of variables. When

a name is made up of several words, such as HelloWorld or interestRate, it is customary to capitalize each word, except possibly the first; this is sometimes referred to as camel case, since the upper case letters in the middle of a name are supposed to look something like the humps on a camel's back.

Finally, I'll note that in addition to simple identifiers, things in Java can have compound names which consist of several simple names separated by periods. (Compound names are also called qualified names.) You've already seen an example: System.out.println. The idea here is that things in Java can contain other things. A compound name is a kind of path to an item through one or more levels of containment. The name System.out.println indicates that something called "System" contains something called "out" which in turn contains something called "println".

Variables

Programs manipulate data that are stored in memory. In machine language, data can only be referred to by giving the numerical address of the location in memory where the data is stored. In a high-level language such as Java, names are used instead of numbers to refer to data. It is the job of the computer to keep track of where in memory the data is actually stored; the programmer only has to remember the name. A name used in this way to refer to data stored in memory is called a ***variable***.

Variables are actually rather subtle. Properly speaking, a variable is not a name for the data itself but for a location in memory that can hold data. You should think of a variable as a container or box where you can store data that you will need to use later. The variable refers directly to the box and only indirectly to the data in the box. Since the data in the box can change, a variable can refer to different data values at different times during the execution of the program, but it always refers to the same box. Confusion can arise, especially for beginning programmers, because when a variable is used in a program in certain ways, it refers to the container, but when it is used in other ways, it refers to the data in the container. You'll see examples of both cases below.

In Java, the only way to get data into a variable—that is, into the box that the variable names is with an assignment statement. An assignment statement takes the form:

<variable > = <expression>;

where <expression> represents anything that refers to or computes a data value. When the computer comes to an assignment statement in the course of executing a program, it evaluates the expression and puts the resulting data value into the variable. For example, consider the simple assignment statement

rate = 0.07;

The <variable> in this assignment statement is rate, and the <expression> is the number 0.07. The computer executes this assignment statement by putting the

number 0.07 in the variable rate, replacing whatever was there before. Now, consider the following more complicated assignment statement, which might come later in the same program:

```
interest = rate * principal;
```

Here, the value of the expression “rate * principal” is being assigned to the variable interest. In the expression, the * is a “multiplication operator” that tells the computer to multiply rate times principal. The names rate and principal are themselves variables, and it is really the values stored in those variables that are to be multiplied. We see that when a variable is used in an expression, it is the value stored in the variable that matters; in this case, the variable seems to refer to the data in the box, rather than to the box itself. When the computer executes this assignment statement, it takes the value of rate, multiplies it by the value of principal, and stores the answer in the box referred to by interest. When a variable is used on the left-hand side of an assignment statement, it refers to the box that is named by the variable.

(Note, by the way, that an assignment statement is a command that is executed by the computer at a certain time. It is not a statement of fact. For example, suppose a program includes the statement “rate = 0.07;”. If the statement “interest = rate * principal;” is executed later in the program, can we say that the principal is multiplied by 0.07? No! The value of rate might have been changed in the meantime by another statement. The meaning of an assignment statement is completely different from the meaning of an equation in mathematics, even though both use the symbol ”=”.)

Types

A variable in Java is designed to hold only one particular type of data; it can legally hold that type of data and no other. The compiler will consider it to be a syntax error if you try to violate this rule by assigning a value of the wrong type to a variable. We say that Java is a strongly typed language because it enforces this rule.

There are eight so-called primitive types built into Java. The primitive types are named **byte, short, int, long, float, double, char,** and **boolean**. The first four types hold integers (whole numbers such as 17, -38477, and 0). The four integer types are distinguished by the ranges of integers they can hold. The float and double types hold real numbers (such as 3.6 and -145.99). Again, the two real types are distinguished by their range and accuracy. A variable of type char holds a single character from the Unicode character set. And a variable of type boolean holds one of the two logical values true or false.

Any data value stored in the computer’s memory must be represented as a binary number, that is as a string of zeros and ones. A single zero or one is called a bit. A string of eight bits is called a byte. Memory is usually measured in terms of bytes. Not surprisingly, the byte data type refers to a single byte of memory. A variable of type byte holds a string of eight bits, which can represent any of the integers between

-128 and 127, inclusive. (There are 256 integers in that range; eight bits can represent 256 two raised to the power eight different values.) As for the other integer types,

- short corresponds to two bytes (16 bits). Variables of type **short** have values in the range -32768 to 32767.
- int corresponds to four bytes (32 bits). Variables of type **int** have values in the range -2147483648 to 2147483647.
- long corresponds to eight bytes (64 bits). Variables of type **long** have values in the range -9223372036854775808 to 9223372036854775807.

You don't have to remember these numbers, but they do give you some idea of the size of integers that you can work with. Usually, for representing integer data you should just stick to the int data type, which is good enough for most purposes.

The **float** data type is represented in four bytes of memory, using a standard method for encoding real numbers. The maximum value for a float is about 10 raised to the power 38. A **float** can have about 7 significant digits. (So that 32.3989231134 and 32.3989234399 would both have to be rounded off to about 32.398923 in order to be stored in a variable of type float.) A double takes up 8 bytes, can range up to about 10 to the power 308, and has about 15 significant digits. Ordinarily, you should stick to the double type for real values.

A variable of type char occupies two bytes in memory. The value of a char variable is a single character such as A, *, x, or a space character. The value can also be a special character such a tab or a carriage return or one of the many Unicode characters that come from different languages. Values of type char are closely related to integer values, since a character is actually stored as a 16-bit integer code number. In fact, we will see that chars in Java can actually be used like integers in certain situations.

It is important to remember that a primitive type value is represented using only a certain, finite number of bits. So, an int can't be an arbitrary integer; it can only be an integer in a certain finite range of values. Similarly, float and double variables can only take on certain values. They are not true real numbers in the mathematical sense. For example, the mathematical constant π can only be approximated by a value of type float or double, since it would require an infinite number of decimal places to represent it exactly. For that matter, many simple numbers such as 1/3 can only be approximated by floats and doubles.

Literals

A data value is stored in the computer as a sequence of bits. In the computer's memory, it doesn't look anything like a value written on this page. You need a way to include constant values in the programs that you write. In a program, you represent

constant values as literals. A literal is something that you can type in a program to represent a value. It is a kind of name for a constant value.

For example, to type a value of type char in a program, you must surround it with a pair of single quote marks, such as ’A’, ’*’, or ’x’. The character and the quote marks make up a literal of type char. Without the quotes, A would be an identifier and * would be a multiplication operator. The quotes are not part of the value and are not stored in the variable; they are just a convention for naming a particular character constant in a program. If you want to store the character A in a variable ch of type char, you could do so with the assignment statement

ch = ’A’;

Certain special characters have special literals that use a backslash, \, as an “escape character.”

In particular, a tab is represented as ’\t’, a carriage return as ’\r’, a linefeed as ’\n’, the single quote character as ’\’’, and the backslash itself as ’\\’. Note that even though you type two characters between the quotes in ’\t’, the value represented by this literal is a single tab character.

Numeric literals are a little more complicated than you might expect. Of course, there are the obvious literals such as 317 and 17.42. But there are other possibilities for expressing numbers in a Java program. First of all, real numbers can be represented in an exponential form such as 1.3e12 or 12.3737e-108. The “e12” and “e-108” represent powers of 10, so that 1.3e12 means 1.3 times 1012 and 12.3737e-108 means 12.3737 times 10−108. This format can be used to express very large and very small numbers. Any numeric literal that contains a decimal point or exponential is a literal of type double. To make a literal of type float, you have to append an “F” or “f” to the end of the number. For example, “1.2F” stands for 1.2 considered as a value of type float. (Occasionally, you need to know this because the rules of Java say that you can’t assign a value of type double to a variable of type float, so you might be confronted with a ridiculous-seeming error message if you try to do something like “x = 1.2;” if x is a variable of type float. You have to say “x = 1.2F;". This is one reason why I advise sticking to type double for real numbers.)

Even for integer literals, there are some complications. Ordinary integers such as 177777 and -32 are literals of type byte, short, or int, depending on their size. You can make a literal of type long by adding “L” as a suffix. For example: 17L or 728476874368L. As another complication, Java allows binary, octal (base-8), and hexadecimal (base-16) literals. I don’t want to cover number bases in detail, but in case you run into them in other people’s programs, it’s worth knowing a few things: Octal numbers use only the digits 0 through 7. In Java, a numeric literal that begins with a 0 is interpreted as an octal number; for example, the octal literal 045 represents the number 37, not the number 45. Octal numbers are rarely used, but you need to be aware of what happens when you start a number with a zero. Hexadecimal numbers

use 16 digits, the usual digits 0 through 9 and the letters A, B, C, D, E, and F. Upper case and lower case letters can be used interchangeably in this context. The letters represent the numbers 10 through 15. In Java, a hexadecimal literal begins with 0x or 0X, as in 0x45 or 0xFF7A. Finally, binary literals start with 0b or 0B and contain only the digits 0 and 1; for example: 0b10110.

As a final complication, numeric literals can include the underscore character ("_"), which can be used to separate groups of digits. For example, the integer constant for two billion could be written 2 000 000 000, which is a good deal easier to decipher than 2000000000. There is no rule about how many digits have to be in each group. Underscores can be especially useful in long binary numbers; for example, 0b1010 1100 1011.

I will note that hexadecimal numbers can also be used in character literals to represent arbitrary Unicode characters. A Unicode literal consists of \u followed by four hexadecimal digits. For example, the character literal '\u00E9' represents the Unicode character that is an "e" with an acute accent.

For the type boolean, there are precisely two literals: true and false. These literals are typed just as I've written them here, without quotes, but they represent values, not variables.

Boolean values occur most often as the values of conditional expressions. For example,

rate > 0.05

is a boolean-valued expression that evaluates to true if the value of the variable rate is greater than 0.05, and to false if the value of rate is less than or equal to 0.05. boolean-valued expressions are used extensively in control structures. Of course, boolean values can also be assigned to variables of type boolean. For example, if test is a variable of type boolean, then both of the following assignment statements are legal:

```
test = true;
test = rate > 0.05;
```

Strings and String Literals

Java has other types in addition to the primitive types, but all the other types represent objects rather than "primitive" data values. For the most part, we are not concerned with objects for the time being. However, there is one predefined object type that is very important: the type String. (String is a type, but not a primitive type; it is in fact the name of a class, and we will return to that aspect of strings in the next section.)

A value of type String is a sequence of characters. You've already seen a string literal: "Hello World!". The double quotes are part of the literal; they have to be typed in the program. However, they are not part of the actual String value, which consists of just

the characters between the quotes. A string can contain any number of characters, even zero. A string with no characters is called the empty string and is represented by the literal "", a pair of double quote marks with nothing between them. Remember the difference between single quotes and double quotes! Single quotes are used for char literals and double quotes for String literals! There is a big difference between the String "A" and the char 'A'.

Within a string literal, special characters can be represented using the backslash notation. Within this context, the double quote is itself a special character. For example, to represent the string value

```
I said, "Are you listening!"
```

with a linefeed at the end, you would have to type the string literal:

```
"I said, \"Are you listening! \"\n"
```

You can also use \t, \r, \\, and Unicode sequences such as \u00E9 to represent other special characters in string literals.

Variables in Programs

A variable can be used in a program only if it has first been declared. A variable declaration statement is used to declare one or more variables and to give them names. When the computer executes a variable declaration, it sets aside memory for the variable and associates the variable's name with that memory. A simple variable declaration takes the form:
<type-name> <variable-name-or-names>;

The <variable-name-or-names> can be a single variable name or a list of variable names separated by commas. (We'll see later that variable declaration statements can actually be somewhat more complicated than this.) Good programming style is to declare only one variable in a declaration statement, unless the variables are closely related in some way. For example:

```
int numberOfStudents;
String name;
double x, y;
boolean isFinished;
char firstInitial, middleInitial, lastInitial;
```

It is also good style to include a comment with each variable declaration to explain its purpose in the program, or to give other information that might be useful to a human reader.
For example:

```
double principal; // Amount of money invested.
```

```
      double interestRate; // Rate as a decimal, not %.
```

Here is a simple program using some variables and assignment statements:

```
/* This class implements a simple program that will compute the amount of
interest that is earned on $17,000 invested at an interest rate of 0.027 for
one year. The interest and the value of the investment after one year are
printed to standard output. */

public class Interest {
      public static void main(String[] args) {
      /* Declare the variables. */
      double principal; // The value of the investment.
      double rate; // The annual interest rate.
      double interest; // Interest earned in one year.
      /* Do the computations. */
      principal = 17000;
      rate = 0.027;
      interest = principal * rate; // Compute the interest.
      principal = principal + interest;
      // Compute value of investment after one year, with interest.
      // (Note: The new value replaces the old value of principal.)
      /* Output the results. */
      System.out.print("The interest earned is $");
      System.out.println(interest);
      System.out.print("The value of the investment after one year is $");
      System.out.println(principal);
      } // end of main()
} // end of class Interest
```

This program uses several subroutine call statements to display information to the user of the program. Two different subroutines are used: System.out.print and System.out.println. The difference between these is that System.out.println adds a linefeed after the end of the information that it displays, while System.out.print does not. Thus, the value of interest, which is displayed by the subroutine call “System.out.println(interest);”, follows on the same line as the string displayed by the previous System.out.print statement. Note that the value to be displayed by System.out.print or System.out.println is provided in parentheses after the subroutine name. This value is called a parameter to the subroutine. A parameter provides a subroutine with information it needs to perform its task. In a subroutine call statement, any parameters are listed in parentheses after the subroutine name. Not all subroutines have parameters. If there are no parameters in a subroutine call statement, the subroutine name must be followed by an empty pair of parentheses.

2.5 Strings, Classes, Objects, and Subroutines

Built-in Subroutines and Functions

Recall that a subroutine is a set of program instructions that have been chunked together and given a name. A subroutine is designed to perform some task. To get that task performed in a program, you can "call" the subroutine using a subroutine call statement. In Java, every subroutine is contained either in a class or in an object. Some classes that are standard parts of the Java language contain predefined subroutines that you can use. A value of type String, which is an object, contains subroutines that can be used to manipulate that string. These subroutines are "built into" the Java language. You can call all these subroutines without understanding how they were written or how they work. Indeed, that's the whole point of subroutines: A subroutine is a "black box" which can be used without knowing what goes on inside.

Let's first consider subroutines that are part of a class. One of the purposes of a class is to group together some variables and subroutines, which are contained in that class. These variables and subroutines are called static members of the class. You've seen one example: In a class that defines a program, the main() routine is a static member of the class. The parts of a class definition that define static members are marked with the reserved word "static", such as the word "static" in `public static void main...`

When a class contains a static variable or subroutine, the name of the class is part of the full name of the variable or subroutine. For example, the standard class named System contains a subroutine named exit. To use that subroutine in your program, you must refer to it as System.exit. This full name consists of the name of the class that contains the subroutine, followed by a period, followed by the name of the subroutine. This subroutine requires an integer as its parameter, so you would actually use it with a subroutine call statement such as

```
System.exit(0);
```

Calling System.exit will terminate the program and shut down the Java Virtual Machine. You could use it if you had some reason to terminate the program before the end of the main routine. (The parameter tells the computer why the program was terminated. A parameter value of 0 indicates that the program ended normally. Any other value indicates that the program was terminated because an error was detected, so you could call System.exit(1) to indicate that the program is ending because of an error. The parameter is sent back to the operating system; in practice, the value is usually ignored by the operating system.)

System is just one of many standard classes that come with Java. Another useful class is called Math. This class gives us an example of a class that contains static variables: It includes the variables Math.PI and Math.E whose values are the mathematical constants π and e. Math also contains a large number of mathematical "functions." Every subroutine performs some specific task. For some subroutines, that task is to compute or retrieve some data value. Subroutines of this type are called functions. We say that a function returns a value. Generally, the returned value is meant to be used somehow in the program that calls the function.

You are familiar with the mathematical function that computes the square root of a number. The corresponding function in Java is called Math.sqrt. This function is a static member subroutine of the class named Math. If x is any numerical value, then Math.sqrt(x) computes and returns the square root of that value. Since Math.sqrt(x) represents a value, it doesn't make sense to put it on a line by itself in a subroutine call statement such as

```
Math.sqrt(x); // This doesn't make sense!
```

What, after all, would the computer do with the value computed by the function in this case? You have to tell the computer to do something with the value. You might tell the computer to display it:

```
System.out.print( Math.sqrt(x) ); // Display the square root of x.
```

or you might use an assignment statement to tell the computer to store that value in a variable:

```
lengthOfSide = Math.sqrt(x);
```

The function call Math.sqrt(x) represents a value of type double, and it can be used anyplace where a numeric literal of type double could be used. The x in this formula represents the parameter to the subroutine; it could be a variable named "x", or it could be replaced by any expression that represents a numerical value. For example, Math.sqrt(2) computes the square root of 2, and Math.sqrt(a*a+b*b) would be legal as long as a and b are numeric variables.

Classes and Objects

Classes can be containers for static variables and subroutines. However classes also have another purpose. They are used to describe objects. In this role, the class is a type, in the same way that int and double are types. That is, the class name can be used to declare variables. Such variables can only hold one type of value. The values in this case are objects. An object is a collection of variables and subroutines. Every object has an associated class that tells what "type" of object it is. The class of an object specifies what subroutines and variables that object contains. All objects defined by the same class are similar in that they contain similar collections of variables and subroutines. For example, an object might represent a point in the plane, and it might contain variables named x and y to represent the coordinates of that point. Every point object would have an x and a y, but different points would have different values for these variables. A class, named Point for example, could exist to define the common structure of all point objects, and all such objects would then be values of type Point.

As another example, let's look again at System.out.println. System is a class, and out is a static variable within that class. However, the value of System.out is an object, and System.out.println is actually the full name of a subroutine that is contained in the object System.out. You don't need to understand it at this point, but the object referred to by System.out is an object of the class PrintStream. PrintStream is another class that is a standard part of Java. Any object of type PrintStream is a destination to

which information can be printed; any object of type PrintStream has a println subroutine that can be used to send information to that destination. The object System.out is just one possible destination, and System.out.println is a subroutine that sends information to that particular destination. Other objects of type PrintStream might send information to other destinations such as files or across a network to other computers. This is object-oriented programming: Many different things which have something in common they can all be used as destinations for output can all be used in the same way through a println subroutine. The PrintStream class expresses the commonalities among all these objects.

The dual role of classes can be confusing, and in practice most classes are designed to perform primarily or exclusively in only one of the two possible roles. Fortunately, you will not need to worry too much about it until we start working with objects in a more serious way, in Chapter 5. By the way, since class names and variable names are used in similar ways, it might be hard to tell which is which. Remember that all the built-in, predefined names in Java follow the rule that class names begin with an upper case letter while variable names begin with a lower case letter. While this is not a formal syntax rule, I strongly recommend that you follow it in your own programming. Subroutine names should also begin with lower case letters. There is no possibility of confusing a variable with a subroutine, since a subroutine name in a program is always followed by a left parenthesis.

As one final general note, you should be aware that subroutines in Java are often referred to as methods. Generally, the term “method” means a subroutine that is contained in a class or in an object. Since this is true of every subroutine in Java, every subroutine in Java is a method. The same is not true for other programming languages, and for the time being, I will prefer to use the more general term, “subroutine.” However, I should note that some people prefer to use the term “method” from the beginning.

2.6 Text Input and Output

Basic Output and Formatted Output

The most basic output function is System.out.print(x), where x can be a value or expression of any type. If the parameter, x, is not already a string, it is converted to a value of type String, and the string is then output to the destination called standard output . (Generally, this means that the string is displayed to the user; however, in GUI programs, it outputs to a place where a typical user is unlikely to see it. Furthermore, standard output can be “redirected” to write to a different output destination.

Nevertheless, for the type of program that we are working with now, the purpose of System.out is to display text to the user.)

System.out.println(x) outputs the same text as System.out.print, but it follows that text by a line feed, which means that any subsequent output will be on the next line. It is possible to use this function with no parameter, System.out.println(), which outputs nothing but a line feed. Note that System.out.println(x) is equivalent to

```
System.out.print(x);
System.out.println();
```

You might have noticed that System.out.print outputs real numbers with as many digits after the decimal point as necessary, so that for example is output as 3.141592653589793, and numbers that are supposed to represent money might be output as 1050.0 or 43.575. You might prefer to have these numbers output as, for example, 3.14159, 1050.00, and 43.58. Java has a “formatted output” capability that makes it easy to control how real numbers and other values are printed. A lot of formatting options are available. I will cover just a few of the simplest and most commonly used possibilities here.

The function System.out.printf can be used to produce formatted output. (The name “printf,” which stands for “print formatted,” is copied from the C and C++ programming languages, where this type of output originated.) System.out.printf takes one or more parameters. The first parameter is a String that specifies the format of the output. This parameter is called the format string. The remaining parameters specify the values that are to be output. Here is a statement that will print a number in the proper format for a dollar amount, where amount is a variable of type double:

```
System.out.printf( "%1.2f", amount );
```

The output format for a value is give by a format specifier in the format string. In this example, the format specifier is %1.2f. The format string (in the simple cases that I cover here) contains one format specifier for each of the values that is to be output. Some typical format specifiers are %d, %12d, %10s, %1.2f, %15.8e and %1.8g. Every format specifier begins with a percent sign (%) and ends with a letter, possibly with some extra formatting information in between. The letter specifies the type of output that is to be produced. For example, in %d and %12d, the “d” specifies that an integer is to be written. The “12” in %12d specifies the minimum number of spaces that should be used for the output. If the integer that is being output takes up fewer than 12 spaces, extra blank spaces are added in front of the integer to bring the total up to 12. We say that the output is “right-justified in a field of length 12.” A very large value is not forced into 12 spaces; if the value has more than 12 digits, all the digits will be printed, with no extra spaces. The specifier %d means the same as %1d that is, an integer will be printed using just as many spaces as necessary. (The “d,” by the way, stands for “decimal” that is, base-10 numbers. You can replace the “d” with an “x” to output an integer value in hexadecimal form.) The letter “s” at the end of a format specifier can

be used with any type of value. It means that the value should be output in its default format, just as it would be in unformatted output. A number, such as the “20” in %20s, can be added to specify the (minimum) number of characters. The “s” stands for “string,” and it can be used for values of type String. It can also be used for values of other types; in that case the value is converted into a String value in the usual way. The format specifiers for values of type double are more complicated. An “f”, as in %1.2f, is used to output a number in “floating-point” form, that is with digits after a decimal point. In %1.2f, the “2” specifies the number of digits to use after the decimal point. The “1” specifies the (minimum) number of characters to output; a “1” in this position effectively means that just as many characters as are necessary should be used. Similarly, %12.3f would specify a floating-point format with 3 digits after the decimal point, right-justified in a field of length 12.

Very large and very small numbers should be written in exponential format, such as 6.00221415e23, representing “6.00221415 times 10 raised to the power 23.” A format specifier such as %15.8e specifies an output in exponential form, with the “8” telling how many digits to use after the decimal point. If you use “g” instead of “e”, the output will be in exponential form for very small values and very large values and in floating-point form for other values. In %1.8g, the 8 gives the total number of digits in the answer, including both the digits before the decimal point and the digits after the decimal point.

For numeric output, the format specifier can include a comma (“,”), which will cause the digits of the number to be separated into groups, to make it easier to read big numbers. In the United States, groups of three digits are separated by commas. For example, if x is one billion, then System.out.printf("%,d",x) will output 1,000,000,000. In other countries, the separator character and the number of digits per group might be different. The comma should come at the beginning of the format specifier, before the field width; for example: %,12.3f. If you want the output to be left-justified instead of right justified, add a minus sign to the beginning of the format specifier: for example, %-20s. In addition to format specifiers, the format string in a printf statement can include other characters. These extra characters are just copied to the output. This can be a convenient way to insert values into the middle of an output string. For example, if x and y are variables of type int, you could say

```
System.out.printf("The product of %d and %d is %d", x, y, x*y);
```

When this statement is executed, the value of x is substituted for the first %d in the string, the value of y for the second %d, and the value of the expression x*y for the third, so the output would be something like “The product of 17 and 42 is 714” (quotation marks not included in output!).

To output a percent sign, use the format specifier %% in the format string. You can use %n to output a line feed. You can also use a backslash, \, as usual in strings to output special characters such as tabs and double quote characters.

A First Text Input Example

For some unfathomable reason, Java has traditionally made it difficult to read data typed in by the user of a program. You've already seen that output can be displayed to the user using the subroutine System.out.print. This subroutine is part of a predefined object called System.out. The purpose of this object is precisely to display output to the user. There is a corresponding object called System.in that exists to read data input by the user, but it provides only very primitive input facilities, and it requires some advanced Java programming skills to use it effectively.

Java 5.0 finally made input a little easier with a new Scanner class. Java 6 introduced the Console class for communicating with the user, but Console has its own problems. (It is not always available, and it can only read strings, not numbers.)

Using Scanner for Input

First, since Scanner is defined in the package java.util, you should add the following import directive to your program at the beginning of the source code file, before the "public class. . . ":

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

Then include the following statement at the beginning of your main() routine:

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

This creates a variable named stdin of type Scanner. (You can use a different name for the variable if you want; "stdin" stands for "standard input.") You can then use stdin in your program to access a variety of subroutines for reading user input. For example, the function stdin.nextInt() reads one value of type int from the user and returns it. There are corresponding methods for reading other types of data, including stdin.nextDouble(), stdin.nextLong(), and stdin.nextBoolean(). (stdin.nextBoolean() will only accept "true" or "false" as input.) These subroutines can read more than one value from a line. As a simple example, here is a Interest.java that uses Scanner for user input

```
import java.util.Scanner;

public class Interest {
      public static void main(String[] args) {
      Scanner stdin = new Scanner( System.in ); // Create the Scanner.
      double principal; // The value of the investment.
      double rate; // The annual interest rate.
      double interest; // The interest earned during the year.
      System.out.print("Enter the initial investment: ");
      principal = stdin.nextDouble();
      System.out.print("Enter the annual interest rate (as a decimal): ");
```

```
        rate = stdin.nextDouble();
        interest = principal * rate; // Compute this year's interest.
        principal = principal + interest; // Add it to principal.
        System.out.printf("The amount of interest is $%1.2f%n", interest);
        System.out.printf("The value after one year is $%1.2f%n", principal);
        } // end of main()
} // end of class Interest
```

2.7 Details of Expressions

This section takes a closer look at expressions. Recall that an expression is a piece of program code that represents or computes a value. An expression can be a literal, a variable, a function call, or several of these things combined with operators such as + and >. The value of an expression can be assigned to a variable, used as a parameter in a subroutine call, or combined with other values into a more complicated expression. (The value can even, in some cases, be ignored, if that's what you want to do; this is more common than you might think.) Expressions are an essential part of programming. So far, this book has dealt only informally with expressions. This section tells you the more-or-less complete story (leaving out some of the less commonly used operators).

The basic building blocks of expressions are literals (such as 674, 3.14, true, and 'X'), variables, and function calls. Recall that a function is a subroutine that returns a value. You've already seen some examples of functions, such as the input routines from the TextIO class and the mathematical functions from the Math class.

The Math class also contains a couple of mathematical constants that are useful in mathematical expressions: Math.PI represents π (the ratio of the circumference of a circle to its diameter), and Math.E represents e (the base of the natural logarithms). These "constants" are actually member variables in Math of type double. They are only approximations for the mathematical constants, which would require an infinite number of digits to specify exactly. The standard class Integer contains a couple of constants related to the int data type: Integer.MAX VALUE is the largest possible int, 2147483647, and Integer.MIN VALUE is the smallest int, -2147483648. Similarly, the class Double contains some constants related to type double. Double.MAX VALUE is the largest value of type double, and Double.MIN VALUE is the smallest positive value. It also has constants to represent infinite values, Double.POSITIVE INFINITY and Double.NEGATIVE INFINITY, and the special value Double.NaN to represent an undefined value. For example, the value of Math.sqrt(-1) is Double.NaN.

Literals, variables, and function calls are simple expressions. More complex expressions can be built up by using operators to combine simpler expressions. Operators include + for adding two numbers, > for comparing two values, and so on.

When several operators appear in an expression, there is a question of precedence, which determines how the operators are grouped for evaluation. For example, in the expression “A + B * C”, B*C is computed first and then the result is added to A. We say that multiplication (*) has higher precedence than addition (+). If the default precedence is not what you want, you can use parentheses to explicitly specify the grouping you want. For example, you could use “(A + B) * C” if you want to add A to B first and then multiply the result by C.

The rest of this section gives details of operators in Java. The number of operators in Java is quite large. I will not cover them all here, but most of the important ones are here.

Operators

They are the characters/symbols used to manipulate data. Operators can have one or more operand on which they perform a function. The operators in java can be classified in to following categories:

Arithmetic Operators

Operator	**Use**
+	Addition of two values Ex: 20+10 gives 30
-	Subtraction of two values Ex: 20-10 gives 10
*	Multiplication of two values Ex: 20*10 gives 200
/	Division of two values Ex: 20/10 gives 2
%	Reminder/Modulus gives reminder of division of two numbers Ex: 21%2 gives 1
++	Increment operator increase value by 1 **
--	Decrement operator decrease value by 1 **

** if we use ++/-- before operand, the increment/decrement is performed first before using the operand and if we use ++/-- after operand, the increment/decrement is performed first after using the operand.

For Example,

```
a=4;
b=++a;  //give 5 in b and 5 in a;
a=4;
b=a--;  //gives 4 in b and 3 in a
```

Assignment Operators

These operators are used to assign value to the operand. = is assignment operator. It assigns value to its operand for Ex: a=5; +=, -=, *=, /= and %= are the shorthand operators. They perform operation as shown below:

Operator	Use	Meaning
=	a=5;	value 5 is assigned to a
+=	a+=5;	it performs a=a+5.
-=	a-=5;	it performs a=a-5.
*=	a*=5;	it performs a=a*5.
/=	a/=5;	it performs a=a/5.
%=	a%=5;	it performs a=a%5.

Relational Operators

They are also called comparison operators. They are used to compare two operands and returns Boolean value.

Operator	Meaning	Use
==	equality	a==b
!=	not equal	a!=b
>	greater than	a>b
<	less than	a<b
>=	greater than or equal to	a>=b
<=	less than or equal to	a<=b

They are used with if…else statement to build a condition. For example (a>b) returns true if a is greater than b else it returns false.

Logical Operators

&&, || and ! are the logical operators. They are used to check for two conditions simultaneously.

Operator	Meaning	Usage
&&	logical and	(a>b && a>c) check both condition
||	logical or	(a>b || a>c) check either of one condition
!	logical not	!(a>b) check not of condition

Bitwise Operators

&, |, ^, << and >> are bitwise operators. They are used to perform bitwise operations.

Operator	Meaning	Usage
&	AND	a&b
|	OR	a|b
^	EXOR	a^b
<<	left shift	a<<b
>>	right shift	a>>b

The AND, OR and EXOR operations are shown below in truth table.

A	b	a&b	a|b	a^b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The shift operator shift the value of operand specific number of time in left(<<) or right(>>). The left operand specifies the value to be shifted and right operand specifies number of shift.

Miscellaneous Operators

- **instance of operator:** it is used to check whether an object is of a specific class type or not. For example,
  String s=”hello”;
  if(s instance of String)
  {
  System.out.println(“s is of String type”);
  }

- **Ternary operator:** ?: is used as a ternary operator. It has three operands. It is shorter replacement of if…else statement.

  Syntax: var=(expression) ? value1:value2;
  Example: c=(a>b)? a: b; It means c is largest of a or b.

Precedence Rules

If you use several operators in one expression, and if you don’t use parentheses to explicitly indicate the order of evaluation, then you have to worry about the precedence rules that determine the order of evaluation. (Advice: don’t confuse yourself or the reader of your program; use parentheses liberally.) Here is a listing of the operators discussed in this section, listed in order from highest precedence (evaluated first) to lowest precedence (evaluated last):

Unary operators:	++, --, !, unary -, unary +, type-cast
Multiplication and division:	*, /, %
Addition and subtraction:	+, -
Relational operators:	<, >, <=, >=
Equality and inequality:	==, !=
Boolean and:	&&
Boolean or:	||
Conditional operator:	?:
Assignment operators:	=, +=, -=, *=, /=, %=

Operators on the same line have the same precedence. When operators of the same precedence are strung together in the absence of parentheses, unary operators and assignment operators are evaluated right-to-left, while the remaining operators are evaluated left-to-right. For example, A*B/C means (A*B)/C, while A=B=C means A=(B=C). (Can you see how the expression A=B=C might be useful, given that the value of B=C as an expression is the same as the value that is assigned to B?)

2.8 Programming Environments

Although the Java language is highly standardized, the procedures for creating, compiling, and editing Java programs vary widely from one programming environment to another.

There are two basic approaches: a command line environment, where the user types commands and the computer respond, and an integrated development environment (IDE), where the user uses the keyboard and mouse to interact with a graphical user interface. While there is essentially just one command line environment for Java programming, there are several common IDEs, including Eclipse, NetBeans, IntelliJ IDEA, and BlueJ. I cannot give complete or definitive information on Java programming environments in this section, but I will try to give enough information to let you compile and run the examples from this textbook using the command line. (Readers are strongly encouraged to read, compile, and run the examples. This textbook can be used with Java 8 and later.

Getting JDK

The basic development system for Java programming is usually referred to as a JDK (Java Development Kit). For this textbook, you need a JDK for Java 8 or later. Note that Java comes in two versions: a Development Kit version (the JDK) and a Runtime Environment version. A Runtime Environment can be used to run Java programs, but it does not allow you to compile your own Java programs. A Development Kit includes the Runtime Environment but also lets you compile

programs. A JDK will include the command line environment that you need to work with Java on the command line. If you decide to use an IDE, you might still need to download a JDK first; note, however, that both the Eclipse IDE and BlueJ now include a JDK, so you do not need to download a separate JDK to use them.

Java was developed by Sun Microsystems, Inc., which was acquired by the Oracle corporation. It is possible to download a JDK directly from Oracle's web site, but starting with Java 11, the Oracle JDK is meant mostly for commercial use. For personal and educational use, it is probably preferable to use OpenJDK, which has the same functionality as the version available from Oracle and is distributed under a fully free, open-source license. Although OpenJDK can be downloaded from https://jdk.java.net/, which is also owned by Oracle, I recommend downloading from AdoptOpenJDK at this address: https://adoptopenjdk.net/

This site has OpenJDKs for a wider range of platforms, and it provides installers for Mac OS and Windows that make it easier to set up Java on those platforms. (The installer for Mac OS is a .pkg file, and the installer for Windows is a .msi file.) The sample programs and exercises in this textbook will work with JDK versions as old as Java 8. If you download a JDK installer for Windows or Mac OS from AdoptOpenJDK, you can just double-click the installer file to start the installation, if it does not start automatically. If you use the default installation, the installer will set up your computer so that you can use the javac and java commands on the command line.

Command Line Environment

Many modern computer users find the command line environment to be pretty alien and unintuitive. It is certainly very different from the graphical user interfaces that most people are used to. However, it takes only a little practice to learn the basics of the command line environment and to become productive using it. It is useful to know how to use the command line, and it is particularly important for computer science students.

To use a command line programming environment, you will have to open a window where you can type in commands. In Windows, you can open such a command window by running a program named cmd. In Mac OS, you want to run the Terminal program, which can be found in the Utilities folder inside the Applications folder. In Linux, there are several possibilities, including a very old program called xterm; but try looking for "Terminal" in your Applications menu.

No matter what type of computer you are using, when you open a command window, it will display a prompt of some sort. Type in a command at the prompt and press return. The computer will carry out the command, displaying any output in the command window, and will then redisplay the prompt so that you can type another command. One of the central concepts in the command line environment is the current

directory or working directory, which contains files that can be used by the commands that you type. (The words “directory” and “folder” mean the same thing.) Often, the name of the current directory is part of the command prompt. You can get a list of the files in the current directory by typing in the command dir (on Windows) or ls (on Linux and Mac OS). When the window first opens, the current directory is your home directory, where your personal files are stored. You can change the current directory using the cd command with the name of the directory that you want to use. For example, if the current directory is your home directory, then you can change into your Desktop directory by typing the command cd Desktop (and then pressing return).

You might want to create a directory (that is, a folder) to hold your Java work. For example, you might create a directory named javawork in your home directory. You can do this using your computer’s GUI; another way is to use the command line: Open a command window. If you want to put your work directory in a different folder from your home directory, cd into the directory where you want to put it. Then enter the command mkdir javawork to make the directory. When you want to work on programming, open a command window and use the cd command to change into your Java work directory. Of course, you can have more than one working directory for your Java work; you can organize your files any way you like.

The most basic commands for using Java on the command line are javac and java. The javac command is used to compile Java source code, and java is used to run Java programs. These commands, and other commands for working with Java, can be found in a directory named bin inside the JDK directory. If you set things up correctly on your computer, it should recognize these commands when you type them on the command line. Try typing the commands java -version and javac -version. The output from these commands should tell you which version of Java is being used. If you get a message such as “Command not found,” then Java is not correctly configured.

Java should already be configured correctly on Linux, if you have installed Java from the Linux software repositories. The same is true on Mac OS and Windows, if you have used an installer from AdoptOpenJDK.

To test the javac command, create a file HelloWorld.java into your working directory. Type the command:

```
javac HelloWorld.java
```

This will compile HelloWorld.java and will create a bytecode file named HelloWorld.class in the same directory. Note that if the command succeeds, you will not get any response from the computer; it will just redisplay the command prompt to tell you it’s ready for another command. You will then be able to run the program using the java command:

```
java HelloWorld
```

The computer should respond by outputting the message “Hello World!”. Note that although the program is stored in a file named HelloWorld.class, the java command uses the name of the class, HelloWorld, not the name of the file.

Editor

To create your own programs, you will need a text editor. A text editor is a computer program that allows you to create and save documents that contain plain text. It is important that the documents be saved as plain text, that is without any special encoding or formatting information. Word processor documents are not appropriate, unless you can get your word processor to save as plain text. A good text editor can make programming a lot more pleasant.

Linux comes with several text editors. On Windows, you can use notepad in a pinch, but you will probably want something better. For Mac OS, you might download the BBEdit application, which can be used for free. One possibility that will work on any platform is to use jedit , a programmer’s text editor that is itself written in Java and that can be downloaded for free from www.jedit.org. Another popular cross-platform programming editor is Atom, available from atom.io. To work on your programs, you can open a command line window and cd into the working directory where you will store your source code files. Start up your text editor program, such as by double-clicking its icon or selecting it from a Start menu. Type your code into the editor window, or open an existing source code file that you want to modify. Save the file into your working directory. Remember that the name of a Java source code file must end in “.java”, and the rest of the file name must match the name of the class that is defined in the file. Once the file is saved in your working directory, go to the command window and use the javac command to compile it, as discussed above. If there are syntax errors in the code, they will be listed in the command window. Each error message contains the line number in the file where the computer found the error. Go back to the editor and try to fix one or more errors, save your changes, and then try the javac command again. (It’s usually a good idea to just work on the first few errors; sometimes fixing those will make other errors go away.) Remember that when the javac command finally succeeds, you will get no message at all, or possibly just some “warnings”; warnings do not stop a program from running. Then you can use the java command to run your program, as described above. Once you’ve compiled the program, you can run it as many times as you like without recompiling it. That’s really all there is to it: Keep both editor and command-line window open. Edit, save, and compile until you have eliminated all the syntax errors. (Always remember to save the file before compiling it—the compiler only sees the saved file, not the version in the editor window.) When you run the program, you might find that it has semantic errors that

cause it to run incorrectly. In that case, you have to go back to the edit/save/compile loop to try to find and fix the problem.

Integrated Development Environment

Eclipse IDE

In an Integrated Development Environment, everything you need to create, compile, and run programs is integrated into a single package, with a graphical user interface that will be familiar to most computer users. There are a number of different IDEs for Java program development, ranging from fairly simple wrappers around the JDK to highly complex applications with a multitude of features. For a beginning programmer, there is a danger in using an IDE, since the difficulty of learning to use the IDE, on top of the difficulty of learning to program, can be daunting. IDEs have features that are very useful even for a beginning programmer, although a beginner will want to ignore many of their advanced features.

This subsection tells you how to use it for programs that use only standard Java classes. You can download an Eclipse IDE from eclipse.org. When I install Eclipse, I get the “Eclipse IDE for Java Developers” package (not the “installer”) from this web page: https://www.eclipse.org/downloads/packages/

For Windows and Linux, the download is a compressed archive file. You can simply extract the contents of the archive and place the resulting directory wherever you want it on your computer. You will find the Eclipse application in that directory, and you can start Eclipse by double-clicking the application icon. For Mac OS, the download is a .dmg file that contains the Eclipse application. You can open the .dmg file and drag the application to any location that you prefer (probably the Applications folder). Eclipse is a free program. It is itself written in Java. Recent versions of Eclipse include a copy of an OpenJDK (although Eclipse calls it a JRE), so you can use it without downloading a separate JDK. The first time you start Eclipse, you will be asked to specify a workspace, which is the directory where your work will be stored. You can accept the default name, or provide one of your own. You can use multiple workspaces and select the one that you want to use at startup. When a new workspace is first opened, the Eclipse window will be filled by a large “Welcome” screen that includes links to extensive documentation and tutorials. You should close this screen, by clicking the “X” next to the word “Welcome”; you can get back to it later by choosing “Welcome” from the “Help” menu.

The Eclipse GUI consists of one large window that is divided into several sections. Each section contains one or more views. For example, a view can be a text editor, it can be a place where a program can do I/O, or it can contain a list of your projects. If there are several views in one section of the window, then there will be tabs at the top of the section to select the view that is displayed in that section. This will happen, for example, if you have several editor views open at the same time.

Each view displays a different type of information. The whole set of views in the window is called a perspective. Eclipse uses different perspectives, that is, different sets of views of different types of information, for different tasks. For compiling and running programs, the only perspective that you will need is the “Java Perspective,” which is the default. As you become more experienced, you might want to use the “Debug Perspective,” which has features designed to help you find semantic errors in programs. There are small buttons in the Eclipse toolbar that can be used to switch between perspectives.

The Java Perspective includes a large area in the center of the window that contains text editor views. This is where you will create and edit your programs. To the left of this is the Package Explorer view, which will contain a list of your Java projects and source code files. To the right are one or more other views that you might or might not find useful; I usually close them by clicking the small “X” next to the name of each one. Several other views that will certainly be useful appear under different tabs in a section of the window below the editing area. If you accidently close one of the important views, such as the Package Explorer, you can get it back by selecting it from the “Show View” submenu of the “Window” menu. You can also reset the whole window to its default contents by selecting “Reset Perspective” from the “Window” menu.

To do any work in Eclipse, you need a project . To start a Java project, go to the “New” submenu in the “File” menu, and select the “Java Project” command. In the window that pops up, you will need to fill in a “Project Name,” which can be anything you like.

After entering a project name, and changing the options, if necessary, click the “Finish” button. Remember to say “Don’t Create” if Eclipse asks you whether you want to create “module-info.java”. The project should appear in the “Package Explorer” view on the left side of the Eclipse window. Click on the small triangle or plus sign next to the project name to see the contents of the project. Assuming that you use the default settings, there should be a directory named “src,” which is where your Java source code files will go. The project also contains the “JRE System Library”. This is the collection of standard built-in classes that come with Java;

To run any of the sample Java programs from this textbook, you need to copy the source code file into your Eclipse Java project. You can copy-and-paste it into the Eclipse window. (Right-click the file icon (or control-click on Mac OS); select “Copy” from the pop-up menu; then right-click the project’s src folder in the Eclipse window, and select “Paste”. Be sure to paste it into the src folder, not into the project itself; files outside the src folder are not treated as Java source code files.) Alternatively, you can try dragging the file icon from a file browser window onto the src folder in the Eclipse window.

Once a Java program is in the project, you can open it in an editor by double-clicking the file name in the “Package Explorer” view. To run the program, right-click

in the editor window, or on the file name in the Package Explorer view (or control-click in Mac OS). In the menu that pops up, go to the “Run As” submenu, and select “Java Application”. The program will be executed. If the program writes to standard output, the output will appear in the “Console” view, in the area of the Eclipse window below the editing area. If the program uses Scanner for input, you will have to type the required input into the “Console” view—click the “Console” view before you start typing so that the characters that you type will be sent to the correct part of the window. (For an easier way to run a program, find and click the small “Run” button in Eclipse’s tool bar. This will run either the program in the editor window, the program selected in the Package Explorer view, or the program that was run most recently, depending on context.) Note that when you run a program in Eclipse, it is compiled automatically. There is no separate compilation step. You can have more than one program in the same Eclipse project, or you can create additional projects to organize your work better.

To create a new Java program in Eclipse, you must create a new Java class. To do that, right-click the Java project name in the “Project Explorer” view. Go to the “New” submenu of the popup menu, and select “Class”. (Alternatively, there is a small icon in the toolbar at the top of the Eclipse window that you can click to create a new Java class.) In the window that opens, type in the name of the class that you want to create. The class name must be a legal Java identifier. Note that you want the name of the class, not the name of the source code file, so don’t add “.java” at the end of the name. The window also includes an input box labeled “Package” where you can specify the name of a package to contain the class. Most examples in this book use the “default package,” but you can create your own programs in any package. To use the default package, the “Package” input box should be empty. Finally, click the “Finish” button to create the class. The class should appear inside the “src” folder, in a folder corresponding to its package. The new file should automatically open in the editing area so that you can start typing your program.

Eclipse has several features that aid you as you type your code. It will underline any syntax error with a jagged red line, and in some cases will place an error marker in the left border of the edit window. If you hover the mouse cursor over the error marker or over the error itself, a description of the error will appear. Note that you do not have to get rid of every error immediately as you type; many errors will go away as you type in more of the program!

If an error marker displays a small “light bulb,” Eclipse is offering to try to fix the error for you. Click the light bulb or simply hover your mouse over the actual error to get a list of possible fixes, then click the fix that you want to apply. For example, if you use an undeclared variable in your program, Eclipse will offer to declare it for you. You can actually use this error-correcting feature to get Eclipse to write certain types of code for you! Unfortunately, you’ll find that you won’t understand a lot of the proposed

fixes until you learn more about the Java language, and it is not a good idea to apply a fix that you don't understand often that will just make things worse in the end.

Eclipse will also look for spelling errors in comments and will underline them with jagged red lines. Hover your mouse over the error to get a list of possible correct spellings.

Another essential Eclipse feature is content assist. Content assist can be invoked by typing Control-Space. It will offer possible completions of whatever you are typing at the moment. For example, if you type part of an identifier and hit Control-Space, you will get a list of identifiers that start with the characters that you have typed; use the up and down arrow keys to select one of the items in the list, and press Return or Enter. (You can also click an item with the mouse to select it, or hit Escape to dismiss the list.) If there is only one possible completion when you hit Control-Space, it will be inserted automatically. By default, Content Assist will also pop up automatically, after a short delay, when you type a period or certain other characters. For example, if you type "class-name" and pause for just a fraction of a second, you will get a list of all the subroutines in the class. You can disable it in the Eclipse Preferences. (Look under Java / Editor / Content Assist, and turn off the "Enable auto activation" option.) You can still call up Code Assist manually with Control-Space.

Once you have an error-free program, you can run it as described above. If you find a problem when you run it, it's very easy to go back to the editor, make changes, and run it again.

BlueJ

BlueJ is a small IDE that is designed specifically for people who are learning to program. It is much less complex than Eclipse, but it does have some features that make it useful for education. BlueJ can be downloaded from bluej.org. There are installers available for Windows, MacOS, and Windows. As of August 2021, the installers include OpenJDK 11 as well as JavaFX 11, so you will not need to do any additional downloading or configuration. There is also a generic installer that requires you to download a JDK and JavaFX separately. When you run the generic installer, BlueJ will ask you to input the locations of the JDK and JavaFX. (The current version of the generic installer in July 2021 did not work for me with OpenJDK 16 but did work with OpenJDK 15. It will certainly work with OpenJDK 11.)

In BlueJ, you can begin a project with the "New Project" command in the "Project" menu. A BlueJ project is simply a folder. When you create a project, you will have to select a folder name that does not already exist. The folder will be created and a window will be opened to show the contents of the folder. Files are shown as icons in the BlueJ window. You can drag .java files from the file system onto that window to add files to the project; they will be copied into the project folder as well as shown in

the window. You can also copy files directly into the project folder, but BlueJ won't see them until the next time you open the project. When you restart BlueJ, it should show the project that you were working on most recently, but you can open any project with a command from the "Project" menu.

There is a button in the project window for creating a new class. An icon for the class is added to the window, and a .java source code file is created in the project folder. The file is not automatically opened for editing. To edit a file, double-click its icon in the project window. An editor will be opened in a separate window. (A newly created class will contain some default code that you probably don't want; you can erase it and add a main() routine instead.) The BlueJ editor does not show errors as you type. Errors will be reported when you compile the program. Also, it does not offer automatic fixes for errors. It has a less capable version of Eclipse's Content Assist, which seems only to work for getting a list of available subroutines in a class or object; call up the list by hitting Control-Space after typing the period following the name of a class or object.

An editor window contains a button for compiling the program in the window. There is also a compile button in the project window, which compiles all the classes in the project.

To run a program, it must already be compiled. Right-click the icon of a compiled program. In the menu that pops up, you will see "void main(String[] args)". Select that option from the menu to run the program. Just click "OK" in the dialog box that pops up. A separate window will open for input/output.

One of the neatest features of BlueJ is that you can actually use it to run any subroutine, not just main. If a class contains other subroutines, you will see them in the list that you get by right-clicking its icon. A pop-up dialog allows you to enter any parameters required by the routine, and if the routine is a function, you will get another dialog box after the routine has been executed to tell you its return value. This allows easy testing of individual subroutines. Furthermore, you can also use BlueJ to create new objects from a class. An icon for the object will be added at the bottom of the project window, and you can right-click that icon to get a list of subroutines in the object.

2.9 Let Us Sum Up

In this unit learner you have learned about the Basic Java Application, Variables and the Primitive Types, Strings, Classes, Objects, and Subroutines. We have also learned how to use text Input and Output and details of Expressions. We have discussed three programming environments i.e. command line, eclipse and BlueJ

2.10 Further Reading

1. “Java 2: The Complete Reference” by Herbert Schildt, McGraw Hill Publications.
2. “Effective Java” by Joshua Bloch, Pearson Education.

2.11 Assignments

1. Briefly explain what is meant by the syntax and the semantics of a programming language. Give an example to illustrate the difference between a syntax error and a semantics error.
2. What does the computer do when it executes a variable declaration statement. Give an example.
3. What is a type, as this term relates to programming?
4. What is the boolean type? Where are boolean values used? What are its possible values?
5. Give the meaning of each of the following Java operators: ++, &&, !=
6. Explain what is meant by an assignment statement, and give an example. What are assignment statements used for?
7. What is meant by precedence of operators?
8. What is a literal?
9. In Java, classes have two fundamentally different purposes. What are they?
10. Explain why the value of the expression 2 + 3 + "test" is the string "5test" while the value of the expression "test" + 2 + 3 is the string "test23". What is the value of "test" + 2 * 3 ?
11. What is the purpose of an import directive, such as import java.util.Scanner?
12. Write a complete program that asks the user to enter the number of “widgets” they want to buy and the cost per widget. The program should then output the total cost for all the widgets. Use System.out.printf to print the cost, with two digits after the decimal point. You do not need to include any comments in the program.

Unit 3: Programming in the Small II: Control

3

Unit Structure

3.1 Learning Objectives

After studying this unit, learner should be able to understand

- How to build complex programs with more interesting behaviour.
- How to use two types of control structures, loops and branches
- How to use one of the most common data structures: arrays.

3.2 Introduction

The basic building blocks of programs—variables, expressions, assignment statements, and subroutine call statements—were covered in the previous chapter. Starting with this chapter, we look at how these building blocks can be put together to build complex programs with more interesting behavior.

Since we are still working on the level of “programming in the small” in this chapter, we are interested in the kind of complexity that can occur within a single subroutine. On this level, complexity is provided by control structures. The two types of control structures, loops and branches, can be used to repeat a sequence of statements over and over or to choose among two or more possible courses of action. Java includes several control structures of each type, and we will look at each of them in some detail.

Program complexity can be seen not just in control structures but also in data structures. A data structure is an organized collection of data, chunked together so that it can be treated as a unit. This chapter includes an introduction to one of the most common data structures: arrays.

The ability of a computer to perform complex tasks is built on just a few ways of combining simple commands into control structures. In Java, there are just six such structures that are used to determine the normal flow of control in a program and, in fact, just three of them would be enough to write programs to perform any task. The six control structures are: the block, the while loop, the do..while loop, the for loop, the if statement , and the switch statement . Each of these structures is considered to be a single “statement,” but a structured statement that can contain one or more other statements inside itself.

3.3 Blocks

The block is the simplest type of structured statement. Its purpose is simply to group a sequence of statements into a single statement. The format of a block is:

```
{
```

```
<statements>
}
```

That is, it consists of a sequence of statements enclosed between a pair of braces, “{” and “}”. In fact, it is possible for a block to contain no statements at all; such a block is called an empty block, and can actually be useful at times. An empty block consists of nothing but an empty pair of braces. Block statements usually occur inside other statements, where their purpose is to group together several statements into a unit. However, a block can be legally used wherever a statement can occur. There is one place where a block is required: As you might have already noticed in the case of the main subroutine of a program, the definition of a subroutine is a block, since it is a sequence of statements enclosed inside a pair of braces.

I should probably note again at this point that Java is what is called a free-format language. There are no syntax rules about how the language has to be arranged on a page. So, for example, you could write an entire block on one line if you want. But as a matter of good programming style, you should lay out your program on the page in a way that will make its structure as clear as possible. In general, this means putting one statement per line and using indentation to indicate statements that are contained inside control structures. This is the format that I will use in my examples.

Here are two examples of blocks:

```
{
      System.out.print("The answer is ");
      System.out.println(ans);
}

// This block exchanges the values of x and y
{
      int temp; // A temporary variable for use in this block.
      temp = x; // Save a copy of the value of x in temp.
      x = y; // Copy the value of y into x.
      y = temp; // Copy the value of temp into y.
}
```

In the second example, a variable, temp, is declared inside the block. This is perfectly legal, and it is good style to declare a variable inside a block if that variable is used nowhere else but inside the block. A variable declared inside a block is completely inaccessible and invisible from outside that block. When the computer executes the variable declaration statement, it allocates memory to hold the value of the variable (at least conceptually). When the block ends, that memory is discarded (that is, made available for reuse). The variable is said to be local to the block. There is a general concept called the “scope” of an identifier. The scope of an identifier is the part of the program in which that identifier is valid. The scope of a variable defined inside a block

is limited to that block, and more specifically to the part of the block that comes after the declaration of the variable.

3.4 The if statements

Conditional statements are used to run block of java code based on a condition. The java has various ways to execute conditional statements. They are using if, if…else, if else ladder, nested if…else, and switch…case.

if…else and its variations

The syntax of if…else is,

```
if(condition)
{
   Code block
}
else
{
  Code block
}
```

We can omit the braces ({…}), if the code block has only one program statement.

if…else statements of java are identical to C/C++. We can use if without else. For example, to check whether a is even we can use following statements.

```
if(a%2==0)
      System.out.println(a+" is even");
```

We can also use if…else together. For example, to check whether a is even or odd we can use following code.

```
if(a%2==0)
      System.out.println(a+" is even");
else
      System.out.println(a+" is odd");
```

If we use if…else inside if or else block, it will be nested if… else. For example to find out largest of three numbers a, b and c the following code can be used.

```
if(a>b)
{
  if(a>c)
   System.out.println("a is greatest");
  else
   System.out.println("c is greatest");
}
else
```

```
{
  If(b>c)
   System.out.println("a is greatest");
else
  System.out.println("c is greatest");
}
```

We can also use if…else in ladder pattern. For example from current time if you want a java program to wish "good morning", "good afternoon", "good evening" or "good night", we can use following if…else ladder.

```
if(current_time>5 && current_time<12)
      System.out.println("good morning");
else if(current_time>12 && current_time<5)
      System.out.println("good afternoon");
else if(current_time>5 && current_time<8)
      System.out.println("good evening");
else
      System.out.println("good night");
```

3.5 The switch statements

switch…case can be used to execute different code block for different value of input. For example if based on input value of arithmetic operator we want to perform the operation, we may use following code in java. In switch…case, each case should end with break statement. And default case is match if input is not match with any case.

```
switch(opr)
{
      case '+':   System.out.println(a+b);
                  break;
      case '-':   System.out.println(a-b);
                  break;
      case '*':   System.out.println(a*b);
                  break;
      case '/':   System.out.println(a/b);
                  break;
      case '%':   System.out.println(a%b);
                  break;
      default:    System.out.println("Invalid operation");
}
```

Examples: A program to which reads two integers and perform the arithmetic operation on them based on user's choice.

```
import java.util.Scanner;

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

```
        {
              int ch=0;
              Scanner sc=new Scanner(System.in);
              System.out.println("Enter a:");
              int a=sc.nextInt();
              System.out.println("Enter b:");
              int b=sc.nextInt();

              System.out.println("1. add");
              System.out.println("2. subtract");
              System.out.println("3. multiply");
              System.out.println("4. divide");
              System.out.println("Enter your choice:");
              ch=sc.nextInt();
              if(ch!=5)
              {
                    switch(ch)
                    {
                    case 1: System.out.println(a+b); break;
                    case 2: System.out.println(a-b); break;
                    case 3: System.out.println(a*b); break;
                    case 4: System.out.println(a/b); break;
                    default: System.out.println("Invalid choice");
                    }
              }
        }
}
```

3.6 The for statement

In a program when we want to execute a code block more than once, we need to put it in a loop. In java loop can be a for loop, while loop and do…while loop. The syntax of these loop are same as C/C++. The Java 5 introduce foreach loop. It is used to access the array or collection elements.

The for loop executes a statement or block of statements repeatedly until a condition is matched. For loops are normally used to execute the code block for more than one number of times. The syntax of for loop is given below.

```
for (initialization; test; increment)
{
    statements;
}
```

We can omit the braces if for loop has only one statement. As you can see in the syntax for loop has three parts in bracket.

- **initialization** is used to initialize the counter used in loop to keep track on number of iterations. for example, int i=0 or i=0.

- **test** must be the condition which must be true to enter in the loop. If the condition is false the loop terminates. Test is used to control the iteration count. For example, i<10 terminates the loop when i is greater or equal to 10.
- **increment** is used to change value of variable used in initialization

For example, the below for loop prints “Hello” 10 times with value of i each time. The output will print Hello0, Hello1 ... Hello9.

```
for(int i=0;i<10;i++)
  System.out.println(“Hello”+i);
```

3.7 While and do-while

while and do…while loops are also used to repeatedly execute a block of Java code until a condition is true. The syntax of these loops is same as C/C++.

The only difference between while and do…while loop is the timing of checking the condition. The while loop checks the condition before entering the loop. If condition is true it enters. The do…while loop first enter into the loop and check condition at the end.

They syntax of these loops are

```
while(test)
{
      Statements;
}
do
{
      Statements;
}
while(test);
```

The example in above section can be implemented using while and do…while as below.

```
int i=0;
while(i<10)
{
      System.out.println(“Hello”+i);
      i++;
}
```

OR

```
int i=0;
do
{
      System.out.println(“Hello”+i);
```

```
        i++;
}
while(i<10);
```

Use of continue and break in loops

In any loop, we can use break to terminate the loop and continue to skip existing iteration and start new iteration of the loop.

We can further understand the break and continue using example.

```
for(int i = 0; i <  5; i++)
{
        if ( i < 3 )
                System.out.println( "Hello" + i );
        else
                break;
}
```

In above loop the Hello will be printed for i = 0, 1 and 2. The loop terminates as soon as (i >= 3) because we used break in else part. Here loop will be executed three times only.

The use of continue explained in following code block.

```
for(int i = 0; i <  5; i++)
{
        if ( i ==3 )
                continue;
        else
                System.out.println( "Hello" + i );
}
```

In above example, the loop will be executed 5 times. However hello will be print only four times. Because when i==3 we use continue that means all the statements in a loop after continue will not be executed and next iteration is stared after increasing i.

Labeled loops

Loop can also have a loop inside it. This is called nesting of loop. When we are using nest loop the inside loop is called inner loop and outside loop is called outer loop. When we use break in inner loop the inner loop will be terminated. But if we want to terminate outer loop by using break statement in inner loop, we have to used the concept of labeled loop and continue/break with label.

For example

```
i = 0;
while( i < 3)
{
        j = 0;
        while( j < 3)
        {
```

```
            if(j==2)
                  break;
            j++;
      }
      i++;
}
```

In above example, the inner while loop will be break when j is 2. Inner loop will execute twice. Now if we want to break outer loop when in inner loop j is 2, we should use following code

```
i = 0;
outer:
while( i < 3)
{
      j = 0;
      while( j < 3)
      {
            if(j==2)
                  break outer;
            j++;
      }
      i++;
}
```

Here we have labeled outer loop with label outer: and with break w have to used label of outer loop.

Similarly continue can also be used with labeled loop.

Example:

```
public class Exa2
{
      public static void main(String args[])
      {
      first: for (int i = 0; i < 3; i++)
      {
            for (int j = 0; j< 3; j++)
            {
                  if(i == 1)
                    continue first;
                  System.out.print(" [i = " + i + ", j = " + j + "] ");
            }
      }
      System.out.println();
      second: for (int i = 0; i < 3; i++)
      {
       for (int j = 0; j< 3; j++)
       {
            if(i == 1)
                  break second;
            System.out.print(" [i = " + i + ", j = " + j + "] ");
       }
     }
```

```
}
```

3.8 Introduction to Exceptions and try..catch

In addition to the control structures that determine the normal flow of control in a program, Java has a way to deal with “exceptional” cases that throw the flow of control off its normal track. When an error occurs during the execution of a program, the default behavior is to terminate the program and to print an error message. However, Java makes it possible to “catch” such errors and program a response different from simply letting the program crash. This is done with the try..catch statement. In this section, we will take a preliminary and incomplete look the try..catch statement, leaving out a lot of the rather complex syntax of this statement.

Exceptions

The term exception is used to refer to the type of event that one might want to handle with a try..catch. An exception is an exception to the normal flow of control in the program. The term is used in preference to “error” because in some cases, an exception might not be considered to be an error at all. You can sometimes think of an exception as just another way to organize a program.

Exceptions in Java are represented as objects of type Exception. Actual exceptions are usually defined by subclasses of Exception. Different subclasses represent different types of exceptions. We will look at only two types of exception in this section: NumberFormatException and IllegalArgumentException.

A NumberFormatException can occur when an attempt is made to convert a string into a number. Such conversions are done by the functions Integer.parseInt and Double.parseDouble. Consider the function call Integer.parseInt(str) where str is a variable of type String. If the value of str is the string "42", then the function call will correctly convert the string into the int 42. However, if the value of str is, say, "fred", the function call will fail because "fred" is not a legal string representation of an int value. In this case, an exception of type NumberFormatException occurs. If nothing is done to handle the exception, the program will crash.

An IllegalArgumentException can occur when an illegal value is passed as a parameter to a subroutine. For example, if a subroutine requires that a parameter be greater than or equal to zero, an IllegalArgumentException might occur when a negative value is passed to the subroutine. How to respond to the illegal value is up to the person who wrote the subroutine, so we can’t simply say that every illegal parameter value will result in an IllegalArgumentException. However, it is a common response.

try..catch

When an exception occurs, we say that the exception is “thrown.” For example, we say that Integer.parseInt(str) throws an exception of type NumberFormatException when the value of str is illegal. When an exception is thrown, it is possible to “catch” the exception and prevent it from crashing the program. This is done with a try..catch statement. In simplified form, the syntax for a try..catch statement can be:

```
try {
      <statements-1>
}
catch (<exception-class-name> <variable-name>) {
      <statements-2>
}
```

The <exception-class-name> could be NumberFormatException, IllegalArgumentException, or some other exception class. When the computer executes this try..catch statement, it executes <statements-1>, the statements inside the try part. If no exception occurs during the execution of <statements-1>, then the computer just skips over the catch part and proceeds with the rest of the program. However, if an exception of type <exception-class-name> occurs

during the execution of <statements-1>, the computer immediately jumps from the point where the exception occurs to the catch part and executes <statements-2>, skipping any remaining statements in <statements-1>. Note that only one type of exception is caught; if some other type of exception occurs during the execution of <statements-1>, it will crash the program as usual.

During the execution of <statements-2>, the <variable-name> represents the exception object, so that you can, for example, print it out. The exception object contains information about the cause of the exception. This includes an error message, which will be displayed if you print out the exception object.

After the end of the catch part, the computer proceeds with the rest of the program; the exception has been caught and handled and does not crash the program.

By the way, note that the braces, { and }, are part of the syntax of the try..catch statement. They are required even if there is only one statement between the braces. This is different from the other statements we have seen, where the braces around a single statement

are optional.

As an example, suppose that str is a variable of type String whose value might or might not represent a legal real number. Then we could say:

```
double x;
try {
      x = Double.parseDouble(str);
      System.out.println( "The number is " + x );
}
catch ( NumberFormatException e ) {
      System.out.println( "Not a legal number." );
```

```
            x = Double.NaN;
        }
```

If an error is thrown by the call to Double.parseDouble(str), then the output statement in the try part is skipped, and the statement in the catch part is executed. (In this example, I set x to be the value Double.NaN when an exception occurs. Double.NaN is the special “not-a-number” value for type double.)

It’s not always a good idea to catch exceptions and continue with the program. Often that can just lead to an even bigger mess later on, and it might be better just to let the exception crash the program at the point where it occurs. However, sometimes it’s possible to recover from an error.

3.9 Introduction to Arrays

In previous sections of this chapter, we have already covered all of Java’s control structures. But before moving on to the next chapter, we will take preliminary looks at two additional topics that are at least somewhat related to control structures.

This section is an introduction to arrays. Arrays are a basic and very commonly used data structure, and array processing is often an exercise in using control structures. The next section will introduce computer graphics and will allow you to apply what you know about control structures in another context.

Creating and Using Arrays

A data structure consists of a number of data items chunked together so that they can be treated as a unit. An array is a data structure in which the items are arranged as a numbered sequence, so that each individual item can be referred to by its position number. In Java but not in some other programming languages—all the items must be of the same type, and the numbering always starts at zero. You will need to learn several new terms to talk about arrays: The number of items in an array is called the length of the array. The type of the individual items in an array is called the base type of the array. And the position number of an item in an array is called the index of that item.

Suppose that you want to write a program that will process the names of, say, one thousand people. You will need a way to deal with all that data. Before you knew about arrays, you might have thought that the program would need a thousand variables to hold the thousand names, and if you wanted to print out all the names, you would need a thousand print statements. Clearly, that would be ridiculous! In reality, you can put all the names into an array. The array is represented by a single variable, but it holds the entire list of names. The length of the array would be 1000, since there are 1000 individual names. The base type of the array would be String

since the items in the array are strings. The first name would be at index 0 in the array, the second name at index 1, and so on, up to the thousandth name at index 999.

The base type of an array can be any Java type, but for now, we will stick to arrays whose base type is String or one of the eight primitive types. If the base type of an array is int, it is referred to as an “array of ints.” An array with base type String is referred to as an “array of Strings.” However, an array is not, properly speaking, a list of integers or strings or other values. It is better thought of as a list of variables of type int, or a list of variables of type String, or of some other type. As always, there is some potential for confusion between the two uses of a variable: as a name for a memory location and as a name for the value stored in that memory location. Each position in an array acts as a variable. Each position can hold a value of a specified type (the base type of the array), just as a variable can hold a value. The value can be changed at any time, just as the value of a variable can be changed. The items in an array really, the individual variables that make up the array—are more often referred to as the elements of the array.

As I mentioned above, when you use an array in a program, you can use a variable to refer to the array as a whole. But you often need to refer to the individual elements of the array. The name for an element of an array is based on the name for the array and the index number of the element. The syntax for referring to an element looks, for example, like this: namelist[7]. Here, namelist is the variable that names the array as a whole, and namelist[7] refers to the element at index 7 in that array. That is, to refer to an element of an array, you use the array name, followed by element index enclosed in square brackets. An element name of this form can be used like any other variable: You can assign a value to it, print it out, use it in an expression, and so on.

An array also contains a kind of variable representing its length. For example, you can refer to the length of the array namelist as namelist.length. However, you cannot assign a value to namelist.length, since the length of an array cannot be changed.

Before you can use a variable to refer to an array, that variable must be declared, and it must have a type. For an array of Strings, for example, the type for the array variable would be String[ ], and for an array of ints, it would be int[ ]. In general, an array type consists of the base type of the array followed by a pair of empty square brackets. Array types can be used to declare variables; for example,

```
String[] namelist;
int[] A;
double[] prices;
```

and variables declared in this way can refer to arrays. However, declaring a variable does not make the actual array. Like all variables, an array variable has to be assigned a value before it can be used. In this case, the value is an array. Arrays have to be created using a special syntax. (The syntax is related to the fact that arrays in Java

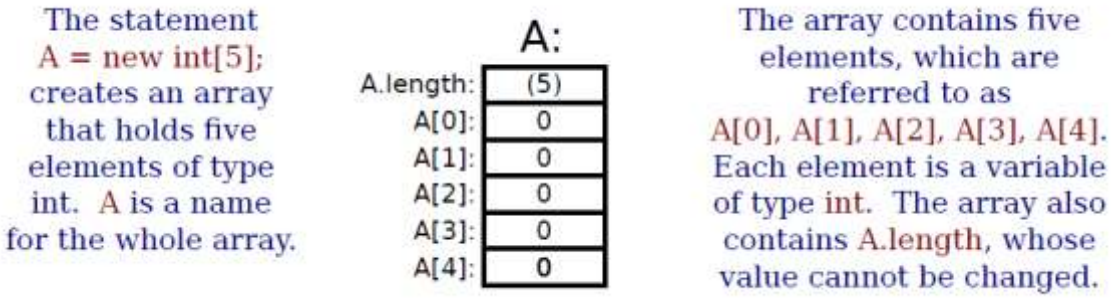
are actually objects, but that doesn't need to concern us here.) Arrays are created with an operator named new. Here are some examples:

```
namelist = new String[1000];
A = new int[5];
prices = new double[100];
```

The general syntax is

```
<array-variable> = new <base-type>[<array-length>];
```

The length of the array can be given as either an integer or an integer-valued expression. For example, after the assignment statement "A = new int[5];", A is an array containing the five integer elements A[0], A[1], A[2], A[3], and A[4]. Also, A.length would have the value 5. It's useful to have a picture in mind:



When you create an array of int, each element of the array is automatically initialized to zero. Any array of numbers is filled with zeros when it is created. An array of boolean is filled with the value false. And an array of char is filled with the character that has Unicode code number zero. (For an array of String, the initial value is null, a special value used for objects.)

Arrays and For Loops

A lot of the real power of arrays comes from the fact that the index of an element can be given by an integer variable or even an integer-valued expression. For example, if list is an array and i is a variable of type int, then you can use list[i] and even list[2*i+1] as variable names. The meaning of list[i] depends on the value of i. This becomes especially useful when we want to process all the elements of an array, since that can be done with a for loop.

For example, to print out all the items in an array, list, we can just write

```
int i; // the array index
for (i = 0; i < list.length; i++) {
      System.out.println( list[i] );
}
```

The first time through the loop, i is 0, and list[i] refers to list[0]. So, it is the value stored in the variable list[0] that is printed. The second time through the loop, i is 1, and the

value stored in list[1] is printed. If the length of the list is 5, then the loop ends after printing the value of list[4], when i becomes equal to 5 and the continuation condition "i < list.length" is no longer true. This is a typical example of using a loop to process an array.

Let's look at a few more examples. Suppose that A is an array of double, and we want to find the average of all the elements of the array. We can use a for loop to add up the numbers, and then divide by the length of the array to get the average:

```
double total; // The sum of the numbers in the array.
double average; // The average of the numbers.
int i; // The array index.
total = 0;
for ( i = 0; i < A.length; i++ ) {
      total = total + A[i]; // Add element number i to the total.
}
average = total / A.length; // A.length is the number of items
```

Another typical problem is to find the largest number in the array A. The strategy is to go through the array, keeping track of the largest number found so far. We'll store the largest number found so far in a variable called max. As we look through the array, whenever we find a number larger than the current value of max, we change the value of max to that larger value.

After the whole array has been processed, max is the largest item in the array overall. The only question is, what should the original value of max be? One possibility is to start with max equal to A[0], and then to look through the rest of the array, starting from A[1], for larger items:

```
double max; // The largest number seen so far.
max = A[0]; // At first, the largest number seen is A[0].
int i;
for ( i = 1; i < A.length; i++ ) {
      if (A[i] > max) {
            max = A[i];
      }
}
// at this point, max is the largest item in A
```

Sometimes, you only want to process some elements of the array. In that case, you can use an if statement inside the for loop to decide whether or not to process a given element. Let's look at the problem of averaging the elements of an array, but this time, suppose that we only want to average the non-zero elements. In this case, the number of items that we add up can be less than the length of the array, so we will need to keep a count of the number of items added to the sum:

```
double total; // The sum of the non-zero numbers in the array.
int count; // The number of non-zero numbers.
double average; // The average of the non-zero numbers.
int i;
total = 0;
count = 0;
```

```
for ( i = 0; i < A.length; i++ ) {
      if ( A[i] != 0 ) {
            total = total + A[i]; // Add element to the total
            count = count + 1; // and count it.
      }
}
if (count == 0) {
      System.out.println("There were no non-zero elements.");
}
else {
    average = total / count; // Divide by number of items
    System.out.printf("Average of %d elements is %1.5g%n", count, average);
}
```

Two-dimensional Arrays

The arrays that we have considered so far are “one-dimensional.” This means that the array consists of a sequence of elements that can be thought of as being laid out along a line. It is also possible to have two-dimensional arrays, where the elements can be laid out in a rectangular grid. In a two-dimensional, or “2D,” array, the elements can be arranged in rows and columns. Here, for example, is a 2D array of int that has five rows and seven columns:

	0	1	2	3	4	5	6
0	13	7	33	54	-5	-1	92
1	-3	0	8	42	18	0	67
2	44	78	90	79	-5	72	22
3	43	-6	17	100	1	-12	12
4	2	0	58	58	36	21	87

This 5-by-7 grid contains a total of 35 elements. The rows in a 2D array are numbered 0, 1, 2, . . . , up to the number of rows minus one. Similarly, the columns are numbered from zero up to the number of columns minus one. Each individual element in the array can be picked out by specifying its row number and its column number. (The illustration shown here is not what the array actually looks like in the computer’s memory, but it does show the logical structure of the array.)

In Java, the syntax for two-dimensional arrays is similar to the syntax for one-dimensional arrays, except that an extra index is involved, since picking out an element requires both a row number and a column number. For example, if A is a 2D array of int, then A[3][2] would be the element in row 3, column 2. That would pick out the number 17 in the array shown above.

The type for A would be given as int[ ][ ], with two pairs of empty brackets. To declare the array variable and create the array, you could say,

```
int[][] A;
A = new int[5][7];
```

The second line creates a 2D array with 5 rows and 7 columns. Two-dimensional arrays are often processed using nested for loops. For example, the following code segment will print out the elements of A in neat columns:

```
int row, col; // loop-control-variables for accessing rows and columns in A
for ( row = 0; row < 5; row++ ) {
      for ( col = 0; col < 7; col++ ) {
            System.out.printf( "%7d", A[row][col] );
      }
      System.out.println();
}
```

The base type of a 2D array can be anything, so you can have arrays of type double[ ][ ], String[ ][ ], and so on.

There are some natural uses for 2D arrays. For example, a 2D array can be used to store the contents of the board in a game such as chess or checkers. But sometimes two-dimensional arrays are used in problems in which the grid is not so visually obvious. Consider a company that owns 25 stores. Suppose that the company has data about the profit earned at each store for each month in the year 2018. If the stores are numbered from 0 to 24, and if the twelve months from January 2018 through December 2018 are numbered from 0 to 11, then the profit data could be stored in an array, profit, created as follows:

```
double[][] profit;
profit = new double[25][12];
```

profit[3][2] would be the amount of profit earned at store number 3 in March, and more generally, profit[storeNum][monthNum] would be the amount of profit earned in store number storeNum in month number monthNum (where the numbering, remember, starts from zero).

Let’s assume that the profit array has already been filled with data. This data can be processed in a lot of interesting ways. For example, the total profit for the company for the whole year from all its stores—can be calculated by adding up all the entries in the array:

```
double totalProfit; // Company’s total profit in 2018.
int store, month; // variables for looping through the stores and the months
totalProfit = 0;
for ( store = 0; store < 25; store++ ) {
      for ( month = 0; month < 12; month++ )
            totalProfit += profit[store][month];
}
```

Sometimes it is necessary to process a single row or a single column of an array, not the entire array. For example, to compute the total profit earned by the company in December, that is, in month number 11, you could use the loop:

```
double decemberProfit;
int storeNum;
decemberProfit = 0.0;
for ( storeNum = 0; storeNum < 25; storeNum++ ) {
      decemberProfit += profit[storeNum][11];
}
```

Two-dimensional arrays are sometimes useful, but they are much less common than one-dimensional arrays. Java actually allows arrays of even higher dimension, but they are only rarely encountered in practice.

3.10 Let Us Sum Up

In this unit we have learned about how the basic building blocks of programs can be put together to build complex programs with more interesting behavior. We have discussed the two types of control structures, loops and branches, that can be used to repeat a sequence of statements over and over or to choose among two or more possible courses of action. We have looked at each of them in some detail. We have also get an introduction to one of the most common data structures: arrays.

3.11 Further Reading

1. “Java 2: The Complete Reference” by Herbert Schildt, McGraw Hill Publications.
2. “Effective Java” by Joshua Bloch, Pearson Education.

3.12 Assignments

1. What is a block statement? How are block statements used in Java programs?
2. What is the main difference between a while loop and a do..while loop?
3. Write a for loop that will print out all the multiples of 3 from 3 to 36, that is: 3 6 9 12 15 18 21 24 27 30 33 36.
4. Write a Java program to ask the user to enter an integer, read the user’s response, and tell the user whether the number entered is even or odd.
5. Suppose that s1 and s2 are variables of type String, whose values are expected to be string representations of values of type int. Write a code segment that will compute and print the integer sum of those values, or will print an error message if the values cannot successfully be converted into integers. (Use a try..catch statement.)
6. Show the exact output that would be produced by the following main() routine:

```
public static void main(String[] args) {
   int N;
   N = 1;
   while (N <= 32) {
      N = 2 * N;
      System.out.println(N);
   }
}
```

7. Show the exact output produced by the following main() routine:

```
public static void main(String[] args) {
   int x,y;
   x = 5;
   y = 1;
   while (x > 0) {
      x = x - 1;
      y = y * x;
      System.out.println(y);
   }
}
```

8. What output is produced by the following program segment? Why?

```
String name;
int i;
boolean startWord;
name = "Himanshu N. Patel";
startWord = true;
for (i = 0; i < name.length(); i++) {
   if (startWord)
         System.out.println(name.charAt(i));
   if (name.charAt(i) == ' ')
         startWord = true;
   else
   startWord = false;
}
```

9. Suppose that numbers is an array of type int[ ]. Write a code segment that will count and output the number of times that the number 42 occurs in the array.
10. Define the range of an array of numbers to be the maximum value in the array minus the minimum value. Suppose that raceTimes is an array of type double[ ]. Write a code segment that will find and print the range of raceTimes.

Unit 4: Programming in the Large I: Subroutines

4

Unit Structure

4.1 Learning Objectives

After studying this unit, learner should be able to understand

- How to break up a complex program into manageable pieces called subroutines.
- static subroutines.
- Parameters and
- Return values

4.2 Introduction

One way to break up a complex program into manageable pieces is to use subroutines. A subroutine consists of the instructions for carrying out a certain task, grouped together and given a name. Elsewhere in the program, that name can be used as a stand-in for the whole set of instructions. As a computer executes a program, whenever it encounters a subroutine name, it executes all the instructions necessary to carry out the task associated with that subroutine.

Subroutines can be used over and over, at different places in the program. A subroutine can even be used inside another subroutine. This allows you to write simple subroutines and then use them to help write more complex subroutines, which can then be used in turn in other subroutines. In this way, very complex programs can be built up step-by-step, where each step in the construction is reasonably simple.

Subroutines in Java can be either static or non-static. This chapter covers static subroutines. Non-static subroutines, which are used in true object-oriented programming, will be covered in the next chapter.

4.3 Black Boxes

A subroutine consists of instructions for performing some task, chunked together and given a name. “Chunking” allows you to deal with a potentially very complicated task as a single concept. Instead of worrying about the many, many steps that the computer might have to go though to perform that task, you just need to remember the name of the subroutine. Whenever you want your program to perform the task, you just call the subroutine. Subroutines are a major tool for dealing with complexity.

A subroutine is sometimes said to be a “black box” because you can’t see what’s “inside” it (or, to be more precise, you usually don’t want to see inside it, because then you would have to deal with all the complexity that the subroutine is meant to hide). Of course, a black box that has no way of interacting with the rest of the world would be pretty useless. A black box needs some kind of interface with the rest of the world, which allows some interaction between what’s inside the box and

what's outside. A physical black box might have buttons on the outside that you can push, dials that you can set, and slots that can be used for passing information back and forth. Since we are trying to hide complexity, not create it, we have the first rule of black boxes:

The interface of a black box should be fairly straight-forward, well-defined, and easy to understand.

Are there any examples of black boxes in the real world? Yes; in fact, you are surrounded by them. Your television, your car, your mobile phone, your refrigerator.... You can turn your television on and off, change channels, and set the volume by using elements of the television's interface on/off switch, remote control, don't forget to plug in the power without understanding anything about how the thing actually works. The same goes for a mobile phone, although the interface in that case is a lot more complicated.

Now, a black box does have an inside—the code in a subroutine that actually performs the task, or all the electronics inside your television set. The inside of a black box is called its implementation. The second rule of black boxes is that:

To use a black box, you shouldn't need to know anything about its implementation; all you need to know is its interface.

In fact, it should be possible to change the implementation, as long as the behavior of the box, as seen from the outside, remains unchanged. For example, when the insides of TV sets went from using vacuum tubes to using transistors, the users of the sets didn't need to know about it or even know what it means. Similarly, it should be possible to rewrite the inside of a subroutine, to use more efficient code for example, without affecting the programs that use that subroutine.

Of course, to have a black box, someone must have designed and built the implementation in the first place. The black box idea works to the advantage of the implementor as well as the user of the black box. After all, the black box might be used in an unlimited number of different situations. The implementor of the black box doesn't need to know about any of that. The implementor just needs to make sure that the box performs its assigned task and interfaces correctly with the rest of the world. This is the third rule of black boxes:

The implementor of a black box should not need to know anything about the larger systems in which the box will be used.

In a way, a black box divides the world into two parts: the inside (implementation) and the outside. The interface is at the boundary, connecting those two parts.

You should keep in mind that subroutines are not the only example of black boxes in programming. For example, a class is also a black box. We'll see that a class can have a "public" part, representing its interface, and a "private" part that is entirely inside its hidden implementation. All the principles of black boxes apply to classes as well as to subroutines.

4.4 Static Subroutines and Static Variables

Every subroutine in Java must be defined inside some class. This makes Java rather unusual among programming languages, since most languages allow free-floating, independent subroutines. One purpose of a class is to group together related subroutines and variables. Perhaps the designers of Java felt that everything must be related to something. As a less philosophical motivation, Java's designers wanted to place firm controls on the ways things are named, since a Java program potentially has access to a huge number of subroutines created by many different programmers. The fact that those subroutines are grouped into named classes (and classes are grouped into named "packages," as we will see later) helps control the confusion that might result from so many different names.

There is a basic distinction in Java between static and non-static subroutines. A class definition can contain the source code for both types of subroutine, but what's done with them when the program runs is very different. Static subroutines are easier to understand: In a running program, a static subroutine is a member of the class itself. Non-static subroutine definitions, on the other hand, are only there to be used when objects are created, and the subroutines themselves become members of the objects. Non-static subroutines only become relevant when you are working with objects. The distinction between static and non-static also applies to variables and to other things that can occur in class definitions. This chapter will deal with static subroutines and static variables almost exclusively. We'll turn to non-static
stuff and to object-oriented programming in the next chapter.

A subroutine that is in a class or object is often called a method, and "method" is the term that most people prefer for subroutines in Java. I will start using the term "method" occasionally, but I will continue to prefer the more general term "subroutine" in this chapter, at least for static subroutines. However, you should start thinking of the terms "method" and "subroutine" as being essentially synonymous as far as Java is concerned. Other terms that you might see used to refer to subroutines are "procedures" and "functions." (I generally use the term "function" only for subroutines that compute and return a value, but in some programming languages, it is used to refer to subroutines in general.)

Subroutine Definitions

A subroutine must be defined somewhere. The definition has to include the name of the subroutine, enough information to make it possible to call the subroutine, and the code that will be executed each time the subroutine is called. A subroutine definition in Java takes the form:

```
<modifiers> <return-type> <subroutine-name> ( <parameter-list> ) {
      <statements>
}
```

The <statements> between the braces, { and }, in a subroutine definition make up the body of the subroutine. These statements are the inside, or implementation part, of the "black box," as discussed in the previous section. They are the instructions that the computer executes when the method is called.

The <modifiers> that can occur at the beginning of a subroutine definition are words that set certain characteristics of the subroutine, such as whether it is static or not. The modifiers that you've seen so far are "static" and "public". There are only about a half-dozen possible modifiers altogether.

If the subroutine is a function, whose job is to compute some value, then the <return-type> is used to specify the type of value that is returned by the function. It can be a type name such as String or int or even an array type such as double[ ]. If the subroutine is not a function, then the <return-type> is replaced by the special value void, which indicates that no value is returned. The term "void" is meant to indicate that the return value is empty or non-existent.

Finally, we come to the <parameter-list> of the method. Parameters are part of the interface of a subroutine. They represent information that is passed into the subroutine from outside, to be used by the subroutine's internal computations. For a concrete example, imagine a class named Television that includes a method named changeChannel(). The immediate question is: What channel should it change to? A parameter can be used to answer this question. If a channel number is an integer, the type of the parameter would be int, and the declaration of the changeChannel() method might look like

```
public void changeChannel(int channelNum) { ... }
```

This declaration specifies that changeChannel() has a parameter named channelNum of type int. However, channelNum does not yet have any particular value. A value for channelNum is provided when the subroutine is called; for example: changeChannel(17); The parameter list in a subroutine can be empty, or it can consist of one or more parameter declarations of the form <type> <parameter-name>. If there are several declarations, they are separated by commas. Note that each declaration can name only one parameter. For example, if you want two parameters of type double, you have to say "double x, double y", rather than "double x, y". Parameters are covered in more detail in the next section.

Here are a few examples of subroutine definitions, leaving out the statements that define what the subroutines do:

```
public static void playGame() {
// "public" and "static" are modifiers; "void" is the
// return-type; "playGame" is the subroutine-name;
// the parameter-list is empty.
. . . // Statements that define what playGame does go here.
}

int getNextN(int N) {
// There are no modifiers; "int" is the return-type;
// "getNextN" is the subroutine-name; the parameter-list
// includes one parameter whose name is "N" and whose
// type is "int".
. . . // Statements that define what getNextN does go here.
}
```

```
static boolean lessThan(double x, double y) {
// "static" is a modifier; "boolean" is the
// return-type; "lessThan" is the subroutine-name;
// the parameter-list includes two parameters whose names are
// "x" and "y", and the type of each of these parameters
// is "double".
. . . // Statements that define what lessThan does go here.
}
```

In the second example given here, getNextN is a non-static method, since its definition does not include the modifier “static” and so it’s not an example that we should be looking at in this chapter! The other modifier shown in the examples is “public”. This modifier indicates that the method can be called from anywhere in a program, even from outside the class where the method is defined. There is another modifier, “private”, which indicates that the method can be called only from inside the same class. The modifiers public and private are called access specifiers. If no access specifier is given for a method, then by default, that method can be called from anywhere in the package that contains the class, but not from outside that package. There is one other access modifier, protected.

Note, by the way, that the main() routine of a program follows the usual syntax rules for a subroutine. In

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

the modifiers are public and static, the return type is void, the subroutine name is main, and the parameter list is “String[] args”. In this case, the type for the parameter is the array type String[ ].

You’ve already had some experience with filling in the implementation of a subroutine. In this chapter, you’ll learn all about writing your own complete subroutine definitions, including the interface part.

Calling Subroutines

When you define a subroutine, all you are doing is telling the computer that the subroutine exists and what it does. The subroutine doesn’t actually get executed until it is called. (This is true even for the main() routine in a class—even though you don’t call it, it is called by the system when the system runs your program.) For example, the playGame() method given as an example above could be called using the following subroutine call statement:

```
playGame();
```

This statement could occur anywhere in the same class that includes the definition of playGame(), whether in a main() method or in some other subroutine. Since playGame() is a public method, it can also be called from other classes, but in that case, you have to tell the computer which class it comes from. Since playGame() is a static method, its full name includes the name of the class in which it is defined. Let’s say, for example, that playGame() is defined in a class named Poker. Then to call playGame() from outside the Poker class, you would have to say Poker.playGame();

The use of the class name here tells the computer which class to look in to find the method. It also lets you distinguish between Poker.playGame() and other potential playGame() methods defined in other classes, such as Roulette.playGame() or Blackjack.playGame().

More generally, a subroutine call statement for a static subroutine takes the form

```
<subroutine-name> (<parameters>);
```

if the subroutine that is being called is in the same class, or

```
<class-name>.<subroutine-name>(<parameters>);
```

if the subroutine is defined elsewhere, in a different class. (Non-static methods belong to objects rather than classes, and they are called using objects instead of class names. More on that later.) Note that the parameter list can be empty, as in the playGame() example, but the parentheses must be there even if there is nothing between them. The number of parameters that you provide when you call a subroutine must match the number specified in the parameter list in the subroutine definition, and the types of the parameters in the call statement must match the types in the subroutine definition.

Member Variables

A class can include other things besides subroutines. In particular, it can also include variable declarations. Of course, you can declare variables inside subroutines. Those are called local variables. However, you can also have variables that are not part of any subroutine. To distinguish such variables from local variables, we call them member variables, since they are members of a class. Another term for them is global variable.

Just as with subroutines, member variables can be either static or non-static. In this chapter, we'll stick to static variables. A static member variable belongs to the class as a whole, and it exists as long as the class exists. Memory is allocated for the variable when the class is first loaded by the Java interpreter. Any assignment statement that assigns a value to the variable changes the content of that memory, no matter where that assignment statement is located in the program. Any time the variable is used in an expression, the value is fetched from that same memory, no matter where the expression is located in the program. This means that the value of a static member variable can be set in one subroutine and used in another subroutine. Static member variables are "shared" by all the static subroutines in the class. A local variable in a subroutine, on the other hand, exists only while that subroutine is being executed, and is completely inaccessible from outside that one subroutine.

The declaration of a member variable looks just like the declaration of a local variable except for two things: The member variable is declared outside any subroutine (although it still has to be inside a class), and the declaration can be marked with modifiers such as static, public, and private. Since we are only working with static member variables for now, every declaration of a member variable in this chapter will

include the modifier static. They might also be marked as public or private. For example:

```
static String usersName;
public static int numberOfPlayers;
private static double velocity, time;
```

A static member variable that is not declared to be private can be accessed from outside the class where it is defined, as well as inside. When it is used in some other class, it must be referred to with a compound identifier of the form hclass-namei.hvariable-namei. For example, the System class contains the public static member variable named out, and you use this variable in your own classes by referring to System.out. Similarly, Math.PI is a public static member variable in the Math class. If numberOfPlayers is a public static member variable in a class named Poker, then code in the Poker class would refer to it simply as numberOfPlayers, while code in another class would refer to it as Poker.numberOfPlayers.

As an example, let's add a couple of static member variables to the GuessingGame class that we wrote earlier in this section. We add a variable named gamesPlayed to keep track of how many games the user has played and another variable named gamesWon to keep track of the number of games that the user has won. The variables are declared as static member variables:

```
static int gamesPlayed;
static int gamesWon;
```

In the playGame() routine, we always add 1 to gamesPlayed, and we add 1 to gamesWon if the user wins the game. At the end of the main() routine, we print out the values of both variables. It would be impossible to do the same thing with local variables, since both subroutines need to access the variables, and local variables exist in only one subroutine. Furthermore, global variables keep their values between one subroutine call and the next. Local variables do not; a local variable gets a new value each time that the subroutine that contains it is called.

When you declare a local variable in a subroutine, you have to assign a value to that variable before you can do anything with it. Member variables, on the other hand are automatically initialized with a default value. The default values are the same as those that are used when initializing the elements of an array: For numeric variables, the default value is zero; for Boolean variables, the default is false; for char variables, it's the character that has Unicode code
number zero; and for objects, such as Strings, the default initial value is the special value null.

Since they are of type int, the static member variables gamesPlayed and gamesWon automatically get zero as their initial value. This happens to be the correct initial value for a variable that is being used as a counter. You can, of course, assign a value to a variable at the beginning of the main() routine if you are not satisfied with the default initial value, or if you want to make the initial value more explicit.

4.5 Parameters

If a subroutine is a black box, then a parameter is something that provides a mechanism for passing information from the outside world into the box. Parameters are part of the interface of a subroutine. They allow you to customize the behavior of a subroutine to adapt it to a particular situation.

As an analogy, consider a thermostat a black box whose task it is to keep your house at a certain temperature. The thermostat has a parameter, namely the dial that is used to set the desired temperature. The thermostat always performs the same task: maintaining a constant temperature. However, the exact task that it performs that is, which temperature it maintains is customized by the setting on its dial.

Formal and Actual Parameters

Parameters in a subroutine definition are called formal parameters or dummy parameters. The parameters that are passed to a subroutine when it is called are called actual parameters or arguments. When a subroutine is called, the actual parameters in the subroutine call statement are evaluated and the values are assigned to the formal parameters in the subroutine's definition. Then the body of the subroutine is executed.

A formal parameter must be a name, that is, a simple identifier. A formal parameter is very much like a variable, and like a variable it has a specified type such as int, boolean, String, or double[ ]. An actual parameter is a value, and so it can be specified by any expression, provided that the expression computes a value of the correct type. The type of the actual parameter must be one that could legally be assigned to the formal parameter with an assignment statement. For example, if the formal parameter is of type double, then it would be legal to pass an int as the actual parameter since ints can legally be assigned to doubles. When you call a subroutine, you must provide one actual parameter for each formal parameter in the subroutine's definition.

Consider, for example, a subroutine

```
static void doTask(int N, double x, boolean test) {
      // statements to perform the task go here
}
```

This subroutine might be called with the statement

```
doTask(17, Math.sqrt(z+1), z >= 10);
```

When the computer executes this statement, it has essentially the same effect as the block of statements:

```
{
      int N; // Allocate memory locations for the formal parameters.
      double x;
      boolean test;
```

```
        N = 17; // Assign 17 to the first formal parameter, N.
        x = Math.sqrt(z+1); // Compute Math.sqrt(z+1), and assign it to
        // the second formal parameter, x.
        test = (z >= 10); // Evaluate "z >= 10" and assign the resulting
        // true/false value to the third formal
        // parameter, test.
        // statements to perform the task go here
}
```

(There are a few technical differences between this and "doTask(17,Math.sqrt(z+1),z>=10);" besides the amount of typing because of questions about scope of variables and what happens when several variables or parameters have the same name.)

Beginning programming students often find parameters to be surprisingly confusing. Calling a subroutine that already exists is not a problem—the idea of providing information to the subroutine in a parameter is clear enough. Writing the subroutine definition is another matter. A common beginner's mistake is to assign values to the formal parameters at the beginning of the subroutine, or to ask the user to input their values. This represents a fundamental misunderstanding. By the time the computer starts executing the statements in the subroutine, the formal parameters have already been assigned initial values! The computer automatically assigns values to the formal parameters before it starts executing the code inside the subroutine. The values come from the actual parameters in the subroutine call statement.

Remember that a subroutine is not independent. It is called by some other routine, and it is the subroutine call statement's responsibility to provide appropriate values for the parameters.

Overloading

In order to call a subroutine legally, you need to know its name, you need to know how many formal parameters it has, and you need to know the type of each parameter. This information is called the subroutine's signature. The signature of the subroutine doTask, used as an example above, can be expressed as: doTask(int,double,boolean). Note that the signature does not include the names of the parameters; in fact, if you just want to use the subroutine, you don't even need to know what the formal parameter names are, so the names are not part of the interface.

Java is somewhat unusual in that it allows two different subroutines in the same class to have the same name, provided that their signatures are different. When this happens, we say that the name of the subroutine is overloaded because it has several different meanings. The computer doesn't get the subroutines mixed up. It can tell which one you want to call by the number and types of the actual parameters that you provide in the subroutine call statement.
You have already seen overloading used with System.out. This object includes many different methods named println, for example. These methods all have different signatures, such as:

```
println(int) println(double)
println(char) println(boolean)
```

```
println()
```

The computer knows which of these subroutines you want to use based on the type of the actual parameter that you provide. System.out.println(17) calls the subroutine with signature println(int), while System.out.println('A') calls the subroutine with signature println(char). Of course all these different subroutines are semantically related, which is why it is acceptable programming style to use the same name for them all. But as far as the computer is concerned, printing out an int is very different from printing out a char, which is different from printing out a boolean, and so forth—so that each of these operations requires a different subroutine.

Note, by the way, that the signature does not include the subroutine's return type. It is illegal to have two subroutines in the same class that have the same signature but that have different return types. For example, it would be a syntax error for a class to contain two subroutines defined as:

```
int getln() { ... }
double getln() { ... }
```

This is why in the TextIO class, the subroutines for reading different types are not all named getln(). In a given class, there can only be one routine that has the name getln with no parameters. So, the input routines in TextIO are distinguished by having different names, such as getlnInt() and getlnDouble().

Subroutine Examples

Let's do a few examples of writing small subroutines to perform assigned tasks. Of course, this is only one side of programming with subroutines. The task performed by a subroutine is always a subtask in a larger program. The art of designing those programs of deciding how to break them up into subtasks—is the other side of programming with subroutines.

As a first example, let's write a subroutine to compute and print out all the divisors of a given positive integer. The integer will be a parameter to the subroutine. Remember that the syntax of any subroutine is:

```
<modifiers> <return-type> <subroutine-name> (<parameter-list>) {
      <statements>
}
```

Writing a subroutine always means filling out this format. In this case, the statement of the problem tells us that there is one parameter, of type int, and it tells us what the statements in the body of the subroutine should do. Since we are only working with static subroutines for now, we'll need to use static as a modifier. We could add an access modifier (public or private), but in the absence of any instructions, I'll leave it out. Since we are not told to return a value, the return type is void. Since no names are specified, we'll have to make up names for the formal parameter and for the subroutine itself. I'll use N for the parameter and printDivisors for the subroutine name. The subroutine will look like

```
      static void printDivisors( int N ) {
```

```
            <statements>
      }
```

and all we have left to do is to write the statements that make up the body of the routine. This is not difficult. Just remember that you have to write the body assuming that N already has a value! The algorithm is: “For each possible divisor D in the range from 1 to N, if D evenly divides N, then print D.” Written in Java, this becomes:

```
/* Print all the divisors of N. We assume that N is a positive integer. */

static void printDivisors( int N ) {
      int D; // One of the possible divisors of N.
      System.out.println("The divisors of " + N + " are:");
      for ( D = 1; D <= N; D++ ) {
      if ( N % D == 0 ) // Does D evenly divide N?
      System.out.println(D);
      }
}
```

I’ve added a comment before the subroutine definition indicating the contract of the Subroutine that is, what it does and what assumptions it makes. The contract includes the assumption that N is a positive integer. It is up to the caller of the subroutine to make sure that this assumption is satisfied.

As a second short example, consider the problem: Write a private subroutine named printRow. It should have a parameter ch of type char and a parameter N of type int. The subroutine should print out a line of text containing N copies of the character ch. Here, we are told the name of the subroutine and the names of the two parameters, and we are told that the subroutine is private, so we don’t have much choice about the first line of the subroutine definition. The task in this case is pretty simple, so the body of the subroutine is easy to write. The complete subroutine is given by

```
/* Write one line of output containing N copies of the character ch.
If N <= 0, an empty line is output. */

private static void printRow( char ch, int N ) {
      int i; // Loop-control variable for counting off the copies.
      for ( i = 1; i <= N; i++ ) {
            System.out.print( ch );
      }
      System.out.println();
}
```

Note that in this case, the contract makes no assumption about N, but it makes it clear what will happen in all cases, including the unexpected case that N <= 0.

Finally, let’s do an example that shows how one subroutine can build on another. Let’s write a subroutine that takes a String as a parameter. For each character in the string, it should print a line of output containing 25 copies of that character. It should use the printRow() subroutine to produce the output.

Again, we get to choose a name for the subroutine and a name for the parameter. I’ll call the subroutine printRowsFromString and the parameter str. The

algorithm is pretty clear: For each position i in the string str, call printRow(str.charAt(i),25) to print one line of the output. So, we get:

```
/* For each character in str, write a line of output containing 25 copies of
that character. */

private static void printRowsFromString( String str ) {
      int i; // Loop-control variable for counting off the chars.
      for ( i = 0; i < str.length(); i++ ) {
            printRow( str.charAt(i), 25 );
      }
}
```

We could then use printRowsFromString in a main() routine such as

```
public static void main(String[] args) {
      String inputLine; // Line of text input by user.
      System.out.print("Enter a line of text: ");
      inputLine = TextIO.getln();
      System.out.println();
      printRowsFromString( inputLine );
}
```

Of course, the three routines, main(), printRowsFromString(), and printRow(), would have to be collected together inside the same class. The program is rather useless, but it does demonstrate the use of subroutines. You'll find the program in the file RowsOfChars.java, if you want to take a look.

Array Parameters

It's possible for the type of a parameter to be an array type. This means that an entire array of values can be passed to the subroutine as a single parameter. For example, we might want a subroutine to print all the values in an integer array in a neat format, separated by commas and enclosed in a pair of square brackets. To tell it which array to print, the subroutine would have a parameter of type int[ ]:

```
static void printValuesInList( int[] list )
{
      System.out.print('[');
      int i;
      for ( i = 0; i < list.length; i++ ) {
            if ( i > 0 )
                  System.out.print(','); // No comma in front of list[0]
            System.out.print(list[i]);
      }
      System.out.println(']');
}
```

To use this subroutine, you need an actual array. Here is a legal, though not very realistic, code segment that creates an array just to pass it as an argument to the subroutine:

```
int[] numbers;
numbers = new int[3];
numbers[0] = 42;
numbers[1] = 17;
numbers[2] = 256;
printValuesInList( numbers );
```

The output produced by the last statement would be [42,17,256].

Command-line Arguments

The main routine of a program has a parameter of type String[ ]. When the main routine is called, some actual array of String must be passed to main() as the value of the parameter. The system provides the actual parameter when it calls main(), so the values come from outside the program. Where do the strings in the array come from, and what do they mean? The strings in the array are command-line arguments from the command that was used to run the program. When using a command-line interface, the user types a command to tell the system to execute a program. The user can include extra input in this command, beyond the name of the program. This extra input becomes the command-line arguments. The system takes the command-line arguments, puts them into an array of strings, and passes that array to main().

For example, if the name of the program is myProg, then the user can type “java myProg” to execute the program. In this case, there are no command-line arguments. But if the user types the command

```
java myProg one two three
```

then the command-line arguments are the strings “one”, “two”, and “three”. The system puts these strings into an array of Strings and passes that array as a parameter to the main() routine.

Here, for example, is a short program that simply prints out any command line arguments entered by the user:

```
public class CLDemo {
      public static void main(String[] args) {
            System.out.println("You entered " + args.length
            + " command-line arguments");
            if (args.length > 0) {
                  System.out.println("They were:");
                  int i;
                  for ( i = 0; i < args.length; i++ )
                        System.out.println(" " + args[i]);
            }
      } // end main()
} // end class CLDemo
```

Note that the parameter, args, can be an array of length zero. This just means that the user did not include any command-line arguments when running the program. In practice, command-line arguments are often used to pass the names of files to a program.

Global and Local Variables

We now have three different sorts of variables that can be used inside a subroutine: local variables declared in the subroutine, formal parameter names, and static member variables that are declared outside the subroutine.

Local variables have no connection to the outside world; they are purely part of the internal working of the subroutine.

Parameters are used to “drop” values into the subroutine when it is called, but once the subroutine starts executing, parameters act much like local variables. Changes made inside a subroutine to a formal parameter have no effect on the rest of the program (at least if the type of the parameter is one of the primitive types—things are more complicated in the case of arrays and objects, as we’ll see later).

Things are different when a subroutine uses a variable that is defined outside the subroutine. That variable exists independently of the subroutine, and it is accessible to other parts of the program as well. Such a variable is said to be global to the subroutine, as opposed to the local variables defined inside the subroutine. A global variable can be used in the entire class in which it is defined and, if it is not private, in other classes as well. Changes made to a global variable can have effects that extend outside the subroutine where the changes are made. You’ve seen how this works in the last example in the previous section, where the values of the global variables, gamesPlayed and gamesWon, are computed inside a subroutine and are used in the main() routine.

It’s not always bad to use global variables in subroutines, but you should realize that the global variable then has to be considered part of the subroutine’s interface. The subroutine uses the global variable to communicate with the rest of the program. This is a kind of sneaky, back-door communication that is less visible than communication done through parameters, and it risks violating the rule that the interface of a black box should be straightforward and easy to understand. So before you use a global variable in a subroutine, you should consider whether it’s really necessary.

I don’t advise you to take an absolute stand against using global variables inside subroutines. There is at least one good reason to do it: If you think of the class as a whole as being a kind of black box, it can be very reasonable to let the subroutines inside that box be a little sneaky about communicating with each other, if that will make the class as a whole look simpler from the outside.

4.6 Return Values

A subroutine that returns a value is called a function. A given function can only return a value of a specified type, called the return type of the function. A function call generally occurs in a position where the computer is expecting to find a value, such as the right side of an assignment statement, as an actual parameter in a subroutine call, or in the middle of some larger expression. A boolean-valued function can even be used as the test condition in an if, while, for or do..while statement.

The return statements

A function takes the same form as a regular subroutine, except that you have to specify the value that is to be returned by the subroutine. This is done with a return statement, which has the following syntax:

```
return <expression>;
```

Such a return statement can only occur inside the definition of a function, and the type of the <expression> must match the return type that was specified for the function. (More exactly, it must be an expression that could legally be assigned to a variable whose type is specified by the return type of the function.) When the computer executes this return statement, it evaluates the expression, terminates execution of the function, and uses the value of the expression as the returned value of the function.

For example, consider the function definition

```
static double pythagoras(double x, double y) {
      // Computes the length of the hypotenuse of a right
      // triangle, where the sides of the triangle are x and y.
      return Math.sqrt( x*x + y*y );
}
```

Suppose the computer executes the statement “totalLength = 17 + pythagoras(12,5);”. When it gets to the term pythagoras(12,5), it assigns the actual parameters 12 and 5 to the formal parameters x and y in the function. In the body of the function, it evaluates Math.sqrt(12.0*12.0 + 5.0*5.0), which works out to 13.0. This value is “returned” by the function, so the 13.0 essentially replaces the function call in the assignment statement, which then has the same effect as the statement “totalLength = 17+13.0 ”. The return value is added to 17, and the result, 30.0, is stored in the variable, totalLength.

Note that a return statement does not have to be the last statement in the function definition. At any point in the function where you know the value that you want to return, you can return it. Returning a value will end the function immediately, skipping any subsequent statements in the function. However, it must be the case that the function definitely does return some value, no matter what path the execution of the function takes through the code.

You can use a return statement inside an ordinary subroutine, one with declared return type “void”. Since a void subroutine does not return a value, the return statement does not include an expression; it simply takes the form “return;”. The effect of this statement is to terminate execution of the subroutine and return control back to the point in the program from which the subroutine was called. This can be convenient if you want to terminate execution somewhere in the middle of the subroutine, but return statements are optional in non-function subroutines. In a function, on the other hand, a return statement, with expression, is always required.

Note that a return inside a loop will end the loop as well as the subroutine that contains it. Similarly, a return in a switch statement breaks out of the switch statement as well as the subroutine. So, you will sometimes use return in contexts where you are used to seeing a break.

4.7 Lambda Expressions

A lambda expression represents an anonymous subroutine, that is, one without a name. But it does have a formal parameter list and a definition. The full syntax is:

```
(<parameter-list>) -> {<statements>}
```

As with a regular subroutine, the <parameter-list> can be empty, or it can be a list of parameter declarations, separated by commas, where each declaration consists of a type followed by a parameter name. However, the syntax can often be simplified. First of all, the parameter typescan be omitted, as long as they can be deduced from the context. For example, if the lambda expression is known to be of type FunctionR2R, then the parameter type must be double, so it is unnecessary to specify the parameter type in the lambda expression. Next, if there is exactly one parameter and if its type is not specified, then the parentheses around the parameter list can be omitted. On the right-hand side of the ->, if the only thing between the braces, { and }, is a single subroutine call statement, then the braces can be omitted. And if the right-hand side has the form { return <expression>; }, then you can omit everything except the <expression>.

For example, suppose that we want a lambda expression to represent a function that computes the square of a double value. The type of such a function can be the FunctionR2R interface given above. If sqr is a variable of type FunctionR2R, then the value of the function can be a lambda expression, which can be written in any of the following forms:

```
sqr = (double x) -> { return x*x; };
sqr = (x) -> { return x*x; };
sqr = x -> { return x*x; };
sqr = x -> x*x;
sqr = (double fred) -> fred*fred;
sqr = (z) -> z*z;
```

The last two statements are there to emphasize that the parameter names in a lambda expression are dummy parameters; their names are irrelevant. The six lambda expressions in these statements all define exactly the same function. Note that the parameter type double can be omitted because the compiler knows that sqr is of type FunctionR2R, and a FunctionR2R requires a parameter of type double. A lambda expression can only be used in a context where the compiler can deduce its type, and the parameter type has to be included only in a case where leaving it out would make the type of the lambda expression ambiguous.

Now, in Java, the variable sqr as defined here is not quite a function. It is a value of type FunctionR2R, which means that it contains a function named valueAt, as specified in the definition of interface FunctionR2R. The full name of that function is sqr.valueAt, and we must use that name to call the function. For example: sqr.valueAt(42) or sqr.valueAt(x) + sqr.valueAt(y).

When a lambda expression has two parameters, the parentheses are not optional. Here is an example of using the ArrayProcessor interface, which also demonstrates a lambda expression with a multiline definition:

```
ArrayProcessor concat;
concat = (A,n) -> { // parentheses around (A,n) are required!
String str;
str = "";
for (int i = 0; i < n; i++)
str += A[i];
System.out.println(str);
}; // The semicolon marks the end of the assignment statement;
// it is not part of the lambda expression.
String[] nums;
nums = new String[4];
nums[0] = "One";
nums[1] = "Two";
nums[2] = "Three";
nums[3] = "Four";
for (int i = 1; i < nums.length; i++) {
      concat.process( nums, i );
}
```

This will print out

```
One
OneTwo
OneTwoThree
OneTwoThreeFour
```

Things get more interesting when a lambda expression is used as an actual parameter, which is the most common use in practice. For example, suppose that the following function is defined:

```
/* For a function f, compute f(start) + f(start+1) + ... + f(end).
*  The value of end should be >= the value of start.
*/
static double sum( FunctionR2R f, int start, int end ) {
      double total = 0;
      for (int n = start; n <= end; n++) {
            total = total + f.valueAt( n );
      }
      return total;
}
```

Note that since f is a value of type FunctionR2R, the value of f at n is actually written as f.valueAt(n). When the function sum is called, the first parameter can be given as a lambda expression. For example:

```
System.out.print("The sum of n squared for n from 1 to 100 is ");
System.out.println( sum( x -> x*x, 1, 100 ) );
System.out.print("Sum of 2 raised to the power n, for n from 1 to 10 is ");
System.out.println( sum( num -> Math.pow(2,num), 1, 10 ) );
```

As another example, suppose that we have a subroutine that performs a given task several times. The task can be specified as a value of type Runnable:

```
static void doSeveralTimes( Runnable task, int repCount ) {
      for (int i = 0; i < repCount; i++) {
            task.run(); // Perform the task!
      }
}
```

We could then say “Hello World” ten times by calling

```
doSeveralTimes( () -> System.out.println("Hello World"), 10 );
```

Note that for a lambda expression of type Runnable, the parameter list is given as an empty pair of parentheses. Here is an example in which the syntax is getting rather complicated:

```
doSeveralTimes( () -> {
      // count from 1 up to some random number between 5 and 25
      int count = 5 + (int)(21*Math.random());
      for (int i = 1; i <= count; i++) {
            System.out.print(i + " ");
      }
      System.out.println();
}, 100);
```

This is a single subroutine call statement in which the first parameter is a lambda expression that extends over multiple lines. The second parameter is 100, and the semicolon on the last line ends the subroutine call statement.

We have seen examples of assigning a lambda expression to a variable and of using one as an actual parameter. Here is an example in which a lambda expression is the return value of a function:

```
static FunctionR2R makePowerFunction( int n ) {
      return x -> Math.pow(x,n);
}
```

Then makePowerFunction(2) returns a FunctionR2R that computes the square of its parameter, while makePowerFunction(10) returns a FunctionR2R that computes the 10^{th} power of its parameter. This example also illustrates the fact that a lambda expression can use other variables in addition to its parameter, such as n in this case (although there are some restrictions on when that can be done).

4.8 APIs, Packages, Modules, and Javadoc

As computers and their user interfaces have become easier to use, they have also become more complex for programmers to deal with. You can write programs for a simple console-style user interface using just a few subroutines that write output to the console and read the user’s typed replies. A modern graphical user interface, with windows, buttons, scroll bars, menus, text-input boxes, and so on, might make things easier for the user, but it forces the programmer to cope with a hugely expanded array of possibilities. The programmer sees this increased complexity in the form of great numbers of subroutines that are provided for managing the user interface, as well as for other purposes.

The Java programming language is supplemented by a large, standard API. You’ve seen part of this API already, in the form of mathematical subroutines such as Math.sqrt(), the String data type and its associated routines, and the System.out.print() routines. The standard Java API includes routines for working with graphical user

interfaces, for network communication, for reading and writing files, and more. It's tempting to think of these routines as being part of the Java language, but they are technically subroutines that have been written and made available for use in Java programs.

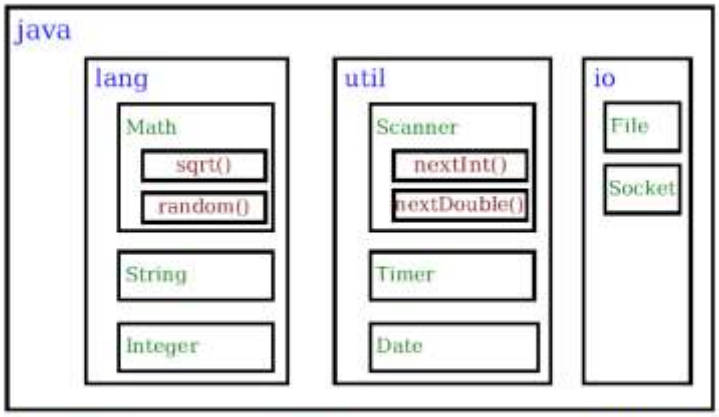
Java is platform-independent. That is, the same program can run on platforms as diverse as Mac OS, Windows, Linux, and others. The same Java API must work on all these platforms. But notice that it is the interface that is platform-independent; the implementation of some parts of the API varies from one platform to another. A Java system on a particular computer includes implementations of all the standard API routines. A Java program includes only calls to those routines. When the Java interpreter executes a program and encounters a call to one of the standard routines, it will pull up and execute the implementation of that routine which is appropriate for the particular platform on which it is running. This is a very powerful idea. It means that you only need to learn one API to program for a wide variety of platforms.

Java's Standard Packages

Like all subroutines in Java, the routines in the standard API are grouped into classes. To provide larger-scale organization, classes in Java can be grouped into packages. You can have even higher levels of grouping, since packages can also contain other packages. In fact, the entire standard Java API is implemented in several packages. One of these, which is named "java", contains several non-GUI packages as well as the original AWT graphical user interface classes. Another package, "javax", contains the classes used by the Swing graphical user interface as well as many other classes. And "javafx" contains the JavaFX API that is used for GUI programming in this textbook.

A package can contain both classes and other packages. A package that is contained in another package is sometimes called a "sub-package." Both the java package and the javafx package contain sub-packages. One of the sub-packages of java, for example, is named "util". Since util is contained within java, its full name is actually java.util. This package contains a variety of utility classes, including the Scanner class that was discussed earlier. The java package includes several other sub-packages, such as java.io, which provides facilities for input/output, and java.net, which deals with network communication. The most basic package is called java.lang. This package contains fundamental classes such as String, Math, Integer, and Double.

It might be helpful to look at a graphical representation of the levels of nesting in the java package, its sub-packages, the classes in those sub-packages, and the subroutines in those classes. This is not a complete picture, since it shows only a very few of the many items in each element:



Subroutines nested in classes nested in two layers of packages.
The full name of sqrt() is java.lang.Math.sqrt().

Similarly, the package javafx contains a package javafx.scene, which in turn contains javafx.scene.control. This package contains classes that represent GUI components such as buttons and input boxes. Another subpackage, javafx.scene.paint, contains class Color and other classes that define ways to fill and stroke a shape.

The standard Java API includes thousands of classes in hundreds of packages. Many of the classes are rather obscure or very specialized, but you might want to browse through the documentation to see what is available. As I write this, the documentation for the complete basic API for Java 8 can be found at

https://docs.oracle.com/javase/8/docs/api/

and for JavaFX at

https://docs.oracle.com/javase/8/javafx/api/toc.htm

Using Classes from Packages

Let's say that you want to use the class javafx.scene.paint.Color in a program that you are writing. This is the full name of class Color in package javafx.scene.paint. Like any class, javafx.scene.paint.Color is a type, which means that you can use it to declare variables and parameters and to specify the return type of a function. One way to do this is to use the full name of the class as the name of the type. For example, suppose that you want to declare a variable named rectColor of type Color. You could say:

```
javafx.scene.paint.Color rectColor;
```

This is just an ordinary variable declaration of the form "htype-namei hvariable-namei;". Of course, using the full name of every class can get tiresome, and you will hardly ever see full names like this used in a program. Java makes it possible to avoid using the full name of a class by importing the class. If you put

```
import javafx.scene.paint.Color;
```

at the beginning of a Java source code file, then, in the rest of the file, you can abbreviate the full name javafx.scene.paint.Color to just the simple name of the class,

which is Color. Note that the import line comes at the start of a file (after the package statement, if there is one) and is not inside any class. Although it is sometimes referred to as a statement, it is more properly called an import directive since it is not a statement in the usual sense. The import directive “import javafx.scene.paint.Color” would allow you to say

```
Color rectColor;
```

to declare the variable. Note that the only effect of the import directive is to allow you to use simple class names instead of full “package.class” names. You aren’t really importing anything substantial; if you leave out the import directive, you can still access the class you just have to use its full name. There is a shortcut for importing all the classes from a given package. For example, you can import all the classes from java.util by saying

```
import java.util.*;
```

The “*” is a wildcard that matches every class in the package. (However, it does not match sub-packages; for example, you cannot import the entire contents of all the sub-packages of the javafx package by saying import javafx.*.)

Some programmers think that using a wildcard in an import statement is bad style, since it can make a large number of class names available that you are not going to use and might not even know about. They think it is better to explicitly import each individual class that you want to use. In my own programming, I often use wildcards to import all the classes from the most relevant packages, and use individual imports when I am using just one or two classes from a given package.

A program that works with networking might include the line “import java.net.*;”, while one that reads or writes files might use “import java.io.*;”. But when you start importing lots of packages in this way, you have to be careful about one thing: It’s possible for two classes that are in different packages to have the same name. For example, both the java.awt package and the java.util package contain a class named List. If you import both java.awt.* and java.util.*, the simple name List will be ambiguous. If you try to declare a variable of type List, you will get a compiler error message about an ambiguous class name. You can still use both classes in your program: Use the full name of the class, either java.awt.List or java.util.List. Another solution, of course, is to use import to import the individual classes you need, instead of importing entire packages.

Because the package java.lang is so fundamental, all the classes in java.lang are automatically imported into every program. It’s as if every program began with the statement “import java.lang.*;”. This is why we have been able to use the class name String instead of java.lang.String, and Math.sqrt() instead of java.lang.Math.sqrt(). It would still, however, be perfectly legal to use the longer forms of the names.

Programmers can create new packages. Suppose that you want some classes that you are writing to be in a package named utilities. Then the source code files that defines those classes must begin with the line

```
package utilities;
```

This would come even before any import directive in that file. Furthermore, the source code file would be placed in a folder with the same name as the package, “utilities” in this example. And a class that is in a sub-package must be in a subfolder. For example, a class in a package named utilities.net would be in folder named “net” inside a folder named “utilities”. A class that is in a package automatically has access to other classes in the same package; that is, a class doesn’t have to import classes from the package in which it is defined.

4.9 Let Us Sum Up

In this unit we have learned about subroutine which consists of the instructions for carrying out a certain task, grouped together and given a name. We discussed subroutine as a Black Boxes and also discussed about Static Subroutines and Static Variables, different types of parameters, return Value of subroutine and Lambda Expressions. We have also discussed about APIs and Packages.

4.10 Further Reading

1. “Java 2: The Complete Reference” by Herbert Schildt, McGraw Hill Publications.
2. “Effective Java” by Joshua Bloch, Pearson Education.

4.11 Assignments

1. A “black box” has an interface and an implementation. Explain what is meant by the terms interface and implementation.
2. A subroutine is said to have a contract. What is meant by the contract of a subroutine? When you want to use a subroutine, why is it important to understand its contract? The contract has both “syntactic” and “semantic” aspects. What is the syntactic aspect? What is the semantic aspect?
3. Briefly explain how subroutines can be useful in the top-down design of programs.
4. Discuss the concept of parameters. What are parameters for? What is the difference between formal parameters and actual parameters?
5. Give two different reasons for using named constants (declared with the final modifier).
6. What is an API? Give an example.
7. What might the following expression mean in a program? (a,b) -> a*a + b*b + 1
8. Suppose that SupplyInt is a functional interface that defines the method public int get(). Write a lambda expression of type SupplyInt that gets a random integer in the range 1 to 6 inclusive. Write another lambda expression of type SupplyInt that gets an int by asking the user to enter an integer and then returning the user’s response.
9. Write a subroutine named “stars” that will output a line of stars to standard output. (A star is the character “*”.) The number of stars should be given as a parameter to the subroutine. Use a for loop. For example, the command “stars(20)” would output **********

10. Write a main() routine that uses the subroutine that you wrote for Question 7 to output 10 lines of stars with 1 star in the first line, 2 stars in the second line, and so on, as shown below.

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

Block-2: Programming in the Large

Unit 1: Programming in the Large II: Objects and Classes

1

Unit Structure

1.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand and use the class and object
- Function of Garbage collector
- Use method overloading, nested function and static members of class
- Explain the usage of wrapper classes.

1.2 CLASS, OBJECT & METHOD

An object is an entity which has several attributes and behavior. A number of objects sharing same attributes and behavior form a Class. For example: parrot, peacock, hen, dove are objects of class birds. They have attributes like colour, eating habit, shape of beak etc and behavior like fly, build nest, lay eggs etc. in java we can create a class using class keyword and declare various variables in it for its attributes and create a function for its behavior. In java for creating a class, the class keyword is used. The attributes of the class can be defined as member variable of the class and behaviour of class can be methods of class in java.

1.3 DEFINING CLASS

In java, class can be defined using class keyword follow by class name as shown in example. The definition of class is written within braces. The class name should start with capital letter. If class name has multiple words first letter of each word should be capital. For example: Student, Bird, StringBuffer etc.

```
class Student
{

}
```

1.4 ADDING VARIABLES

We can add variables in class by declaring them within class. for each attribute of class we can create variable in it. For example class Student can have attributes

like rollNumber, name, course etc. The variable name in class should be in lower case. If variable name has more than one word each word should start with capital letter except first word. For example rollNumber. The student class can be created as follows

```
class Student
{
  int rollNumber;
  String name;
  String course;
}
```

1.5 ADDING METHODS

We can define methods in class. The syntax is return type then name of method followed by arguments in bracket ( ). The function definition is written within braces. For example in Student class we can create two functions getData for assigning values to its variable and printData to print is variables.

```
class Student
{
      int rollNumber;
     String name;
     1String course;
     void getData(int r, Srting n, String c)
     {
      rollNumber = r;
     name = n;
     course = c;
     }
     Void printData()
     {
      System.out.println( rollNumber + ”  “ + name + ”   “ + course );
     }
}
```

1.6 CREATING OBJECTS

After defining class, we can use it by creating its object. This is also called instantiation of class. the new keyword is used for creating object of class.

For example,

```
ClassName x = new ClassName ( );
```

In this example ClassName is the name of class created in your program.

Example

```
class Student
{
        int rollNember;
        String name;
        String course;
        void getData(int r, String n, String c)
        {
                rollNumber = r;
                name = n;
                course = c;
        }
        void printData()
        {
                System.out.println ( rollNumber );
                System.out.println ( name );
                System.out.println ( course );
        }
        }
        class Exa_Cls
        {
        Public static void main(String args[])
        {
                Student s1 = new Student( );    //object s1 is created
                S1.getData(1, ”manan” , ”civil” );
                s1.printData();
        }
}
```

1.7 CONSTRUCTOR

In java, we can define Constrictors in a class. Constructor is a function which has same name as class name. This function will be called when we create object using new keyword. The constructors are mainly used to initialize the attributes/variables of the class. Constructor can be default constructor or parameterized constructor. In default constructor, nothing is passed as an argument. However in parameterized constructor the parameter values must be passed as arguments of constructor function.

For example:

```
class Student
{
      int rollNember;
```

```
        String name;
        String course;
        Student()         //default constructor
        {
          rollNumber = 0;
          name = "";
          course = "";
        }
        Student(int r, String n, String c)       //parameterized constructor
        {
          rollNumber = r;
          name = n;
          course = c;
        }
        void printData()
        {
          System.out.println( rollNumber );
          System.out.println( name );
          System.out.println( course );
        }
}

class Exa_Cls
{
        Public static void main(String args[])
        {
                Student s1 = new Student(1,"manan","civil");
                s1.printData();
        }
}
```

1.8 THIS KEYWORD

this is reference variable of java which points to the current object. It can also be used to point instance of the current class as shown in following example.

```
class  abc
{
        int a,b,c;
        abc(){ a = 0; b = 0; c = 0;}
        abc( int a,int b, int c)
        {
                this.a = a;
                this.b = b;
                this.c = c;
        }
}
class MyExa
{
```

```
        public static void main(String args[])
        {
                abc x = new abc(1,2,3);
        }
}
```

In above example, in class abc, this.a, this.b and this.c are referring the variable of class abc and a,b and c are the parameters of constructor.

1.9 GARBAGE COLLECTION

In C, when we allocate memory at runtime using malloc() function, at the end of program we have to free them using free() function. Similarly in C++, when we create memory for any object/variable using new, we should free them using delete.

In java when we are creating memory for reference variable/object, programmer don't care about destroying them. There is a special component in JVM called garbage collector which will take care of deletion of all memory occupied by java programs. It frees the heap memory occupied by reference variables which are not in use. Java has an automatic garbage collection.

1.10 FINALIZE() METHOD

The finalize() is a method of java.lan.Object class which is called by garbage collector for the which is identified to be destroyed. It is because there are no reference to that object in program. In a class we can override (redefine) the finalize method to perform the cleanup of system resources.

1.11 ACCESSING CLASS MEMBERS

To access the member variables and methods of the class, we should create the object of the class using new keyword. And using the object name and variable/method name separated by . we can access the member variable the example is shown in section 2.7 and 2.6.

1.12 METHODS OVERLOADING

Method overloading is the feature of object oriented programming. It is used to implement polymorphism. In java in a same class we can define more than one method with same but different signature, this concept is called method overloading. In method overloading same method can be used in different manners. For example in class Add, we can define 3 addition methods shown below,

```
class Add
{
        int  addition(int a, int b){ return ( a + b ); }
        float addition(float a, float b) { return ( a + b ); }
        String addition(String a, String b) { return a + b;  }
}
public class Sum
{
        public static void main(String args[])
        {
        Sum s1 = new Sum();
        System.out.println(s1.addition(10, 20));
        System.out.println(s1.addition(10.56 ,20.78));
        System.out.println(s1.addition("abc", "def"));
}

}
```

1.13 NESTING OF METHODS

When a method of class calling the other method of the same class is called nesting of methods. The following example uses nesting of method.

```
import java.util.Scanner;
class Circle
{
        int radius;
        void getRadius()
        {
        Scanner sc=new Scanner(System.in);
        Radius = sc.nextInt();
        }
        double area()
        {
        getRadius();
        return(3.14*radius*radius);
        }
}
```

```
public class Exa
{
        public static void main(String args[])
        {
        Circle c1 = new Circle();
        System.out.println (c1.area());
        }
}
```

1.14 WRAPPER CLASSES

Wrapper classes are the classes whose objects wrap the primitive data types. To treak primitive data type as a Class and Object, java provide a wrapper class for each primitive data types. The following is the list of wrapper classes and their corresponding primitive data types.

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

Table-7 list of wrapper classes and their corresponding primitive data types

Advantages of wrapper class:

1). They convert a primitive data type into object when we need to pass them as reference argument to the function. By default the primitive data types are passed as value into the function.

2). The Vector can store objects only. If we want to store primitive data values in Vector, we need to convert them into objects.

Autoboxing is an important concept related to wrapper classes. Autoboxing is an automatic conversion of primitive data types into object of its wrapper class. The reverse process of autoboxing is called unboxing. Unboxing is automatical conversion of object of wrapper class into its corresponding primitive data type.

For example

1)
```
int a = 5;
Integer aa = a;        //autoboxing
```

2)
```
Vector v1 = new Vector();
v1.add(24);   //autoboxing 24 into Integer object
v1.add(89);
int n=v1.firstElemen();  //unboxing
```

Example

```
class Exa3
{
	public static void main(String args[])
	{
	//Autoboxing
	byte a = 10;
	Byte aobj = new Byte(a);

	int b = 289;
	Integer bobj = new Integer(b);

	float c = 508.5f;
	Float cobj = new Float(c);

	double d = 90.3;
	Double dobj = new Double(d);

	char e='x';
	Character eobj=e;

	System.out.println("Autoboxing");
	System.out.println(aobj);
	System.out.println(bobj);
	System.out.println(cobj);
	System.out.println(dobj);
	System.out.println(eobj);

	//Unboxing
	byte v = aobj;
	int w = bobj;
	float x = cobj;
	double y = dobj;
	char z = eobj;

	System.out.println("Unboxing");
	System.out.println(v);
	System.out.println(w);
	System.out.println(x);
	System.out.println(y);
```

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

➢ **Methods of wrapper classes**

The following are some of the methods of wrapper class.

- **valueOf(String s)**

  All wrapper class except Character class have this function. It is a static function hence called using class name. This function coverts a String representation of any primitive value into its corresponding wrapper class object.

  Example:

  ```
  Integer a=Integer.valueOf(“100”);
  Byte b=Byte.valueOf(“8”);
  Double c=Double.valueOf(“10.80”);
  ```

- **valueOf(String s, int radix):**

  This is a static function of Byte, Short, Inetger and Long wrapper class. This function converts a string into corresponding wrapper class object. However the String stores the value represented in radix form. Radix 2 is for binary, 8 is for octal, 16 is for hexadecimal and so on.

  For example

  Integer a=Integer.valueOf(“101”,2);//store 7 in a because 101 is binary of 7.

- **valueOf(primitive_data_type x):**

  All wrapper classes have this static function which converts a primitive data value into its corresponding wrapper class object.

  For example

  ```
  Integer a = Integer.valueOf(100);
  Double b = Double.valueOf(34.6);
  ```

Example

```
public class ExWrap1
{
	public static void main(String args[])
	{
	// example of valueOf
	System.out.println(" valueOf converts String into Wrapper class object");
	Integer a=Integer.valueOf("100");
	Byte b=Byte.valueOf("8");
	Double c=Double.valueOf("10.80");
	System.out.println("Integer: " + a);
	System.out.println("Byte: " + b);
	System.out.println("Double: " + c);

	System.out.println(" valueOf converts String with differnt base into Wrapper class
				object");
	Integer a1=Integer.valueOf("1110",2);
	System.out.println("Integer: " + a1);

	System.out.println(" valueOf converts primitive data type into Wrapper class
				object");
	Integer a2 = Integer.valueOf(100);
	Double b2 = Double.valueOf(34.6);
	System.out.println("Integer: " + a2);
	System.out.println("Integer: " + b2);

	}
}
```

- **Primitive data type conversion functions**

public byte byteValue(), public short shortValue(), public int intValue(), public long longValue(), public float floatValue(), public float doubleValue() are the non static functions. They need object of Wrapper class to call. The numeric wrapper classes like Byte, Short, Integer, Long, Float, and Double has these all methods defined in them. These methods are used to return corresponding primitive data type value.

For example,

```
Integer x = new Integer(189);
int  y = x.intValue();
byte z = x.byteValue();
float a = x.floatValue();
```

Example:

```
public class ExWrap2
{
	public static void main(String args[])
	{
	System.out.println(" xxxValue functions converts one numeric datatype into other
				");
	Integer x = new Integer(122);

	int  y = x.intValue();
	byte z = x.byteValue();
	float a = x.floatValue();
	System.out.println(" int :" + y);
	System.out.println(" byte :" + z);
	System.out.println(" float :" + a);

	}
}
```

- **String to primitive data type conversion functions**

public static int parseInt(String s), public static byte parseByte(String s), public static short parseShort(String s), public static long parseLong(String s), public static float parseFloat(String s), public static double parseDoublet(String s), public static boolean parseBoolean(String s)

All the wrapper class except Character class has parse function. This function is used to convert a String argument into corresponding primitive data type value.
For example:

```
int x  = Integer.parseInt(“123”);
double y = Double.parseDouble(“123.56”);
boolean z = Boolean.parseBoolean(“false”);
```

The parse function has one more version which is,
public static int parseInt(String s, int radix) for Integer class.

Similarly the wrapper classes Byte, Short and Long have this function. It converts a String s, which represents a number with base radix into primitive data types.

For example,

int x=Integer.parseInt(“1111”,2); //this converts a binary 1111 into integer.

This function can be used to convert string representation of binary (radix 2), octal(radix 8) or hexadecimal (radix 16) number into decimal value.

Example:

```
public class ExWrap3
{
	public static void main(String args[])
	{
	System.out.println(" parseXXX functions converts String to primitive data type");

	int x  = Integer.parseInt("123");
	double y = Double.parseDouble("123.56");
	boolean z = Boolean.parseBoolean("false");

	System.out.println(" int :"  +  x);
	System.out.println(" double :" + y);
	System.out.println(" boolean :" + z);

	System.out.println(" parseXXX functions converts a String representation of a
                                number with  base radix into primitive data types.");
	int x1=Integer.parseInt("1111",2);
	System.out.println(" decimal of 1111 is int :" + x1);

	}
}
```

- **public String toString()**

every wrapper class has this function. It is used to convert a wrapper class object into String.

For example,

```
Double d=new Double(123.88);
String s=d.toString();   //stores “123.88” into s
```

- **public static String toString(primitive p)**

every wrapper class has this function. It is used to convert a primitive data type value into String.

For example,

```
String s=Double.toString(123.89);
```

Example:

```
public class ExWrap4
{
public static void main(String args[])
{
System.out.println( "non static toString functions converts wrapper object to String ");

Double d = new Double(123.88);
String s = d.toString();

System.out.println(" String : "  +  s);

System.out.println(" static toString functions converts primitive data type into String
");
String x1 = Double.toString(123.89);
System.out.println(" String :"  +  x1);

}
}
```

1.15 LET US SUM UP

class: a non primitive data type which encapsulates variables and function in it.

object: an instance of class or variable of type class. The new keyword is used to create object.

member variable: list of variables defined in class

member function: methods/functions defined within class

constructor: It is a function of a class having same name as class name. It is called to initialized object when it is created.

Garbage collection: it automatically frees the unnecessary memory area of the program.

finalize(): this method will be called by garbage collector before destroying the object.

method overloading: In a class we can write more than one method with same name and different signature.

wrapper classes: For each primitive data type there is a class in java which is called wrapper class. The wrapper class wraps the primitive data value as an object and can have various data conversion functions.

1.16 CHECK YOUR PROGRESS

- True-False with reason:

1. Class and object are same.
2. Static member function can be called without object.
3. We can enhance capacity of Vector at rum time.
4. Constructor function can have any name.
5. We can write only one constructor function for a class.
6. We can not call static function inside non static function.
7. Instance variables are shared by all objects of the class
8. A[1] refer to the first element of the array
9. Array can be initialized.
10. We can implement matrix using single dimensional array.

- Answer the followings:

1. List all wrapper class.
2. How can we create an object of wrapper class?
3. How can we create an array of 10 integers?
4. How can we create an object of a class?
5. Give example of method overloading.
6. How can we convert a string “102” into a number?
7. How can we find size of a vector object?
8. Compare class variable and instance variable
9. Compare Vector and array.
10. Compare class and object.

- Identify the class and its attributes and methods from following problem statement.

1. In school software, they are storing information of each students and staff.
2. In library software, they are allowing issue and return of the book by library members.
3. We want to design software for restaurant bill generation.

1.17 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

- True-False with reason:

1. False. Objects are the instance of class.
2. True.
3. True.
4. False. Constructor function must have name as class name.
5. False. We can write multiple constructor for a class with different argument in each.
6. False. We can call static function inside non static function.
7. False. Class variables/static variables are shared by all objects of the class
8. False. A[0] refer to the first element of the array
9. True.
10. False. We can implement matrix using two dimensional array

- Answer the followings:

1. Wrapper Classes:
   Boolean, Byte, Short, Integer, Long, Float, Double, Character
2. To create an object of wrapper class:
   Boolean a=true;
   Boolean x=a;
3. To create an array of 10 integers :
   int[] a=new int[10];
4. To create an object of a class:
   Class_Name obj= new Class_Name();
5. Example of method overloading:

```
class Ex_Add
{
    static int add(int a,int b){return a+b;}
    static int add(int a,int b,int c){return a+b+c;}
}
class ExOverloading
{
```

```
    public static void main(String[] args)
    {
        System.out.println(Adder.add(11,11));
        System.out.println(Adder.add(11,11,11));
    }
}
```

6. To convert a string “102” into a number:

   int a= Integer.parseInt(“102”);

7. The size() method of Vector class in Java is used to get the size of the Vector.
8. Class variable v/s Instance variable

Class variable	Instance variable
They are static member variables of class	They are non static member variables of class
They are shared among all object of class	They are separately created for each object
To access class variable class name is used.	To access instance variable object name is used.

9. Vector v/s array.

Vector	Array
Vector is resizable array	The length of an Array is fixed.
Vector is synchronized	Array is not synchronized.
Vector can store any type of objects	Array can store same type of objects
Vector is slow to access.	Array supports efficient random access to the members

10. Class v/s object.

Class	Object
It is a blueprint/structure of object.	It is an instance of class
Class is a group of similar entities	Object is a real world entity
Class is declared once	Object is created many times as per requirement.
Class doesn't allocated memory when it is created.	Object allocates memory when it is created.

- Identify the class and its attributes and methods from following problem statement.

1. In school software, they are storing information of each students and staff.

   Class name : Student

   Attributes : enrollment number, name, course, address, phone number, semester

   Methods: enroll_course(int enr_no, String crs), print_data(), get_data()

   Class name : Staff

   Attributes : Employ ID, name, designation, address, phone number, qualification

   Methods: enroll_course(int enr_no, String crs), print_data(), get_data()

2. In library software, they are allowing issue and return of the book by library members.

   a. Class name: Member
   b. Attributes : Library ID, name, address, phone number

c. Methods: add_member(), searchMember(), printAllMembers(), deleteMember()
d. Class name: Book
e. Attributes : bookID, title, author, publisher, price, qty
f. Methods: addBook(), searchBook(), printAllBooks(), deleteBook()
g. Class name: Book_transaction
h. Attributes: bookID, Library ID, date_issue, date_return, fine.
i. Methods : bookIssue(), bookReturn()

3. We want to design software for restaurant bill generation.

a. Customer : custId, custName, custAddr, custPhone
b. Methods : addCust(), searchCust(), deleteCust()
c. Item: itemID, itemName, itemCategory, itemPrice
d. Methods : addItem(), searchItem(), deleteItem()
e. Bill : billID, custID, itemID, qty, billDate, billAmount
f. Methods : billGeneration(), billPayment(), printBill()

2.21 FURTHER READING

1. "Java 2: The Complete Reference" by Herbert Schildt, McGraw Hill Publications.

2. "Effective Java" by Joshua Bloch, Pearson Education.

2.22 ASSIGNMENTS

- Write java program for following:
  1) Create a class name meter which represents a distance in meter and centimeter. Also create class name kilometer which represents distance in km and meter. In both class write a function which converts one class to other.
  2) Create a class name Doctor with properties and methods. The properties can be name, phone number, qualification, specialization etc. The methods include getting information of doctor and printing them.
  3) Sort numbers in descending order.

4) Create a menu driven program for matrix operations like add, subtract, and multiply.
5) Find maximum and minimum from the n numbers.
6) Create a class student with necessary properties, methods and constructor. Overload a function name search in this class which allows us to search student based on roll number, name and city.
7) To print transpose of matrix.
8) To implement pop and push operation of stack using array.

Unit 2: Programming in the Large III: Inheritance and Interface

2

Unit Structure

2.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand the inheritance and its types
- Implementation of various types of inheritance in java.
- Use of final keyword with variables, function and class.
- Use of abstract class and abstract function to implement polymorphism.
- Use of function overriding and its implementation
- Understand Object class and its functions.

2.2 INHERITANCE

Using inheritance a class can inherit attributes and methods of the other class. It is like a child inherits the features of parents. It helps us to reuse an already available class, which is called reusability, an important feature of Object oriented programming.

The class which inherits the properties and method of existing class is called subclass or child class and the existing class is called super class or parent class.

The inheritance can be of various types. They are single inheritance, multiple inheritance, multilevel inheritance, hierarchical inheritance and hybrid inheritance.

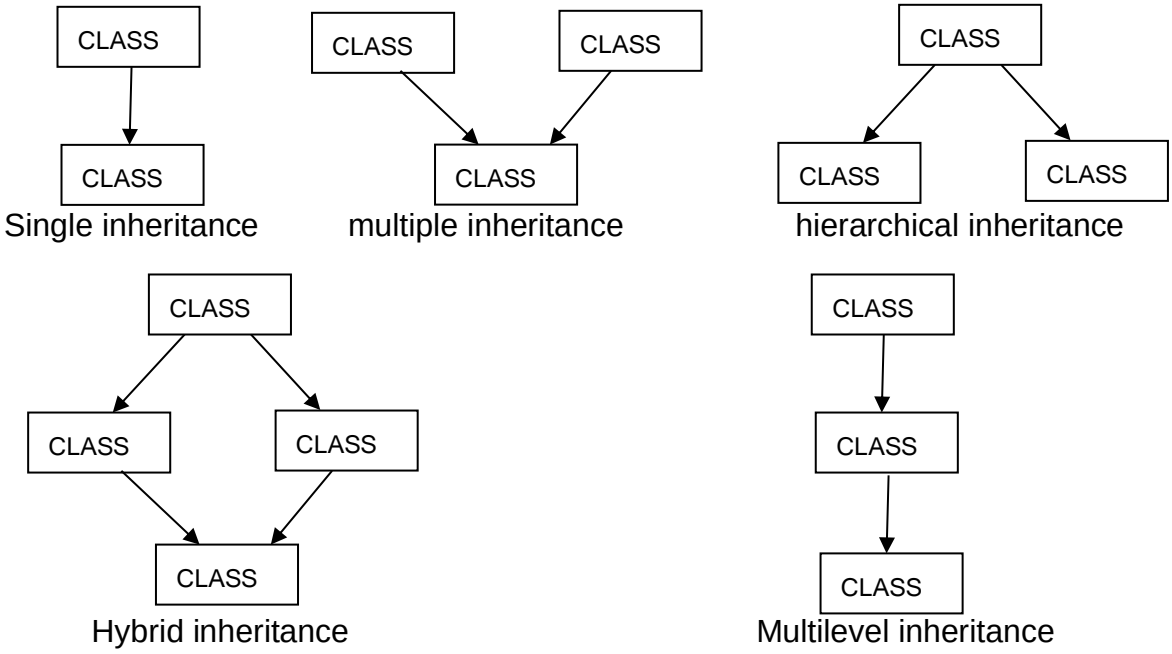


Figure-25 Pictorially view of type of inheritance

- **Single inheritance** : Class B is inherited from Class A.

- **Multiple inheritance** : Class C is inherited from both Class A and Class B.
- **Hierarchical inheritance**: Class B and Class C are inherited from a single Class A.
- **Multilevel inheritance**: Class B inherited from Class A and Class C is inherited from Class B. here class C has properties and methods of Class A also through Class B.
- **Hybrid inheritance**: it is combination of two inheritance that are hierarchical and multiple inheritance

In java we can implement single inheritance, hierarchical inheritance and multilevel inheritance using class.For implementation of multiple and hybrid inheritance in java interfaces are used.

2.3 SUB CLASS

In java for implementing inheritance extends keyword is used. The syntax is as below,

```
class X
{
}
class Y extends X
{

}
```

Here X is a super class or parent class and Y is a sub class or child class. Class Y inherits class X.

Example,

```
class A
{
	int a;
	A() {a = 0; }
	A( int x ) { a = x; }
	void printA() { System.out.println(“ a = “ + a); }
}
class B extends A
{
	int b;
	B() { a = 0;  b = 0; }
	B(int x, int y) { a = x;  b = y; }
	void printB()
```

```
        {
        printA();
        System.out.println(“ b = “ + b);
        }
}
```

The above example shows single inheritance.

2.4 SUBCLASS CONSTRUCTOR

In above example, the class B has its own constructor in which it initializes the value of parameters of both class B (child) as well as class A (parent). We can also call constructor of parent class in child class for that super keyword is used. The super keyword is used to store reference of parent class object in child class. The method and properties of parent class can be accessed using super keyword in child class.

For example:

```
class A
{
        int a;
        A() {a = 0; }
        A( int x ) { a = x; }
        void printA() { System.out.println(“ a = “ + a); }
}
class B extends A
{
        int b;
        B() {super();  b = 0; }
        B(int x, int y) { super(x);  b = y; }
        void printB()
        {
        super.printA();
        System.out.println(“ b = “ + b);
        }
}
```

Example:

```
class Person
{
        String name;
        String address;
        int phno;
        Person(){ name = ""; address = ""; phno = 0;}
        Person(String n, String a, int p){ name = n; address = a; phno = p;}
        void printP()
```

```
        {
        System.out.println("Name : " + name);
        System.out.println("Address : " + address);
        System.out.println("Phone Number : " + phno);
        }
}
class Student extends Person
{
         int rollNumber;
         String course;
         Student(){ super(); rollNumber = 0; course = "";}
         Student(String n, String a, int p,int r, String c)
        {
         super(n,a,p);
         rollNumber = r;
         course = c;
        }
        void printS()
        {
        printP();
        System.out.println("Roll number : " + rollNumber);
        System.out.println("Course : " + course);
        }

}
public class ExSimple
{
        public static void main(String args[])
        {
        Student s1=new Student("Aryan","Surat",34567890,12,"Computer");
        s1.printS();
        }
}
```

2.5 HIERARCHICAL INHERITANCE

This inheritance can be implemented using extends key word in java. In hierarchical inheritance more than one child can be inherited from the same parent class.

For example,

```
class Parent

{

}
```

```
class child1 extends Parent

{

}

class child2 extends Parent

{

}
```

Here, class Parent is the super class/parent class, which has two children class Child1 and Child2. Child1 and Child2 are also called sibling as they have same parent.

Example:

```
class A
{
	int a;
	A() {a = 0; }
	A( int x ) { a = x; }
	void printA() { System.out.println(“ a = “ + a); }
}

class B extends A
{
	int b;
	B() {super();  b = 0; }
	B(int x, int y) { super(x);  b = y; }
	void printB()
	{
	super.printA();
	System.out.println(“ b = “ + b);
	}
}

class C extends A
{
	int c;
	C() {super();  c = 0; }
	C(int x, int z) { super(x);  c = z; }
	void printB()
	{
	super.printA();
	System.out.println(“ c = “ + c);
	}
```

```
}
public class ExInh
{
        public static void main(String args[])
        {
         B b1 = new B(10,20);
         C c1 = new C(23,34);
         b1. printB();
         c1. printC();
        }
}
```

2.6 OVERRIDING METHODS

When a child class inherits a parent class, we can redefine a method of parent class in child class. This concept is called method overriding.

For example,

```
class A
{
        int a;
        A() {a = 0; }
        A( int x ) { a = x; }
        void printData() { System.out.println(“ a = “ + a); }
}
class B extends A
{
        int b;
        B() {super();  b = 0; }
        B(int x, int y) { super(x);  b = y; }
        void printData()          //the method of parent class is redefined
        {
        System.out.println(“ a = “ + a);
        System.out.println(“ b = “ + b);
        }
}
public class ExInh
{
        public static void main(String args[])
        {
         A a1 = new A(10);
         B b1 = new B(23,34);
         a1. printData();
         b1. printData();
```

```
    }
}
```

In above example printData() method of parent class A is override in child class B. when we call printData method using object of child class, the method of child class will be called. When we call same method using object of parent class, the parent class printData method will be called.

We can also call child class method using reference of parent class. That means when parent class refer parent object it will call parent class's method. And when parent class refers child class object, it will call child class's method.

It is decided at run time which method will be called using reference of parent class. This concept is called dynamic binding or dynamic method dispatch.

For example,

```
class B extends A
{
    int b;
    B() {super();  b = 0; }
    B(int x, int y) { super(x);  b = y; }
    void printData()          //the method of parent class is redefined
    {
    System.out.println(" a = " + a);
    System.out.println(" b = " + b);
    }
}

public class ExInh
{
    public static void main(String args[])
    {
     A a1=new A(10);
     B b1=new B(23,34);
     b1.printData();
     a1=b1;
     b1.printData();
    }
}
```

In this example in main method b1.printData() method is called twice. Both time it will run different method. This is due to runtime binding of object with class. It decides at run time which method will be called. This can also be an example of polymorphism.

2.7 FINAL VARIABLE

In java, when a variable is declared as final, it is constant. We have to assign value to this variable while declaring them final. We cannot change value of final variables in our program. Final variables are same as constant variables of C++ and C.

For example,

```
final int N=50;
```

Using final variable in java program

```
public class ExInh
{
        public static void main(String args[])
        {
        final int x = 80;
        int[] a = new int [ x ];        //we can use x but can not modify it
        x = 90;         //this gives compilation error as x is constant
        }
}
```

2.8 FINAL METHOD

We can also declare method of a class final. If any method of class defined final it cannot be override/redefine in its child class. Final methods of parent calss can not be overridden in child class. The final methods cannot be changed outside the class.

For example,

```
class A
{
        int a;
        A() {a = 0; }
        A( int x ) { a = x; }
        final void printData() { System.out.println(" a = " + a); } // can not override
}
class B extends A
{
        int b;
        B() {super();  b = 0; }
        B(int x, int y) { super(x);  b = y; }
        void printB()
        {
        Super.printData();
```

```
        System.out.println(“ b = “ + b);
        }
}
```

However, if we override the method declared final it gives us a compilation error.

For example,

```
class A
{
         int a;
         A(){ a = 0;}
         A(int x){ a = x;}
         final void printA()
        {
         System.out.println(" a = " + a);
        }
}
class B extends A
{
         int b;
         B(){ super(); b = 0;}
        B(int x, int y) { super(x); b = y;}
        int printA(){ return a+b;}
}
public class ExFinal
{
        public static void main(String args[])
        {
        B b1 = new B(10,20);
        System.out.println(b1.printA());
        }
}
```

2.9 FINAL CLASS

In java class can also be final. The final class restrict them from inheritance. We cannot inherit a class if it is declared as final.

```
final class A
{
}
```

We can not create any class B which inherits class A.
For example,

```
final class A
{
         int a;
         A(){ a=0;}
```

```
        A(int x){ a=x;}
        void printA()
       {
        System.out.println(" a = " + a);
       }
}
class B extends A
{
        int b;
        B(){ super(); b=0;}
       B(int x, int y) { super(x); b=y;}
}
public class ExFinal
{
        public static void main(String args[])
        {
        B b1=new B(10,20);
        b1.printA();
        }
}
```

This program gives compilation error because we try to inherit final class A in this program.

2.10 ABSTRACT CLASS

Abstract class is used to implement abstraction which is am important OOP concept. It is used to create a class with partial implementation. The subclass of abstract class must complete the implementation left in abstract class.

In java, abstract class can be created using abstract keyword. The abstract class is a class which has at least one method declared as an abstract method.

Abstract methods are the methods of a class which are declared in class and have no definition. These methods must be defined in the child classes which are inherited from that abstract class.

We cannot create an object of abstract class. The abstract class can define constructor, non abstract methods, static methods as well as final methods.

Abstract class enforces inheritance. To use abstract class we have to create a class which inherits an abstract class. We can access the method of subclass using the parent class reference.

For example,

```
abstract class A
{
        int a;
        A() {a = 0; }
        A( int x ) { a = x; }
        abstract void printData();
}
class B extends A
{
        int b;
        B() {super();  b = 0; }
        B(int x, int y) { super(x);  b = y; }
        void printData()       //definition of abstract method of parent class A
        {
        System.out.println(“ a = “ + a);
        System.out.println(“ b = “ + b);
        }
}
```

Example:

```
abstract class Shape
{
        double ar;
        double peri;
        Shape()
        {
         ar = 0.0;
         peri = 0.0;
        }
        final double PI=3.14;
        abstract void area();
        abstract void perimeter();
        void printArea()
        {
        System.out.println("Area : " + ar);
        }
        void printPerimeter()
        {
        System.out.println("Perimeter : " + peri);
        }

}

class Circle extends Shape
{
         int r;
         Circle(){ r = 0; }
         Circle(int r){ this.r = r; }
         void area()
```

```
        {
         ar = PI*r*r;
        }
         void perimeter()
        {
         peri = 2*PI*r;
        }

}

class Square extends Shape
{
         int s;
         Square(){ s = 0; }
         Square(int s){ this.s = s; }
         void area()
        {
         ar = s * s;
        }
         void perimeter()
        {
         peri = 4 * s;
        }

}

public class ExInh1
{
        public static void main(String args[])
        {
        Shape c1=new Circle(2);
        c1.area();
        c1.printArea();
        c1.perimeter();
        c1.printPerimeter();
        c1=new Square(2);
        c1.area();
        c1.printArea();
        c1.perimeter();
        c1.printPerimeter();
        }
}
```

In the above example in main method, we are using reference of Shape class to call methods of child class. When reference of Shape (c1) refers to circle object(line 1) it calls method of Circle class. Same reference can also be used to call the method of Square class. You can see in the main method, line number 2,3,4,5 and line 7,8,9,10 are same in syntax but the line 2,3,4 and 5 calls method of Circle class where as late

four lines calls methods of square class. Thus same line code can be executed differently which is an implementation of polymorphism concept of OOP.

2.11 MULTIPLE INHERITANCE

The multiple inheritance cannot be implemented in java using class. We have to use interface. For creating interface we need to use interface keyword. Interface are created with declaration of methods and constant variables in it. All the methods of interface are either abstract or final. All variables in interface are final and static. We cannot create an instance of interface. We need to implement it in its child class. Interface can extend other interface. A class can implement one or more interface using implement keyword. By default, all the method in interface are abstract and all the variables are final and static. The method in interface must be declared public.

For example multiple inheritance can be implemented as below,

```
interface A
{
        int x = 5;
        public void getData();
        public void printData();
}

interface B
{
        int y = 2;
        public void getD();
        public void printD();
}

class C implements A,B
{
        int [] data;
        C () { int [] data = new int[ x + y ]; }
        public void getData()
        {
        for( int i = 0; i < x ; i++)
        data[i] = 10 * ( i + 1 );
        }
        public void printData()
        {
        for( int i = 0; i < x ; i++)
        System.out.println( data[i] );
        }
```

```
        public void getD()
        {
        for( int i = x; i < x+y ; i++)
        data[i] = 10 * ( i + 1 );
        }
        public void printD()
        {
        for( int i = x; i < x+y ; i++)
        System.out.println( data[i] );
         }
}
```

Example:

```
interface Shape
{
        double PI=3.14;
        public double area();
        public double perimeter();
        public void printData();
}

class Circle implements Shape
{
         int r;
         Circle(){ r = 0; }
         Circle(int r){ this.r = r; }
        public double area()
        {
          return PI*r*r;
        }
        public double perimeter()
        {
         return 2*PI*r;
        }
        public void printData()
        {
        System.out.println("Area : " + area());
        System.out.println("Perimeter : " + perimeter());
        }
}

class Square implements Shape
{
         int s;
         Square(){ s = 0; }
         Square(int s){ this.s = s; }
        public double area()
        {
         return s*s*1.0;
```

```
        }
        public double perimeter()
        {
         return 4.0*s;
        }
        public void printData()
        {
        System.out.println("Area : " + area());
        System.out.println("Perimeter : " + perimeter());
        }
}
public class ExInf
{
        public static void main(String args[])
        {
        Shape c1=new Circle(5);
        c1.printData();
        }
}
```

2.12 THE OBJECT CLASS

Object class is available in java.lang package in java. Any class created in java, automatically derived from Object class. Hence methods of Object class available in all classes of java. The Object class is root of all class.

Some of the methods of Object class:

- **String toString():**
  it converts an object into String. It returns a string consists of name of class, '@' and hashcode of the object. We can customize the output of toString() function by overriding it in our class.

  Example,

```
class A
{
int a;
A() { a = 0; }
A( int x ) { a = x; }
}
public class ObjEx
{
public static void main(String args[])
{
A x1 = new A( 5 );
```

```
System.out.println( " toString " + x1.toString() );
}
}
```

- **Example of overriding toString()**

```
class A
{
int a;
A() { a = 0; }
A( int x ) { a = x; }
public String toString()
{
return " Object of Class A ";
}
}
public class ObjEx
{
public static void main(String args[])
{
A x1 = new A( 5 );
System.out.println( " toString " + x1.toString() );
}
}
```

- **int hashCode():**

  it is used to get hashvalue of object which can be used to search for object. Hashcode is unique for each object.

  Example,

```
public class ObjEx
{
public static void main(String args[])
{
String s = new String(" Hello ");
Class c = s.getClass();
System.out.println ( " class of object s is :" + c.getName() );

}
}
```

- **boolean equals(Object obj):**

  compare object obj with this object and returns true if equal else false.

  For example,

```
class A
{
int a;
A() { a = 0; }
A( int x ) { a = x; }
}
public class ObjEx
{
public static void main(String args[])
{
A x1 = new A( 5 );
A x2 = x1;
if ( x1.equals(x2))
System.out.println( " x1 and x2 are equal" );
else
System.out.println( " x1 and x2 are not equal" );

}
}
```

- **Class getClass():**

  Returns Class object of this object. The Class object has method name getName() which returns name of class which of the type of this object.

  For example:

```
A a = new A(15);
Class c = a.getClass();
// print A as class name
System.out.println( “ class of object s is :” + c.getName());
```

  Example,

```
public class ObjEx
{
public static void main(String args[])
{
String s = new String(" Hello ");
Class c = s.getClass();
System.out.println( " class of object s is :" + c.getName());

}
}
```

- **finalize():**

  This method is called before call of garbage collector in java. this method is called for each object once.

for example,

```
class A
{
int a;
A(){ a = 0; }
A(int x) { a = x; }
protected void finalize()
{
System.out.println(“ finalize method is called “ );
}
}
public class ExFin
{
public static void main(String args[])
{
A a1=new A(4);
a1 = null;
System.gc();
}
}
```

- Object clone():

  This method returns an object that is same as this object. For using clone() function the class must implements Cloneable interface and implements a function name clone in it. Also the method which calling clone function must handle the CloneNotSupportedException. We will discuss more about Exception in unit 6 of this book.

  For example,

```
class A implements Cloneable
{
int a;
A() { a = 0; }
A( int x ) { a = x; }
 public Object clone() throws CloneNotSupportedException
   {
      return super.clone();
   }
public void printData()
{
System.out.println(" a : " + a );
}

}
public class ObjEx
```

```
{
public static void main(String args[]) throws CloneNotSupportedException
{
A x1 = new A( 5 );
A x2 = ( A ) x1.clone();
x1.printData();
x2.printData();

}
}
```

2.13 LET US SUM UP

Inheritance: it is an important object oriented programming features in which we can reuse existing class by adding new features and methods in it.

Subclass: in inheritance the class which derives the existing class is called subclass

Super class: in inheritance class from which a subclass is derived is called super class

super keyword: in child class, super is a reference to object of parent class. We can access parent class properties and method using super key word.

method overriding: The function of parent class and be redefined in child class, this is called method overriding.

final variable: it is used to define constant variables in java.

final function: it is a function of parent class which cannot be overridden in child class.

final class: the final class cannot be inherited. We cannot create a child of final class.

abstract class: Abstract class is a class which has at least one abstract method declared in it. This class cannot be instantiated. We have to inherit this class to used it.

abstract function: this methods have only signature in class. The subclass which inherits the parent class must define all the abstract methods in it.

Interface: it must have only static final variables and abstract and final methods in it. It supports multiple inheritance in java.

Object class: Object class is available in java.lang package library. It is the parent of each class created in java program

2.14 CHECK YOUR PROGRESS

- True-False with reason

1. extends keyword is used to inherit a class.
2. implements keyword is used to inherit a class.
3. abstract class cannot be inherited.
4. Final class cannot be inherited.
5. Final method cannot be overloaded.
6. Interface and class are same.
7. Object class is parent of each class created in java.
8. Interface can have at least one abstract function in it.
9. All the variable declared in interface are final and static.
10. All variables declared in abstract class are final.
11. Method overriding is writing more than one method in a class with same name.
12. Super is a reference to object which is accessing the variable or method of class.
13. We can call constructor of parent class using super keyword.
14. Multilevel inheritance is not supported in java using class.
15. Multiple inheritance is possible using interface.

2.15 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

- True-False with reason

1. True
2. False. implements keyword is used to inherit an interface.
3. False. abstract class has to be inherited.
4. True
5. False. Final methods cannot be overridden.
6. False. Class does not support multiple inheritance where as interface does.
7. True
8. False. Interface has all abstract functions in it.

9. True
10. False. At least one method in abstract class must be abstract.
11. False. Method overriding is writing a definition of a parent class's method in subclass.
12. False. Super is a reference to object which is accessing the variable or method of parent class
13. True.
14. False. Multilevel inheritance is supported in java using class.
15. True.

2.16 FURTHER READING

1. Java Inheritance (Subclass and Superclass) - W3Schools https://www.w3schools.com/java/java_inheritance.asp

2. Inheritance in Java OOPs with Example - Guru99 https://www.guru99.com/java-class-inheritance.html

3. "Java 2: The Complete Reference" by Herbert Schildt, McGraw Hill Publications.

4. "Effective Java" by Joshua Bloch, Pearson Education

2.17 ASSIGNMENTS

- Write java program for following

1) Create a class to find out the Area and perimeter of rectangle.
2) Create a class quadrilateral and create two methods each for calculating area & perimeter of the quadrilateral with one & two parameters respectively Check number is even or odd.
3) Define a class student with the following specifications:
   Private members of the class:
   Admission Number - An Integer
   Name - string of 20 characters
   Class - Integer
   Roll Number - Integer
   Public members of the class:

getdata() - To input the data

showdata() - To display the data

Write a program to define an array of 10 objects of this class, input the data in this array and then display this list.

4) A class STUDENT has 3 data members:

Name, Roll Number, Marks of 5 subjects, Stream

and member functions to input and display data. It also has a function member to assign stream on the basis of the table given below:

Average Marks	Stream
96% or more	Computer Science
91% - 95%	Electronics
86% - 90%	Mechanical
81% - 85%	Electrical
75% - 80%	Chemical
71% - 75%	Civil

Declare a structure STUDENT and define the member functions.

Write a program to define a structure STUDENT and input the marks of n (<=20) students and for each student allot the stream.

5) Define a POINT class for two-dimensional points (x, y). Include constructors, a negate() function to transform the point into its negative, a norm() function to return the point's distance from the origin (0,0), and a print() function besides the functions to input and display the coordinates of the point. Use this class in a menu driven program to perform various operations on a point.

6) Write a program implement a class 'Complex' of complex numbers. The class should be include member functions to add and subtract two complex numbers.

7) Write a Program to implement a sphere class with appropriate members and member function to find the surface area and the volume. (Surface = 4 π r2 and Volume = 4/ 3 π r3).

8) Write a program to implement an Account Class with member functions to Compute Interest, Show Balance, Withdraw and Deposit amount from the Account.

Unit 3: More on class and object

3

Unit Structure

3.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Understand and use of access modifiers.
- Use of recursion in java program.
- Input data using command line argument.
- To manipulate the string using String and StringBuffer class
- Understand various types of inner class and its usage.

3.2 VISIBILITY CONTROL

Visibility control means controlling the access of java class, data members and methods of class, constructor, and variables. The visibility control can be implemented with the help of access modifier. The access modifier restrict the access of class, constructor, data members and methods of the class and variables in its scope. There are four access modifiers in java:

1. Default – no keyword specified
2. Private- using private keyword.
3. Protected- using protected keyword
4. Public- using public keyword.

Access modifier / Scope	Default	private	protected	public
In same class	Yes	Yes	Yes	Yes
In child class of same package	Yes	No	Yes	Yes
In other class of same package	Yes	No	Yes	Yes
In child class of other package	No	No	Yes	Yes
In other class of other packages	No	No	No	Yes

Table-8 Scope of aceess modifiers

3.3 PUBLIC ACCESS

The variable, class, and methods must be declared public using public access modifier. for this it uses the keyword public. This access specifier has the widest scope. The public class, methods or variables can be access from everywhere. There is no restriction for public data. The public class, method and variable can be access within class, sub class, class outside the package and class within the package.

```
class A
{
        public int a;
        public A() { a = 0; }
        public A(int x) { a = x; }
        public void printA()
        {
        System.out.println(“ a = “ + a);
        }
}
public class ExDefault      // if a class containing main method is public, we have to
                            //create that class name.java file for that program.
                            // Ex:  ExDefault.java for this program
{
        public static void main(String args[])
        {
        A a1 = new A(9);
        A1.printA();
        }
}
```

3.4 FRIENDLY ACCESS

It is also called Default access. When no access modifier used to declare any class, method or data member, it has friendly access. They can only be accessed within all classes of the same package i.e. the package in which the class is created.

For Example,

```
class A
{
        int a;
        A() { a = 0; }
        A(int x) { a = x; }
        void printA()
        {
        System.out.println(“ a = “ + a);
```

```
        }
        }
        class ExDefault
        {
        public static void main(String args[])
        {
        A a1 = new A(9);
        A1.printA();   //printA can be access other class in same package
        //similarly we can access data member a also.
        }
}
```

3.5 PROTECTED ACCESS

It uses protected keyword to assign protected access modifier. The methods and member variables of a class can be declared protected. It means they can be access within class, within package, and only subclass of the package and subclass of the outside package.

For Example,

```
class A
{
        protected int a;
        A() { a = 0; }
        A(int x) { a = x; }
        protected void printA()
        {
        System.out.println(" a = " + a);
        }
}
class B extends A
{
        B(){ super(); }
        B(int x) { super(x); }
        void printB()
        {
        printA(); //can access in subclass of A, as it is protected
        }
}
class ExDefault
{
public static void main(String args[])
{
A a1 = new A(9);
A1.printA();   //printA can be access other class in same package as it is protected
a1.a=10;   //we can access data member a here because it is protected.
```

```
}
}
```

3.6 PRIVATE ACCESS

The private keyword is used to declare private access modifier. The methods and member variables of class declared as private can be access within class only.

For Example,

```
class A
{
        private int a;
        A() { a = 0; }
        A(int x) { a = x; }
        void printA()
        {
        System.out.println(“ a = “ + a);
        }
}
class ExDefault
{
        public static void main(String args[])
        {
        A a1 = new A(9);
        A1.printA();    //printA can be access other class in same package
        a1.a=10;   //we can not access data member a here because it is private.
        }
}
```

3.7 RULE OF THUMB

- All important member variables of class should be declared private.
- The final variable should declare public.
- The methods can be declared private only when you want not others to access it.
- Private and protected access modifier should not be used for top level classes. They can be used for inner class declared in nested classes.

3.8 OBJECT AS PARAMETERS

In java, class can have member functions and constructors defined in it. We can pass various arguments to this function. The arguments can be various primitive data types, array or objects. We can pass object of any class as an argument to the function.

For example,

```
class A
{
        int a;
        A() { a = 0; }
        A(int x) { a = x; }
        void printA()
        {
        System.out.println(“ a  =  “  +  a);
        }
}
public class ExObjArg
{
        public static void main(String args[])
        {
        A a1 = new A(9);
        A a2 = new A(8);
        add(a1 , a2);
        }
        static void add(A a1, A a2) //function with objects as arguments
        {
        int sum = a1.a + a2.a;
        System.out.println(“ sum  =  “  +  sum);
        }
}
```

3.9 RETURNING OBJECT

A member function of a class can also return an object of any class. The return type of that function must be class type.

For example,

```
class A
{
        int a;
        A() { a = 0; }
        A(int x) { a = x; }
        void printA()
        {
```

```
        System.out.println(" a  =  "  +  a);
        }
}
public class ExObjArg
{
  public static void main(String args[])
  {
  A a1 = new A(9);
  A a2 = new A(8);
  A a3 = add(a1 , a2);
  a3.printA();
  }
  static A add(A a1, A a2)   //function with objects as arguments and returns object
  {
  A a3 = new A();
  a3.a = a1.a + a2.a;
  return a3;
  }
}
```

3.10 RECURSION

A function can call itself within its definition. Such function is called recursive function. Programming approach which solves problems using recursive function is called recursion.

For example,

```
long  factorial( int n)
{
        long fact = 1;
        if( n == 1 || n == 0)
         return fact;
        else
        fact = n * factorial(n-1);
        return fact;
}
```

Example

```
public class Factorial
{
        public static void main(String args[])
        {
        long f = factorial(5);
        System.out.println(f);
        }
        static long  factorial( int n)
        {
```

```
        long fact = 1;
        if( n == 1 || n == 0)
         return fact;
        else
        fact = n * factorial(n-1);

        return fact;
        }
}
```

3.11 NESTED AND INNER CLASS

In java, class can also be created within a class. This is called nested class. Here the class which holds class definition inside it is called outer class and the class inside the outer class is called inner class.

Nested class can be categorized into two. Non-static nested class and static nested class. In nonstatic nested class, the inner class is not static. In static nested class the inner class is declared as static.

NON STATIC NESTED CLASS

Non static nested class are also categorized into three types

1) Inner class
2) Method local inner class
3) Anonymous inner class

- **Inner class**

It is simple to create an inner class. We have to create a class definition inside the class. The inner class can be declared private or public. If it is declared private, we cannot access it outside the outer class. We have to use inner class within the outer class only. The public inner class can be access outside the outer class and the other class also.

For example (private inner class),

```
class OuterClass {
  int n;
  private class InnerClass {
    public void sayHello() {
      System.out.println("Hello, from inner class");
```

```
    }
  }
   void useInner() {
    InnerClass x = new InnerClass();
    x.sayHello();
  }
}
  public class ExInnerClass {
  public static void main(String args[]) {
    OuterClass a = new OuterClass();
    a.useInner();
  }
}
```

For example (public inner class),

In this example the inner class is used to get private member variable of the outer class.

```
class OuterClass {
  private int n = 50;
  public class InnerClass {
    public int getN() {
      return n;
    }
  }
}
public class ExInnerClass1 {
  public static void main(String args[]) {
    OuterClass x = new OuterClass();
    OuterClass.InnerClass y = x.new InnerClass();
    System.out.println(y.getN());
  }
}
```

- **Method local inner class**

In this type of inner class, we can create a class within a function. This class will be the local to the method. This class can be used only inside the method.

For example,

```
class OuterClass {
  void innerFun() {
    int n = 50;
    class InnerClass {                  //class inside the method innerFun()
      public void sayHello() {
        System.out.println("Hello from method inner class ");          }
    }
```

```
    InnerClass x = new InnerClass();
    x.sayHello();
  }
}
public class ExInnerClass2
{
  public static void main(String args[]) {
    OuterClass y = new OuterClass();
    y.innerFun();
  }
}
```

- **Anonymous inner class**

This type of inner class is use to declare without class name. They are declared and instantiate simultaneously. They are mainly used to override the abstract methods of class or interface.

Example,

```
abstract class InnerClass {
  public abstract void sayHello();
}
public class ExInnerClass3 {
  public static void main(String args[]) {
    InnerClass x = new InnerClass() {
      public void sayHello() {
        System.out.println("Hello from anonymous inner class");
      }
    };
    x.sayHello();
  }
}
```

STATIC NESTED CLASS

In static nested class, the inner class is declared; hence it is a static member of the outer class. To access this class, the object of outer class is not required. This static inner class can not access the member variables and methods of outer class.

For example,

```
class OuterClass
{
        static class InnerClass
```

```
        {
        void sayHello()
        {
        System.out.println(" Hello, this is inner class");
        }
        }
}
public class ExStNested
{
        public static void main(String args[])
        {
        OuterClass.InnerClass x=new OuterClass.InnerClass();
        x.sayHello();
        }
}
```

3.12 STRING CLASS

Strings are the sequence of characters. We can store name, address etc. as string. In java string is treated as an object of String class which is available in java.lang package. String class has various methods using which we can create and manipulate the strings.

To create a string in java program following syntax is used.

```
String s=”Hello”;
```

Here “Hello” is a string literal. For each string literal java compiler creates a String object. We can create a String object using new keyword and constructor. In java strings are non-mutable (non modifiable).

```
String s=new String(“Hello”);
```

We can also create a String from an array of characters.

```
char[] s1 = { 'h', 'e', 'l', 'l', 'o', '.' };
        String s = new String(s1);
```

Functions of String class

The following is the list of methods of String class

1) **char charAt(int index)** : this function returns the character at the index position

2) **int compareTo(String str)** : it compares a String with the other String we pass as an argument lexicographically and return difference of those two strings.

3) **int compareToIgnoreCase(String str)** : it compares strings , ignoring case.

4) **String concat(String str)** : it concatenate a string with the specified string passed as an argument.

5) **boolean contentEquals(StringBuffer sb)** : returns true if content of String and StringBuffer is same.

6) **static String copyValueOf(char[] data)** : returns a String having sequence of characters stored in an array.

7) **static String copyValueOf(char[] data, int offset, int count)**: returns a String having count number of characters stored in an array starting from offset.

8) **boolean endsWith(String suffix)** : returns true if String ends with specified String.

9) **boolean equals(String str)** :returns true is both String objects have same content.

10) **boolean equalsIgnoreCase(String anotherString)** : returns true if both String are equal ignoring case. Ex: Hello and hello are equal for this function.

11) **byte getBytes()** : returns a byte array containing String characters.

12) **int hashCode()** : returns a hashcode of the String

13) **int indexOf(int ch)** : returns position of character ch(first occurrence) in the String.

14) **int indexOf(int ch, int fromIndex)** : returns position of character ch in the String after fromIndex.

15) **int indexOf(String str)** : returns position of String str(first occurrence) in the String.

16) **int indexOf(String str, int fromIndex)** : returns position of String str in the String after fromIndex.

17) **int lastIndexOf(int ch)** :returns the position of last occurrence of ch in String.

18) **int lastIndexOf(int ch, int fromIndex)** : returns the position of last occurrence of ch in String. It searches in backward starting from the fromIndex.

19) **int lastIndexOf(String str)** : returns the position of last occurrence of str in String.

20) **int lastIndexOf(String str, int fromIndex)** : returns the position of last occurrence of str in String. It searches in backward starting from the fromIndex.

21) **int length()** :returns the length of the String.

22) **String replace(char oldChar, char newChar)** : returns a new String in which oldChar is replace with newChar in specified String.

23) **boolean startsWith(String prefix)** :returns true if String start with strings specified by prefix.

24) **String substring(int beginIndex)** :return a new String that is a substring start with beginIndex till the end.

25) **String substring(int beginIndex, int endIndex)** : return a new String that is a substring start with beginIndex till the endIndex.

26) **char[] toCharArray()** :converts a string into character array.

27) **String toLowerCase()** : returns a new String which is lower case conversion of specified String.

28) **String toUpperCase()** : returns a new String which is upper case conversion of specified String.

29) **String trim()** :returns a copy of String after removing starting and ending spaces.

30) **static String valueOf(primitive data type x)** :returns String conversion of primitive data value.

Example,

```
public class ExString {
    public static void main( String args [])
    {
    String s1 = "hello";
    String s2 = "whatsup";
    String s3 = new String( "Hello" );
    char[] s4  = { 'a' , 'b' };
    System.out.println( "charAt : " + s1.charAt(2));
    System.out.println( "compareTo s1 and s3: " + s1.compareTo(s3));
```

```
        System.out.println( "compareTo ignore case s1 and s3: " +
                                s1.compareToIgnoreCase(s3));
        System.out.println( "concat s1 and s2: " + s1.concat(s2));
        System.out.println( "copy value of: " + String.copyValueOf(s4));
        System.out.println( "ends with o: " + s1.endsWith("o"));
        System.out.println( "equal s1 and s3: " + s1.equals(s3));
        System.out.println( "euals ignore case s1 and s3: " + s1.equalsIgnoreCase(s3));
        byte[] b = s1.getBytes();
        System.out.println( "hash code: " + s1.hashCode());
        System.out.println( "indexOf: " + s1.indexOf('l'));
        System.out.println( "Last indexOf: " + s1.lastIndexOf('l'));
        System.out.println( "String length: " + s1.length());
        System.out.println( "replace: " + s1.replace('l','i'));
        System.out.println( "starts with : " + s1.endsWith("h"));
        System.out.println( "substring: " + s1.substring(3));
        char ar1 [] = s1.toCharArray();
        System.out.println(" Uppercase: " + s1.toUpperCase());
        System.out.println(" Lowercase: " + s1.toLowerCase());
        System.out.println(" valueOf: " + String.valueOf(123));

        }
}
```

3.13 STRINGBUFFER CLASS

StringBuffer is also a class of java.lang package. It is also used to create and manipulate the strings in java. The StringBuffer is used to create mutable strings.

We can create StringBuffer using following constructors,

StringBuffer() : creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str) : creates a string buffer with the specified string.
StringBuffer(int capacity) : creates an empty string buffer with the specified capacity as length.

1) **StringBuffer append(String s)**: is used to append the specified string with this string.
2) **StringBuffer insert(int offset, String s)**: is used to insert a string with this string at the specified position.
3) **StringBuffer replace(int startIndex, int endIndex, String str)**: is used to replace the string from specified startIndex and endIndex.

4) **StringBuffer delete(int startIndex, int endIndex)**: is used to delete the string from specified startIndex and endIndex.

5) **StringBuffer reverse():** is used to reverse the string.

6) **int capacity()**: is used to return the current capacity.

7) **char charAt(int index)**: is used to return the character at the specified position.

8) **int length()**: is used to return the length of the string i.e. total number of characters.

9) **String substring(int beginIndex)**: is used to return the substring from the specified beginIndex.

10) **String substring(int beginIndex, int endIndex)**: is used to return the substring from the specified beginIndex and endIndex.

Example,

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

        StringBuffer s1 = new StringBuffer("Hello");
        s1.append( " world" );
        System.out.println( s1 );
        s1.insert ( 1 , "!!!");
        System.out.println( s1 );
        s1.replace ( 2, 4, "***" );
        System.out.println( s1 );
        s1.delete( 2, 4);
        System.out.println( s1 );
        s1.reverse();
        System.out.println( s1 );
        System.out.println( s1.capacity() );
        System.out.println( s1.charAt(2) );
        System.out.println( s1.length() );
        System.out.println( s1.substring(2,4) );

        }
}
```

3.14 COMMAND LINE ARGUMENTS

Command line arguments are the arguments pass to java program when we run it. They are always in form of String. We can pass one or more string separated by space while running java program using java.exe. The java program accepts those

strings in a String array as a parameter of main method. These arguments passed from the console to main method and can be received in the java program. They can be used as an input.

For example,

In this example, the command line arguments stored inside the array args[] and can be used inside the program.

```
public class ExCmdArg{
        public static void main(String args[])
        {
        for( int j = 0; j < args.length ; j++)
         System.out.println(args[j]);
        }
}
```

Here, the strings “abc”, “xyz”, “hello” and “aryan” are command line arguments**.**

3.15 GENERIC IN JAVA

Java Generics were introduced in JDK 5.0 with the aim of reducing bugs and adding an extra layer of abstraction over types. Generics in Java is similar to templates in C++. The idea is to allow type (user defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

BaseType <Type> obj = new BaseType <Type>()

For example,

```
class Test<T>         // generic class
{
   T obj;
   Test(T obj) {  this.obj = obj;  }
   public T getObject()  { return this.obj; }
}
class ExGen
{
   public static void main (String[] args)
   {
```

```
        Test <Integer> Obj1 =  new Test<Integer>(15);
        System.out.println( Obj1.getObject() );

         Test <String> Obj2 =  new Test<String>( " Hello world " );
        System.out.println( Obj2.getObject() );
    }
}
```

3.16 LET US SUM UP

Access modifier: They are the key words which are use to restrict access of class member variables and methods.

public: This key word allows access of member functions and methods of class everywhere outside the class.

private: This key word allows access of member functions and methods of class only inside the class in which they are declared.

protected: This key word allows access of member functions and methods of class inside the class in which they are declared and in subclass of the class.

default: This key word allows access of member functions and methods of class only inside the classes of the package in which class resides.

recursion: It is a call of the function in itself**.**

nested class: We can create a class as a member of the class. This concept is called nested class.

outer class and inner class: In nested class, the class in which the member class is defined is called outer class. And the member class is called inner class.

String class: Strings are the non mutable sequence of characters.

StringBuffer class: They are mutable sequence of characters.

Command line arguments: They can be used to input in java program while running them on command prompt.

Generic: The idea of Generic is to allow type to be a parameter to methods, classes and interfaces like template of C++.

3.17 CHECK YOUR PROGRESS

- True-False with reason.

1. The recursion is calling a member function of a class into other member function.

2. Command line arguments can be used to give input to program.
3. Nested class is defining more than one class in same java program file.
4. We can not declare a class static.
5. Private member of the class can be accessed outside the class which is subclass.
6. Protected members of the class can be accessed inside the class in which they are declared.
7. Public members of a class can be accessed from everywhere.
8. For default access modifier the friendly keyword is used.
9. We can only pass strings as a command line arguments.
10. String is non mutable series of characters.

3.18 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

- True-False with reason

1. False. The recursion is calling a function of a class into the function itself.
2. True
3. False. Nested class is defining a class inside a class.
4. False. We can declare inner class of nested class static.
5. False. Private members of a class can be accessed inside a class only.
6. False. Protected members of the class can be accessed inside the class in which they are declared and inside the subclass.
7. True.
8. False. No keyword is used for default access.
9. True.
10. True.

3.19 FURTHER READING

1) "Java 2: The Complete Reference" by Herbert Schildt, McGraw Hill Publications.
2) "Effective Java" by Joshua Bloch, Pearson Education
3) Nested classes in Java | Core Java Tutorial' | Studytonight https://www.studytonight.com/ java/ nested-classes.php
4) What Is Recursion in Java Programming ? - dummies

5) https:// www.dummies.com/ programming/ java/ what-is-recursion-in-java-programming/

3.20 ASSIGNMENTS

- Write java program for following

1) Print Fibonacci series up to n elements using recursion.
2) Implement binary search using recursion.
3) Create a class Student with attributes roll number, name, address, phone numbers. To store address, create an Address class. An Address class can have PhoneNumbers inner class which holds all the phone numbers associated with an address and may have some extra functionality, like returning the best phone number (the most used one). All classes have get methods and print methods to input and print the data values respectively.
4) Find GCD of a number using recursion.
5) Create a class Customer with properties customer ID, name, address, phone number, date of birth and function to get and print these attributes. Inherit the class Account from Customer class with account number, account type, rate of interest and balance properties and functions to get and print these properties. Also in account class implement the deposit and withdraw function. Use appropriate access modifier with attributes and methods of each class.
6) Implement the above example considering customer as an abstract class which is inherited as Account class. Also inherit the Loan class from the Customer class which has properties like loan number, rate of interest, loan duration, loan amount, date of installment etc and methods to get and print value of these attributes. The Loan class also has method to pay installment. Use appropriate access modifier with attributes and methods of each class. After implementation of classes show their use in main method.
7) For educational institute design an application in which a Person with person id, name, address, department and phone number can be a faculty or a student. A faculty can have other attributes like degree, designation, specialization, experience etc. A student can have attributes like semester; results etc. create appropriate classes and methods in the classes. Use

appropriate access modifier with attributes and methods of each class. Also show their use in main method.

8) Implement a java program to input a String from command line argument and convert it into upper case without using toUpperCase function.

9) Implement a MyString class with your own reverse, getBytes and parseInt function in it. (Do not use readymade functions available in String class).

10) To input a paragraph from console and convert word at even position into upper case.

11) To input a paragraph from console and replace a word “is” with “are” in the input paragraph.

Block-3 Data Structure

Unit 1: ArrayList 1

Unit Structure

1.1 LEARNING OBJECTIVES

After studying this unit student should be able to:

- Define and use ArrayList
- Define and use Searching
- Define Sorting and
- Understand Insertion and Bubble Sort and
- Use Association List

1.2 INTRODUCTION

Computers get a lot of their power from working with data structures. A data structure is an organized collection of related data. An object is a data structure, but this type of data structure consisting of a fairly small number of named instance variables is just the beginning. In many cases, programmers build complicated data structures by hand, by linking objects together. In particular, we will look at the important topic of algorithms for searching and sorting an array.
An array has a fixed size that can't be changed after the array is created. But in many cases, it is useful to have a data structure that can grow and shrink as necessary. In this chapter, we will look at a standard class, ArrayList, that represents such a data structure.

1.3 ArrayList

Java has a feature called "parameterized types" that makes it possible to avoid the multitude of classes, and Java has a single class named ArrayList that implements the dynamic array pattern for all data types.

1.4 ArrayList and Parameterized Types

Java has a standard type with the rather odd name ArrayList<String> that represents dynamic arrays of Strings. Similarly, there is a type ArrayList<Button> that can be used to represent dynamic arrays of Buttons. And if Player is a class representing players in a game, then the type ArrayList<Player> can be used to represent a dynamic array of Players.

It might look like we still have a multitude of classes here, but in fact there is only one class, the ArrayList class, defined in the package java.util. But ArrayList is a parameterized type. A parameterized type can take a type parameter, so that from the single class ArrayList, we get a multitude of types including ArrayList<String>,

ArrayList<Button>, and in fact ArrayList<T> for any object type T. The type parameter T must be an object type such as a class name or an interface name. It cannot be a primitive type. This means that, unfortunately, you can not have an ArrayList of int or an ArrayList of char.

Consider the type ArrayList<String>. As a type, it can be used to declare variables, such as

```
ArrayList<String> namelist;
```

It can also be used as the type of a formal parameter in a subroutine definition, or as the return type of a subroutine. It can be used with the new operator to create objects:

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

The object created in this way is of type ArrayList<String> and represents a dynamic list of strings. It has instance methods such as namelist.add(str) for adding a String to the list, namelist.get(i) for getting the string at index i, and namelist.size() for getting the number of items currently in the list.

But we can also use ArrayList with other types. If Player is a class representing players in a game, we can create a list of players with

```
ArrayList<Player> playerList = new ArrayList<Player>();
```

Then to add a player, plr, to the game, we just have to say playerList.add(plr). And we can remove player number k with playerList.remove(k).

Furthermore if playerList is a local variable, then its declaration can be abbreviated (in Java 10 or later) to

```
var playlerList = new ArrayList<Player>();
```

using the alternative declaration syntax. The Java compiler uses the initial value that is assigned to playerList to deduce that its type is ArrayList<Player>.

When you use a type such as ArrayList<T>, the compiler will ensure that only objects of type T can be added to the list. An attempt to add an object that is not of type T will be a syntax error, and the program will not compile. However, note that objects belonging to a subclass of T can be added to the list, since objects belonging to a subclass of T are still considered to be of type T. Thus, for example, a variable of type ArrayList<Pane> can be used to hold objects of type BorderPane, TilePane, GridPane, or any other subclass of Pane. (Of course, this is the same way arrays work:

An object of type T[ ] can hold objects belonging to any subclass of T.) Similarly, if T is an interface, then any object that implements interface T can be added to the list.

An object of type ArrayList<T> has all of the instance methods that you would expect in a dynamic array implementation. Here are some of the most useful. Suppose that list is a variable of type ArrayList<T>. Then we have:

- list.size() — This function returns the current size of the list, that is, the number of items currently in the list. The only valid positions in the list are numbers in the range 0 to list.size()-1. Note that the size can be zero. A call to the default constructor new ArrayList<T>() creates a list of size zero.
- list.add(obj) — Adds an object onto the end of the list, increasing the size by 1. The parameter, obj, can refer to an object of type T, or it can be null.
- list.get(N) — This function returns the value stored at position N in the list. The return type of this function is T. N must be an integer in the range 0 to list.size()-1. If N is outside this range, an error of type IndexOutOfBoundsException occurs. Calling this function is similar to referring to A[N] for an array, A, except that you can't use list.get(N) on the left side of an assignment statement
- list.set(N, obj) — Assigns the object, obj, to position N in the ArrayList, replacing the item previously stored at position N. The parameter obj must be of type T. The integer N must be in the range from 0 to list.size()-1. A call to this function is equivalent to the command A[N] = obj for an array A.
- list.clear() — Removes all items from the list, setting its size to zero.
- list.remove(N) — For an integer, N, this removes the N-th item in the ArrayList. N must be in the range 0 to list.size()-1. Any items in the list that come after the removed item are moved up one position. The size of the list decreases by 1.
- list.remove(obj) — If the specified object occurs somewhere in the list, it is removed from the list. Any items in the list that come after the removed item are moved up one position. The size of the ArrayList decreases by 1. If obj occurs more than once in the list, only the first copy is removed. If obj does not occur in the list, nothing happens; this is not an error.
- list.indexOf(obj) — A function that searches for the object, obj, in the list. If the object is found in the list, then the first position number where it is found is returned. If the object is not found, then -1 is returned.

For the last two methods listed here, obj is compared to an item in the list by calling obj.equals(item), unless obj is null. This means, for example, that strings are tested for equality by checking the contents of the strings, not their location in memory.

Java comes with several parameterized classes representing different data structures. Those classes make up the Java Collection Framework. By the way, ArrayList can also be used as a non-parametrized type. This means that you can declare variables and create objects of type ArrayList such as

```
ArrayList list = new ArrayList();
```

The effect of this is similar to declaring list to be of type ArrayList<Object>. That is, list can hold any object that belongs to a subclass of Object. Since every class is a subclass of Object, this means that any object can be stored in list.

1.5 Programming With ArrayList

As a simple first example, we can redo ReverseWithDynamicArray.java, from the previous section, using an ArrayList instead of a custom dynamic array class. In this case, we want to store integers in the list, so we should use ArrayList<Integer>. Here is the complete program:

```
import java.util.Scanner;
import java.util.ArrayList;
/**
* Reads a list of non-zero numbers from the user, then prints
* out the input numbers in the reverse of the order in which
* the were entered. There is no limit on the number of inputs.
*/
public class ReverseWithArrayList {
      public static void main(String[] args) {
            ArrayList<Integer> list;
            list = new ArrayList<Integer>();
            Scanner in = new Scanner(System.in);
            System.out.println("Enter some integers. Enter 0 to end.");
            while (true) {
                  System.out.print("? ");
                  int number = in.nextInt();
                  if (number == 0)
                        break;
                  list.add(number);
            }
            System.out.println();
            System.out.println("Your numbers in reverse are:");
            for (int i = list.size() - 1; i >= 0; i--) {
                  System.out.printf("%10d%n", list.get(i));
            }
      }
}
```

As illustrated in this example, ArrayLists are commonly processed using for loops, in much the same way that arrays are processed. for example, the following loop prints out all the items for a variable namelist of type ArrayList<String>:

```
for (int i = 0; i < namelist.size(); i++ ) {
      String item = namelist.get(i);
      System.out.println(item);
```

```
}
```

You can also use for-each loops with ArrayLists, so this example could also be written

```
for (String item : namelist ) {
      System.out.println(item);
}
```

When working with wrapper classes, the loop control variable in the for-each loop can be a primitive type variable. This works because of unboxing. For example, if numbers is of type ArrayList<Double>, then the following loop can be used to add up all the values in the list:

```
double sum = 0;
for ( double num : numbers ) {
      sum = sum + num;
}
```

This will work as long as none of the items in the list are null. If there is a possibility of null values, then you will want to use a loop control variable of type Double and test for nulls. For example, to add up all the non-null values in the list:

```
double sum;
for ( Double num : numbers ) {
      if ( num != null ) {
            sum=sum + num; // Here, num is SAFELY unboxed to get a double.
      }
}
```

1.6 Searching

There is an obvious algorithm for searching for a particular item in an array: Look at each item in the array in turn, and check whether that item is the one you are looking for. If so, the search is finished. If you look at every item without finding the one you want, then you can be sure that the item is not in the array. It's easy to write a subroutine to implement this algorithm. Let's say the array that you want to search is an array of ints. Here is a method that will search the array for a specified integer. If the integer is found, the method returns the index of the location in the array where it is found. If the integer is not in the array, the method returns the value -1 as a signal that the integer could not be found:

```
/**
* Searches the array A for the integer N. If N is not in the array,
* then -1 is returned. If N is in the array, then the return value is
* the first integer i that satisfies A[i] == N.
*/
```

```
static int find(int[] A, int N) {
      for (int index = 0; index < A.length; index++) {
            if ( A[index] == N )
                  return index; // N has been found at this index!
      }
      // If we get this far, then N has not been found
      // anywhere in the array. Return a value of -1.
      return -1;
}
```

This method of searching an array by looking at each item in turn is called linear search. If nothing is known about the order of the items in the array, then there is really no better alternative algorithm. But if the elements in the array are known to be in increasing or decreasing order, then a much faster search algorithm can be used. An array in which the elements are in order is said to be sorted. Of course, it takes some work to sort an array, but if the array is to be searched many times, then the work done in sorting it can really pay off.

Binary search is a method for searching for a given item in a sorted array. Although the implementation is not trivial, the basic idea is simple: If you are searching for an item in a sorted list, then it is possible to eliminate half of the items in the list by inspecting a single item. For example, suppose that you are looking for the number 42 in a sorted array of 1000 integers. Let's assume that the array is sorted into increasing order. Suppose you check item number 500 in the array, and find that the item is 93. Since 42 is less than 93, and since the elements in the array are in increasing order, we can conclude that if 42 occurs in the array at all, then it must occur somewhere before location 500. All the locations numbered 500 or above contain values that are greater than or equal to 93. These locations can be eliminated as possible locations of the number 42.

The next obvious step is to check location 250. If the number at that location is, say, -21, then you can eliminate locations before 250 and limit further search to locations between 251 and 499. The next test will limit the search to about 125 locations, and the one after that to about 62. After just 10 steps, there is only one location left. This is a whole lot better than looking through every element in the array. If there were a million items, it would still take only 20 steps for binary search to search the array! (Mathematically, the number of steps is approximately equal to the logarithm, in the base 2, of the number of items in the array.)

In order to make binary search into a Java subroutine that searches an array A for an item N, we just have to keep track of the range of locations that could possibly contain N. At each step, as we eliminate possibilities, we reduce the size of this range. The basic operation is to look at the item in the middle of the range. If this item is greater than N, then the second half of the range can be eliminated. If it is less than N, then the first half of the range can be eliminated. If the number in the middle just

happens to be N exactly, then the search is finished. If the size of the range decreases to zero, then the number N does not occur in the array. Here is a subroutine that implements this idea:

```
/**
* Searches the array A for the integer N.
* Precondition: A must be sorted into increasing order.
* Postcondition: If N is in the array, then the return value, i,
* satisfies A[i] == N. If N is not in the array, then the
* return value is -1.
*/
static int binarySearch(int[] A, int N) {
      int lowestPossibleLoc = 0;
      int highestPossibleLoc = A.length - 1;
      while (highestPossibleLoc >= lowestPossibleLoc) {
            int middle = (lowestPossibleLoc + highestPossibleLoc) / 2;
            if (A[middle] == N) {
                  // N has been found at this index!
                  return middle;
            }
            else if (A[middle] > N) {
                  // eliminate locations >= middle
                  highestPossibleLoc = middle - 1;
            }
            else {
                  // eliminate locations <= middle
                  lowestPossibleLoc = middle + 1;
            }
      }
      // At this point, highestPossibleLoc < LowestPossibleLoc,
      // which means that N is known to be not in the array. Return
      // a -1 to indicate that N could not be found in the array.
      return -1;
}
```

1.7 Association Lists

One particularly common application of searching is with association lists. The standard example of an association list is a dictionary. A dictionary associates definitions with words. Given a word, you can use the dictionary to look up its definition. We can think of the dictionary as being a list of pairs of the form (w,d), where w is a word and d is its definition. A general association list is a list of pairs (k,v), where k is some “key” value, and v is a value associated to that key. In general, we want to assume that no two pairs in the list have the same key. There are two basic operations on association lists: Given a key, k, find the value v associated with k, if any. And given a key, k, and a value v, add the pair (k,v) to the association list (replacing the pair, if any, that had the same key value). The two operations are usually called get and put.

Association lists are very widely used in computer science. For example, a compiler has to keep track of the location in memory associated with each variable. It

can do this with an association list in which each key is a variable name and the associated value is the address of that variable in memory. Another example would be a mailing list, if we think of it as associating an address to each name on the list. As a related example, consider a phone directory that associates a phone number to each name. We'll look at a highly simplified version of this example. (This is not meant to be a realistic way to implement a phone directory!)

The items in the phone directory's association list could be objects belonging to the class:

```
class PhoneEntry {
      String name;
      String phoneNum;
}
```

The data for a phone directory consists of an array of type PhoneEntry[ ] and an integer variable to keep track of how many entries are actually stored in the directory. The technique of dynamic arrays can be used in order to avoid putting an arbitrary limit on the number of entries that the phone directory can hold. Using an ArrayList would be another possibility. A PhoneDirectory class should include instance methods that implement the "get" and "put" operations. Here is one possible simple definition of the class:

```
/*A PhoneDirectory holds a list of names with a phone number for
  each name. It is possible to find the number associated with
  a given name, and to specify the phone number for a given name. */

public class PhoneDirectory {

      // An object of type PhoneEntry holds one name/number pair.
      private static class PhoneEntry {
            String name; // The name.
            String number; // The associated phone number.
      }

      private PhoneEntry[] data; // Array that holds the name/number pairs.
      private int dataCount; // The number of pairs stored in the array.

      // Constructor creates an initially empty directory.
      public PhoneDirectory() {
            data = new PhoneEntry[1];
            dataCount = 0;
      }
      /* Looks for a name/number pair with a given name. If found, the index
      of the pair in the data array is returned. If no pair contains the
      given name, then the return value is -1. This private method is
      used internally in getNumber() and putNumber(). */

      private int find (String name) {
```

```
            for (int i = 0; i < dataCount; i++) {
                  if (data[i].name.equals(name))
                        return i; // The name has been found in position i.
            }
            return -1; // The name does not exist in the array.
      }

      /*Finds the phone number, if any, for a given name. @return The phone
      number associated with the name; if the name does not occur in the
      phone directory, then the return value is null. */

      public String getNumber( String name ) {
            int position = find(name);
            if (position == -1)
                  return null; // no phone entry for the given name.
            else
                  return data[position].number;
      }
      /* Associates a given name with a given phone number. If the name
      already exists in the phone directory, then the new number replaces
      the old one. Otherwise, a new name/number pair is added. The name and
      number should both be non-null. An IllegalArgumentException is thrown
      if this is not the case. */

      public void putNumber( String name, String number ) {
            if (name == null || number == null)
                  throw new IllegalArgumentException("name and number cannot
            be null");
            int i = find(name);
            if (i >= 0) {
                  // The name already exists, in position i in the array.
                  // Just replace the old number at that pos. with the new.
                  data[i].number = number;
            }
            else {
            // Add a new name/number pair to the array. If the array is
            // already full, first create a new, larger array.
            if (dataCount == data.length) {
                  data = Arrays.copyOf( data, 2*data.length );
            }
            PhoneEntry newEntry = new PhoneEntry(); // Create a new pair.
            newEntry.name = name;
            newEntry.number = number;
            data[dataCount] = newEntry; // Add the new pair to the array.
            dataCount++;
            }
      }
} // end class PhoneDirectory
```

The class defines a private instance method, find(), that uses linear search to find the position of a given name in the array of name/number pairs. The find() method is used both in the getNumber() method and in the putNumber() method. Note in

particular that putNumber(name,number) has to check whether the name is in the phone directory. If so, it just changes the number in the existing entry; if not, it has to create a new phone entry and add it to the array.

This class could use a lot of improvement. For one thing, it would be nice to use binary search instead of simple linear search in the getNumber method. However, we could only do that if the list of PhoneEntries were sorted into alphabetical order according to name. In fact, it's really not all that hard to keep the list of entries in sorted order, as you'll see in the next subsection.

1.8 Insertion Sort

We've seen that there are good reasons for sorting arrays. There are many algorithms available for doing so. One of the easiest to understand is the insertion sort algorithm. This technique is also applicable to the problem of keeping a list in sorted order as you add new items to the list. Let's consider that case first:

Suppose you have a sorted list and you want to add an item to that list. If you want to make sure that the modified list is still sorted, then the item must be inserted into the right location, with all the smaller items coming before it and all the bigger items after it. This will mean moving each of the bigger items up one space to make room for the new item.

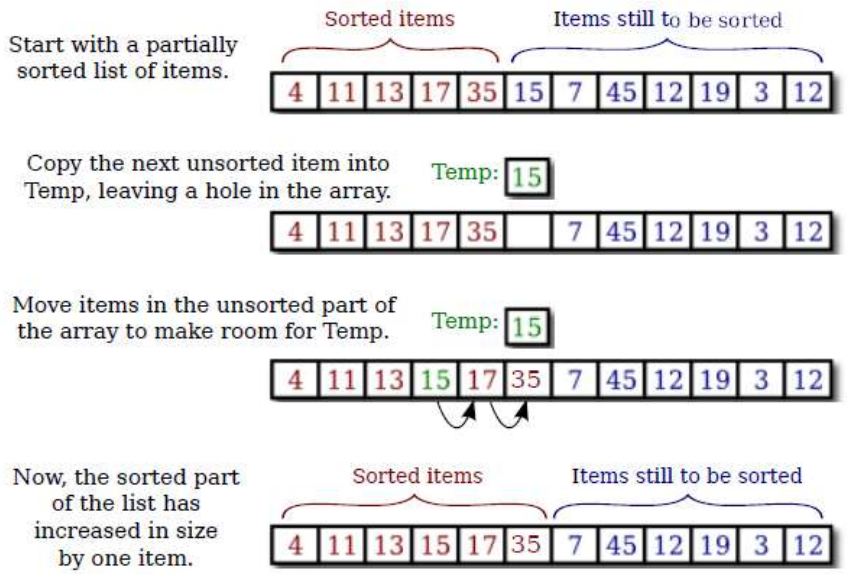
```
/* Precondition: itemsInArray is the number of items that are stored in A.
These items must be in increasing order (A[0] <= A[1] <= ... <=
A[itemsInArray-1]). The array size is at least one greater than itemsInArray.
Postcondition: The number of items has increased by one, newItem has been
added to the array, and all the items in the array are still in increasing
order. Note: To complete the process of inserting an item in the array, the
variable that counts the number of items in the array must be incremented,
after calling this subroutine. */

static void insert(int[] A, int itemsInArray, int newItem) {
      int loc = itemsInArray - 1; // Start at the end of the array.
      /* Move items bigger than newItem up one space;
      Stop when a smaller item is encountered or when the
      beginning of the array (loc == 0) is reached. */
      while (loc >= 0 && A[loc] > newItem) {
            A[loc + 1] = A[loc]; // Bump item from A[loc] up to loc+1.
            loc = loc - 1; // Go on to next location.
      }
      A[loc + 1] = newItem; // Put newItem in last vacated space.
}
```

Conceptually, this could be extended to a sorting method if we were to take all the items out of an unsorted array, and then insert them back into the array one-by-one, keeping the list in sorted order as we do so. Each insertion can be done using the insert routine given above.

In the actual algorithm, we don't really take all the items from the array; we just remember what part of the array has been sorted:

```
static void insertionSort(int[] A) {
      // Sort the array A into increasing order.
      int itemsSorted; // Number of items that have been sorted so far.
      for (itemsSorted = 1; itemsSorted < A.length; itemsSorted++) {
            // Assume that items A[0], A[1], ... A[itemsSorted-1]
            // have already been sorted. Insert A[itemsSorted]
            // into the sorted part of the list.
            int temp = A[itemsSorted]; // The item to be inserted.
            int loc = itemsSorted - 1; // Start at end of list.
            while (loc >= 0 && A[loc] > temp) {
                  A[loc + 1] = A[loc]; // Bump item A[loc] up to loc+1.
                  loc = loc - 1; // Go on to next location.
            }
            A[loc + 1] = temp; // Put temp in last vacated space.
      }
}
```

Here is an illustration of one stage in insertion sort. It shows what happens during one execution of the for loop in the above method, when itemsSorted is 5:



1.9 Selection Sort

Another typical sorting method uses the idea of finding the biggest item in the list and moving it to the end—which is where it belongs if the list is to be in increasing order. Once the biggest item is in its correct location, you can then apply the same idea to the remaining items. That is, find the next-biggest item, and move it into the next-to-last space, and so forth. This algorithm is called selection sort . It's easy to write:

```
static void selectionSort(int[] A) {
      // Sort A into increasing order, using selection sort
      for (int lastPlace = A.length-1; lastPlace > 0; lastPlace--) {
      // Find the largest item among A[0], A[1], ...,
            // A[lastPlace], and move it into position lastPlace
            // by swapping it with the number that is currently
            // in position lastPlace.
            int maxLoc = 0; // Location of largest item seen so far.
            for (int j = 1; j <= lastPlace; j++) {
                  if (A[j] > A[maxLoc]) {
                        // Since A[j] is bigger than the maximum we've seen
                        // so far, j is the new location of the max value
                        // we've seen so far.
                        maxLoc = j;
                  }
            }
            int temp = A[maxLoc]; // Swap largest item with A[lastPlace].
            A[maxLoc] = A[lastPlace];
            A[lastPlace] = temp;
      } // end of for loop
}
```

Insertion sort and selection sort are suitable for sorting fairly small arrays (up to a few hundred elements, say). There are more complicated sorting algorithms that are much faster than insertion sort and selection sort for large arrays, to the same degree that binary search is faster than linear search. The standard method Arrays.sort uses these fast-sorting algorithms.

1.10 Let Us Sum Up

In this chapter, we have looked at a standard class, ArrayList, that represents such a data structure. In particular, we have looked at the important topic of algorithms for searching and sorting an array. We have discussed linear search, binary search, insertion sort and selection sort.

1.11 Further Reading

- Java Collections Framework Tutorials - BeginnersBook.com https://beginnersbook.com/java-collections-tutorials/
- Java Collections Framework | Collections in Java With Examples https://www.edureka.co/blog/java-collections/
- "Java 2: The Complete Reference" by Herbert Schildt, McGraw Hill Publications.
- "Effective Java" by Joshua Bloch, Pearson Education.

1.12 Assignments

1. What is meant by the basetype of an array?
2. What does it mean to sort an array?
3. What is the main advantage of binary search over linear search? What is the main disadvantage?
4. What does it mean to say that ArrayList is a parameterized type?
5. Suppose that a variable strlst has been declared as
   ArrayList<String> strlst = new ArrayList<String>();
   Assume that the list is not empty and that all the items in the list are non-null. Write a code segment that will find and print the string in the list that comes first in lexicographic order.
6. Write a complete static method that finds the largest value in an array of ints. The method should have one parameter, which is an array of type int[]. The largest number in the array should be returned as the value of the method.

Unit 2: Linked Data Structures

2

Unit Structure

2.1 Learning Objectives

After studying this unit learner should be able to

- Define Linked List
- List various operations on Linked List
- Insert into a Linked List
- Delete element from a Linked List

2.2 Introduction

Every useful object contains instance variables. When the type of an instance variable is given by a class or interface name, the variable can hold a reference to another object. Such a reference is also called a pointer, and we say that the variable points to the object. (Of course, any variable that can contain a reference to an object can also contain the special value null, which points to nowhere.) When one object contains an instance variable that points to another object, we think of the objects as being “linked” by the pointer. Data structures of great complexity can be constructed by linking objects together.

2.3 Recursive Linking

Something interesting happens when an object contains an instance variable that can refer to another object of the same type. In that case, the definition of the object’s class is recursive. Such recursion arises naturally in many cases. For example, consider a class designed to represent employees at a company. Suppose that every employee except the boss has a supervisor, who is another employee of the company. Then the Employee class would naturally contain an instance variable of type Employee that points to the employee’s supervisor:

```
/* An object of type Employee holds data about one employee. */

public class Employee {
      String name; // Name of the employee.
      Employee supervisor; // The employee’s supervisor.
      .
      . // (Other instance variables and methods.)
      .
} // end class Employee
```

If emp is a variable of type Employee, then emp.supervisor is another variable of type Employee. If emp refers to the boss, then the value of emp.supervisor should be null to indicate the fact that the boss has no supervisor. If we wanted to print out the name of the employee’s supervisor, for example, we could use the following Java statement:

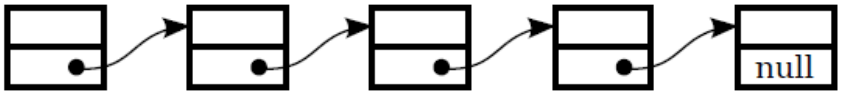
```
if ( emp.supervisor == null) {
      System.out.println(emp.name + " is the boss and has no supervisor!" );
}
else {
      System.out.print( "The supervisor of " + emp.name + " is " );
      System.out.println( emp.supervisor.name );
}
```

Now, suppose that we want to know how many levels of supervisors there are between a given employee and the boss. We just have to follow the chain of command through a series of supervisor links, and count how many steps it takes to get to the boss:

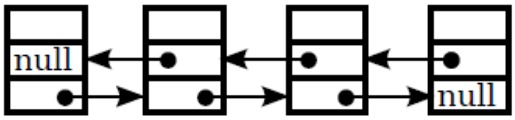
```
if ( emp.supervisor == null ) {
      System.out.println( emp.name + " is the boss!" );
}
else {
      Employee runner; // For "running" up the chain of command.
      runner = emp.supervisor;
      if ( runner.supervisor == null) {
            System.out.println(emp.name + " reports directly to the boss.");
      }
      else {
            int count = 0;
            while ( runner.supervisor != null ) {
                  count++; // Count the supervisor on this level.
                  runner = runner.supervisor; // Move up to the next level.
            }
            System.out.println( "There are " + count
            + " supervisors between " + emp.name + " and the boss.");
      }
}
```

As the while loop is executed, runner points in turn to the original employee (emp), then to emp's supervisor, then to the supervisor of emp's supervisor, and so on. The count variable is incremented each time runner "visits" a new employee. The loop ends when runner.supervisor is null, which indicates that runner has reached the boss. At that point, count has counted the number of steps between emp and the boss.

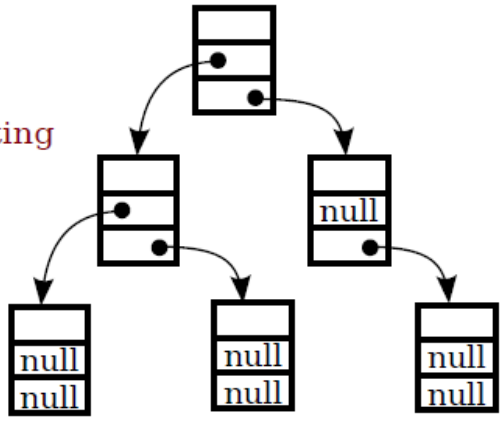
In this example, the supervisor variable is quite natural and useful. In fact, data structures that are built by linking objects together are so useful that they are a major topic of study in computer science. We'll be looking at a few typical examples. In this section and the next, we'll be looking at linked lists. A linked list consists of a chain of objects of the same type, linked together by pointers from one object to the next. This is much like the chain of supervisors between emp and the boss in the above example. It's also possible to have more complex situations, in which one object can contain links to several other objects.



When an object contains a reference to an object of the same type, then several objects can be linked together into a list. Each object refers to the next object.



Things get even more interesting when an object contains two references to objects of the same type. In that case, more complicated data structures can be constructed.



2.4 Linked Lists

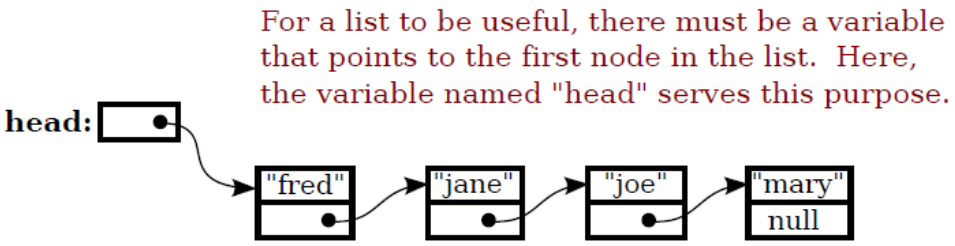
For most of the examples in the rest of this section, linked lists will be constructed out of objects belonging to the class Node which is defined as follows:

```
class Node {
      String item;
      Node next;
}
```

The term node is often used to refer to one of the objects in a linked data structure. Objects of type Node can be chained together as shown in the top part of the above illustration. Each node holds a String and a pointer to the next node in the list (if any). The last node in such a list can always be identified by the fact that the instance variable next in the last node holds the value null instead of a pointer to another node. The purpose of the chain of nodes is to represent a list of strings. The first string in the list is stored in the first node, the second string is stored in the second node, and so on. The pointers and the node objects are used to build the structure, but the data that we want to represent is the list of strings. Of course, we could just as easily represent a list of integers or a list of Buttons or a list of any other type of data by changing the type of the item that is stored in each node.

Although the Nodes in this example are very simple, we can use them to illustrate the common operations on linked lists. Typical operations include deleting nodes from the list, inserting new nodes into the list, and searching for a specified String among the items in the list. We will look at subroutines to perform all of these operations, among others.

For a linked list to be used in a program, that program needs a variable that refers to the first node in the list. It only needs a pointer to the first node since all the other nodes in the list can be accessed by starting at the first node and following links along the list from one node to the next. In my examples, I will always use a variable named head, of type Node, that points to the first node in the linked list. When the list is empty, the value of head is null.



2.5 Basic Linked List Processing

It is very common to want to process all the items in a linked list in some way. The common pattern is to start at the head of the list, then move from each node to the next by following the pointer in the node, stopping when the null that marks the end of the list is reached. If head is a variable of type Node that points to the first node in the list, then the general form of the code for processing all the items in a linked list is:

```
Node runner; // A pointer that will be used to traverse the list.
runner = head; // Start with runner pointing to the head of the list.
while ( runner != null ) { // Continue until null is encountered.
  process( runner.item ); // Do something with the item in the current node.
  runner = runner.next; // Move on to the next node in the list.
}
```

Our only access to the list is through the variable head, so we start by getting a copy of the value in head with the assignment statement runner = head. We need a copy of head because we are going to change the value of runner. We can't change the value of head, or we would lose our only access to the list! The variable runner will point to each node of the list in turn. When runner points to one of the nodes in the list, runner.next is a pointer to the next node in the list, so the assignment statement runner = runner.next moves the pointer along the list from each node to the next. We know that we've reached the end of the list when runner becomes equal to null. Note that

our list-processing code works even for an empty list, since for an empty list the value of head is null and the body of the while loop is not executed at all. As an example, we can print all the strings in a list of Strings by saying:

```
Node runner = head;
while ( runner != null ) {
      System.out.println( runner.item );
      runner = runner.next;
}
```

The while loop can, by the way, be rewritten as a for loop. Remember that even though the loop control variable in a for loop is often numerical, that is not a requirement. Here is a for loop that is equivalent to the above while loop:

```
for ( Node runner = head; runner != null; runner = runner.next ) {
      System.out.println( runner.item );
}
```

Similarly, we can traverse a list of integers to add up all the numbers in the list. A linked list of integers can be constructed using the class

```
public class IntNode {
      int item; // One of the integers in the list.
      IntNode next; // Pointer to the next node in the list.
}
```

If head is a variable of type IntNode that points to a linked list of integers, we can find the sum of the integers in the list using:

```
int sum = 0;
IntNode runner = head;
while ( runner != null ) {
      sum = sum + runner.item; // Add current item to the sum.
      runner = runner.next;
}
System.out.println("The sum of the list of items is " + sum);
```

It is also possible to use recursion to process a linked list. Recursion is rarely the natural way to process a list, since it’s so easy to use a loop to traverse the list. However, understanding how to apply recursion to lists can help with understanding the recursive processing of more complex data structures. A non-empty linked list can be thought of as consisting of two parts: the head of the list, which is just the first node in the list, and the tail of the list, which consists of the remainder of the list after the head. Note that the tail is itself a linked list and that it is shorter than the original list (by one node). This is a natural setup for recursion, where the problem of processing a list can be divided into processing the head and recursively processing the tail. The

base case occurs in the case of an empty list (or sometimes in the case of a list of length one). For example, here is a recursive algorithm for adding up the numbers in a linked list of integers:

```
if the list is empty then
      return 0 (since there are no numbers to be added up)
otherwise
      let listsum = the number in the head node
      let tailsum be the sum of the numbers in the tail list (recursively)
      add tailsum to listsum
      return listsum
```

One remaining question is, how do we get the tail of a non-empty linked list? If head is a variable that points to the head node of the list, then head.next is a variable that points to the second node of the list—and that node is in fact the first node of the tail. So, we can view head.next as a pointer to the tail of the list. One special case is when the original list consists of a single node. In that case, the tail of the list is empty, and head.next is null. Since an empty list is represented by a null pointer, head.next represents the tail of the list even in this special case. This allows us to write a recursive list-summing function in Java as

```
/* Compute the sum of all the integers in a linked list of integers. @param
head a pointer to the first node in the linked list */

public static int addItemsInList( IntNode head ) {
if ( head == null ) {
      // Base case: The list is empty, so the sum is zero.
      return 0;
}
else {
      // Recursive case: The list is non-empty. Find the sum of
      // the tail list, and add that to the item in the head node.
      // (Note that this case could be written simply as
      // return head.item + addItemsInList( head.next );)
      int listsum = head.item;
      int tailsum = addItemsInList( head.next );
      listsum = listsum + tailsum;
      return listsum;
      }
}
```

I will finish by presenting a list-processing problem that is easy to solve with recursion, but quite tricky to solve without it. The problem is to print out all the strings in a linked list of strings in the reverse of the order in which they occur in the list. Note that when we do this, the item in the head of a list is printed out after all the items in the tail of the list. This leads to the following recursive routine. You should convince yourself that it works, and you should think about trying to do the same thing without using recursion:

```
public static void printReversed( Node head ) {
      if ( head == null ) {
```

```
        // Base case: The list is empty, and there is nothing to print.
        return;
        }
        else {
              // Recursive case: The list is non-empty.
              printReversed( head.next );
              // Print strings from tail, in reverse order.
              System.out.println( head.item );
              // Then print string from head node.
        }
}
```

In the rest of this section, we'll look at a few more advanced operations on a linked list of strings. The subroutines that we consider are instance methods in a class that I wrote named StringList. An object of type StringList represents a linked list of strings. The class has a private instance variable named head of type Node that points to the first node in the list, or is null if the list is empty. Instance methods in class StringList access head as a global variable. The source code for StringList is in the file StringList.java, and it is used in a sample program named ListDemo.java, so you can take a look at the code in context if you want.

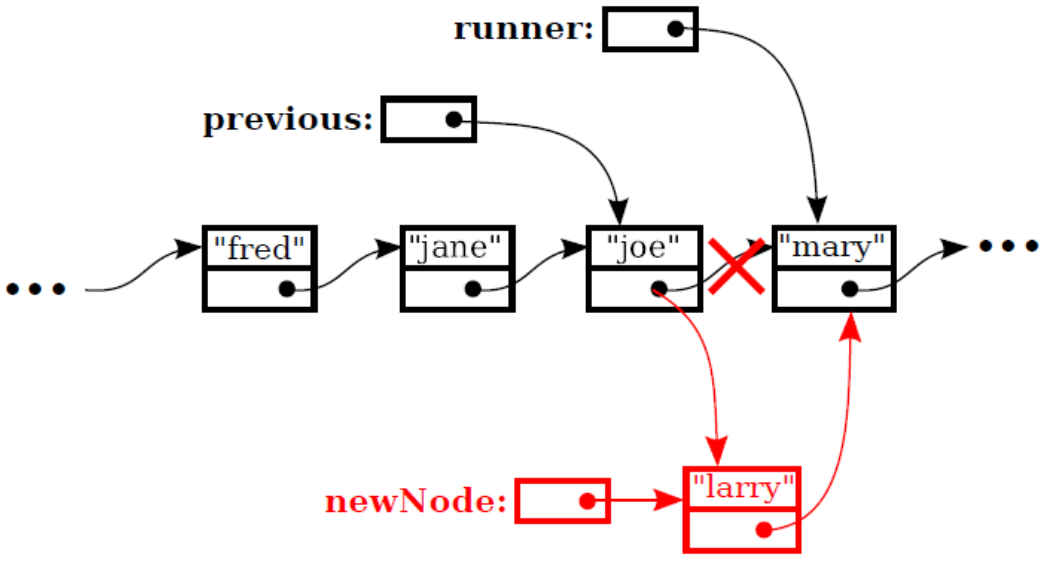
One of the methods in the StringList class searches the list, looking for a specified string. If the string that we are looking for is searchItem, then we have to compare searchItem to each item in the list. This is an example of basic list traversal and processing. However, in this case, we can stop processing if we find the item that we are looking for.

```
/**
* Searches the list for a specified item.
* @param searchItem the item that is to be searched for
* @return true if searchItem is one of the items in the list or false if
* searchItem does not occur in the list.
*/

public boolean find(String searchItem) {
      Node runner; // A pointer for traversing the list.
      runner = head; // Start by looking at the head of the list.
      // (head is an instance variable!)
      while (runner != null ) {
            // Go through the list looking at the string in each
            // node. If the string is the one we are looking for,
            // return true, since the string has been found in the list.
            if (runner.item.equals(searchItem))
                  return true;
            runner = runner.next; // Move on to the next node.
      }
      // At this point, we have looked at all the items in the list
      // without finding searchItem. Return false to indicate that
      // the item does not exist in the list.
      return false;
} // end find ()
```

It is possible that the list is empty, that is, that the value of head is null. We should be careful that this case is handled properly. In the above code, if head is null, then the body of the while loop is never executed at all, so no nodes are processed and the return value is false. This is exactly what we want when the list is empty, since the searchItem can't occur in an empty list.

2.6 Inserting into a Linked List

The problem of inserting a new item into a linked list is more difficult, at least in the case where the item is inserted into the middle of the list. (In fact, it's probably the most difficult operation on linked data structures that you'll encounter in this chapter.) In the StringList class, the items in the nodes of the linked list are kept in increasing order. When a new item is inserted into the list, it must be inserted at the correct position according to this ordering. This means that, usually, we will have to insert the new item somewhere in the middle of the list, between two existing nodes. To do this, it's convenient to have two variables of type Node, which refer to the existing nodes that will lie on either side of the new node. In the following illustration, these variables are previous and runner. Another variable, newNode, refers to the new node. In order to do the insertion, the link from previous to runner must be "broken," and new links from previous to newNode and from newNode to runner must be added:



Inserting a new node
into the middle of a list.

Once we have previous and runner pointing to the right nodes, the command "previous.next = newNode;" can be used to make previous.next point to the new node.

And the command “newNode.next = runner” will set newNode.next to point to the correct place. However, before we can use these commands, we need to set up runner and previous as shown in the illustration. The idea is to start at the first node of the list, and then move along the list past all the items that are less than the new item. While doing this, we have to be aware of the danger of “falling off the end of the list.” That is, we can’t continue if runner reaches the end of the list and becomes null. If insertItem is the item that is to be inserted, and if we assume that it does, in fact, belong somewhere in the middle of the list, then the following code would correctly position previous and runner:

```
Node runner, previous;
previous = head; // Start at the beginning of the list.
runner = head.next;
while ( runner != null && runner.item.compareTo(insertItem) < 0 ) {
      previous = runner; // "previous = previous.next" would also work
      runner = runner.next;
}
```

This is fine, except that the assumption that the new node is inserted into the middle of the list is not always valid. It might be that insertItem is less than the first item of the list. In that case, the new node must be inserted at the head of the list. This can be done with the instructions

```
newNode.next = head; // Make newNode.next point to the old head.
head = newNode; // Make newNode the new head of the list.
```

It is also possible that the list is empty. In that case, newNode will become the first and only node in the list. This can be accomplished simply by setting head = newNode. The following insert() method from the StringList class covers all of these possibilities:

```
/*Insert a specified item into the list, keeping the list in order.
* @param insertItem the item that is to be inserted.*/

public void insert(String insertItem) {
      Node newNode; // A Node to contain the new item.
      newNode = new Node();
      newNode.item = insertItem; // (N.B. newNode.next is null.)
      if ( head == null ) {
            // The new item is the first (and only) one in the list.
            // Set head to point to it.
            head = newNode;
      }
      else if ( head.item.compareTo(insertItem) >= 0 ) {
            // The new item is less than the first item in the list,
            // so it has to be inserted at the head of the list.
      newNode.next = head;
      head = newNode;
      }
      else {
```

```
        // The new item belongs somewhere after the first item
        // in the list. Search for its proper position and insert it.
        Node runner; // A node for traversing the list.
        Node previous; // Always points to the node preceding runner.
        runner = head.next; // Start by looking at the SECOND position.
        previous = head;
        while ( runner != null && runner.item.compareTo(insertItem) < 0 ) {
              // Move previous and runner along the list until runner
              // falls off the end or hits a list element that is
              // greater than or equal to insertItem. When this
              // loop ends, previous indicates the position where
              // insertItem must be inserted.
              previous = runner;
              runner = runner.next;
        }
        newNode.next = runner; // Insert newNode after previous.
        previous.next = newNode;
        }
} // end insert()
```

If you were paying close attention to the above discussion, you might have noticed that there is one special case which is not mentioned. What happens if the new node has to be inserted at the end of the list? This will happen if all the items in the list are less than the new item. In fact, this case is already handled correctly by the subroutine, in the last part of the if statement. If insertItem is greater than all the items in the list, then the while loop will end when runner has traversed the entire list and become null. However, when that happens, previous will be left pointing to the last node in the list. Setting previous.next = newNode adds newNode onto the end of the list. Since runner is null, the command newNode.next = runner sets newNode.next to null, which is exactly what is needed to mark the end of the list.

2.7 Deleting from a Linked List

The delete operation is similar to insert, although a little simpler. There are still special cases to consider. When the first node in the list is to be deleted, then the value of head has to be changed to point to what was previously the second node in the list. Since head.next refers to the second node in the list, this can be done by setting head = head.next. (Once again, you should check that this works when head.next is null, that is, when there is no second node in the list. In that case, the list becomes empty.) If the node that is being deleted is in the middle of the list, then we can set up previous and runner with runner pointing to the node that is to be deleted and with previous pointing to the node that precedes that node in the list. Once that is done, the command “previous.next = runner.next;” will delete the node. The deleted node will be garbage collected. I encourage you to draw a picture for yourself to illustrate this operation. Here is the complete code for the delete() method:

```
/*Delete a specified item from the list, if that item is present.
```

```
* If multiple copies of the item are present in the list, only
* the one that comes first in the list is deleted.
* @param deleteItem the item to be deleted
* @return true if the item was found and deleted, or false if the item
* was not in the list. */

public boolean delete(String deleteItem) {
      if ( head == null ) {
      // The list is empty, so it certainly doesn't contain deleteString.
            return false;
      }
      else if ( head.item.equals(deleteItem) ) {
            // The string is the first item of the list. Remove it.
            head = head.next;
            return true;
      }
      else {
            // The string, if it occurs at all, is somewhere beyond the
            // first element of the list. Search the list.
            Node runner; // A node for traversing the list.
            Node previous; // Always points to the node preceding runner.
            runner = head.next; // Start by looking at the SECOND list node.
            previous = head;
            while (runner!=null && runner.item.compareTo(deleteItem) < 0) {
                  // Move previous and runner along the list until runner
                  // falls off the end or hits a list element that is
                  // greater than or equal to deleteItem. When this
                  // loop ends, runner indicates the position where
                  // deleteItem must be, if it is in the list.
                  previous = runner;
                  runner = runner.next;
            }
            if (runner!=null && runner.item.equals(deleteItem) ) {
                  // Runner points to the node that is to be deleted.
                  // Remove it by changing the pointer in the previous node.
                  previous.next = runner.next;
                  return true;
            }
            else {
                  // The item does not exist in the list.
                  return false;
            }
      }
} // end delete ()
```

2.8 Let Us Sum Up

We have discussed about dynamic data structure Linked List, various operations that can be performed on Linked List such as insert data into a Linked List and delete element from a Linked List

2.9 Further Reading

- Java Collections Framework Tutorials - BeginnersBook.com
  https://beginnersbook.com/java-collections-tutorials/
- Java Collections Framework | Collections in Java With Examples
  https://www.edureka.co/blog/java-collections/

2.10 Assignments

- Explain what is meant by a recursive subroutine.
- Consider the following subroutine:

```
static void printStuff(int level) {
      if (level == 0) {
            System.out.print("*");
      }
      else {
            System.out.print("[");
            printStuff(level - 1);
            System.out.print(",");
            printStuff(level - 1);
            System.out.print("]");
      }
}
```

  Show the output that would be produced by the subroutine calls printStuff(0), printStuff(1), printStuff(2), and printStuff(3).

- Suppose that a linked list is formed from objects that belong to the class

```
class ListNode {
   int item; // An item in the list.
   ListNode next; // Pointer to next item in the list.
}
```

  Write a subroutine that will count the number of zeros that occur in a given linked list of ints. The subroutine should have a parameter of type ListNode and should return a value of type int.
- Let ListNode be defined as in the previous problem. Suppose that head is a variable of type ListNode that points to the first node in a linked list. Write a code segment that will add the number 42 in a new node at the end of the list. Assume that the list is not empty. (There is no “tail pointer” for the list.)

Block-4

Streams and Multithreaded Programming

Unit 1: Input/Output Streams, Files, and Networking

1

Unit Structure

1.1 LEARNING OBJECTIVES

After studying this unit student should be able to understand:

- Input/Output Streams
- Files and Programming with Files
- Networking

1.2 INTRODUCTION

Computer programs are only useful if they interact with the rest of the world in some way. This interaction is referred to as input/output, or I/O (pronounced “eye-oh”). Up until now, this book has concentrated on just one type of interaction: interaction with the user, through either a graphical user interface or a command-line interface. But the user is only one possible source of information and only one possible destination for information. Most important, aside from files, is that it supports communication over network connections. In Java, the most common type of input/output involving files and networks is based on I/O streams, which are objects that support I/O commands that are similar to those that you have already used. In fact, standard output (System.out) and standard input (System.in) are examples of I/O streams.

Working with files and networks requires familiarity with exceptions. Many of the subroutines that are used can throw checked exceptions, which require mandatory exception handling. This generally means calling the subroutine in a try..catch statement that can deal with the exception if one occurs. Effective network communication also requires the use of threads. We will look at the basic networking API in this chapter.

1.3 I/O Streams, Readers, and Writers

Without the ability to interact with the rest of the world, a program would be useless. The interaction of a program with the rest of the world is referred to as input/output or I/O. Historically, one of the hardest parts of programming language design has been coming up with good facilities for doing input and output. A computer can be connected to many different types of input and output devices. If a programming language had to deal with each type of device as a special case, the complexity would be overwhelming. One of the major achievements in the history of programming has been to come up with good abstractions for representing I/O devices. In Java, the main I/O abstractions are called I/O streams. Other I/O abstractions, such as “files” and “channels” also exist, but in this section, we will look only at streams. Every stream represents either a source of input or a destination to which output can be sent.

1.4 Character and Byte Streams

When dealing with input/output, you have to keep in mind that there are two broad categories of data: machine-formatted data and human-readable text. Machine-formatted data is represented in binary form, the same way that data is represented inside the computer, that is, as strings of zeros and ones. Human-readable data is in the form of characters. When you read a number such as 3.141592654, you are reading a sequence of characters and interpreting them as a number. The same number would be represented in the computer as a bit-string that you would find unrecognizable.

To deal with the two broad categories of data representation, Java has two broad categories of streams: byte streams for machine-formatted data and character streams for human-readable data. There are several predefined classes that represent streams of each type. An object that outputs data to a byte stream belongs to one of the subclasses of the abstract class OutputStream. Objects that read data from a byte stream belong to subclasses of the abstract class InputStream. If you write numbers to an OutputStream, you won't be able to read the resulting data yourself. But the data can be read back into the computer with an InputStream. The writing and reading of the data will be very efficient, since there is no translation involved: the bits that are used to represent the data inside the computer are simply copied to and from the streams.

For reading and writing human-readable character data, the main classes are the abstract classes Reader and Writer. All character stream classes are subclasses of one of these. If a number is to be written to a Writer stream, the computer must translate it into a human-readable sequence of characters that represents that number. Reading a number from a Reader stream into a numeric variable also involves a translation, from a character sequence into the appropriate bit string. (Even if the data you are working with consists of characters in the first place, such as words from a text editor, there might still be some translation. Characters are stored in the computer as 16-bit Unicode values. For people who use the English alphabets, character data is generally stored in files in ASCII code, which uses only 8 bits per character. The Reader and Writer classes take care of this translation, and can also handle non-western alphabets and characters in non-alphabetic written languages such as Chinese.)

Byte streams can be useful for direct machine-to-machine communication, and they can sometimes be useful for storing data in files, especially when large amounts of data need to be stored efficiently, such as in large databases. However, binary data is fragile in the sense that its meaning is not self-evident. When faced with a long series of zeros and ones, you have to know what information it is meant to represent and how that information is encoded before you will be able to interpret it. Of course, the same is true to some extent for character data, since characters, like any other kind of data, have to be coded as binary numbers to be stored or processed by a computer. But the binary encoding of character data has been standardized and is well understood, and data expressed in character form can be made meaningful to human readers. The current trend seems to be towards increased use of character data, represented in a way that will make its meaning as self-evident as possible. We'll

look at one way this is done in Section 11.5. I should note that the original version of Java did not have character streams, and that for ASCII-encoded character data, byte streams are largely interchangeable with character streams. In fact, the standard input and output streams, System.in and System.out, are byte streams rather than character streams. However, you should prefer Readers and Writers rather than InputStreams and OutputStreams when working with character data, even when working with the standard ASCII character set.

The standard I/O stream classes discussed in this section are defined in the package java.io, along with several supporting classes. You must import the classes from this package if you want to use them in your program. That means either importing individual classes or putting the directive "import java.io.*;" at the beginning of your source file. I/O streams are used for working with files and for doing communication over a network. They can also be used for communication between two concurrently running threads, and there are stream classes for reading and writing data stored in the computer's memory.

The beauty of the stream abstraction is that it is as easy to write data to a file or to send data over a network as it is to print information on the screen.

* * *

The basic I/O classes Reader, Writer, InputStream, and OutputStream provide only very primitive I/O operations. For example, the InputStream class declares an abstract instance method

```
public int read() throws IOException
```

for reading one byte of data, as a number in the range 0 to 255, from an input stream. If the end of the input stream is encountered, the read() method will return the value -1 instead. If some error occurs during the input attempt, an exception of type IOException is thrown. Since IOException is a checked exception, this means that you can't use the read() method except inside a try statement or in a subroutine that is itself declared with a "throws IOException" clause.

The InputStream class also defines methods for reading multiple bytes of data in one step into an array of bytes, which can be a lot more efficient that reading individual bytes. However, InputStream provides no convenient methods for reading other types of data, such as int or double, from a stream. This is not a problem because you will rarely use an object of type InputStream itself. Instead, you'll use subclasses of InputStream that add more convenient input methods to InputStream's rather primitive capabilities. Similarly, the OutputStream class defines a primitive output method for writing one byte of data to an output stream. The method is defined as:

```
public void write(int b) throws IOException
```

The parameter is of type int rather than byte, but the parameter value is type-cast to type byte before it is written; this effectively discards all but the eight low order bits of b. Again, in practice, you will almost always use higher-level output operations defined in some subclass of OutputStream.

The Reader and Writer classes provide the analogous low-level read and write methods. As in the byte stream classes, the parameter of the write(c) method in Writer and the return value of the read() method in Reader are of type int, but in these character-oriented classes, the I/O operations read and write characters rather than bytes. The return value of read() is -1 if the end of the input stream has been reached. Otherwise, the return value must be type-cast to type char to obtain the character that was read. In practice, you will ordinarily use higher level I/O operations provided by sub-classes of Reader and Writer, as discussed below.

1.5 PrintWriter

One of the neat things about Java's I/O package is that it lets you add capabilities to a stream by "wrapping" it in another stream object that provides those capabilities. The wrapper object is also a stream, so you can read from or write to it—but you can do so using fancier operations than those available for basic streams.

For example, PrintWriter is a subclass of Writer that provides convenient methods for outputting human-readable character representations of all of Java's basic data types. If you have an object belonging to the Writer class, or any of its subclasses, and you would like to use PrintWriter methods to output data to that Writer, all you have to do is wrap the Writer in a PrintWriter object. You do this by constructing a new PrintWriter object, using the Writer as input to the constructor. For example, if charSink is of type Writer, then you could say

```
PrintWriter printableCharSink = new PrintWriter(charSink);
```

In fact, the parameter to the constructor can also be an OutputStream or a File, and the constructor will build a PrintWriter that can write to that output destination. When you output data to the PrintWriter printableCharSink, using the high-level output methods in PrintWriter, that data will go to exactly the same place as data written directly to charSink. You've just provided a better interface to the same output destination. For example, this allows you to use PrintWriter methods to send data to a file or over a network connection.

For the record, if out is a variable of type PrintWriter, then the following methods are defined:

out.print(x)	prints the value of x, represented in the form of a string of characters, to the output stream; x can be an expression of any type, including both primitive types and object types. An object is converted to string form using its toString() method. A null value is represented by the string "null".
out.println()	outputs an end-of-line to the output stream.
out.println(x)	outputs the value of x, followed by an end-of-line; this is equivalent to out.print(x) followed by out.println().

out.printf(formatString, x1, x2, ...)	does formatted output of x1, x2, ... to the output stream. The first parameter is a string that specifies the format of the output. There can be any number of additional parameters, of any type, but the types of the parameters must match the formatting directives in the format string.
out.flush()	ensures that characters that have been written with the above methods are actually sent to the output destination. In some cases, notably when writing to a file or to the network, it might be necessary to call this method to force the output to actually appear at the destination.

Note that none of these methods will ever throw an IOException. Instead, the PrintWriter class includes the method

```
public boolean checkError()
```

which will return true if any error has been encountered while writing to the stream. The PrintWriter class catches any IOExceptions internally, and sets the value of an internal error flag if one occurs. The checkError() method can be used to check the error flag. This allows you to use PrintWriter methods without worrying about catching exceptions. On the other hand, to write a fully robust program, you should call checkError() to test for possible errors whenever you use a PrintWriter.

1.6 Data Streams

When you use a PrintWriter to output data to a stream, the data is converted into the sequence of characters that represents the data in human-readable form. Suppose you want to output the data in byte-oriented, machine-formatted form. The java.io package includes a byte-stream class, DataOutputStream, that can be used for writing data values to streams in internal, binary-number format. DataOutputStream bears the same relationship to OutputStream that PrintWriter bears to Writer. That is, whereas OutputStream only has methods for outputting bytes, DataOutputStream has methods writeDouble(double x) for outputting values of type double, writeInt(int x) for outputting values of type int, and so on. Furthermore, you can wrap any OutputStream in a DataOutputStream so that you can use the higher level output methods on it. For example, if byteSink is of type OutputStream, you could say

```
DataOutputStream dataSink = new DataOutputStream(byteSink);
```

to wrap byteSink in a DataOutputStream.

For input of machine-readable data, such as that created by writing to a DataOutputStream, java.io provides the class DataInputStream. You can wrap any InputStream in a DataInputStream object to provide it with the ability to read data of various types from the byte-stream. The methods in the DataInputStream for reading binary data are called readDouble(), readInt(), and so on. Data written by a DataOutputStream is guaranteed to be in a format that can be read by a DataInputStream. This is true even if the data stream is created on one type of computer and read on another type of computer. The cross-platform compatibility of

binary data is a major aspect of Java's platform independence.

In some circumstances, you might need to read character data from an InputStream or write character data to an OutputStream. This is not a problem, since characters, like all data, are ultimately represented as binary numbers. However, for character data, it is convenient to use Reader and Writer instead of InputStream and OutputStream. To make this possible, you can wrap a byte stream in a character stream. If byteSource is a variable of type InputStream and byteSink is of type OutputStream, then the statements

```
Reader charSource = new InputStreamReader( byteSource );
Writer charSink = new OutputStreamWriter( byteSink );
```

create character streams that can be used to read character data from and write character data to the byte streams. In particular, the standard input stream System.in, which is of type InputStream for historical reasons, can be wrapped in a Reader to make it easier to read character data from standard input:

```
Reader charIn = new InputStreamReader(System.in );
```

As another application, the input and output streams that are associated with a network connection are byte streams rather than character streams, but the byte streams can be wrapped in character streams to make it easy to send and receive character data over the network.

There are various ways for characters to be encoded as binary data. A particular encoding is known as a charset or character set . Charsets have standardized names such as "UTF-16," "UTF-8," and "ISO-8859-1." In UTF-16, characters are encoded as 16-bit UNICODE values; this is the character set that is used internally by Java. UTF-8 is a way of encoding UNICODE characters using 8 bits for common ASCII characters and longer codes for other characters. ISO-8859-1, also known as "Latin-1," is an 8-bit encoding that includes ASCII characters as well as certain accented characters that are used in several European languages. Readers and Writers use the default charset for the computer on which they are running, unless you specify a different one. That can be done, for example, in a constructor such as Writer charSink = new OutputStreamWriter(byteSink, "ISO-8859-1"); Certainly, the existence of a variety of charset encodings has made text processing more complicated—unfortunate for us English-speakers but essential for people who use non-Western character sets. Ordinarily, you don't have to worry about this, but it's a good idea to be aware that different charsets exist in case you run into textual data encoded in a non-default way.

1.7 Reading Text

Much I/O is done in the form of human-readable characters. In view of this, it is surprising that Java does not provide a standard character input class that can read character data in a manner that is reasonably symmetrical with the character output capabilities of PrintWriter. There is one basic case that is easily handled by the standard class BufferedReader, which has a method

```
public String readLine() throws IOException
```

that reads one line of text from its input source. If the end of the stream has been reached, the return value is null. When a line of text is read, the end-of-line marker is read from the input stream, but it is not part of the string that is returned. Different input streams use different characters as end-of-line markers, but the readLine method can deal with all the common cases.

BufferedReader also defines the instance method lines() which returns a value of type Stream<String> that can be used with the stream API. A convenient way to process all the lines from a BufferedReader, reader, is to use the forEach() operator on the stream of lines: reader.lines().forEachOrdered(action), where action is a consumer of strings, usually given as a lambda expression.

Line-by-line processing is very common. Any Reader can be wrapped in a BufferedReader to make it easy to read full lines of text. If reader is of type Reader, then a BufferedReader wrapper can be created for reader with

```
BufferedReader in = new BufferedReader( reader );
```

This can be combined with the InputStreamReader class that was mentioned above to read lines of text from an InputStream. For example, we can apply this to System.in: BufferedReader in; // BufferedReader for reading from standard input.

```
in = new BufferedReader( new InputStreamReader( System.in ) );
try {
      String line = in.readLine();
      while ( line != null ) {
            processOneLineOfInput( line );
            line = in.readLine();
      }
}
catch (IOException e) {
}
```

This code segment reads and processes lines from standard input until an end-of-stream is encountered. (An end-of-stream is possible even for interactive input. For example, on at least some computers, typing a Control-D generates an end-of-stream on the standard input stream.) The try..catch statement is necessary because the readLine method can throw an exception of type IOException, which requires mandatory exception handling; an alternative to try..catch would be to declare that the method that contains the code “throws IOException”. Also, remember that BufferedReader, InputStreamReader, and IOException must be imported from the package java.io.

Note that the main purpose of BufferedReader is not simply to make it easier to read lines of text. Some I/O devices work most efficiently if data is read or written in large chunks, instead of as individual bytes or characters. A BufferedReader reads a chunk of data, and stores it in internal memory. The internal memory is known as a buffer. When you read from the BufferedReader, it will take data from the buffer if possible, and it will only go back to its input source for more data when the buffer is emptied. There is also a BufferedWriter class, and there are buffered stream classes for byte streams as well.

1.8 Serialized Object I/O

The classes PrintWriter, Scanner, DataInputStream, and DataOutputStream allow you to easily input and output all of Java's primitive data types. But what happens when you want to read and write objects? Traditionally, you would have to come up with some way of encoding your object as a sequence of data values belonging to the primitive types, which can then be output as bytes or characters. This is called serializing the object. On input, you have to read the serialized data and somehow reconstitute a copy of the original object. For complex objects, this can all be a major chore. However, you can get Java to do a lot of the work for you by using the classes ObjectInputStream and ObjectOutputStream. These are subclasses of InputStream and OutputStream that can be used for reading and writing serialized objects.

ObjectInputStream and ObjectOutputStream are wrapper classes that can be wrapped around arbitrary InputStreams and OutputStreams. This makes it possible to do object input and output on any byte stream. The methods for object I/O are readObject(), in ObjectInputStream, and writeObject(Object obj), in ObjectOutputStream. Both of these methods can throw IOExceptions. Note that readObject() returns a value of type Object, which generally has to be type-cast to the actual type of the object that was read.

ObjectOutputStream also has methods writeInt(), writeDouble(), and so on, for outputting primitive type values to the stream, and ObjectInputStream has corresponding methods for reading primitive type values. These primitive type values can be interspersed with objects in the data. In the file, the primitive types will be represented in their internal binary format.

Object streams are byte streams. The objects are represented in binary, machine-readable form. This is good for efficiency, but it does suffer from the fragility that is often seen in binary data. They suffer from the additional problem that the binary format of Java objects is very specific to Java, so the data in object streams is not easily available to programs written in other programming languages. For these reasons, object streams are appropriate mostly for short-term storage of objects and for transmitting objects over a network connection from one Java program to another. For long-term storage and for communication with non-Java programs, other approaches to object serialization are usually better.

ObjectInputStream and ObjectOutputStream only work with objects that implement an interface named Serializable. Furthermore, all of the instance variables in the object must be serializable. However, there is little work involved in making an object serializable, since the Serializable interface does not declare any methods. It exists only as a marker for the compiler, to tell it that the object is meant to be writable and readable. You only need to add the words "implements Serializable" to your class definitions. Many of Java's standard classes are already declared to be serializable. One warning about using ObjectOutputStreams: These streams are optimized to avoid writing the same object more than once. When an object is encountered for a second time, only a reference to the first occurrence is written. Unfortunately, if the object has

been modified in the meantime, the new data will not be written. That is, the modified value will not be written correctly to the stream. Because of this, ObjectOutputStreams are meant mainly for use with “immutable” objects that can’t be changed after they are created. (Strings are an example of this.) However, if you do need to write mutable objects to an ObjectOutputStream, and if it is possible that you will write the same object more than once, you can ensure that the full, correct version of the object will be written by calling the stream’s reset() method before writing the object to the stream.

1.9 Files

The data and programs in a computer’s main memory survive only as long as the power is on. For more permanent storage, computers use files, which are collections of data stored on a hard disk, on a USB memory stick, on a CD-ROM, or on some other type of storage device. Files are organized into directories (also called folders). A directory can hold other directories, as well as files. Both directories and files have names that are used to identify them.

Programs can read data from existing files. They can create new files and can write data to files. In Java, such input and output can be done using I/O streams. Human-readable character data can be read from a file using an object belonging to the class FileReader, which is a subclass of Reader. Similarly, data can be written to a file in human-readable format through an object of type FileWriter, a subclass of Writer. For files that store data in machine format, the appropriate I/O classes are FileInputStream and FileOutputStream. In this section, I will only discuss character-oriented file I/O using the FileReader and FileWriter classes. However, FileInputStream and FileOutputStream are used in an exactly parallel fashion. All these classes are defined in the java.io package.

Reading and Writing Files

The FileReader class has a constructor which takes the name of a file as a parameter and creates an input stream that can be used for reading from that file. This constructor will throw an exception of type FileNotFoundException if the file doesn’t exist. For example, suppose you have a file named “data.txt”, and you want your program to read data from that file. You could do the following to create an input stream for the file:

```
FileReader data; // (Declare the variable before the
// try statement, or else the variable
// is local to the try block and you won't
// be able to use it later in the program.)
try {
      data = new FileReader("data.txt"); // create the stream
}
catch (FileNotFoundException e) {
      ... // do something to handle the error---maybe, end the program
}
```

The FileNotFoundException class is a subclass of IOException, so it would be acceptable to catch IOExceptions in the above try...catch statement. More generally,

just about any error that can occur during input/output operations can be caught by a catch clause that handles IOException.

Once you have successfully created a FileReader, you can start reading data from it. But since FileReaders have only the primitive input methods inherited from the basic Reader class, you will probably want to wrap your FileReader in a Scanner, in a BufferedReader, or in some other wrapper class.

To create a BufferedReader for reading from a file named data.dat, you could say:

```
BufferedReader data;
try {
data = new BufferedReader( new FileReader("data.dat") );
}
catch (FileNotFoundException e) {
... // handle the exception
}
```

Wrapping a Reader in a BufferedReader lets you easily read lines of text from the file, and thebuffering can make the input more efficient.

To use a Scanner to read from the file, you can construct the scanner in a similar way. However, it is more common to construct it more directly from an object of type File (to be covered below):

```
Scanner in;
try {
      in = new Scanner( new File("data.dat") );
}
catch (FileNotFoundException e) {
... // handle the exception
}
```

Working with output files is no more difficult than this. You simply create an object belonging to the class FileWriter. You will probably want to wrap this output stream in an object of type PrintWriter. For example, suppose you want to write data to a file named “result.dat”. Since the constructor for FileWriter can throw an exception of type IOException, you should use a try..catch statement:

```
PrintWriter result;
try {
      result = new PrintWriter(new FileWriter("result.dat"));
}
catch (IOException e) {
      ... // handle the exception
}
```

However, as with Scanner, it is more common to use a constructor that takes a File as parameter; this will automatically wrap the File in a FileWriter before creating the PrintWriter:

```
PrintWriter result;
try {
      result = new PrintWriter(new File("result.dat"));
}
```

```
catch (IOException e) {
      ... // handle the exception
}
```

You can even use just a String as the parameter to the constructor, and it will be interpreted as a file name (but you should remember that a String in the Scanner constructor does not name a file; instead the file will read characters from the string itself).

If no file named result.dat exists, a new file will be created. If the file already exists, then the current contents of the file will be erased and replaced with the data that your program writes to the file. This will be done without any warning. To avoid overwriting a file that already exists, you can check whether a file of the same name already exists before trying to create the stream, as discussed later in this section. An IOException might occur in the PrintWriter constructor if, for example, you are trying to create a file on a disk that is "write-protected," meaning that it cannot be modified.

When you are finished with a PrintWriter, you should call its flush() method, such as "result.flush()", to make sure that all the output has been sent to its destination. If you forget to do this, you might find that some of the data that you have written to a file has not actually shown up in the file.

After you are finished using a file, it's a good idea to close the file, to tell the operating system that you are finished using it. You can close a file by calling the close() method of the associated PrintWriter, BufferedReader, or Scanner. Once a file has been closed, it is no longer possible to read data from it or write data to it, unless you open it again as a new I/O stream. (Note that for most I/O stream classes, including BufferedReader the close() method can throw an IOException, which must be handled; however, PrintWriter and Scanner override this method so that it cannot throw such exceptions.) If you forget to close a file, the file will ordinarily be closed automatically when the program terminates or when the file object is garbage collected, but it is better not to depend on this. Note that calling close() should automatically call flush() before the file is closed. (I have seen that fail, but not recently.)

As a complete example, here is a program that will read numbers from a file named data.dat, and will then write out the same numbers in reverse order to another file named result.dat. It is assumed that data.dat contains only real numbers. The input file is read using a Scanner. Exception-handling is used to check for problems along the way. Although the application is not a particularly useful one, this program demonstrates the basics of working with files.

```
import java.io.*;
import java.util.ArrayList;
import java.util.Scanner;

/* Reads numbers from a file named data.dat and writes them to a file named
result.dat in reverse order. The input file should contain only real numbers.
*/
public class ReverseFileWithScanner {
      public static void main(String[] args) {
            Scanner data; // For reading the data.
            PrintWriter result; // Character output stream for writing data.
```

```
            ArrayList<Double> numbers; // An ArrayList for holding the data.
            numbers = new ArrayList<Double>();
            try { // Create the input stream.
                  data = new Scanner(new File("data.dat"));
            }
            catch (FileNotFoundException e) {
                  System.out.println("Can't find file data.dat!");
                  return; // End the program by returning from main().
            }
            try { // Create the output stream.
                  result = new PrintWriter("result.dat");
            }
            catch (FileNotFoundException e) {
                  System.out.println("Can't open file result.dat!");
                  System.out.println("Error: " + e);
                  data.close(); // Close the input file.
                  return; // End the program.
            }
            while ( data.hasNextDouble() ) { // Read until end-of-file.
                  double inputNumber = data.nextDouble();
                  numbers.add( inputNumber );
            }
            // Output the numbers in reverse order.
            for (int i = numbers.size()-1; i >= 0; i--)
                  result.println(numbers.get(i));
                  System.out.println("Done!");
                  data.close();
                  result.close();
      } // end of main()
} // end class ReverseFileWithScanner
```

Note that this program will simply stop reading data from the file if it encounters anything other than a number in the input. That will not be considered to be an error.

Files and Directories

Java has the class java.io.File. An object belonging to this class does not actually represent a file! Precisely speaking, an object of type File represents a file name rather than a file as such. The file to which the name refers might or might not exist. Directories are treated in the same way as files, so a File object can represent a directory just as easily as it can represent a file.

A File object has a constructor, "new File(String)", that creates a File object from a path name. The name can be a simple name, a relative path, or an absolute path. For example, new File("data.dat") creates a File object that refers to a file named data.dat, in the current directory. Another constructor, "new File(File,String)", has two parameters. The first is a File object that refers to a directory. The second can be the name of the file in that directory or a relative path from that directory to the file.

File objects contain several useful instance methods. Assuming that file is a variable of type File, here are some of the methods that are available:

- file.exists() – This boolean-valued function returns true if the file named by the

- File object already exists. You can use this method if you want to avoid overwriting the contents of an existing file when you create a new output stream. The boolean function
- file.canRead() – returns true if the file exists and the program has permission to read the file. And
- file.canWrite() is true if the program has permission to write to the file.
- file.isDirectory() – This boolean-valued function returns true if the File object refers to a directory. It returns false if it refers to a regular file or if no file with the given name exists.
- file.delete() – Deletes the file, if it exists. Returns a boolean value to indicate whether the file was successfully deleted.
- file.list() – If the File object refers to a directory, this function returns an array of type String[] containing the names of the files in that directory. Otherwise, it returns null. The method file.listFiles() is similar, except that it returns an array of File instead of an array of String.

Here, for example, is a program that will list the names of all the files in a directory specified by the user. In this example, I have used a Scanner to read the user's input:

```
import java.io.File;
import java.util.Scanner;

/* This program lists the files in a directory specified by the user. The
user is asked to type in a directory name. If the name entered by the user
is not a directory, a message is printed and the program ends. */

public class DirectoryList {
      public static void main(String[] args) {
            String directoryName; // Directory name entered by the user.
            File directory; // File object referring to the directory.
            String[] files; // Array of file names in the directory.
            Scanner scanner; // For reading a line of input from the user.
            scanner = new Scanner(System.in);// scanner reads from std input.
            System.out.print("Enter a directory name: ");
            directoryName = scanner.nextLine().trim();
            directory = new File(directoryName);
            if (directory.isDirectory() == false) {
                  if (directory.exists() == false)
                        System.out.println("There is no such directory!");
                  else
                        System.out.println("That file is not a directory.");
            }
            else {
                  files = directory.list();
                  System.out.println("Files in dir \"" + dir + "\":");
                  for (int i = 0; i < files.length; i++)
                        System.out.println(" " + files[i]);
            }
      } // end main()
} // end class DirectoryList
```

All the classes that are used for reading data from files and writing data to files have constructors that take a File object as a parameter. For example, if file is a variable of type File, and you want to read character data from that file, you can create a FileReader to do so by saying new FileReader(file).

1.10 Networking

As far as a program is concerned, a network is just another possible source of input data, and another place where data can be output. That does oversimplify things, because networks are not as easy to work with as files are. But in Java, you can do network communication using input streams and output streams, just as you can use such streams to communicate with the user or to work with files. Nevertheless, opening a network connection between two computers is a bit tricky, since there are two computers involved and they have to somehow agree to open a connection. And when each computer can send data to the other, synchronizing communication can be a problem. But the fundamentals are the same as for other forms of I/O.

One of the standard Java packages is called java.net. This package includes several classes that can be used for networking. Two different styles of network I/O are supported. One of these, which is fairly high-level, is based on the World Wide Web, and provides the sort of network communication capability that is used by a Web browser when it downloads pages for you to view. The main classes for this style of networking are java.net.URL and java.net.URLConnection. An object of type URL is an abstract representation of a Universal Resource Locator, which is an address for an HTML document or other resource. A URLConnection represents a network connection to such a resource.

The second style of I/O, which is more general and more important, views the network at a lower level. It is based on the idea of a socket. A socket is used by a program to establish a connection with another program on a network. Communication over a network involves two sockets, one on each of the computers involved in the communication. Java uses a class called java.net.Socket to represent sockets that are used for network communication. The term “socket” presumably comes from an image of physically plugging a wire into a computer to establish a connection to a network, but it is important to understand that a socket, as the term is used here, is simply an object belonging to the class Socket. In particular, a program can have several sockets at the same time, each connecting it to another program running on some other computer on the network or even running on the same computer. All these connections use the same physical network connection.

This section gives a brief introduction to these basic networking classes, and shows how they relate to input and output streams

The URL class is used to represent resources on the World Wide Web. Every resource has an address, which identifies it uniquely and contains enough information for a Web browser to find the resource on the network and retrieve it. The address is called a “url” or “universal resource locator.” (URLs can actually refer to resources from other sources besides the web; after all, they are “universal.” For example, they can represent files on your computer.)
An object belonging to the URL class represents such an address. Once you have a URL object, you can use it to open a URLConnection to the resource at that address. A url is ordinarily specified as a string, such as http://math.hws.edu/eck/index.html. There are also relative url’s. A relative url specifies the location of a resource relative

to the location of another url, which is called the base or context for the relative url. For example, if the context is given by the url http://math.hws.edu/eck/, then the incomplete, relative url "index.html" would really refer to http://math.hws.edu/eck/index.html.

An object of the class URL is not simply a string, but it can be constructed from a string representation of a url. A URL object can also be constructed from another URL object, representing a context, plus a string that specifies a url relative to that context. These constructors have prototypes

```
public URL(String urlName) throws MalformedURLException
```

and

```
public URL(URL context, String relativeName) throws MalformedURLException
```

Note that these constructors will throw an exception of type MalformedURLException if the specified strings don't represent legal url's. The MalformedURLException class is a subclass of IOException, and it requires mandatory exception handling.

Once you have a valid URL object, you can call its openConnection() method to set up a connection. This method returns a URLConnection. The URLConnection object can, in turn, be used to create an InputStream for reading data from the resource represented by the URL. This is done by calling its getInputStream() method. For example:

```
URL url = new URL(urlAddressString);
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
```

The openConnection() and getInputStream() methods can both throw exceptions of type IOException. Once the InputStream has been created, you can read from it in the usual way, including wrapping it in another input stream type, such as BufferedReader, or using a Scanner. Reading from the stream can, of course, generate exceptions.

One of the other useful instance methods in the URLConnection class is getContentType(), which returns a String that describes the type of information available from the URL. The return value can be null if the type of information is not yet known or if it is not possible to determine the type. The type might not be available until after the input stream has been created, so you should generally call getContentType() after getInputStream(). The string returned by getContentType() is in a format called a mime type. Mime types include "text/plain", "text/html", "image/jpeg", "image/png", and many others. All mime types contain two parts: a general type, such as "text" or "image", and a more specific type within that general category, such as "html" or "png". If you are only interested in text data, for example, you can check whether the string returned by getContentType() starts with "text". (Mime types were first introduced to describe the content of email messages. The name stands for "Multipurpose Internet Mail Extensions." They are now used almost universally to specify the type of information in a file or other resource.)

Let's look at a short example that uses all this to read the data from a URL. This subroutine opens a connection to a specified URL, checks that the type of data at the URL is text, and then copies the text onto the screen. Many of the operations in this subroutine can throw exceptions. They are handled by declaring that the subroutine "throws IOException" and leaving it up to the main program to decide what to do when an error occurs.

```
static void readTextFromURL( String urlString ) throws IOException {

      /* Open a connection to the URL, and get an input stream
      for reading data from the URL. */

      URL url = new URL(urlString);
      URLConnection connection = url.openConnection();
      InputStream urlData = connection.getInputStream();

      /* Check that the content is some type of text. Note:
      connection.getContentType() method should be called
      after getInputStream(). */

      String contentType = connection.getContentType();
      System.out.println("Stream opened with content type: " + contentType);
      System.out.println();

      if (contentType == null || contentType.startsWith("text") == false)
            throw new IOException("URL does not refer to a text file.");
      System.out.println("Fetching context from " + urlString + " ...");
      System.out.println();

      /* Copy lines of text from the input stream to the screen, until
      end-of-file is encountered (or an error occurs). */

      BufferedReader in; // For reading from the connection's input stream.
      in = new BufferedReader( new InputStreamReader(urlData) );
      while (true) {
            String line = in.readLine();
            if (line == null)
            break;
            System.out.println(line);
      }
      in.close();
} // end readTextFromURL()
```

Sockets in Java

To implement TCP/IP connections, the java.net package provides two classes, ServerSocket and Socket. A ServerSocket represents a listening socket that waits for connection requests from clients. A Socket represents one endpoint of an actual network connection. A Socket can be a client socket that sends a connection request to a server. But a Socket can also be created by a server to handle a connection request from a client. This allows the server to create multiple sockets and handle multiple connections. A ServerSocket does not itself participate in connections; it just listens for connection requests and creates Sockets to handle the actual connections. When you construct a ServerSocket object, you have to specify the port number on which the server will listen. The specification for the constructor is

```
public ServerSocket(int port) throws IOException
```

The port number must be in the range 0 through 65535, and should generally be greater than 1024. The constructor might throw a SecurityException if a port number smaller than 1024 is specified. An IOException can also occur if, for example, the specified port number is already in use. (A parameter value of 0 in this method tells the server socket to listen on any available port.)

As soon as a ServerSocket is created, it starts listening for connection requests. The accept() method in the ServerSocket class accepts such a request, establishes a connection with the client, and returns a Socket that can be used for communication with the client. The accept() method has the form

```
public Socket accept() throws IOException
```

When you call the accept() method, it will not return until a connection request is received (or until some error occurs). The method is said to block while waiting for the connection. (While the method is blocked, the program—or more exactly, the thread—that called the method can't do anything else. If there are other threads in the same program, they can proceed.) You can call accept() repeatedly to accept multiple connection requests. The ServerSocket will continue listening for connections until it is closed, using its close() method, or until some error occurs, or until the program is terminated in some way.

Suppose that you want a server to listen on port 1728, and that you want it to continue to accept connections as long as the program is running. Suppose that you've written a method provideService(Socket) to handle the communication with one client. Then the basic form of the server program would be:

```
try {
      ServerSocket server = new ServerSocket(1728);
      while (true) {
            Socket connection = server.accept();
            provideService(connection);
      }
}
catch (IOException e) {
      System.out.println("Server shut down with error: " + e);
}
```

On the client side, a client socket is created using a constructor in the Socket class. To connect to a server on a known computer and port, you would use the constructor

```
public Socket(String computer, int port) throws IOException
```

The first parameter can be either an IP number or a domain name. This constructor will block until the connection is established or until an error occurs.

Once you have a connected socket, no matter how it was created, you can use the Socket methods getInputStream() and getOutputStream() to obtain streams that can be used for communication over the connection. These methods return objects of type InputStream and OutputStream, respectively. Keeping all this in mind, here is the outline of a method for working with a client connection:

```
/* Open a client connection to a specified server computer and port number
on the server, and then do communication through the connection. */

void doClientConnection(String computerName, int serverPort) {
      Socket connection;
      InputStream in;
      OutputStream out;
      try {
            connection = new Socket(computerName,serverPort);
            in = connection.getInputStream();
            out = connection.getOutputStream();
      }
      catch (IOException e) {
            System.out.println(
            "Attempt to create connection failed with error: " + e);
            return;
      }
      .
      . // Use the streams, in and out, to communicate with the server.
      .
      try {
            connection.close();
            // (Alternatively, you might depend on the server
            // to close the connection.)
      }
      catch (IOException e) {

      }
} // end doClientConnection()
```

All this makes network communication sound easier than it really is. (And if you think it sounded hard, then it's even harder.) If networks were completely reliable, things would be almost as easy as I've described. The problem, though, is to write robust programs that can deal with network and human error. I won't go into detail. However, what I've covered here should give you the basic ideas of network programming, and it is enough to write some simple network applications.

1.11 Let Us Sum Up

In this unit we have learned about I/O Streams, Readers, and Writers. We discussed Character and Byte Streams, PrintWriters and Data Streams. We also learned about how to Read Text and Serialized Object I/O, Files, Directories and Networking.

1.12 Further Reading

- Core Java for Beginners: 1 (X-Team) by Sharanam Shah, Vaishali Shah 1st edition
- Java Programming for Beginners by Mark Lassoff
- Core Java Programming-A Practical Approach by Tushar B. Kute
- Java: The Complete Reference by Schildt Herbert. Ninth Edition

1.13 Assignments

- In Java, input/output is done using I/O streams. I/O streams are an abstraction. Explain what this means and why it is important.
- Java has two types of I/O stream: character streams and byte streams. Why? What is the difference between the two types of streams?
- What is a file? Why are files necessary?
- What is the point of the following statement?
  out = new PrintWriter(new FileWriter("data.dat"));
  Why would you need a statement that involves two different stream classes, PrintWriter and FileWriter?

- The package java.io includes a class named URL. What does an object of type URL represent, and how is it used?
- What is the purpose of the FileChooser class?
- Explain what is meant by the client / server model of network communication.
- What is a socket?
- What is a ServerSocket and how is it used?
- Write a complete program that will display the first ten lines from a text file. The lines should be written to standard output, System.out. The file name is given as the command-line argument args[0]. You can assume that the file contains at least ten lines. Don't bother to make the program robust. Do not use TextIO to process the file; read from the file using methods covered in this chapter.

Unit 2: Threads and Multiprocessing

2

Unit Structure

2.1 LEARNING OBJECTIVE

- Understand purpose of multitasking and multithreading
- Describe java's multithreading model

2.2 OUTCOMES

After learning the contents of this chapter, the reader must be able to :

- Describe the concept of multithreading
- Explain the Java thread model
- Create and use threads in program
- Describe how to set the thread priorities

2.3 INTRODUCTION

Multitasking – performing multiple tasks/jobs simultaneously/concurrently. There are two types of concurrency- Real and Apparent. Personal Computer has only a single CPU; so, you might have a question, how it can execute more than one task at the same time? With single microprocessor systems, only a single task can run at a time. But multitasking system increase the utilization of CPU. The CPU quickly switches back and forth between several tasks to create an illusion that the tasks are performing/ executing at the same time. For example, a user/system can request the operating system to execute program P1, P2 and P3 by having it spawn a separate process for each program and scheduled it independently. These programs can run in a concurrent manner, depending upon the multiprocessing (multiprogramming) features supported by the operating system. A process is memory image/context of a program that is created when the program is executed.
In single-processor systems support apparent concurrency only. Real concurrency is not supported by it. Apparent concurrency is the characteristic exhibited when multiple tasks execute. There are two types of multitasking –
1. Process based multitasking and
2. Thread based multitasking.

A thread is single sequence of execution that can run independently in an application. Uses of thread in programs are good in terms of resource utilization of the system on which application(s) is running. There are several advantages of thread based multitasking, so Java programming language support thread based multitasking.

This unit covers the very important concept of multithreading in programming. Multithreading differs from multiprocessing. Multithreaded programming is very useful in network and Internet applications development. In this unit you will learn what is multithreading, how thread works, how to write programs in Java using multithreading.

Also, in this unit will be explained about thread-properties, synchronization, and interthread communication.

2.4 MULTITHREADING: AN INTRODUCTION AND ADVANTAGES

A multithreaded program contains two or more parts that can run simultaneously. Each such part of a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. This means that multiples threads are simultaneously execute multiple sequences of instructions. Each instruction sequence has its own unique flow of control that is independent of all others. These independently executed instruction sequences are known as threads. Threads allow multiple activities to proceed concurrently in the same program. . For example, a text editor can edit text at the same time that it is auto save a document, as long as these two actions are being performed by two separate threads. But remember, threads are not complete processes in themselves.

The Java Virtual Machine supports multithreaded programming, which allows you to write programs that execute many tasks simultaneously. The Java run-time provides simple solution for multithread synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

➢ Advantages of Multithreading

The advantages of multithreading are:

1. Concurrency can be used within a process to implement multiple instances of simultaneous task.
2. Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process.
3. Multithreading requires less processing overhead than multiprocessing because concurrent threads are able to share common resources more efficiently.
4. Multithreading enables programmers to write very efficient programs that make maximum use of the CPU.
5. Inter-thread communication is less expensive.

2.5 THE MAIN THREAD

When you execute a java program, usually a single non-daemon thread begins running immediately. This is called the "main" thread of your program, because it is

the one that is executed when your program begins. The main thread is very important for two reasons:

1. It is the thread from which other “child” threads will be spawned. And,
2. It must be the last thread to finish execution because it performs various cleanup and shutdown actions.

The main thread is created automatically when your program is started. The main thread of Java programs is accessed and controlled through methods of **Thread** class.

You can get a reference of current running thread by calling currentThread( ) method of the Thread class, which is a static method.

The signature of the method is:

public static Thread currentThread();

By using this method, you obtain a reference to the thread in which this method is called. Once you have a reference to the thread, you can control it.

For example, the following code segment obtain a reference of the main thread and get the name of the main thread is by calling getName () and rename it “MyMainThread” using method setName(String).

// Program-1

```
class ThreadDemo {
        public static void main(String [] args){
                Thread t  = Thread.currentThread();
                System.out.println("Current thread name is: " + t.getName());
                t.setName("MyMainThread");
                System.out.println("New name is: " + t.getName());
        }
}
```

Output:

```
Current thread name is: main
New name is: MyMainThread
```

In java every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Check Your Progress 1

1) How does multithreading achieved on a computer with a single CPU?
2) Name two ways to create a thread
3) Make suitable change in “Program-1” and find out a priority of the main thread as well as the name of thread group in which the main thread belong.

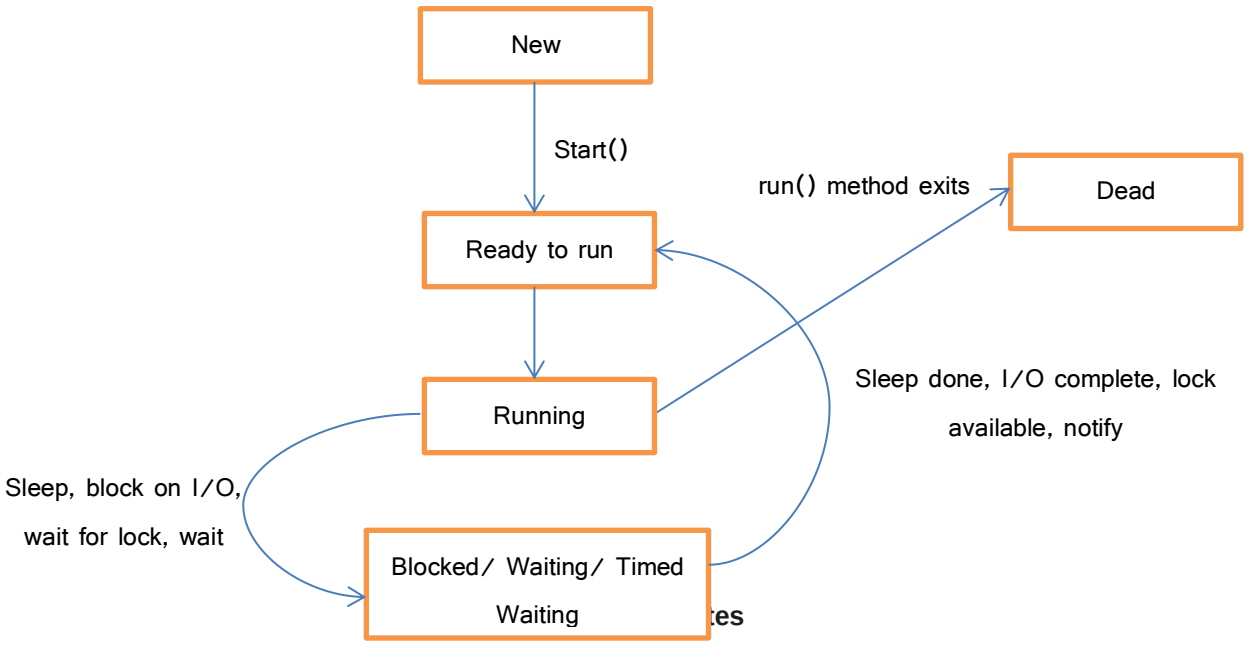
4) How would you re-start a dead Thread?
5) State the advantages of multithreading.
6) Write an application that executes two threads. One thread which is display 'A' every 1000 milliseconds, and the another display 'B' every 3000 milliseconds. Create the first thread by implementing Runnable interface and the second one by extending Thread class.

2.6 THE JAVA THREAD MODEL

The Java run-time environment depends on threads for many things, and all the class libraries are designed with multithreading in mind. For that, Java uses threads to enable the entire environment to be asynchronous. This helps to you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum and preventing the waste of CPU cycles.

2.7 THREAD STATES AND LIFE CYCLE

Thread pass through several stages during its life cycle. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.



The thread exists as an object; threads have several well-defined states in addition to the dead states. These states are:

- **New Thread**

When a new thread (thread object) is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread in the new state, it's code is yet to be run and hasn't started to execute.

- **Runnable State**

A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

- **Running State**

Threads are born to run, and a thread is said to be in the running state when it is actually executing means thread gets CPU. It may leave this state for a number of reasons.

- **Blocked/Waiting/Timed Waiting state**

When a thread is temporarily inactive, then it's in one of the following states:

- Blocked
- Waiting
- Timed Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the threads which is blocked for that section and moves it to the runnable state. A thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

A thread lies in timed waiting/temporality sleep state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to time waiting state.

- **Dead State**

A thread terminates because of either of the following reasons:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagate beyond the run method.

A thread that lies in this state does no longer consume any cycles of CPU. After a thread reaches the dead state, then it is not possible to restart it.

2.8 THE THREAD CLASS AND RUNNABLE INTERFACE

Java's multithreading organization is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution. To create a new thread, your program will either extend Thread or implement the Runnable interface. The Thread class defines several methods that help manage threads. Some of that will be used in this chapter are follows:

- **Constructors of Thread class**

Thread()

Thread(Runnable target)

Thread (Runnable target, String name)

Thread(String name)

Thread(ThreadGroup group, Runnable target)

Thread(ThreadGroup group, Runnable target, String name)

Thread(ThreadGroup group, Runnable target, String name, long stackSize) Thread(ThreadGroup group, String name)

- **Methods of Thread class**

Methods	**Description**
public static Thread **currentThread**()	Returns a reference to the currently executing thread object.
public String getName()	Obtain a thread's name
public int getPriority()	Obtain a thread's priority
public boolean isAlive()	Determine if a thread is still running
public void join()	Wait for a thread to terminate
public void run()	Entry point for the thread and execution of it begins.
public void sleep()	Suspend a thread for a period of time
public void start()	Start a thread by calling its run method.
public void setName(String name)	Change name of the thread
public void setPriority(int priority)	Changes the priority of thread
public static void **yield**()	Used to pause temporarily to currently executing thread object and allow other threads to execute.

public static int **activeCount**()	Returns the number of active threads in the current thread's thread group.

Table-9 Methods of Thread cladd

The Thread class defines three int static constants that are used to specify the priority of a thread. These are MAX_PRIORITY, MIN_PRIORITY, and NORM_PRIORITY. They represent the maximum, minimum and normal thread priorities.

2.9 THREAD CREATION

Java has built support to create a thread by instantiating an object of type **Thread.** Java lets you create a thread one of two ways:

1. By **extending** the **Thread class.**
2. By **implementing** the **Runnable interface**.

Thread class in the java.lang package allows you to create and manage threads. The thread class provides the capability to create thread objects, each with its own separate flow of control. The signature of the class is:

public class java.lang.Thread extends java.lang.Object implements java.lang.Runnable

- **Extending Thread class**

In the first approach, you create a child of the java.lang.Thread class and override the run( ) method.

```
class EvenThread extends Thread{
        public void run(){
                //Logic for the thread
        }
}
```

Here the class EvenThread extends Thread. The logic for the thread is written in run() method. The complexity of run() method may be simple or complex is depending on what would you like to performed in you thread.

The program can create an object of the thread by

EvenThread et = new EvenThread(); // Instantiates the EvenThread class

When you create an instance of child of Thread class, you invoke start( ) method to cause the thread to execute. The start( ) method is inherited from the Thread class. It register the thread with scheduler and invokes the run( ) method. Your logic for the thread is implemented in the run() method.

Et.start(); // invokes the start() method of that object to start execution of thread.

Now let us see the program given below for creating threads by inheriting the Thread class. The program prints even numbers after every one second interval.

// Program-2

```
class EvenThread extends Thread {
        EvenThread(String name){
                super(name);}
        public void run(){
                for(int i=1; i<11; i++){
                        if(i%2==0)
                                System.out.println(this.getName() + " :" + i);
                        try{
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                        System.out.println (" Thread is Interrupted");
                        }
                }
        }
}
class ThreadDemoOne{

        public static void main(String [] args){
                EvenThread et1 = new EvenThread("Thread 1 : ");
                et1.start();
                EvenThread et2 = new EvenThread("Thread 2 : ");
                et2.start();
                while(et1.isAlive() || et2.isAlive()){}
        }
}
```

Output :

```
Thread 1 :  2
Thread 2 :  2
Thread 1 :  4
Thread 2 :  4
Thread 2 :  6
Thread 1 :  6
Thread 2 :  8
Thread 1 :  8
```

Thread 2 : 10

Thread 1 : 10

Above output shows how two threads execute in sequence, displaying information on the console. The program creates two threads of execution, et1, and et2. The threads display even numbers from 1 to 10, by interval of 1 second.

- **Implementing Runnable**

There is another way to create thread. Declare a class that implements java.lang.Runnable interface. The Runnable interface contain on one method, that is public void run(). The run ( ) provides entry point into your thread.

```
class EvenRunnable implements Runnable{
        public void run(){
                //Logic for the thread
        }
}
```

The program can start an instance of the thread by using following code:

```
EvenRunnable et = new EvenRunnable ();
Thread t = new Thread(et);
t.start();
```

The first statement creates an object of EvenRunnable class. The second statement creates an object of thread class. A reference of EvenRunnble object is provided as argument to the constructor. The last statement starts the thread.

Now let us see the program given below for creating threads by implementing Runnable.

// Program-3

```
class EvenRunnable implements Runnable {
String name="";
EvenRunnable (String name){
                this.name = name;
        }
        public void run(){
                for(int i=1; i<11; i++){
                        if(i%2==0)
                        System.out.println(Thread.currentThread().getName() + " :" + i);
                        try{
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                        System.out.println (" Thread is interrupted");
```

```
                }
            }
        }
}
class ThreadDemoTwo{
        public static void main(String [] args){
                EvenThread et1 = new EvenThread("Thread 1 : ");
                Thread t1 = new Thread(et1);
                t1.start();
                EvenThread et2 = new EvenThread("Thread 2 : ");
                Thread t2 = new Thread(et2);
                t2.start();
                while(t1.isAlive() || t2.isAlive()){}
        }
}
```

Output:

```
Thread 1 :  2
Thread 2 :  2
Thread 1 :  4
Thread 2 :  4
Thread 2 :  6
Thread 1 :  6
Thread 2 :  8
Thread 1 :  8
Thread 2 :  10
Thread 1 :  10
```

This program is similar to previous program and also gives same output. The advantage of using the Runnable interface is that your class does not need to extend the thread class. This is a very helpful feature when you create multithreaded program in that your class already extending for some other class. The only disadvantage of this approach is that you have to do some more work to create and execute your own threads.

- **Choosing an Approach**

At this point, you might be questioning why Java has two ways to create child threads, and which approach is better.

Extending Thread class allows you to modify other overridable methods of the Thread class, if should you wish to do so. Extending Thread class will not give you

an option to extend any other class. But if you implement Runnable interface you could extend other classes in your class. Advantages of implementing Runnable are

1. You have freedom to extend any other class
2. You can implement more interfaces
3. You can use you Runnable implementation in thread pools

2.10 THREAD PRIORITIES

In java every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread. Thread priority is an integer value that specifies the relative priority of one thread to another. A thread can voluntarily relinquish control. Threads relinquish control by explicitly yielding, sleeping, or blocking on pending Input/ Output operations. In this scenario, all other threads are examined, and the highest- priority thread that is ready to run gets the chance to use the CPU.

A higher-priority thread can preempt a low priority thread. In this case, a lower-priority thread that does not yield the processor is forcibly pre-empted. In cases where two threads with the same priority are competing for CPU cycles, the situation is handled differently by different operating systems.

Java thread class has defined three constants NORM_PRIORITY, MIN_PRIORITY and MAX_PRIORITY. Any thread priority lies between MIN_PRIORITY and MAX_PRIORITY. The value of NORM_PRIORITY is 5, MIN_PRIORITY is 1 and MAX_PRIORITY is 10.

// Program-5

```
class ThreadPriorityDemo
{
    public static void main (String [] args)
    {
    try
      {
          Thread t1 = new Thread("Thread1");
          Thread t2 = new Thread("Thread2");
          System.out.println ("Before any change in default priority :");
          System.out.println("The Priority of "+ t1.getName() +" is "+ t1.getPriority());
          System.out.println("The Priority of "+ t1.getName() +" is "+ t2.getPriority());

          //change in priority
          t1.setPriority(7);
```

```
            t2.setPriority(8);
            System.out.println ("After changing in Priority :");
            System.out.println("The Priority of "+ t1.getName() +" is "+
                                t1.getPriority());
            System.out.println("The Priority of "+t1.getName() +" is "+ t2.getPriority());

              } catch (Exception e) {
                    System.out.println("main thread interrupted");
        }
        }
}
```

Output:

Before any change in default priority :

The Priority of Thread1 is 5

The Priority of Thread1 is 5

After changing in priority :

The Priority of Thread1 is 7

The Priority of Thread1 is 8

Check Your Progress 2

1) How can we create a Thread in Java?
2) How can we pause the execution of a Thread for specific time?
3) What do you understand about Thread Priority?

2.11 LET US SUM UP

This chapter described the functioning of multithreading in Java. Also, you have learned what the main thread, its purpose and when it is created in a Java program. Various states of threads are described in this chapter. This chapter also explained how threads are created using Thread class and Runnable interface. It explained how thread priority is used to determine which thread is to execute next.

Block-4

Introduction to GUI Programming

Unit 1: AWT Controls 1

Unit Structure

1.1 LEARNING OBJECTIVE

The objective of this unit is to make the students,

- To learn, understand various AWT Component and container hierarchy
- To learn, understand various container class and its methods
- To learn, understand and define various AWT components / controls and its methods

1.2 OUTCOMES

After learning the contents of this chapter, the students will be able to:

- Use container as per their requirement for GUI designing
- Use different AWT controls and its various methods in programs;

1.3 INTRODUCTION

Abstract Window Toolkit (AWT) is a application program interfaces (API's) to create graphical user interface (GUI).

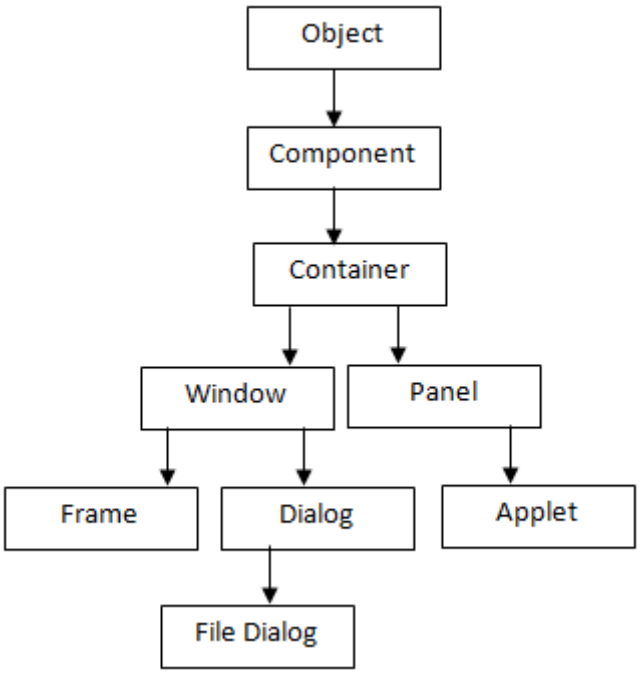


Figure-97 AWT Class Hierarchy

GUI contains objects like buttons, label, textField, scrollbars that can be added to containers like frames, panels and applets. AWT API is part of the Java Foundation Classes (JFC), a GUI class library. The AWT is contained in Java.awt package.

The Container is a component as it extends Component class. It inherits all the methods of Component class. Components can be added to the component i.e container.

As we can see in the above class hierarchy, Container is the super class of all the Java containers. The class signature is as follows:

public class Container extends Component

Controls are placed on the GUI by adding them to a container. A container is also a component. We can create and add these controls to the container without knowing anything about creating containers. Throughout this unit we will use Frame as a container for all of our controls. To add a control to a container, we need to:

1. First, create an object of the control

2. Second, after creating the control, add the control to the container.

The general form of add( ) method is:

add(Component compt)

compt is an instance of the control that we want to add. Once a control is added, it will automatically be visible whenever its parent container is displayed.

Sometimes, we need to remove a control from the container then, remove( ) method helps us to do. This method is defined by Container class.

void remove(Component compt)

compt is the control we want to remove. We can remove all the controls from the container by calling removeAll( ) method.

1.4 AWT COMPONENTS

Now, we will learn about the basic User Interface components (controls) like labels, buttons, check boxes, choice menus, text fields etc.

1.4.1 FRAME

The AWT Frame is a top-level window which is used to hold other child components in it. Components such as a button, checkbox, radio button, menu, list, table etc. A Frame can have a Title Window with Minimize, Maximize and Close buttons. The default layout of the AWT Frame is BorderLayout. So, if we add components to a Frame without calling it's setLayout() method, these controls are automatically added to the center region using BorderLayout manager.

- **Constructor:**

public Frame(): This constructor allows us to create a Frame window without name.

public Frame(String name): This constructor allows us to create a Frame window with a specified name.

- **Method:**

public void add(Component compt): This method adds the component compt, to the container Frame.

public void setLayout(LayoutManager object): This method allows to set the layout of the components in a container, Frame.

public void remove(Component compt): This method allows to remove a component, compt, from the container Frame.

public void setSize(int widthPixel, int heightPixel): This method allows to set the size of the Frame in terms of pixels.

1.4.2 BUTTON

Buttons are used to fire events in a GUI application. The Button class is used to create buttons. The default layout for a container is flow layout. To create a button we will use one of the following constructors:

Button(): This constructor allows to create a button with no text label.

Button(String): This constructor allows to create a button with the given string as label.

When a button is pressed or clicked, an ActionEvent is fired and leads to implementation of the ActionListener interface.

Note: The Layout Manager helps to organize controls on the container. It is discussed in next unit.

Example:

```
import java.awt.*;
public class buttonTest extends Frame
{
        Button first, second, third;
        buttonTest(String str)
        {
                super(str);
                setLayout(new FlowLayout());
                first = new Button("BAOU");
                second = new Button("MCA");
                third = new Button("GVP");
                add(first);
                add(second);
                add(third);
        }
        public static void main(String arg[])
        {
                Frame frm=new buttonTest("AWT Button");
                frm.setSize(250,250);
                frm.setVisible(true);
        }
}
```

Output:

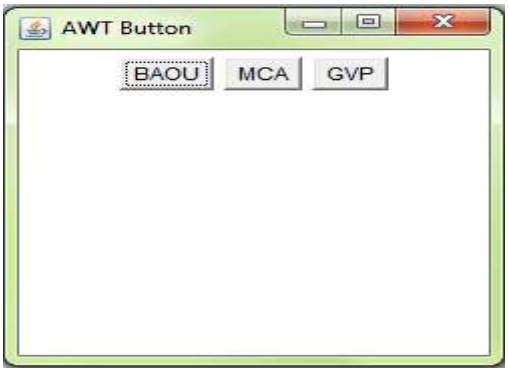


Figure-98 Output of program

1.4.3 LABEL

Labels can be created using the Label class. Labels are basically used to caption the components on a given interface. Label cannot be modified directly by the user. To create a Label we will use one of the following constructors:

Label( ): This constructor allows to create a label with its string aligned to the left.

Label(String): This constructor allows to create a label initialized with the specified string, and aligned to the left.

Label(String, int): This constructor allows to create a label with specified text and alignment. Alignment may be Label.Right, Label.Left and Label.Center.

getText( ) and setText() method is used to retrieve the label text and set the text of the label respectively.

Example:

```
import java.awt.*;
public class labelTest extends Frame
{
        labelTest(String str) {
                super(str);
                setLayout(new FlowLayout());
                Label one = new Label("BAOU");
                Label two = new Label("MCA");
                Label three = new Label("GVP");
                // add labels to Frame
                add(one);
                add(two);
                add(three);
        }
        public static void main(String arg[]){
                Frame frm=new labelTest("AWT Label");
```

```
                frm.setSize(250,200);
                frm.setVisible(true);
        }
}
```

Output:

Figure-99 Output of program

The output from the LabelTest program shows that the labels are arranged as we have added to the container.

1.4.4 CHECKBOX

Check Boxes are the controls allowing the user to select multiple selections from the given choice. For example, if a user wants to specify hobbies then CheckBox is the best control to use. It can be either “Checked” or “UnChecked”.

Check boxes are created using the Checkbox class. To create a check box we can use one of the following constructors:

Checkbox():This constructor allows to create an unlabeled checkbox that is not checked.

Checkbox(String): This constructor allows to create an unchecked checkbox with the given label as its string.

We can use the setState(boolean) method to set the status of the Checkbox. We can specify a true as argument for checked checkboxes and false for unchecked checkboxes. To get the current state of a check box, we can call boolean getState( ) method.

When a check box is selected or deselected, an ItemEvent is fired and leads to implementation of the ItemListener interface.

Example:

```
import java.awt.*;
```

```
public class checkBoxTest extends Frame
{
        Checkbox MCA, BCA, MscIT, Bsc;
        checkBoxTest(String str)
        {
                super(str);
                setLayout( new FlowLayout());
                MCA = new Checkbox("BAOU", null, true);
                BCA = new Checkbox("GVP");
                MscIT = new Checkbox("MCA");
                Bsc = new Checkbox("PGDCA");
                add(MCA);
                add(BCA);
                add(MscIT);
                add(Bsc);
        }
public static void main(String arg[])
 {
          Frame frm=new checkBoxTest("AWT CheckBox");
          frm.setSize(300,200);
          frm.setVisible(true);
   }
}
```

Output:

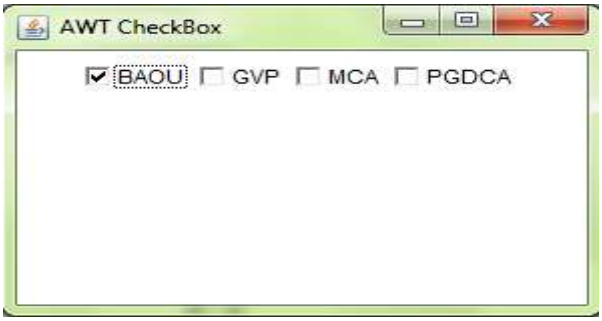


Figure-100 Output of program

As we can see in the above output window that the first BAOU checkbox displayed checked while others are unchecked.

1.4.5 CHECKBOXGROUP

CheckboxGroup is also known as a radio button or exclusive check boxes. Check Boxes group allows the user to select single choice from the given choice. For example, if a user wants to specify gender (Male / Female) then CheckboxGroup is the best choice. It can be either “Checked” or “UnChecked”.

We can create CheckboxGroup object as follows:

CheckboxGroup cbg = new CheckboxGroup ();

To create radio button, we have to use this object as an extra argument to the Checkbox constructor. For example,

Checkbox (String, CheckboxGroup, Boolean): It will allow us to create a checkbox with the given string that belongs to the CheckboxGroup specified in the second argument. If the last argument is true then the radio button will be checked and false otherwise.

We can determine currently selected check box in a group by calling getSelectedCheckbox( ) method as follows:.

Checkbox getSelectedCheckbox( )

We can set a check box by calling setSelectedCheckbox( ) method as follows:

void setSelectedCheckbox(Checkbox cb)

Here, cb is the check box that we want to be selected and at the same time previously selected check box will be turned off.

Example:

```
import java.awt.*;
public class ChBoxGroup extends Frame
{
    Checkbox mca, mba, mbbs, msc;
    CheckboxGroup cbg;
    ChBoxGroup(String str)
    {
        super(str);
        setLayout(new FlowLayout());
        cbg = new CheckboxGroup();
        mca = new Checkbox("MCA", cbg, false);
        mba = new Checkbox("MBA", cbg, false);
        mbbs= new Checkbox("MBBS", cbg, true);
        msc = new Checkbox("MSc", cbg, false);
        add(mca);
        add(mba);
        add(mbbs);
```

```
                add(msc);
        }
        public static void main(String arg[])
        {
          Frame frm=new ChBoxGroup("AWT CheckboxGroup");
          frm.setSize(300,200);
          frm.setVisible(true);
    }
}
```

Output:

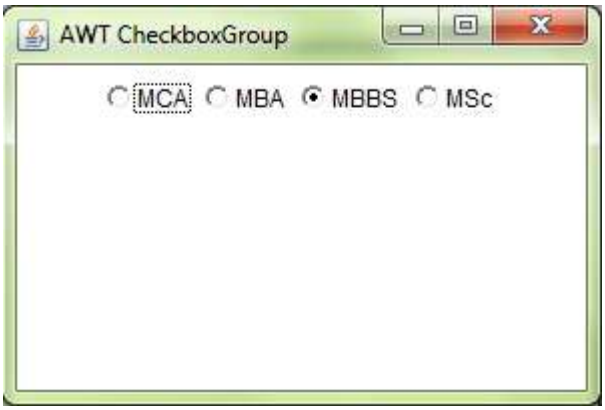


Figure-101 Output of program

The output generated by the ChBoxGroup is shown above. Note that the check boxes are now displayed in circular shape.

1.4.6 CHOICE

Choice control is created from the Choice class. This component enables a single item to be selected from a drop-down list. We can create a choice control to hold the list, as shown below:

Choice city = new Choice():

Items are added to the Choice control by using addItem(String) method. The following code adds three items to the city choice control.

city.addItem(“Ahmedbad”);

city.addItem(“Vadodara”);

city.addItem(“Surat”);

After adding the items to the Choice, it is added to the container like any other control using the add() method. The following example shows a Frame that contains a list of subjects in a MSc IT course.

To get the item currently selected, we may call either getSelectedItem() or getSelectedIndex() methods as shown here:

String getSelectedItem( )

int getSelectedIndex( )

The getSelectedItem() method will return a string containing the name of the item. While getSelectedIndex( ) will return the index of the item. The first item will be at index 0. By default, the selected item will be the first item. To get the number of items in the list we can call getItemCount() method. We can get the name associated with the item at the specified index by calling getItem( ) method as shown here:

String getItem(int index)

When a choice is selected, an ItemEvent is generated and leads to implementation of the ItemListener interface.

Example:

```
import java.awt.*;
public class choiceTest extends Frame
{
        Choice master, bachelor;
        choiceTest(String str)
        {
                super(str);
                setLayout(new FlowLayout());
                master = new Choice();
                bachelor = new Choice();
                master.add("MCA");
                master.add("MBA");
                master.add("MBBS");
                master.add("MSc");
                bachelor.add("BCA");
                bachelor.add("BBA");
                bachelor.add("BSc");
           add(master);
                add(bachelor);
        }
        public static void main(String arg[])
        {
          Frame frm=new choiceTest("AWT Choice");
          frm.setSize(300,200);
```

```
        frm.setVisible(true);
    }
}
```

Output:

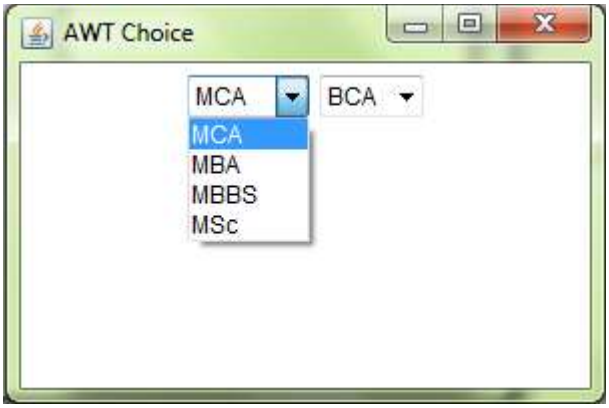


Figure-102 Output of program

The output generated by the above program shows two choice control named Master and Bachelor.

1.4.7 TEXTFIELD

TextField is a subclass of TextComponent class. This control allows user to provide textual data through GUI. AWT provides two classes to accept the user input, i.e TextField and TextArea. The TextField allows a single line of text to be entered and does not have scrollbars. TextField control allows us to enter the text and edit the text. To create a text field one of the following constructors are used:

TextField(): This constructor allows to create an empty TextField with no specified width.

TextField(String): This constructor allows to create a text field initialized with the given string.

TextField(String, int): This constructor allows to create a text field with specified text and specified width.

For example, the following line creates a text field 25 characters wide with the specified string:

TextField txtName = new TextField (“BAOU”, 15);

add(txtName);

To get the string contained in the text field, call getText() method. To set the text, call setText( ) method as follows:

String getText( )

void setText(String str)

setEditable(boolean ed): If ed is true, the text field may be modified. If it is false, the text cannot be modified.

Boolean isEditable(): This method returns true if the text in text filed may be changed and false otherwise.

Example:

```
import java.awt.*;
public class txtFieldTest extends Frame
{
        TextField txtname, txtpass;
        txtFieldTest(String str)
        {
                super(str);
                setLayout(new FlowLayout());
                Label name = new Label("Name: ", Label.RIGHT);
                Label pass = new Label("Password: ", Label.RIGHT);
                txtname = new TextField(12);
                txtpass = new TextField(8);
                txtpass.setEchoChar('*');
                add(name);
                add(txtname);
                add(pass);
                add(txtpass);
        }
public static void main(String arg[])
 {
          Frame frm=new txtFieldTest("AWT TextField");
          frm.setSize(250,200);
          frm.setVisible(true);
   }
}
```

Output:

Figure-103 Output of program

1.4.8 TextArea

The TextArea control allows us to enter more than one line of text. TextArea control have horizontal and vertical scrollbars to scroll through the text. We can use one of the following constructors to create a text area:

TextArea(): creates an empty text area with unspecified width and height.

TextArea(int, int): creates an empty text area with indicated number of lines and specified width in characters.

TextArea(String): This constructor allows to create a text area with the specified string.

TextArea(String, int, int): This constructor allows to create a text area containing the specified text and specified number of lines and width in the characters.

TextArea is a subclass of TextComponent so it inherits the getText(), setText(), getSelectedText(), select(), isEditable() and setEditable() methods.

TextArea class supports two more methods as follows:

insertText(String, int): It is used to insert specified strings at the character index specified by the second argument.

replaceText(String, int, int): It is used to replace text between given integer position specified by second and third argument with the specified string.

void append(String str): This append( ) method appends the string specified by str at the end of the current text.

Example:

```
import java.awt.*;
public class txtAreaTest extends Frame
{
        txtAreaTest(String str)
        {
                super(str);
                setLayout(new FlowLayout());
```

```
                String val ="Baba Saheb Ambedkar Open University and Gujarat Vidyapith";
                TextArea text = new TextArea(val, 10, 30);
                add(text);
        }
        public static void main(String arg[])
        {
          Frame frm=new txtAreaTest("AWT TextArea");
          frm.setSize(250,200);
          frm.setVisible(true);
    }
}
```

Output:

Figure-104 Output of program

In the above output we can see that scrollbar allows us to scroll through the textarea.

1.4.9 SCROLL BAR

Scroll bar controls are used to select values between a specified minimum and maximum. Scroll bars may be horizontal or vertical. The current value of the scroll bar relative to its minimum and maximum values will be specified by the slider box. The slider box can be dragged by the user to a new position. Scroll bar controls are encapsulated by the Scrollbar class. Scrollbar constructors are:

Scrollbar( ) : This will allow us to create a vertical scroll bar.

Scrollbar(int style)

Scrollbar(int style, int initialValue, int thumbSize, int minVal, int maxVal)

The second and third constructor will allow us to provide the orientation of the scroll bar. The style may be Scrollbar.VERTICAL or Scrollbar.HORIZONTAL. The initial value of the scroll bar will be specified by initialValue. The number of units

represented by the height of the thumb is specified by thumbSize. The minimum and maximum values for the scroll bar are specified by minVal and maxVal.

If we construct a scroll bar by one of the first two constructors, then we need to provide its parameters by using setValues( ) method as shown here:

void setValues(int initialValue, int thumbSize, int min, int max)

To get the current value of the scroll bar we can call getValue() method. It will return the current setting. To set the current value we can call setValue() method as follows:

int getValue( )

void setValue(int newValue)

We can also get the minimum and maximum values by getMinimum( ) and getMaximum( ) methods as shown here:

int getMinimum( ) and int getMaximum( )

They return the requested quantity. To handle scroll bar events, we need to implement the AdjustmentListener interface.

Example:

```
import java.awt.*;
class scrollBarTest extends Frame
{
        scrollBarTest(String str)
        {
        super(str);
        setLayout(new FlowLayout());
        //Horizontal Scrollbar with min value 0,max value 200,initial value 50 and visible amount 10
        Label Horzlbl =new Label("Horizontal Scrollbar");
        Scrollbar hzsb = new Scrollbar(Scrollbar.HORIZONTAL,50,10,0,200);
        //Vertical Scrollbar with min value 0,max value 255,initial value 10 and visible amount 5
        Label vertlbl =new Label("Vertical Scrollbar");
        Scrollbar vtsb = new Scrollbar(Scrollbar.VERTICAL,30,15,0,255);
        add(Horzlbl);
        add(hzsb);
        add(vertlbl);
        add(vtsb);
    }
public static void main(String arg[])
```

```
 {
        Frame frm=new scrollBarTest("AWT Scrollbar");
        frm.setSize(250,200);
        frm.setVisible(true);
  }
}
```

Output:

Figure-105 Output of program

1.4.10 LISTS

The List class provides us a compact, multiple-choice and scrolling selection list. A List control allows us to show any number of choices in the visible window compare to a choice object, which shows only the single selected item in the menu. It also allows multiple selections. List constructors are:

List( ): This constructor allows us to create a List control that will allow only one item to be selected at any one time.

List(int numRows): In this constructor, the value of numRows specifies the number of items from the list will always be visible

List(int numRows, boolean multiSelect): In this constructor, if multiSelect is true, then the user can select two or more items at a time. If it is false, then only one item can be selected.

To add a selection to the list we have to call add( ) method as follows:

void add(String name)

void add(String name, int index)

In both the forms, name is the name of the item added to the list. The first constructor will add items to the end of the list. The second constructor will add the item at the index specified by index.

The getSelectedItem( ) method will return a string containing the name of the item selected. In case of more item is selected or no selection has been made then null will be returned. getSelectedIndex( ) method will return the index of the item

selected. In case of more item is selected or no selection has been made then –1 will be returned.

We must use either getSelectedItems() or getSelectedIndexes( ) methods for lists allowing multiple selection as shown here:

String[ ] getSelectedItems()

int[ ] getSelectedIndexes( )

To get the number of items in the list, call getItemCount( ) method as shown here:

int getItemCount( )

We can obtain the name associated with the item at the specified index by calling getItem( ) method.

String getItem(int index)

To handle the list events, we need to implement the ActionListener interface. When a List item is double-clicked, an ActionEvent object is generated. When an item is selected or deselected with a single click, an ItemEvent object is generated. Following example shows one multiple choice and the other single choice:

Example:

```
import java.awt.*;
public class ListTest extends Frame {
        List master, bachelor;
        ListTest(String str)   {
                super(str);
                setLayout(new FlowLayout());
                master = new List(13, true);
                bachelor = new List(13, false);
                master.add("MCA");
                master.add("MBA");
                master.add("MBBS");
                master.add("MSc");
                bachelor.add("BCA");
                bachelor.add("BBA");
                bachelor.add("BSc");
                 bachelor.select(1);
                //add lists to Frame
                add(master);
                add(bachelor);
        }
        public static void main(String arg[])
```

```
        {
          Frame frm=new ListTest("AWT List");
          frm.setSize(1300,200);
          frm.setVisible(true);
    }
}
```

Output:

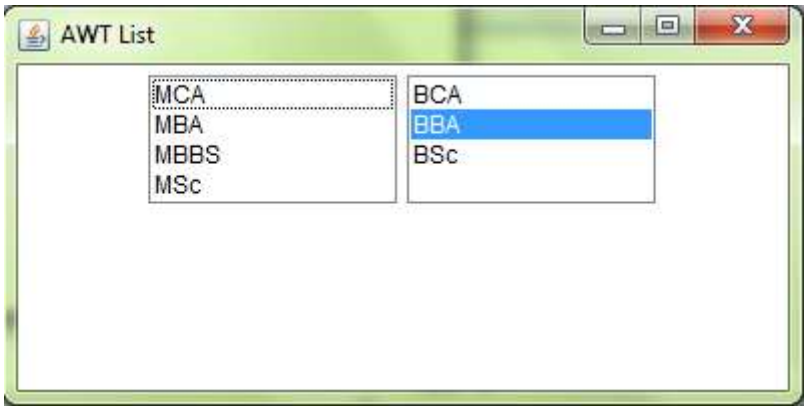


Figure-106 Output of program

As we can see in the output that in second list first index value is selected.

1.4.11 MENU

Menus are mostly used in Windows that contains a list of menu items. When we click on the MenuItem it generates ActionEvent and is handled by ActionListener. AWT Menu and MenuItem are not components as they are not subclasses of java.awt.Component class. They are derived from MenuComponent class. Creation of Menu requires lot of classes like MenuBar, Menu and MenuItem and one is required to be added to the other. The following image depicts Menu hierarchy.

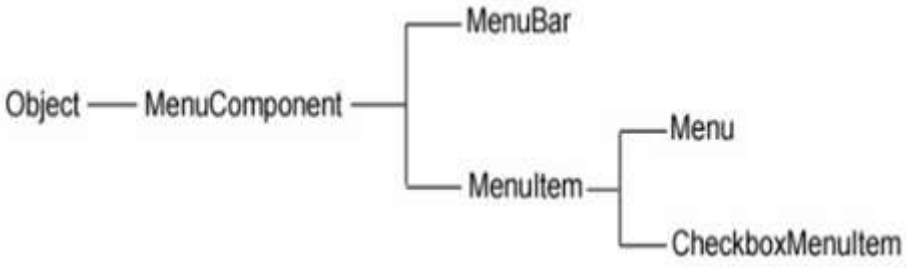


Figure-107 Hierarchy of menu

MenuComponent class is the super most class of all the menu classes same as Component is the super most class for all component classes like Button, choice, Frame etc. MenuBar will hold the menus and Menu will hold menu items. Menus will be placed on menu bar. The following steps will be executed to create AWT Menu.

1. Create menu bar
2. Add (set) menu bar to the frame

3. Create menus
4. Add created menus to menu bar
5. Create menu items
6. Add created menu items to menus
7. At last, if required then handle events

Example:

```
import java.awt.*;
import java.lang.*;
import java.util.*;

public class menuTest extends Frame {
  MenuBar  mbar;
  Menu     file, help;
  MenuItem op, os, pr, sa, mc;
  Label   msg = new Label("Select an option from menu");
  menuTest(String str)
  {
    super(str);
   setLayout(new BorderLayout());
    add("Center", msg);
    mbar = new MenuBar();

    mbar.add(file = new Menu("File"));
    mbar.add(help = new Menu("Help"));
    mbar.setHelpMenu(help);

    file.add(op = new MenuItem("Open"));
    file.add(os = new MenuItem("Save"));
    file.addSeparator();
    file.add(pr = new MenuItem("Print"));

    help.add(sa = new MenuItem("Save As"));
    help.add(mc = new MenuItem("close"));

          setMenuBar(mbar);
  }
```

```
  public static void main(String arg[])   {
            Frame frm=new menuTest("MenuBar");
            frm.setSize(200,200);
            frm.setVisible(true);
    }
}
```

Output:

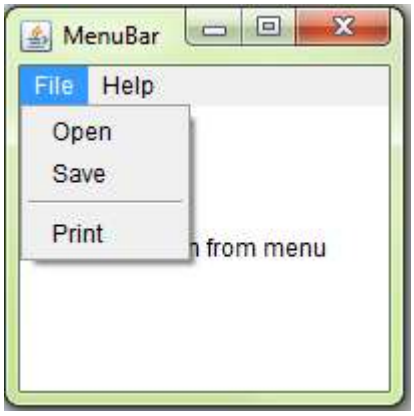


Figure-108 Output of program

1.4.12 CANVAS

The Canvas control is a blank rectangular shape where the application allows us to draw. It inherits the Component class. Canvas is a class from java.awt package on which a user can draw some shapes or display images. A button click or a keyboard key press on the canvas can fire events and these events can be transferred into drawings. The class signature of canvas is as follows:

public class Canvas extends Component implements Accessible

Drawing Oval on Canvas

In the following simple canvas code, a canvas is created and a oval is drawn on it.

Example:

```
import java.awt.*;
public class canvasDraw extends Frame
{
  public canvasDraw(String str)
  {
          super(str);
    CanvasTest ct = new CanvasTest();
    ct.setSize(125, 100);
    ct.setBackground(Color.cyan);
```

```
    add(ct, "North");

    setSize(300, 200);
    setVisible(true);
  }
  public static void main(String args[])
  {
    new canvasDraw("AWT Canvas");
  }
}
class CanvasTest extends Canvas
{
  public void paint(Graphics g)
  {
    g.setColor(Color.blue);
    g.fillRect(65, 5, 1135, 65);
  }
}
```

Output:

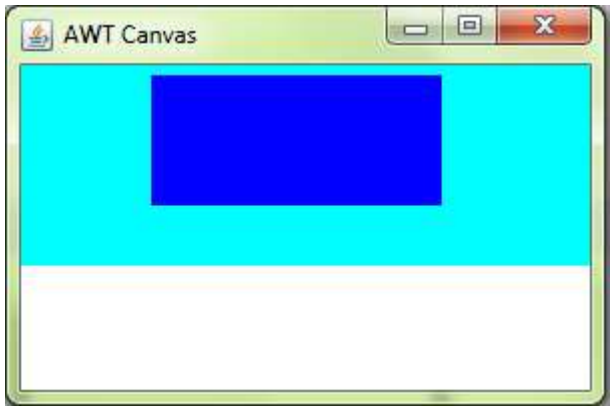


Figure-109 Output of program

In the above program, our class extends the java.awt.Canvas class. Here, CanvasTest extends Canvas. The main class is canvasDraw extends Frame. CanvasTest object is created and added to the frame on North side. Canvas is colored cyan just for identification. The object of Canvas is tied to a frame to draw painting. On the canvas, rectangle object is filled with blue color.

1.4.13 PANEL

Panel class is the simple container class. A panel class provides an area in which an application can contain any other component including other panels. The signature of Panel class is as follows:

public class Panel extends Container

The default layout manager for a panel class is the FlowLayout layout manager and can be changed as per the requirement of the layout. Being the subclass of both Component and Container class, a panel is both a component and a container. As a component it can be added to another container and as a container it can be added with components. It is also known as a child window so it does not have a border.

In the following program, three buttons are added to the north (top) of the frame and three buttons to the south (bottom) of the frame. Without panels, this arrangement is not possible with mere layout managers.

Example:

```
import java.awt.*;
public class PanelTest extends Frame
{
   public PanelTest(String str)
   {
      super(str);
            setLayout(new BorderLayout());

      Panel p1 = new Panel();
      Panel p2 = new Panel();

      p1.setBackground(Color.cyan);
      p2.setLayout(new GridLayout(1, 3, 20, 0));

      Button b1 = new Button("BAOU");
      Button b2 = new Button("GVP");
      Button b3 = new Button("MCA");
      Button b13 = new Button("BCA");
      Button b5 = new Button("MBA");
      Button b6 = new Button("BBA");

      p1.add(b1);
      p1.add(b2);
      p1.add(b3);
```

```
      p2.add(b13);
      p2.add(b5);
      p2.add(b6);

      add(p1, "North");
      add(p2, "South");
         }
    public static void main(String args[])
    {
      Frame fm=new PanelTest("AWT Panel");
           fm.setSize(300, 200);
      fm.setVisible(true);
    }
}
```

Output:

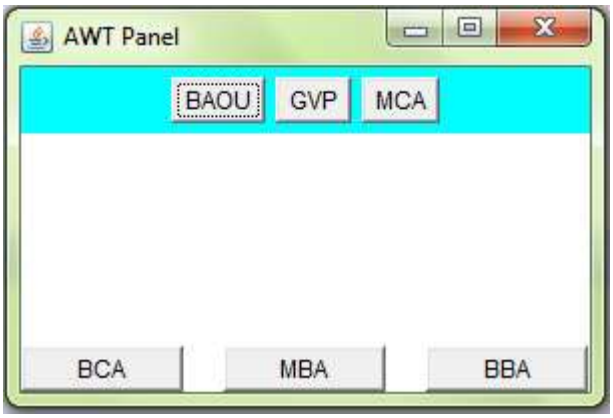


Figure-110 Output of program

1.5 LET US SUM UP

At last, AWT is the bunch of component and containers allowing users for different options to set on their GUI. These components can be created by instantiating their class and making them visualize on the container like Frame, window or Panel. Component will be like buttons, choice, text fields etc. Once these controls are added to the GUI user can interact with them through Event handling. Event Handling is covered in the next sections.

1.6 FURTHER READING

1) Java: The Complete Reference by Schildt Herbert. Ninth Edition
2) Let us Java by Yashavant Kanetkar. 3rd Edition
3) Head First Java: A Brain-Friendly Guide, Kindle Edition by Kathy Sierra, Bert Bates. 2nd
4) Edition
5) https://fresh2refresh.com/java-tutorial/
6) https://www.studytonight.com/java/

1.7 ASSIGNMENTS

1) Define AWT. List various component and containers of AWT.
2) Why AWT Components are known as heavy weight components?
3) What is the difference between Panel and Frame?
4) Discuss any three methods of Checkbox and TextField class.
5) Write a program to design personal information form with the help of AWT controls.

Unit 2: Event Delegation Model 2

Unit Structure

2.1 LEARNING OBJECTIVE

The objective of this unit is to make the students,

- To learn, understand Event
- To learn, understand Event Source and Event Handlers
- To learn, understand and define different Event and Listeners for various kinds of Events
- To learn, understand adapter classes and their importance

2.2 OUTCOMES

After learning the contents of this chapter, the students will be able to:

- Define different events
- Write event source and event handlers to handle the events
- Adapter classes when they are in need of few methods instead of all methods of handlers

2.3 INTRODUCTION

Java provides the platform to develop interactive GUI application using the AWT and Event classes. This unit discusses various event classes for handling various events like button click, checkbox selection etc. We can define an event as the change in the state of an object when something changes within a graphical user interface. If a user check or uncheck redio button, clicks on a button, or write characters into a text field etc. then an event trigger and creates the relevant event object. This mechanism is a part of Java's Event Delegation Model

2.4 EVENT DELEGATION MODEL

The Event Delegation Model is based on the concept of source and listener. A source triggers an event and sends it to one or more registered listeners. On receiving the event notification, listener processes the event and returns it. The important feature is that the source has a list of registered listeners which will be informed as and when event take place. Only the registered listeners will actually receive the notification when a specific event is generated. Generally the event Handling is a three step process:

a. Create controls which can generate events (Event Generators).

b. Build objects that can handle events (Event Handlers).

c. Register event handlers with event generators.

2.4.1 EVENT GENERATORS

It is an object that is responsible to generate a particular kind of event. An event is generated when the internal state of an object is changed. A source may trigger more than one kind of event. Every source must register a list of listeners that are

interested to receive the notifications when an event is generated. Event source has methods to add or remove listeners.

To register (add) a listener the signature of method is:

public void addNameListener(NameListener eventlistener)

To unregister (remove) a listener the signature of method is:

public void removeNameListener(NameListener eventlistener)

where,

Name is the name of the event and eventlistener is a reference to the event listener.

2.4.2 EVENT LISTENER

An event listener is an object which receives notification when an event is triggered. As already said only registered listeners will receive notifications from the event sources about specific kinds of events. The event listener is responsible to receive these notifications and process them. Technically these listeners are interfaces having various abstract methods for event handling. These interfaces needs to be implemented in the class where the object or source will trigger the event.

For example, consider an action event represented by the class ActionEvent, which is triggered when a user clicks a button or the item of a list. At the user's interaction, an ActionEvent object related to the relevant action is created. This object will contain both the event source and the specific action taken by the user. This event object is then passed to the related ActionListener object's method:

void actionPerformed(ActionEvent e)

This method will be executed and returns the appropriate response to the user.

2.4.3 REGISTRATION OF LISTENER FOR EVENTS

As we know, we have implemented the interface and set up the methods which will listen for these events and trigger the functionality accordingly. To perform this, we have to use an event listener. To use an event listener the addActionListener() method will be used on the component that will listen for these events - the button.

button.addWindowListener(this);

2.5 TYPES OF EVENTS

There are various types of events that can happen in a Java program. They are,

Event Class / Type	Generated when	Listener Interface	Methods to implement
Action event	Button is pressed	ActionListener	actionPerformed()
Adjustment event	Scroll bar is manipulated	AdjustmentListener	adjustmentValueChanged()

Component event	A control is hidden, moved, resized, or shown	ComponentListener	componentHidden(), componentMoved(), componentResized(), componentShown()
Container event	A control is added or removed from a container	ContainerListener	componentAdded(), componentRemoved()
Focus event	A control gains or loses focus	FocusListener	focusGained(), focusLost()
Item event	An item is selected or deselected	ItemListener	itemStateChanged()
Key event	A key is pressed, released or typed	KeyListener	keyPressed(), keyReleased(), keyTyped()
Mouse event	Mouse is clicked, pressed or released. Mouse pointer enters, leaves a component	MouseListener	mouseClicked(), mouseEntered(), mouseExited(), mousePressed(), mouseReleased()
Mouse event	Mouse is dragged or moved	MouseMotionListener	mouseDragged(), mouseMoved()
Text event	Text value is changed	TextListener	textValueChanged()
Window event	A window is activated, closed, deactivated, deiconfied, opened or quit	WindowListener	windowActivated(), windowClosed(), windowClosing(), windowDeactivated(), windowDeiconified(), windowIconified(), windowOpened()

Table-10: Event classes and their methods

Each interface has their own methods to use to execute some code when certain events occur. For example, the ActionListener interface has a actionPerformed method that can be used to execute some code when a button is clicked. When we implement an interface, we have to define all of it's abstract methods in the program.

Let's check a simple example that will have window events.

```
import java.awt.*;
import java.awt.event.*;
public class winEvents extends Frame implements WindowListener{
}
```

Now we need to implement the methods of the WindowListener interface to specify what happens during window events.

```
//Window event methods
    public void windowClosing(WindowEvent we)
    { System.out.println("The frame is closing"); }
    public void windowClosed(WindowEvent we)
    { System.out.println("The frame is closed"); }
    public void windowDeactivated(WindowEvent we)
    { System.out.println("The frame is deactivated"); }
```

Example:

In the below program, a frame utilizes all the window event methods. See how the program displays different messages as we perform different actions such as minimize and maximize on the frame.

```
import java.awt.*;
import java.awt.event.*;
public class WinEvents extends Frame implements WindowListener
{
  public WinEvents(String str){
    super(str);
   addWindowListener(this);
  }
  public static void main(String[] args){
    Frame fm = new WinEvents("WindowEvent_Example");
    fm.setSize(250, 250);
    fm.setVisible(true);
  }
  public void windowClosing(WindowEvent we){
    System.out.println("The window is closing.....");
```

```
    ((Window)we.getSource()).dispose();
  }
  public void windowClosed(WindowEvent we){
    System.out.println("The window has been closed!");
    System.exit(0);
  }
  public void windowActivated(WindowEvent we){
    System.out.println("The window has been activated");
  }
  public void windowDeactivated(WindowEvent we){
    System.out.println("The window has been deactivated");
  }
  public void windowDeiconified(WindowEvent we){
    System.out.println("The window has been restored from a minimized state");
  }
  public void windowIconified(WindowEvent we){
    System.out.println("The window has been minimized");
  }
  public void windowOpened(WindowEvent we){
    System.out.println("The window is now visible");
  }
}
```

Output: When we perform different operation on window following output will be displayed.

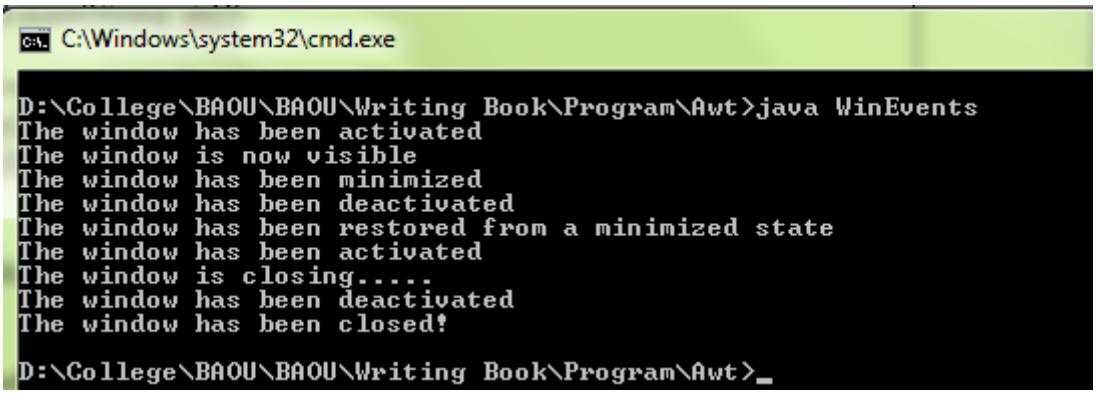


Figure-111: Output of program

After discussing all the window event methods, let us check the key events. The following program depicts the use of KeyListener to handle different key events.

```
import java.awt.BorderLayout;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.Frame;
import java.awt.TextField;
import java.awt.TextArea;
import java.awt.Label;
public class keyListenerTest extends Frame implements KeyListener
{
        TextArea text;
        TextField txtF;
        Label l1;
  keyListenerTest(String str)
  {
          super(str);
          setLayout(null);
          l1=new Label("Enter Key:");
           l1.setBounds(50,50,100,30);
          txtF= new TextField();
          text = new TextArea();
          txtF.addKeyListener(this);
          txtF.setBounds(160,50,100,30);
           text.setBounds(20,100,300,300);
          add(l1);
          add(txtF);
          add(text);
  }
        public void keyPressed(KeyEvent ke)
        {
                text.append("Key is Pressed\n");
        }
        public void keyReleased(KeyEvent ke)
        {
                text.append("Key is Released\n");
        }
```

```
        public void keyTyped(KeyEvent ke)
        {
                text.append("Key is Typed\n");
        }
        public static void main(String args[])
        {
                Frame frame = new keyListenerTest("KeyListener");
                 frame.setSize(350,1400);
                frame.setVisible(true);
        }
}
```

Output:

Figure-112: Output of program

2.6 ADAPTER CLASSES

We have seen that handler class need to implement interface therefore it has to provide implementation of all the methods of that interface. Suppose, an interface has 10 methods, then hander class has to provide implementation of all these 10 methods. Even if the requirement is for one method, class has to provide empty implementation of the remaining 9 methods with null bodies. This becomes a irritating job for a programmer.

Adapter classes are classes that implement all of the methods in their corresponding interfaces with null bodies. If the programmer needs one of the methods of particular interface then he / she can extend an adapter class and override its methods. They belongs to java.awt.event package.

There is an adapter class for listener interfaces having more than one event handling methods. For example, for WindowListener there is a WindowAdapter class and for MouseMotionListener there is a MouseMotionAdapter class and many more.

Adapter classes provide definitions for all the methods (empty bodies) of their corresponding Listener interface. It means that WindowAdapter class implements

WindowListener interface and provide the definition of all methods inside that Listener interface. Consider the following example of WindowAdapter and its corresponding WindowListener interface:

```
public interface WindowListener{
        public void windowOpened ( WindowEvent e )
        public void windowIconified ( WindowEvent e )
        public void windowDeiconified ( WindowEvent e )
        public void windowClosed ( WindowEvent e )
        public void windowActivated ( WindowEvent e )
        public void windowDeactivated ( WindowEvent e )
}
public class WindowAdapter implements WindowListener {
        public void windowOpened ( WindowEvent e ) {}
        public void windowIconified ( WindowEvent e ) {}
        public void windowDeiconified ( WindowEvent e ) {}
        public void windowClosed ( WindowEvent e ) {}
        public void windowActivated ( WindowEvent e ) {}
        public void windowDeactivated ( WindowEvent e ) {}
}
```

Now in the below class WinEvents, if we extend the above handler class then due to inheritance, all the methods of the adapter class will be available inside handler class as adapter classes has already provided implementation with empty bodies. So, we only need to override and provide implementation of method of our interest.

```
public class WinEvents extends WindowAdapter{...}
```

Example: Following program demonstrates the use of WindowAdapter class.

```
import java.awt.*;
import java.awt.event.*;
public class adapterTest extends Frame
{
        Label lblTest;
        adapterTest(String str)
        {
                super(str);
                setLayout(new FlowLayout(FlowLayout.LEFT));
                lblTest = new Label();
```

```
                add(lblTest);
                addMouseListener(new MyAdapter(lblTest));
        }
        public static void main(String str[])          {
                Frame at=new adapterTest("AdapterClass");
                at.setSize(250,250);
                at.setVisible(true);
        }
}
class MyAdapter extends MouseAdapter {
        Label lblTest;
        MyAdapter(Label lbl)
        {
                lblTest = lbl;
        }
        public void mouseClicked(MouseEvent me)
        {
                lblTest.setText("Mouse is Clicked");
        }
}
```

Output:

Figure-113: Output of program

The following example handles action event along with item event when a user clicks a button, check a checkbox or changes a value from choice.

Example:

```
import java.awt.*;
import java.awt.event.*;
public class AwtControl extends Frame
implements ActionListener, ItemListener {
        Button button;
        Checkbox mca;
        Choice city;
        TextField TxtF,TxtF1,TxtF2;
        AwtControl(String str)          {
           super(str);
                setLayout(new FlowLayout());
                button = new Button("BAOU");
                mca = new Checkbox("MCA");
                city = new Choice();
                TxtF = new TextField(50);
                TxtF1 = new TextField(50);
                TxtF2 = new TextField(50);
                button.addActionListener(this);
                mca.addItemListener(this);
                city.addItemListener(this);
                city.addItem("Ahmedabad");
                city.addItem("Sadra");
                city.addItem("Randheja");

                add(button);
                add(mca);
                add(city);
                add(TxtF); add(TxtF1);add(TxtF2);

        }

        public void actionPerformed(ActionEvent e)      {
                String action = e.getActionCommand();
                if(action.equals("BAOU"))
```

```
                {
                        TxtF.setText("BAOU is in Ahmedabad");
                }
        }
        public void itemStateChanged(ItemEvent e)      {
                if (e.getSource() == mca)
                {
                        TxtF1.setText("MCA at Gujarat Vidyapith: " + mca.getState()
+ ".");
                }
                else if (e.getSource() == city)
                {
                        TxtF2.setText(city.getSelectedItem() + " is selected.");
                }
        }
        public static void main(String arg[])
        {
                Frame frame = new AwtControl("AWT Event");
             frame.setSize(1400,200);
                  frame.setVisible(true);
        }
}
```

Output:

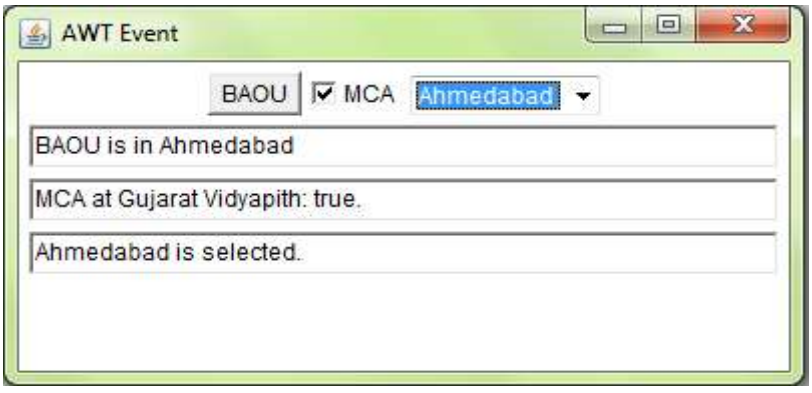


Figure-114: Output of program

2.7 LET US SUM UP

Throughout the unit, we saw that AWT provides various event classes and listener interfaces to fire and handle events triggered through user interaction. We have seen that how ActionEvent class helps to handle the event triggered by button

like controls. Same way, in the unit we have discussed various other Event classes and their respective listener for event handling. At last, we have discussed Adapter class to relieve the programmer from writing all the listener methods.

2.8 FURTHER READING

1) Java: The Complete Reference by Schildt Herbert. Ninth Edition
2) Let Us Java by Yashavant Kanetkar. 3rd Edition
3) Core Java Volume I — Fundamentals by Cay S. HorstMann, Gary Cornell, 9th Edition
4) https://www.javatpoint.com/

2.9 ASSIGNMENTS

1) Discuss Event delegation model with diagram in detail.
2) Explain different methods of KeyListener with its signature.
3) Write a program to implement a single key event with the help of adapter classes.

Unit 3: Graphics Class 3

Unit Structure

3.1 LEARNING OBJECTIVE

The objective of this unit is to make the students,

- To learn, understand and define graphics class and its methods
- To learn, understand and define the Font class and its methods
- To learn, understand and define the Color class and its methods
- To learn, understand the arrangement of AWT controls on the container
- To learn, understand different Layout and its parameters

3.2 OUTCOMES

After learning the contents of this chapter, the students will be able to:

- Use Graphics class and its various methods
- Use Font class and its various methods in programs
- Use Color class and its various methods in programs
- Write a graphical application using graphics, font and color class
- Use Layout Manager to arrange AWT components on the containers

3.3 INTRODUCTION

Java provides the platform to develop graphics based application using the Graphics class. This unit dicusses various java functionalities for painting shapes like rectangle, polygon etc. The unit covers the use of color and fonts. It also demonstrates the filling of object once it is drawn on the container. It also discusses various font family, its style to display the content on the container. It is essential to learn to beautify the components placed on the container area using Font and Color class. Withour the proper arrangement of control on the containers the GUI of the application looks jagged. So, it becomes very important for the programmer to arrangement the controls on the containers. Here, the Layout Manager comes. This unit also discusses different layout techniques to arrange components on the containers. It also covers various techniques to arrange the control manually using setBounds method.

3.4 GRAPHICS CLASS

In the AWT package, the Graphics class provides the foundation for all graphics operations. At one end the graphics context provides the information about drawing operations like the background and foreground colors, font and the location and dimensions of the region of a component. At the other end, the Graphics class provides methods for drawing simple shapes, text, and images at the destination.

To draw any object a program requires a valid graphics context in the form of instance of the Graphics class. Graphics class is an abstract base class, it cannot be instantiated. An instance is created by a component and handed over to the program as an argument to a component's update() and paint() methods. The update() and paint() method should be redefined to perform the desired graphics operations. There are various methods used for drawing different component on the containser. Thay are discussed below.

- **repaint() Method**

The repaint() method requests for a component to be repainted. This method has various forms as shown below:

1. public void repaint();
2. public void repaint(long tm) ; // Specify a period of time in milliseconds

Once a period of time is provided, the painting operation will occur before the time elapses.

3. public void repaint(int x, int y, int w, int h);

We can also provide that only a portion of a component be repainted. It is useful when the paint operation is time-consuming, and only a portion of the display needs to be repainted.

4. public void repaint(long tm, int x, int y, int w, int h);

- **public void update(Graphics g)**

The update() method is called in turn to a repaint() request. This method takes an instance of the Graphics class as an argument. The scope of graphics instance is valid only within the context of the update() method and the methods it calls. The default implementation of the Component class will erase the background and calls the paint() method.

- **public void paint(Graphics g)**

The paint() method is called from an update() method, and is responsible for drawing the graphics. It takes an instance of the Graphics class as an argument.

- **void drawLine(int xStart, int yStart, int xStop, int yStop)**

It draws a straight line, a single pixel wide, between the specified start and end points. The line will be drawn in the current foreground color. This methods works when invoked on a valid Graphics instance and used only within the scope of a component's update() and paint() methods.

- **Retangle**

Rectangle object can be drawn in different ways like,

1. void drawRect(int x, int y, int width, int height)
2. void fillRect(int x, int y, int width, int height)
3. void drawRoundRect(int x, int y, int width, int height, int arcwidth, int archeight)
4. void fillRoundRect(int x, int y, int width, int height, int arcwidth, int archeight)
5. void draw3DRect(int x, int y, int width, int height, boolean raised)
6. void fill3DRect(int x, int y, int width, int height, boolean raised)

All the method requires, the x and y coordinates as parameters to start the rectangle, and the width and height of the rectangle. The width and height must be positive values. Rectangles can be drawn in three different styles: plain, with rounded corners, and with a three-dimensional effect (rarely seen).

The RoundRect methods require an arc width and arc height to control the rounding of the corners. The 3 dimensional methods require an additional parameter that indicates whether or not the rectangle should be raised. These all method works when invoked on a valid Graphics instance and used only within the scope of a component's update() and paint() methods.

- **Ovals and Arcs**

Ovals and Arc object can be drawn in different ways like,

1. void drawOval(int x, int y, int width, int height)
2. void fillOval(int x, int y, int width, int height)
3. void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)
4. void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)

Each of this method requires, the x and y coordinates of the center of the oval or arc, and the width and height of the oval or arc. The width and height must be positive values. The arc methods require a start angle and an arc angle, to specify the beginning of the arc and the size of the arc in degrees.

This methods works when invoked on a valid Graphics instance and used only within the scope of a component's update() and paint() methods.

- **Polygons**

Polygon object can be drawn in different ways like,

1. void drawPolygon(int xPoints[], int yPoints[], int nPoints)
2. void drawPolygon(Polygon p)
3. void fillPolygon(int xPoints[], int yPoints[], int nPoints)
4. void fillPolygon(Polygon p)

Polygons object drawn from a sequence of line segments. Each of this method requires, the coordinates of the endpoints of the line segments that will make the polygon. These endpoints can be specified by first, the two parallel arrays of integers, one representing the x coordinates and the other representing the y coordinates; second is, using an instance of the Polygon class. The Polygon class provides the method addPoint(), which allows a polygon to be organized point by point. These methods works when invoked on a valid Graphics instance and used only within the scope of a component's update() and paint() methods.

3.4.1 COLOR CLASS

The java.awt.Color class provides 13 standard colors as constants. They are: RED, GREEN, BLUE, MAGENTA, CYAN, YELLOW, BLACK, WHITE, GRAY, DARK_GRAY, LIGHT_GRAY, ORANGE and PINK. Colors are created from red, green and blue components of RGB values. The range of RGB will be from 0 to 255 or floating point values from 0.0 to 1.0. We can use the toString() method to print the RGB values of these color (e.g., System.out.println(Color.RED)):

- **Methods**

To implement color in objects or text, two Color methods getColor() and setColor() are used. Method getColor() returns a Color object and setColor() method used to sets the current drawing color.

Now check below program to learn how these methods can be used.

Example:

```
import java.awt.Frame;
import java.awt.Panel;
import java.awt.Graphics;
import java.awt.Polygon;
import java.awt.Color;
public class PictureDraw extends Panel
{
        public void paint(Graphics g)
        {
                //Print a String message
                g.drawString("Welcome to BAOU", 20, 20);
                //draw a Line
                g.drawLine(0, 0, 100, 70);
                //draw a Oval
                g.drawOval(100, 100, 100, 100);
                //draw a rectangle
                g.drawRect(80, 80, 125, 125);
                //draw a Polygon
                int x[] = {35, 155, 35, 155, 35};
                 int y[] = {35, 35, 155, 155, 35};
                 g.drawPolygon(x,y,5);   //points = 5;

                g.setColor(Color.orange);
                Polygon pg = new Polygon();
                  pg.addPoint(220, 30);
                  pg.addPoint(300, 35);
                  pg.addPoint(320, 95);
                  pg.addPoint(275, 70);
                  pg.addPoint(210, 100);
                  pg.addPoint(180, 50);
                   g.drawPolygon(pg);
                g.fillPolygon(pg);
        }
```

```
        public static void main(String[] args)
        {
                Frame f= new Frame("Graphics Control");
                f.add(new PictureDraw());
                f.setSize(600, 1500);
                f.setVisible(true);
                f.setResizable(false);
        }
}
```

Output:

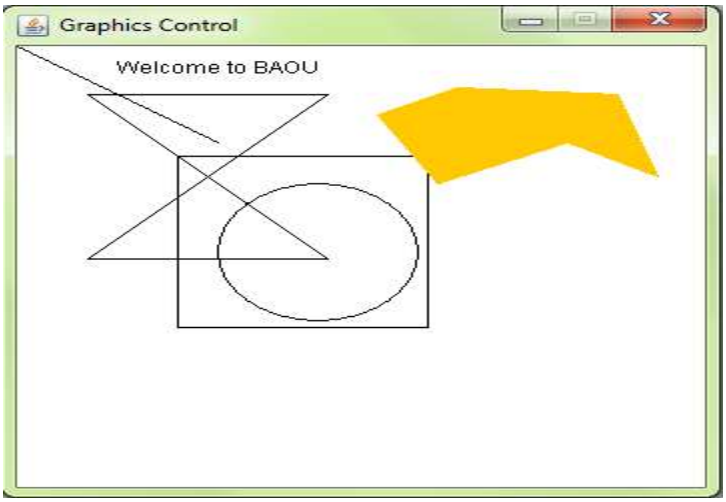


Figure-115: Output of program

3.4.2 FONT CLASS

The java.awt.Font class represents a method of specifying and using fonts. That font will be used to render the texts. The Font class constructor is used to construct a font object using the font's name, style (PLAIN, BOLD, ITALIC, or BOLD + ITALIC) and font size. In java, fonts are named in a platform independent fashion and then mapped to local fonts that are supported by the underlying operating system. The getName() method is used to return the logical Java font name of a particular font and the getFamily() method is used to return the operating system-specific name of the font. In java the standard font names are Courier, Helvetica, TimesRoman etc. There are 3 logical font names. Java will select a font name in the system that matches the general feature of the logical font.

I. Serif: This is often used for blocks of text (example, Times).

II. Sansserif: This is often used for titles (example, Arial or Helvetica).

III. Monospaced: This is often used for computer text (example, Courier).

The logical font family names are “Dialog”, “DialogInput”, “Monospaced”, “Serif”, or “SansSerif” and Physical font names are actual font libraries such as “Arial”, “Times New Roman” in the system. There is logical font names,

standard on all platforms and are mapped to actual fonts on a particular platform.

- **Constructor:**

public Font(String fontName, int fontStyle, int fontSize);

where, fontName represents Font Family name

fontStyle represents Font.PLAIN, Font.BOLD, Font.ITALIC or Font.BOLD or Font.ITALIC

fontSize represents the point size of the font (in pt) (1 inch has 72 pt).

The setFont() method to set the current font for the Graphics context g for rendering texts.

For example,

```
g.drawString("Welcome to BAOU", 15, 25);     // in default font
Font fontTest = new Font(Font.SANS_SERIF, Font.ITALIC, 15);
g.setFont(fontTest);
g.drawString("Gujarat Vidyapith", 10, 50);  // in fontTest
```

We can use GraphicsEnvironment's getAvailableFontFamilyNames() method to list all the font family names; and getAllFonts() method to construct all Font instances (font size of 1 pt).

For example,

```
GraphicsEnvironment fontEnv =
GraphicsEnvironment.getLocalGraphicsEnvironment();
String[] fontList = fontEnv.getAvailableFontFamilyNames();
for (int i = 0; i < fontList.length; i++)
{
    System.out.println(fontList [i]);
}
// Construct all Font instance (with font size of 1)
Font[] fontList = fontEnv.getAllFonts();
for (int i = 0; i < fontList.length; i++)
{
    System.out.print(fontList [i].getFontName() + " : ");
    System.out.print(fontList [i].getFamily() + " : ");
    System.out.print(fontList [i].getName());
}
```

Example:

Now check below program to learn how Font class and its method can be used.

```
import java.awt.Font;
import java.awt.Frame;
import java.awt.Panel;
import java.awt.Graphics;
public class FontClass extends Panel {
   public void paint(Graphics g)
        {
      Font f = new Font("Arial", Font.PLAIN, 18);
      Font fb = new Font("TimesRoman", Font.BOLD, 18);
      Font fi = new Font("Serif", Font.ITALIC, 18);
      Font fbi = new Font("Monospaced", Font.BOLD + Font.ITALIC, 18);

      g.setFont(f);
      g.setFont(fb);
      g.drawString("Welcome to BAOU, Ahmedabad", 10, 50);
      g.setFont(fi);
      g.drawString("This is Dept. of Computer Science", 10, 75);
      g.setFont(fbi);
      g.drawString("This is Gujarat Vidyapith, Ahmedabad", 10, 100);
   }
        public static void main(String s[])
        {
                Frame f= new Frame("Font Usage");
                f.add(new FontClass());
                f.setVisible(true);
                f.setSize(1550,200);
        }
}
```

Output:

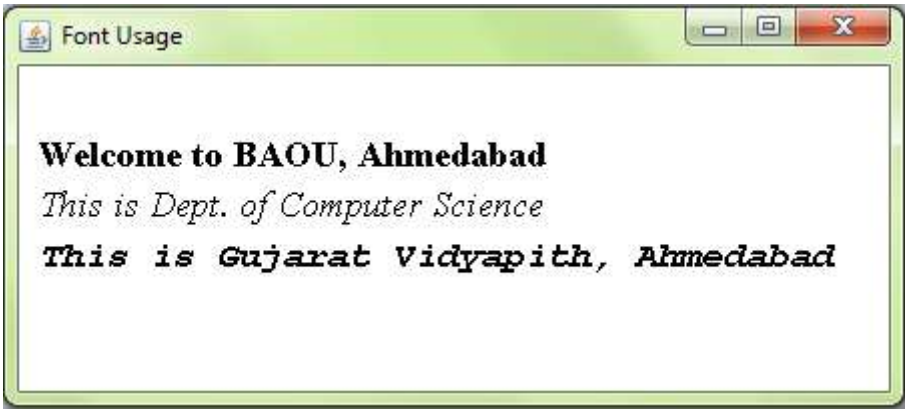


Figure-116: Output of program

3.5 LAYOUT MANAGER

It is possible to position and size the GUI component by hard coding but also challenging and therefore not advised. So, it is advised to use layout manager as it is easier to adjust and rework positions, sizes and the overall look-and-feel of the container. Use of layout managers facilitates a top-level or base container to have its own layout while other containers on top of it have their own layout which is completely independent. Whenever we add any components to a container, the final configuration of size and positioning is ultimately decided by the layout manager of the underlying container. Therefore, anytime a container is resized, its layout manager has to position each of the components within it. JPanel and content panes are the containers base of the GUI application structure and belong to FlowLayout and BorderLayout classes. It is recommended to set layout manager of the container.

LayoutManager is an interface. It is implemented by all the classes of layout managers. The following class represents the layout managers from java.awt package.

1. BorderLayout
2. FlowLayout
3. GridLayout
4. CardLayout
5. GridBagLayout

3.5.1 BORDERLAYOUT

The BorderLayout helps to arrange the components in north, south, east, west and center regions. This is the default layout for frame or window. The BorderLayout has five constants for each region. They are public static final int NORTH, SOUTH, EAST, WEST, CENTER.

Constructors:

1. BorderLayout(): This allows us to create a border layout without gaps between the components.

2. JBorderLayout(int hgap, int vgap): This allows us to create a border layout with the given horizontal and vertical gaps between the components.

Note: In this unit, we have used Frame as the main container in all programs.

Example: The following program depicts the use of BorderLayout.

```
import java.awt.*;
public class BorderLout extends Frame
{
        BorderLout(String title)
        {
                super(title);
                Button b1=new Button("BAOU");;
                Button b2=new Button("GVP");;
                Button b3=new Button("DCS");;
                Button b15=new Button("BCA");;
                Button b5=new Button("MCA");;

                add(b1,BorderLayout.NORTH);
                add(b2,BorderLayout.SOUTH);
                add(b3,BorderLayout.EAST);
                add(b15,BorderLayout.WEST);
                add(b5,BorderLayout.CENTER);
        }
        public static void main(String[] args)
        {
                Frame bly=new BorderLout("Border");
                bly.setSize(300,300);
                bly.setVisible(true);
        }
}
```

Output:

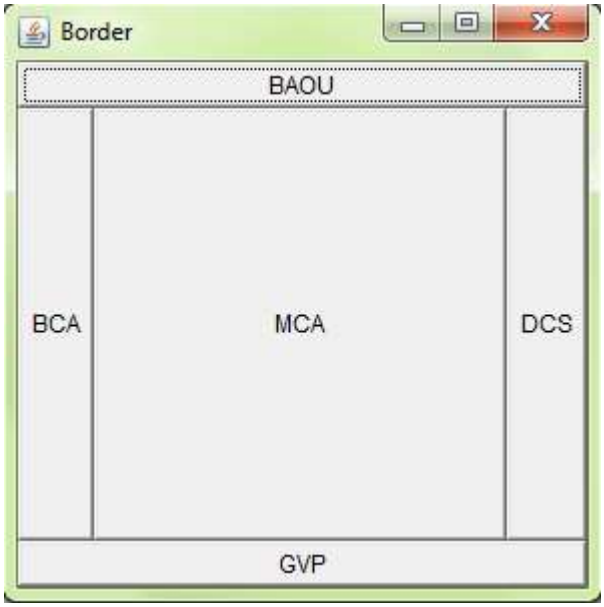


Figure-117: Output of program

3.5.2 FLOWLAYOUT

The FlowLayout is used to arrange the components in a line. As we keeps adding components, it arranges them one after another from left to right in a flow. This layout is the default layout of applet or panel.

2.4.1 Constants of FlowLayout:

2.4.2 There are total five constants used in FlowLayout. They are public static final int LEFT, RIGHT, CENTER, LEADING and TRAILING.

Constructors:

1. **FlowLayout():** It allows us to create a flowlayout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align):** It allows us to create creates a flowlayout with the specified alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap):** It allows us to create a flowlayout with the specified alignment and horizontal and vertical gap.

Example: The following program depicts the use of FlowLayout.

```
import java.awt.*;
public class FlowLout extends Frame
{
        FlowLout(String title)
        {
                super(title);
```

```
                Button b1=new Button("BAOU");
                Button b2=new Button("GVP");
                Button b3=new Button("DCA");
                Button b15=new Button("MCA");
                Button b5=new Button("BCA");

                add(b1);add(b2);add(b3);add(b15);add(b5);
                //setting flow layout of right alignment
                setLayout(new FlowLayout(FlowLayout.RIGHT));

}
public static void main(String[] args) {
   Frame fly=new FlowLout("Flow");
   fly.setSize(250,200);
   fly.setVisible(true);
}
}
```

Output:

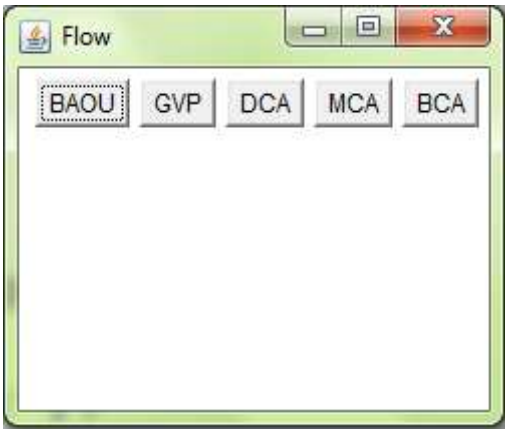


Figure-118: Output of program

3.5.3 GRIDLAYOUT

The GridLayout helps us to arrange the components in rectangular grid. Only one component will be displayed in each rectangle.

Constructors:

1. GridLayout(): This constructor allows us to create a gridlayout with one column per component in a row.

2. GridLayout(int rows, int columns): This constructor allows us to create a gridlayout with the specified rows and columns but without the gaps between the components.
3. GridLayout(int rows, int columns, int hgap, int vgap): This constructor allows us to create a gridlayout with the specified rows, columns, horizontal gap and vertical gap.

Example: The following program depicts the use of GridLayout.

```
import java.awt.*;
public class GridLout extends Frame
{
GridLout(String title){
    super(title);
    Button ba=new Button("A");
    Button bb=new Button("B");
    Button bc=new Button("C");
    Button bd=new Button("D");
    Button be=new Button("E");
    Button bf=new Button("F");
    Button bg=new Button("G");
    Button bh=new Button("H");
    Button bi=new Button("I");

    add(ba);add(bb);add(bc);add(bd);add(be);
    add(bf);add(bg);add(bh);add(bi);
  //setting gridlayout of 3 rows and 3 columns
    setLayout(new GridLayout(3,3));
}
public static void main(String[] args) {
    Frame fyl=new GridLout("Grid");
    fyl.setSize(300,300);
    fyl.setVisible(true);
}
}
```

Output:

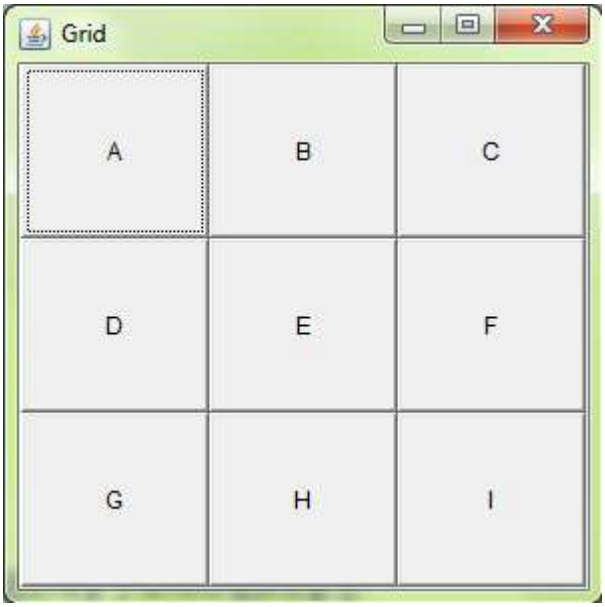


Figure-119: Output of program

2 3.5.4 CARDLAYOUT

The CardLayout class manages the components in such a manner that only one component is visible at a time. It treats each component as a card that is why it is known as CardLayout. There are various methods like next, first, previous, last and show to flip from one card to another card.

2.4.1 Constructors:

1. CardLayout(): This constructor allows us to create a cardlayout with zero horizontal and vertical gap.
2. CardLayout(int hgap, int vgap): This constructor allows us to create a cardlayout with the specified horizontal and vertical gap.

Example: The following program depicts the use of GridLayout. We have used three panels as a card to show different pane.

```
Import java.awt.*;
import java.awt.event.*;
class CardLout extends Frame implements ActionListener {
    CardLayout cardlt = new CardLayout(25,25);
    CardLout(String str)  {
        super(str);
        setLayout(cardlt);
        Button Panel1 = new Button("BAOU");
        Button Panel2 = new Button ("DCS");
        Button Panel3 = new Button("GVP");
        add(Panel1,"BAOU");
```

```
        add(Panel2,"DCS");
        add(Panel3,"GVP");
        Panel1.addActionListener(this);
        Panel2.addActionListener (this);
        Panel3.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
      cardIt.next(this);
    }

   public static void main(String args[])
   {
          CardLout frame = new CardLout("CardLayout");
          frame.setSize(210,170);
          frame.setResizable(false);
          frame.setVisible(true);
   }
}
```

Output:

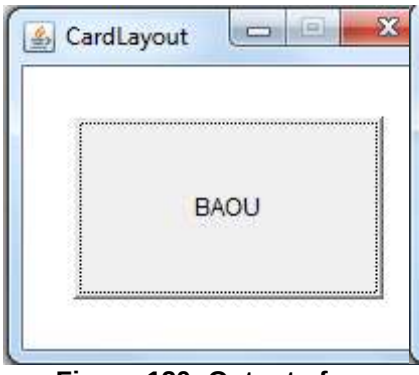


Figure-120: Output of program

Figure-121: Output of program

Figure-122: Output of program

3.5.5 GRIDBAGLAYOUT

The Java GridBagLayout class helps to align components vertically, horizontally or along their baseline. It is also most flexible as well as complex layout managers. It places components in a grid of rows and columns, allowing particular components to span multiple rows or columns. Not all rows and columns necessarily have the same height. It places components in cells in a grid and then uses the

components' preferred sizes to determine how big the cells should be to contain component.

To use a GridBagLayout effectively, we need to customize one or more component's GridBagConstraints. By setting one of its instance variables we can customize a GridBagConstraints object. The instances are:

- gridx, gridy

  This variables specifies the cell at the top most left of the component's display area, where address gridx=0 refers the leftmost column and address gridy=0 refers the top row. GridBagConstraints.RELATIVE is the default value. It specifies that the component placed just to the right of (gridx) or below (gridy) the component.

- gridwidth, gridheight

  This variable specifies the number of cells in a row (for gridwidth) or column (for gridheight) in the component's display area. The default value is 1.

- fill

  This variable is used when the component's display area is larger than the component's requested size to decide whether to resize the component. We can pass NONE (default), HORIZONTAL (will not change its height), VERTICAL (will not change its width) and BOTH (component fill its display area entirely) with GridBagConstrain as valid values of fill.

- ipadx, ipady

  This variable specifies the internal padding. The width of the component will be its minimum width plus ipadx*2 pixels (as the padding applies to both sides of the component). Similarly, the height of the component will be its minimum height plus ipady*2 pixels.

- insets

  This variable specifies the external padding of the component. It will be the minimum amount of space between the component and the edges of its display area.

- anchor

  This variable is helps us when the component is smaller than its display area to decide where to place the component. We can pass CENTER (the default), NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST and NORTHWEST as valid values.

- weightx, weighty

  This variable is used to determine how to distribute space when we want to specify resizing behaviour or change of dimension.

  Example: Below example uses GridBagConstraints instance for all the components the GridBagLayout manages. In real-life, it is recommended that you do not reuse GridBagConstraints. In the example, just before each component is added to the container, the code sets the appropriate instance variables in the GridBagConstraints object. Then after it adds the component

to its container, passing the GridBagConstraints object as the second argument to the add method.

```
import java.awt.*;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

public class gridBagLout extends Frame {
   Button first, second,third,forth,fifth,sixth;
        public static void main(String[] args)       {
      Frame gbl = new gridBagLout(“GridBag Layout”);
                gbl.setSize(300, 300);
                gbl.setVisible(true);
        }
   public gridBagLout(String str)   {
          super(str);
         first=new Button("BAOU");
        second=new Button("DCS");
        third=new Button("MCA");
        forth=new Button("GVP");
        fifth=new Button("Ahmedabad");
        sixth=new Button("Gujarat");

   GridBagConstraints gbc = new GridBagConstraints();
   GridBagLayout layout = new GridBagLayout();
   setLayout(layout);

   gbc.fill = GridBagConstraints.HORIZONTAL;
   gbc.gridx = 0;
   gbc.gridy = 0;
   add(first, gbc);
   gbc.gridx = 1;
   gbc.gridy = 0;
   add(second, gbc);
   gbc.fill = GridBagConstraints.HORIZONTAL;
```

```
    gbc.ipady = 30;
    gbc.gridx = 0;
    gbc.gridy = 1;
    add(third, gbc);
    gbc.gridx = 1;
    gbc.gridy = 1;
    add(forth, gbc);
    gbc.gridx = 0;
    gbc.gridy = 2;
    gbc.fill = GridBagConstraints.HORIZONTAL;
    gbc.gridwidth = 2;  //Merge two columns
    add(fifth, gbc);
    gbc.gridx = 0;
    gbc.gridy = 3;
    gbc.gridwidth = 2;  //Merge two columns
    add(sixth, gbc);
    }
}
```

Output:

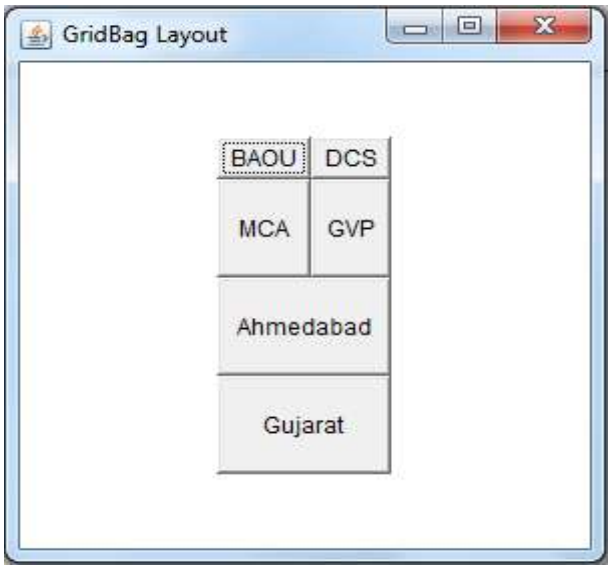


Figure-123: Output of program

- **setBounds() method**

setBounds() method of awt.component class is used to set the size and position of component. When we need to change the size and position of component then we can use this method

Syntax:

public void setBounds(int x, int y, int width, int height)

This parameter puts the upper left corner at location (x, y), where x is the number of pixels from the left of the screen and y is the number from the top of the screen.

Example:

```
import java.awt.*;
public class Setbound extends Frame
{
   Label name;
   TextField user;
   Button login;
   Setbound(String str)
        {
          super(str);
          setLayout(null);
          name=new Label("User_Name:");
          user=new TextField(10);
          login=new Button("Login");

          name.setBounds(50, 50, 75, 30 );
          add(name);
          user.setBounds(130, 50, 180,30 );
          add(user);
          login.setBounds(100, 90, 60, 30 );
          add(login);
  }
  public static void main(String[] args)
        {
     Frame sb=new Setbound("SetBound");
     sb.setSize(350,150);
     sb.setVisible(true);
   }
```

```
}
```

Output:

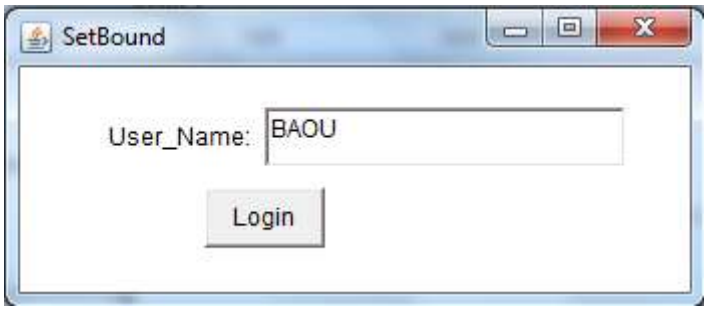


Figure-124: Output of program

3.6 LET US SUM UP

In this unit we have learned the basics of how to paint, including how to use the graphics primitives to draw basic shapes, how to use fonts and font metrics to draw text, and how to use Color objects to change the color of what we are drawing on the container. Graphics, Color and Font classes are the foundation in painting that enables user to do animation inside a container and to work with images. Layout Manager plays a crucial role for arranging components as per the user requirement for designing attractive and user friendly GUI.

3.7 FURTHER READING

1) Core Java Programming-A Practical Approach by Tushar B. Kute
2) Java: The Complete Reference by Schildt Herbert. Ninth Edition
3) Head First Java: A Brain-Friendly Guide, Kindle Edition by Kathy Sierra, Bert Bates. 2nd Edition
4) Java: A Beginner's Guide by Schildt Herbert Sixth Edition
5) Core Java Volume I — Fundamentals by Cay S. HorstMann, Gary Cornell, 9th Edition
6) https://www.codemiles.com/java-examples/fonts-in-java-t2831.html
7) https://courses.cs.washington.edu/courses/cse3151/98au/java/jdk1.2beta15/docs/api/java/awt/Font.html
8) https://www.leepoint.net/GUI-appearance/fonts/10font.html

3.8 ASSIGNMENTS

1) Define Graphics. Explain the importance of Graphics class in java.
2) Differentiate paint(), repaint() and update() method.
3) Explain Font class with proper example to demonstrate the use of font family.
4) How do we can set and get color in java application? Explain through example.
5) What is Layout Manager? Explain different types of layout managers.