JAVA PROGRAMMING

A BEGINNERS GUIDE TO LEARN PROGRAMMING IN JAVA

2018 Edition

BY MR KOTIYANA

Table of Contents

12.3 URL Processing 322

CHAPTER 1 | INTRODUCTION

What This Book is about?

This book was written as an answer for anyone to pick up a programming language and be productive. You will be able to start from scratch without having any previous exposure to any programming language. By the end of this book, you will have the skills to be a capable programmer, or at least know what is involved with how to read and write code. Afterward you should be armed with the knowledge required to feel confident in learning more. You should have general computer skills before you get started. After this you'll know what it takes to at least look at code without your head spinning.

Why Read This Book?

You could go online and find videos and tutorials to learn; however, there is a distinct disadvantage when it comes to learning things in order and in one place.

Most YouTube or tutorial websites either gloss over a topic or dwell at a turtle's pace for an hour on a particular subject. Online content is often brief and doesn't go into much depth on any given topic. It is incomplete or still a work in progress. You'll often find yourself waiting weeks for another video or tutorial to come out.

Most online tutorials for Java are scattered, disordered, and in cohesive. It is difficult to find a good starting point and even more difficult to find a continuous list of tutorials to bring you to any clear understanding of the Java programming language. Just so you know, you should find the act of learning exciting. If not, then you'll have a hard time continuing through to the end of this book. To learn any new skill, a lot of patience is required.

I remember asking an expert programmer how I'd learn to program. He told me to write a compiler. At that time, it seemed rather mean, but now I understand why he said that. It is like telling someone who wants to learn how to drive Formula 1 cars to go compete in a race. In both cases, the "learn" part was left out of the process of learning. It is very difficult to tell someone who wants to learn to write code where to begin.

However, it all really does start with your preparedness to learn. Your motivation must extend beyond the content of this book. You may also have some preconceived notions about what a programming is.

I can't change that, but you should be willing to change your thoughts on the topic if you make discoveries contrary to your knowledge. Keep an open mind.

Computer artists often believe that programming is a technical subject that is incompatible with art. I find the contrary to be true. Programming is an art, much as literature and design is an art. Programming just has a lot of obscure rules that need to be understood for anything to work.

Do I Need to Know Math?

With complex rules in mind, does programming require the knowledge of complex mathematics? Actually, unless you program mathematical software, only a bit of geometry is nice to have. Most of the examples here use only a tiny bit of math to accomplish their purposes.

Mathematics and programming do overlap quite a lot in their methodology. Math taught in schools provides a single solution. Programming results tend to behave a bit like a math proof, but only the proof isn't just another bit of math. Rather, the proof of your code means that your zombies chase after humans.

A considerable amount of math required has been done for you by the computer. It is just up to you to know what math to use, and then plug in the right variables.

Programming as a Form of Expression

There is a deeper connection between words and meaning in programming. This connection is more mechanical, but flexible at the same time. In relation to a programming language, programmers like to call this "expressiveness." When you write words in literature, you infer most if not all of the meaning. When programming inference isn't implied, then you're free to define your own meanings for words and invent words freely.

One common merging of art and code appears within a video game. Anytime characters can react to each other and react to user interaction, which conveys a greater experience. An artist, an animator, or an environment designer should have some ability to convey a richer experience to the audience. A character artist can show a monster to his or her audience; but to really experience the creature, the monster should glare, grunt, and attack in reaction to the audience.

Before You Start your Career in Software Development, I Would Highly Recommend You to Have this Book before Going into Software Development Area:

The Complete Software Developer's Career Guide: How to Learn Your Next Programming Language, Ace Your Programming Interview, and Land The Coding Job Of Your Dreams

- by John Sonmez

Choose Your Store:

[(USA) Amazon.com](#)

[(UK) Amazon.co.uk](#)

[(INDIA) Amazon.in](#)

[(Germany) Amazon.de](#)

[(France) Amazon.fr](#)

[(Spain) Amazon.es](#)

[(Italy) Amazon.it](#)

[(Canada) Amazon.ca](#)

A Brief History of Computer Programming:

How Programming Came to Be

Mechanical Computers:

The first programmable computer is arguably the Babbage machine built by Charles Babbage in the 1820s. Being made of tens of thousands of moving parts and weighing several tons, it was a gigantic mechanical calculator that could display a 31-digit number. It was called the Difference Engine, and in 1824 Babbage won a gold medal from the British Astronomical Society for correcting a set of tables used to chart the movement of planets through the sky. In 1833, a countess named Augusta Ada King of Lovelace, commonly known as Ada Lovelace, met Babbage at a party where she was introduced to the Difference Engine. Several years later, it became evident that she understood the machine better than Charles himself. Despite being housewife to the Earl of Lovelace, she wrote several documents on the operation of the Difference Engine as well as its upgraded version, the Analytical Engine. She is often considered to be the first computer programmer for her work with the Difference Engine and its documentation.

Logic:

In 1848, George Boole gave us Boolean logic. It would take nearly a century between the Difference Engine and the first general programmable computer to make its appearance. Thanks to George, today our computers count in binary (1s and 0s), and our software thinks in terms of true or false. In 1887, Dorr Eugene Felt built a computing machine with buttons; thanks to him, our computers have keyboards. In the 1890s, a tabulating machine used paper with holes punched in it representing 1s and 0s to record the US census. Back then it saved US$2 million and 6 years of work. In the 1900s, between Nicola Tesla and Alexander Graham Bell, modern computers were imminent with the invention of the transistor. In 1936, Konrad Zuse made the Z1 computer, another mechanical computer like that of Babbage, but this time it used a tape with holes punched in it like the tabulating machine. He'd later move on to make the Z3 in 1941.

Computer Science:

In the 1940s, Alan Turing, a British computer scientist, designed and built an encrypted message decoder called the Bombe that used electromechanical switches for helping the Allied forces during World War II. Between the 1930s and the 1950s, Turing informed the computer scientists that computers can, in theory, solve anything calculable, calling this concept Turing completeness. All of these components began to lead toward our modern computer. In the mid-1940s, John Von Neumann demonstrated with the help of his theory that the computer can be built using simple components. This way the software that controls the hardware can add the complex behavior. Thanks to Tesla and Bell, the Z3 was made completely of electronic parts. It included the first use of logic while doing calculations, making it the first complete Turing-complete computer. In 1954, Gordon Teal at Texas Instruments introduced the silicon-based transistor, a key component for building solid-state electronics that are used today in every computer.

Software:

The Z3 was programmed using Plankalkül, the first high-level programming language, invented by Zuse shortly after he finished building the machine. In 1952, Grace Hopper created the first compiler, software that could translate human-readable words into machine operations. For the first time, the programmer didn't need to know how the transistors operated in the hardware to build software for the computer to run. This opened computer programming to a whole new audience. Between the 1950s and the 1970s, computers and computer languages were growing in terms of complexity and capability. In 1958, John McCarthy invented LISP at the Massachusetts Institute of Technology (MIT). Object oriented programming appeared in Simula 67 in 1967. Imperative and procedural programming made its appearance in Pascal in 1970.

Modern Computer Language: Bell Labs, started by Bell in the 1920s, hired Dennis Ritchie in 1967. In 1974, Ritchie published the C programming language that has become the most popular computer programming language. In 1983, C++ made its appearance as "C with Classes." James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. Java was originally designed for interactive television, but it was too advanced for the digital cable television industry at the time. The language was initially called *Oak after* an oak tree that stood outside Gosling's office. Later the project went by the name *Green*and was finally renamed *Java*, from Java coffee. Gosling designed Java with a C/C++-style syntax that system and application programmers would find familiar.

Sun Microsystems released the first public implementation as Java 1.0 in 1995. It promised "Write Once, Run Anywhere" (WORA), providing no-cost run-times on popular platforms. Fairly secure and featuring configurable security, it allowed network-and file-access restrictions. Major web browsers soon incorporated the ability to run *Java applets* within web pages, and Java quickly became popular. The Java 1.0 compiler was re-written in Java by Arthur van Hoff to comply strictly with the Java 1.0 language specification. With the advent of *Java 2* (released initially as J2SE 1.2 in December 1998 – 1999), new versions had multiple configurations built for different types of platforms. *J2EE* included technologies and APIs

for enterprise applications typically run in server environments, while *J2ME* featured APIs optimized for mobile applications. The desktop version was renamed *J2SE*. In 2006, for marketing purposes, Sun renamed new *J2* versions as *Java EE*, *Java ME*, and *Java SE*, respectively.

In 1997, Sun Microsystems approached the ISO/IEC JTC 1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process. Java remains a *de facto* standard, controlled through the Java Community Process. At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System.

On November 13, 2006, Sun released much of its Java virtual machine (JVM) as free and open-source software, (FOSS), under the terms of the GNU General Public License (GPL). On May 8, 2007, Sun finished the process, making all of its JVM's core code available under free software/open-source distribution terms, aside from a small portion of code to which Sun did not hold the copyright.

The Future of Computer Languages: The International Business Machines Corporation (IBM), a US company founded in the 1920s, is currently engineering new technology using particle physics discovered nearly a century ago. Modern spintronics derived from the work of German physicists in the late 1920s. Without particle

physics and physicists such as Friedrich Hermann Hund, we wouldn't understand quantum tunneling, and the solid-state disk or universal serial bus (USB) thumb drive wouldn't exist. After the discovery and confirmation of the Higgs boson, you might ask "Why bother?" This question might not get a good answer for another hundred years, but it'll be worth it. Java is not the last computer language; many others have been introduced recently, most notably an entry by Google called Go.

Microsoft, not to be outdone, has also introduced F#. There are many relatively new programming languages that have been made popular in recent years. Though the languages are different, the principal concepts are all the same. Once the basic ideas are understood, it is easy to apply the knowledge obtained here to new languages and continue to learn. Today quantum computers begin to make their appearance. Dealing with today's transistors, we send and receive determinate values. Quantum computers don't work with 1s and 0s; rather they store multiple values between 0 and 1. To deal with this, new programming languages are currently being developed.

CHAPTER 2 | GETTING STARTED

What Is Programming?

It's all about writing code. Programming is a process in which we organize data and use logic to do something with those data. The data are everything a computer can store; they can range from numbers to zombie characters in a video game.

You do this by writing text into files called source code. Source code written into text files replaces punch cards used by the computing machines half a century ago.

When data are combined with logic and then written into a single file, they're called a class. Classes are also data, and as such can be managed with more logic.
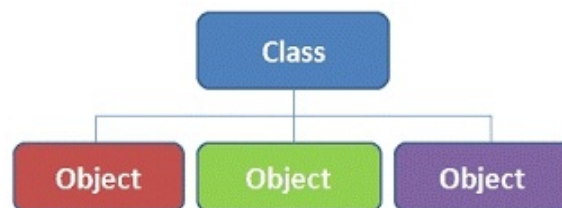
Classes are used to create objects in the computer's memory and can be duplicated to have a life of their own. Classes are used to build objects. Each piece of data within the class becomes a part of that object. Different chunks of data inside of a class are called class members.

Class members can also be chunks of logic called functions or methods.

For Example, in a game with a horde of zombies, each zombie is duplicated or instanced from a zombie class. Each zombie has unique values for each attribute or data element in the class.

This means hit points, and locations are unique for each duplicate zombie object. Objects created from a class are called instances. Similar to families, objects can inherit properties from one another.

The child sometimes called a subclass inherits   attributes from its parent. For instance, the child created from a zombie may inherit the parent's hunger for brains.

To be useful, the child zombie can also add new objects and change the objects it inherited from its parent class. As a result, now the child zombie might have tentacles that the parent didn't have. Objects talk to each other through events and messages.

Shooting at zombies can create an event, or in programmer terms, it "raises" an event. The bullet impact event tells the zombie class to then take necessary steps when hit by a bullet.
Events command the class to take actions on its data, which is where functions come in.

Functions, also known as methods, are sections of logic that act on data. They allow your class to create additional events and talk to yet more objects.

As the player presses the trigger and moves the joystick around, yet more events can be raised and messages can be sent. Events and messages allow the player to interact with your world; logic events and objects together build your game.

What is Data?

Data, in a general sense, is sort of like a noun. Like nouns, data can be a person, place, or thing. Programmers refer to these nouns as objects, and these objects are stored in memory as variables. The word variable infers something that might change, but this isn't always the case. It's better to think of a variable as a space in your computer's memory to put information. When you play word games like MadLibs you might ask for someone's name, an object, an adverb, and a place. Your result could turn out like "Garth ate a jacket, and studiously played at the laundry-mat." In this case the name, object, adverb, and place are variable types. The data is the word you use to assign the variable with.

Programmers use the word **type** to denote what kind of data is going to be stored. Computers aren't fluent in English and don't usually know the difference between the English types noun and adjective, but they do know the difference between letters and a whole variety of numbers. There are many predefined types in java or any other language.

If you add that to the ability to create new types of data, the kinds of data we can store is practically unlimited.

The C# built-in types are sometimes called POD, or plain old data. The term POD came from the original C++ standard which finds its origin dating back to 1979. POD types have not fundamentally changed from their original implementation.

So far we've used the word type several times. Programmers define the word type to describe the variety of data to be stored.

What is a Compiler?

Generally compiling is a term which is often heard by everyone who is associated with programming, even if remotely. A compiler is a software program that transforms high-level source code that is written by a developer in a high-level programming language into a low level object code (binary code) in machine language, which can be understood by the processor. The process of converting high-level programming into machine language is known as compilation.

The processor executes object code, which indicates when binary high and low signals are required in the arithmetic logic unit of the processor.

Let us take an example to understand what does the above **definition** means.

If you ask any person who is associated with programming, that what the first program he/she wrote was, then the obvious answer would be "Hello World". So let us also start with the same.

```
#include<stdio.h>
    Void main()

                                    {

    printf("Hello, world!");

                                    }
```

This most basic program prints or displays the words "Hello,World!" on the computer screen. But there is a problem. It is not that simple. Behind the curtains there is lot of complex things going. Let us peep inside these things. The hard truth is that our computer cannot understand the commands/instructions contained in a source file (helloworld.c), because C is a high-level language which means, it contains various characters, symbols, and words that represent complex, numbers-based instructions for *eg.* printf, main, header files *etc.* The only instructions a computer can execute are those written in machine language, consisting entirely of numbers that is the binary language in terms of 0 and 1. Before our computer can run our C program, our **compiler** should convert our helloworld.c into an object file; then a program called a linker should convert the object file into an executable file.

The drawing below illustrates the process.

Source Files   Object Files

Executable
Files

Compile   Link
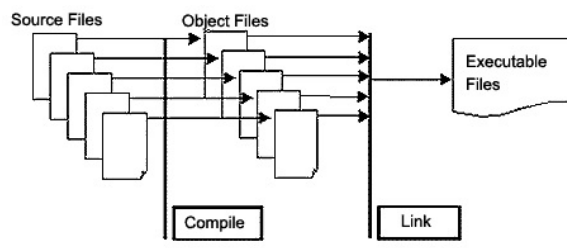
A compiler that converts machine language into high-level natural language is called a decompiler. Compilers that produce the object code meant to run on a system are called cross-compilers. Finally, a compiler that converts one programming language into another is called a language translator.

A compiler executes four major steps: **Scanning**: The scanner reads one character at a time from the source code and keeps track of which character is present in which line.

Lexical Analysis: The compiler converts the sequence of characters that appear in the source code into a series of strings of characters (known as tokens), which are associated by a specific rule by a program called a lexical analyzer. A symbol table is used by the lexical analyzer to store the words in the source code that correspond to the token generated.

Syntactic Analysis: In this step, syntax analysis is performed, which involves preprocessing to determine whether the tokens created during lexical analysis are in proper order as per their usage. The correct order of a set of keywords, which can yield a desired result, is called syntax. The compiler has to check the source code to ensure syntactic accuracy.

Semantic Analysis: This step is comprised of several intermediate steps. First, the structure of tokens is checked, along with their order with respect to the grammar in a given language. The meaning of the token structure is interpreted by the parser and analyzer to finally generate an intermediate code, called object code. The object code includes instructions that represent the processor action for a corresponding token when encountered in the program. Finally, the entire code is parsed and interpreted to check if any optimizations are possible. Once optimizations can be performed, the appropriate modified tokens are inserted in the object code to generate the final object code, which is saved inside a file.

So A *compiler* is a program that reads in as input a program (in some high-level programming language) and outputs machine language code (for some machine architecture). The machine language code can subsequently be executed any

number of times using different input data each time. As a second example, the Java   compiler javac transforms a **.java** source file into a **.class** file that is written in *Java **bytecode**,* which is the machine language for an imaginary machine known as the ***Java Virtual Machine***.

What is an interpreter?

An *interpreter* is a program that reads in as input a source program, along with data for the program, and translates the source program instruction by instruction. For example, the Java interpreter javatranslate a **.class** file into code that can be executed natively on the underlying machine.

Why does Java typically interpret instead of compile?

The main advantage of compilation is that you end up with raw machine language code that can be efficiently executed on your machine. However, it can only be executed on one type of machine architecture (Intel Pentium, PowerPC). A primary advantage of a compiling to an intermediate language like Java bytecode and then interpreting is that you can achieve *platform independence*: you can interpret the same .class file on differently types of machine architectures. However, interpreting the bytecode is typically slower than executing pre-compiled machine language code. A second advantage of using the Java bytecode is that it acts as a buffer between your computer and the program. This enables you to download an untrusted program from the Internet and execute it on your machine with some assurances. Since you are running the Java interpreter (and not raw machine language code), you are protected by a layer of security which guards against malicious programs. It is the combination of Java and the Java bytecode that yield a platform-independent and secure environment, while still embracing a full set of modern programming abstractions.

The Java bytecode and the java interpreter are not inherently specific to the Java programming language. For example, you can use Jython to compile from the Python programming language into Java bytecode, and then use java to interpret it. There are similar ML, Lisp, and Fortran compilers that compile into JAva bytecode. You could also use the Unix program gcj to compile directly from a **.javasource** file into a machine executable file a.out, which can be run natively on any Sparc microprocessors. Additionally, you can design hardware whose machine language is the Java bytecode. Sun Microsystems has done exactly this, making the Java Virtual Machine not so virtual.

Why not use a real machine language instead of the Java bytecode?

The Java bytecode is much simpler than a typical high-level programming language. It is much easier to write a Java bytecode interpreter for a new type of computer than it is to write a full Java compiler.

Programming Environment Setup

When we say Environment Setup, it simply implies a base on top of which we can do our programming. Thus, we need to have the required software setup, i.e., installation on our PC which will be used to write computer programs, compile, and execute them. For example, if you need to browse Internet, then you need the following setup on your machine:

- A working Internet connection to connect to the Internet
- A Web browser such as Internet Explorer, Chrome, Safari, *etc.*

Similarly, you will need the following setup to start with programming using any programming language.

- A text editor to create computer programs.
- A compiler to compile the programs into binary format.
- An interpreter to execute the programs directly.

We recommend you to use IDE (Integrated Development Environment) , the handy, dandy piece of software that acts as text editor, debugger and compiler all in one sometimes-bloated but generally useful package.

Unless you're committed to working in a text editor and a command prompt window (and there are compelling reasons for doing exactly that) you might be looking for some advice on how to choose a good IDE, the pros and cons of various varieties, the relative costs (financial or system resources) of running a particular IDE, what other languages the IDE might handle well, the operating system(s) it runs on and ever so much more.

Here is the list of two most common IDE used by Programmers.

Eclipse

Languages: C, C++, Python, Perl, PHP, Java, Ruby and more
Price: FREE

Eclipse is the free and open-source editor upon which many development frameworks are based. It's one of the granddaddies in its field and comes highly recommended by many a professional developer. Eclipse began as a Java development environment and has greatly expanded through a system of lightweight plugins.

NetBeans

Languages: Java, JavaScript, PHP, Python, Ruby, C, C++ and more
Price: FREE

NetBeans is neck-and-neck with Eclipse as the most-recommended IDE in this category. It's free and open-source, supports tons of languages with more plugins coming all the time, and is incredibly simple to install and use, even for a beginner.

Compilation and Execution

A simple Java project contains a single directory inside which all the Java source files are stored. The Java files are usually not stored directly inside the source directory, but in subdirectories matching their **package structure**. Packages are just a way to group together source files which belong together.

When you compile all the source code in the source, the compiler produces one **.class** file for each **.java** file. The .class contains the compiled version of the **.javafile**. The byte code for the **.java** file in other words.

It is the **.class** files that the Java Virtual Machine can execute. Not the **.java** files. Therefore it is normal to separate the **.java** files from the .class files. This is normally done by instructing the compiler to write the **.class** files into a separate directory. This directory is often called classes, but again, it is not a requirement, and it depends on what build tool or IDE *etc.* you are using.

Compiling the Java Source Code:

You can compile your Java source directly from your IDE (if you use an IDE). Or, you can use the Java compiler that comes with the Java SDK. To compile your Java source code from the command line with the Java compiler, do this:

- Open a command prompt
- Change directory to your project's root directory (not the source directory)
- Make sure the project root directory contains a source directory and a class directory
- Type in the command below (on Windows - other OS'es will look similar though):

```
"c:\Program Files\Java\jdk1.8.0_25\bin\javac" src/myfirstapp/*.java -d classes
```

This command executes the javac command (the Java compiler) which compiles the Java sources in the directory src/myfirstapp . The *.java means the compiler should compile all files in the given directory.

The myfirstapp directory is a package under the root source directory src. If you have multiple packages under the root source directory you will have to run the Java compiler multiple times. A Java IDE handles this for you automatically. So does build tools like Ant, **Maven** or Gradle.

Running the Compiled Java Code:
Once the compiler has done its job, the classes directory will contain the compiled .class files. The package structure (directory structure) from the source directory will be preserved under the class directory. You can run any one of these .class files which have a main() method in it. You can run the .class from inside your Java IDE, or from the command line. From the command line it looks like this:

```
"c:\Program    Files\Java\jdk1.8.0_25\bin\java"    -cp    classes myfirstapp.MyJavaApp
```

The -cp classes flag tells the Java Virtual Machine that all your classes are located under the directory called classes. This is also called the "Class Path" (hence the abbreviation cp).

The name of the class to run is the last argument in the above command –

the myfirstapp.MyJavaApp part.

The JVM needs to know the fully qualified class name (all packages plus class name) to determine where the corresponding .class file is located.

When you run the the class your command line will look similar to this (including the output from the Java app):

```
D:\data\projects\myfirst-java-app>"c:\Program
Files\Java\jdk1.8.0_25\bin\java"
   -cp classes myfirstapp.MyJavaApp
Hello World!

D:\data\projects\myfirst-java-app>
```

(Note, the first command should not have a line break in it. I have put that in only to make it easier to read.

That is it! Now you know how to compile and execute your Java apps! As mentioned earlier, it can be easier using an IDE, but you may not have an IDE everywhere you want to execute the code.

CHAPTER 3 | BASIC TERMS

This section will cover many basic terms and some of the basic ideas behind writing proper code. Java is among the many different programming languages which use the same words and concepts to convey instructions to the computer. Once we get through the first few chapters we'll be introduced to some more of the basics which are directly related to reading and writing code. Before we can get to that we'll need to cover some of the terms and concepts that hold true for many different programming languages, not just Java. The methods and systems that allow converting words into executable code require fairly strict rules. When writing English or chatting online we tend to ignore formatting and punctuation. The brevity allows for faster communication, though only because we as humans have learned many new words and can use context to better interpret the contractions and acronyms in the written form of English used online or in text messages. Computers aren't so smart as to be able to interpret our words so easily.

Tokens

In written English the smallest elements of the language are letters and numbers. Individually, most letters and numbers lack specific meaning. The next larger element after the letter is the word. Each word has more meaning, but complex thoughts are difficult to convey in a single word. To communicate a thought, we use sentences. The words in a sentence each have a specific use, as seen in the diagram below. To convey a concept we use a collection of sentences grouped into a paragraph. And to convey emotion we use a collection of many paragraphs organized into chapters. To tell a story we use a book, a collection of chapters.
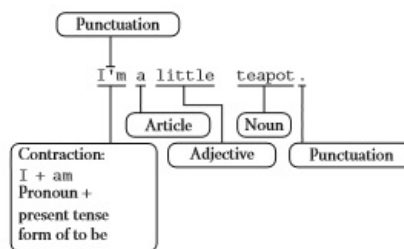
Programming has similar organizational mechanisms. The smallest meaningful element is a token, followed by statements, code blocks, functions, followed by classes and namespaces and eventually a program, or in our case a game. We will begin with the smallest element and work our way up to writing our own classes. However, it's important to know the very smallest element to understand how all the parts fit together before we start writing any complex code.

Writing Java:

Java is an English-based programming language; like any other C-like programming language it can be broken into tokens, the smallest meaningful fragment of text that the computer can understand. Tokens are made of a single character or a series of characters. For instance, in a simple statement like the following there are five tokens.
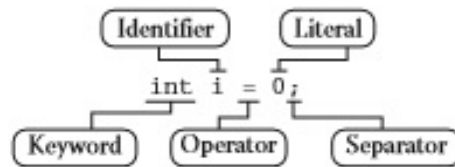
Tokens can be categorized as a keyword, identifier, literal, operator, separator, and white space. The above statement contains five tokens with spacing provided with white space. The keyword int is followed by the identifier i. This is followed by an operator = which assigns the identifier on the left of the operator and the value of the literal 0 on the right. The last token ; is a separator which ends the statement. White spaces and comments, which we have yet to cover, are not considered tokens, although they can act as separators. It's important that proper formatting or style be used.

Let's examine a sample of code:

```
    int i = 0;
  int j = 1;
```

The code shown above is two statements,
the separator keeps the computer from misreading the code as a single statement.
However, there are humans involved

with writing code, and thus code needs to be readable by humans, or any other life-form which might read your code. Proper code means following a specific formatting style. We'll dive into more about proper code style later in this chapter. Each group of characters, or text, is converted by a lexical analyzer, sometimes called a lexer, in the computer and turned a symbol. This process is called tokenization, or as a computer scientist would say, a lexer tokenizes your code into symbols. Once tokenized, the symbols are parsed; this process organizes the tokens into instructions for the computer to follow. This unique vocabulary does frame programmers as a strange group of alien beings with a language all to themselves; I promise, however, conversations about lexical analyzers don't come up very often when talking with most programmers.

This may seem like a great deal of work and we are jumping ahead into more complex computer science topics, but all of this happens behind the scenes so you don't need to see any of this taking place.

It's important to understand what a compiler does, and how it works so you can write code in a way that the computer can understand. This is why a simple typo will stop the lexer from building code. We'll get to writing code to be analyzed, tokenized, and parsed soon enough.

A computer can't interpret or guess at what it is you're trying to do.

Therefore, if you mistype **int i = 0:** and not **int i = 0;** the last token cannot be converted by the lexer into a proper symbol, and thus this statement will not be parsed at all. This code results in an error before the code is even compiled.

Separator Tokens: The most common separator is the ; semicolon. This mark of punctuation is used to end a code statement. Other separators come in pairs. Curly braces are used in pairs to separate groups of code statements. The opening curly brace is the { with the pointy end pointed to the outside of the group; this is paired with the } which closes the curly brace pair. Parentheses start with the (and end with the) while square brackets start with [and end with]. The different opening and closing braces, brackets, and parentheses have specific purposes and are used to help identify what they are being used for. There are angle brackets < and > as well; these are used to surround data types. And don't forget both single quotes " and double quotes "", which are used in pairs as well to surround symbols called strings.

NOTE:
In many word processors a beginning and an ending quote (" and ") are often used. The lexical analyzer doesn't recognize these smart quotes, so the text editor for programming uses straight quote marks (") instead. These subtle changes aren't so subtle once you realize that letters, numbers, and every other character used in programming are all parsed as American Standard Code for Information Interchange (ASCII) or unicode transformation format (UTF)-8, 16, or 32.

Both ASCII and UTF are names given to the database of characters that the computer uses to read text. Computers don't use eyes to read. Rather, they use a database of numbers to look up each character being used. Therefore, to a computer " looks more like 0x201C and " like 0x0022, if we were using UTF-16. Curly braces are used in code to separate statements. Parentheses are usually used to accept

data. Square brackets are often used to help identify arrays. We'll get into arrays later, but imagine something like a spreadsheet or a graph paper till we get to their explanation.

```
int[] arrayNumbers = {1, (int) 3.0, 9000};
```

The above statement assigns three numbers to an array. The curly braces contain a pair of statements separated by a comma. The second statement in the curly braces is converting a number with a dot (.) into a number without a dot in it. This will take a bit of explanation, but it's good to get used to seeing this sort of thing. We'll get to what an array is soon enough, as well as a dot operator.

Often, when learning to program you'll see many strange tokens which might not seem to have any meaning. It's up to you to observe every token you come across and try to understand what it does.

If none of the statements makes sense that's fine. At this point you're not expected to know what all this means yet, but you soon will.

Operator Tokens: Operators are like keywords but use only a few non-word characters. The colon or semi-colon is an operator; unlike keywords, operators change what they do based on where they appear, or how they are used. This behavior shifting is called operator overloading, and it's a complex subject that we'll get into later. It's also important to know that you can make up your own operators when you need to. Commas in Java have a different meaning. They're used to separate different data values like for (int i = 0, j = 1; ; i++, j++) Other Operator Tokens: Operators in general are special characters that take care of specific operations between variables. Notation or the order in which operators and variables appear is normally taught with the following notation or mathematic grammar: a + b = c. However, in programming the following is preferred: c = a + b; the change is made because the = operator in Java works differently from its function commonly taught in math class. The left side of the = operator is the assignment side, and the right side is the operation side. This puts the emphasis on the value assigned rather than how it's assigned. The result is more important than how the problem is solved since the computer is doing the math. In programming the above example should be read as "c is assigned a plus b."

Literals: Literals come in many different forms; similar to operators which appear as things like +, -, or = literals are the tokens that often go between them. Numbers are considered literals, or tokens that are used in a literal manner. Numbers can be called numeric literals. When something is put into quotes, as in "I'm a literal," you are writing a string literal. Literals are common throughout nearly all programming languages and form the base of Java's data types. Right now, we're dealing with only a few different types of literals, but there are many more we will be aware of soon.

Transitive and Non-Transitive Operations: In math class it's sometimes taught how some operations are transitive and others not. For instance 2 + 3 + 4 and 4 + 3 + 2 result in the same values. Where each number appears between the operators doesn't matter, so the + operator is considered transitive. This concept holds true in Java. Results change when we start to mix operators.

```java
int a = 1 + 2 - 4 + 7;  //a = 6
int b = 7 + 4 - 2 + 1; //b = 10
```

This code fragment shows two different results using the same operators but with different number placement. We'll see how to overcome these problems later on when we start using them in our game. Operator order is something to be aware of, and it's important to test each step as we build up our calculations to make sure our results end up with values we expect. To ensure that the operations happen as you expect, either you can use parentheses to contain operands

or you can do each calculation. We'll dive further into more detail with a second look at operators later on, but this will have to wait till we have a better understanding of more of the basics of Java.

CHAPTER 4 | BASIC OF JAVA PROGRAM

Structure of Java Program

A Java program consists of different sections. Some of them are mandatory but some are optional. The optional section can be excluded from the program depending upon the requirements of the programmer.

Documentation Section

It includes the comments to tell the program's purpose. It improves the readability of the program.

Package Statement

It includes statement that provides a package declaration.

Import statements

It includes statements used for referring classes and interfaces that are declared in other packages.

Interface Section

It is similar to a class but only includes constants, method declaration.

Class Section

It describes information about user defines classes present in the program. Every Java program consists of at least one class definition. This class definition declares the main method. It is from where the execution of program actually starts.

1. DOCUMENTATION Section: It includes the comments that improve the readability of the program. A comment is a non-executable statement that helps to read and understand a program especially when your programs get more complex. It is simply a message that exists only for the programmer and is ignored by the compiler. A good program should include comments that describe the purpose of the program, author name, date and time of program creation. This section is optional and comments may appear anywhere in the program.

Java programming language supports three types of comments.

Single line (or end-of line) comment: It starts with a double slash symbol (//) and terminates at the end of the current line. The compiler ignores everything from // to the end of the line.

For example:

// Calculate sum of two numbers

Multiline Comment: Java programmer can use C/C++ comment style that begins with delimiter /* and ends with */. All the text written between the delimiter is ignored by the compiler. This style of comments can be used on part of a line, a whole line or more commonly to define multi-line comment.

For example:

/*calculate sum of two numbers */

Comments cannot be nested. In other words, you cannot comment a line that already includes traditional comment. For example,

/* x = y /* initial value */ + z; */ is wrong.

Documentation comments: This comment style is new in Java. Such comments begin with delimiter /** and end with */. The compiler also ignores this type of comments just like it ignores comments that use / * and */. The main purpose of this type of comment is to automatically generate program documentation. The java doc tool reads these comments and uses them to prepare your program's documentation in HTML format.

For example:

```
/**The text enclosed here will be part of program documentation */
```

2. PACKAGE STATEMENT: Java allows you to group classes in a collection known as package. A package statement includes a statement that provides a package declaration. It must appear as the first statement in the source code file before any class or interface declaration. This statement is optional. For example: Suppose you write the following package declaration as the first statement in the source code file.

package employee;

This statement declares that all classes and interfaces defined in this source file are part of the employee package. Only one package declaration can appear in the source file.

IMPORT STATEMENT: Java contains many predefined classes that are stored into packages. In order to refer these standard predefined classes in your program, you need to use fully qualified name (i.e. Packagename.className). But this is a very tedious task as one need to retype the package path name along with the classname. So a better alternative is to use an import statement.

An import statement is used for referring classes that are declared in other packages. The import statement is written after a package statement but before any class definition. You can import a specific class or all the classes of the package. For example : If you want to import Date class of java.util package using import statement then write

import java.util.Date;

This statement allows the programmer to use the simple classname Date rather than fully qualified classname java.util.Date in the code.

Unlike package statement, you can specify more than one import statement in your program.

For example:

Import java.util.Date; /* imports only the Date class in java.util package */

import java.applet.*; // imports all the classes in java applet

                                // package

INTERFACE SECTION: In the interface section, we specify the interfaces. An interface is similar to a class but contains only constants and method declarations. Interfaces cannot be instantiated. They can only be implemented by classes or extended by other interfaces. It is an optional section and is used when we wish to implement multiple inheritance feature in the program.

```
interface stack
{
void push(int item); // Insert item into stack
int pop(); // Delete an item from stack
}
```

CLASS SECTION: The Class section describes the information about user-defined classes present in the program. A class is a collection of fields (data variables) and methods that operate on the fields. Every program in Java consists of at least one class, the one that contains the main method. The main () method which is from where the execution of program actually starts and follow the statements in the order specified.

The main method can create objects, evaluate expressions, and invoke other methods and much more. On reaching the end of main, the program terminates and control passes back to the operating system.

The class section is mandatory.

After discussing the structure of programs in Java, we shall now discuss a program that displays a string Hello Java on the screen.

```
// Program to display message on the screen
```

```java
class HelloJava

{

    public static void main(String args[])

    {

    System.out.println("Hello Java");

    }

    }
```

The main() Method

A Java program is a sequence of Java instructions that are executed in a certain order. Since the Java instructions are executed in a certain order, a Java program has a start and an end.

To execute your Java program you need to signal to the Java Virtual Machine where to start executing the program. In Java, all instructions (code) have to be located inside a **Java class**. A class is a way of grouping data and instructions that belong together. Thus, a class may contain both **variables** and **methods**. A variable can contain data, and a method groups together a set of operations on data (instructions).

A Java program needs to start its execution somewhere. A Java program starts by executing the main method of some class. You can choose the name of the class to execute, but not the name of the method. The method must always be called main .

Here is how the main method declaration looks when located inside the Java class declaration from earlier:

```
package myjavacode;

public class MyClass {

    public static void main(String[] args) {


    }

}
```

The three keywords **public**, **static** and **void** have a special meaning. Don't worry about them right now. Just remember that a main() method declaration needs these three keywords.

After the three keywords you have the method name. To recap, a method is a set of instructions that can be executed as if they were a single operation. By "**calling**" (executing) a method you execute all the instructions inside that method.

After the method name comes first a left parenthesis, and then a list of parameters. Parameters are variables (data / values) we can pass to the method which may be used by the instructions in the method to customize its behaviour. A main method must always take an array of String objects.

You declare an array of String objects like this:

```
String[] stringArray
```

Don't worry about what a **String** is, or what an **array** is. That will be explained in later texts. Also, it doesn't matter what name you give the parameter. In the main() method example earlier I called the String array parameter args, and in the second example I called it stringArray. You can choose the name freely.

After the method's parameter list comes first a left curly bracket ({), then some empty space, and then a right curly bracket (}). Inside the curly brackets you locate the Java instructions that are to be executed when the main method is executed. This is also referred to as the *method body*. In the example above there are no instructions to be executed. The method is empty.

Let us insert a single instruction into the main method body. Here is an example of how that could look:

```java
package myjavacode;

public class MyClass {

    public static void main(String[] args) {
        System.out.println("Hello World, Java Program");

    }

}
```

Now the main method contains this single Java instruction:

```java
System.out.println("Hello World, Java Program");
```

This instruction will print out the text Hello World, Java Program to the *console*. If you run your Java program from the command line, then you will see the output in the command line console (the textual interface to your computer). If you run your Java program from inside an IDE, the IDE normally catches all output to the console and makes it visible to you somewhere inside the IDE.

Running The main() Method:

When you start a Java program you usually do so via the command line (console). You call the java command that comes with the JRE, and tells it what Java class to execute, and what arguments to pass to the main() method. The Java application is then executed inside the JVM (or **by** the JVM some would claim). Here is a diagram illustrating this:

A command line executing the java command, which in turn executes a Java main program.
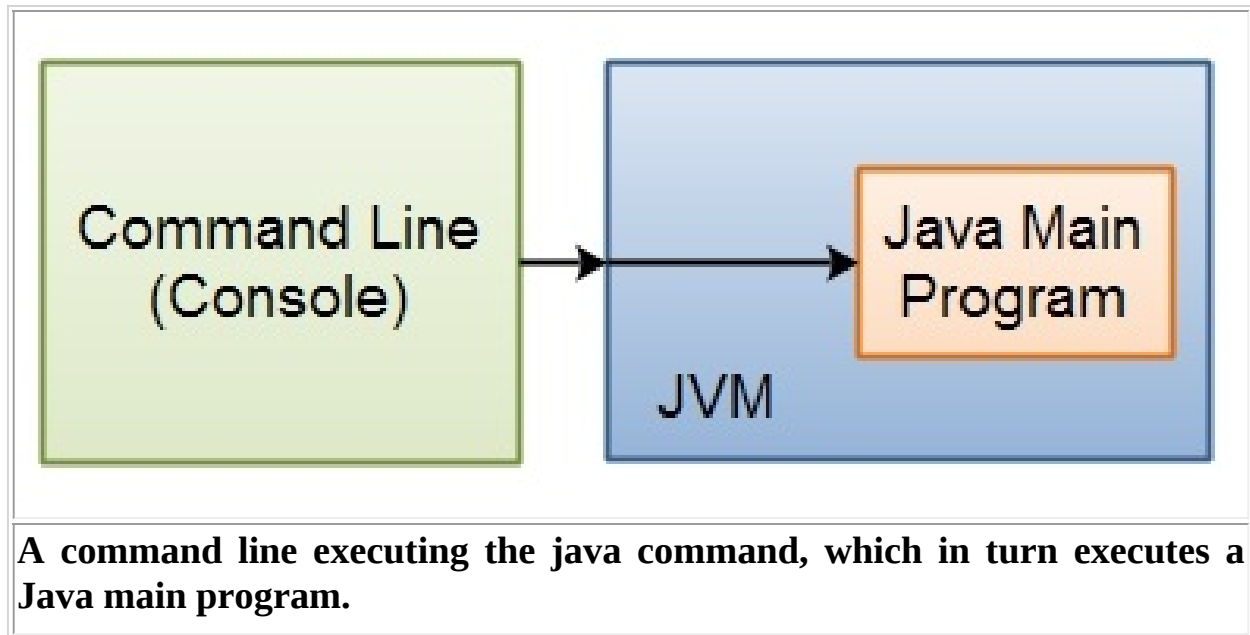
Here is an example command line:

```
java -cp classes myjavacode.MyClass
```

The first part of this command is the java command. This command starts up the JVM. In some cases you may have to specify the full path to where the java command is located on your computer (typically inside the bin subdirectory of the Java install dir).

The second and third arguments (-cp classes) tells the JVM in what directory the compiled Java classes are located (cp means class path). In this case the compiled Java classes are located in a directory named classes .

The fourth argument is the name of the Java class the JVM is to execute. Notice how the class name also contains the name of the package the class is located in (the "fully qualified class name").

Passing Arguments to the main() Method:

You can pass arguments from the command line to the main() method. This command line shows how:

```
java -cp classes myjavacode.MyClass Hello World
```

When the JVM executes the main() method of the myjavacode.MyClass , the String array passed as parameter to the main() method will contain two Strings: "Hello" and "World".

The main() method can access the arguments from the command line like this:

```
package myjavacode;

public class MyClass {

    public static void main(String[] args) {
        System.out.println( args[0] );
        System.out.println( args[1] );

    }

}
```

Notice the references to element 0 and element 1 in the args array (args[0] and args[1]). args[0] will contain the String (text) Hello and args[1] will contain the String World .

Variables and arrays will be explained in more detail in later texts. Don't worry if you don't fully understand them at this point.

The Java Main Class:

If only a single Java class in your Java program contains a main() method, then the class containing the main() method is often referred to as the *main class*.

You can have as many classes as you want in your project with a main() method in. But, the Java Virtual Machine can only be instructed to run one of them at a time. You can still call the other main() methods from inside the main() method the Java Virtual Machine executes (you haven't seen how yet) and you can also start up multiple virtual machines which each execute a single main() method.

Access Control

All good writers including those who write software know that a piece of work isn't good until it's been rewritten, often many times. If you leave a piece of code in a drawer for a while and come back to it, you may see a much better way to do it. This is one of the prime motivations for refactoring, which rewrites working code in order to make it more readable, understandable, and thus maintainable.1

There is a tension, however, in this desire to change and improve your code. There are often consumers (client programmers) who rely on some aspect of your code staying the same. So you want to change it; they want it to stay the same. Thus a primary consideration in object oriented design is to "separate the things that change from the things that stay the same."

This is particularly important for libraries. Consumers of that library must rely on the part they use, and know that they won't need to rewrite code if a new version of the library comes out. On the flip side, the library creator must have the freedom to make modifications and improvements with the certainty that the client code won't be affected by those changes.

This can be achieved through convention. For example, the library programmer must agree not to remove existing methods when modifying a class in the library, since that would break the client programmer's code. The reverse situation is thornier, however. In the case of a field, how can the library creator know which fields have been

accessed by client programmers? This is also true with methods that are only part of the implementation of a class, and not meant to be used directly by the client programmer. What if the library creator wants to rip out an old implementation and put in a new one? Changing any of those members might break a client programmer's code. Thus the library creator is in a strait jacket and can't change anything.

To solve this problem, Java provides access specifiers to allow the library creator to say what is available to the client programmer and what is not. The levels of access control from "most access" to "least access" are **public**, **protected**, **package access** (which has no keyword), and **private**. From the previous paragraph you might think that, as a library designer, you'll want to keep everything as "**private**" as possible, and expose only the methods that you want the client programmer to use. This is exactly right, even though it's often counterintuitive for people who program in other languages (especially C) and who are used to accessing everything without restriction.

Package in java: Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, *etc.*

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are −

- **java.lang** − bundles the fundamental classes

- **java.io** − classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, *etc.* It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Creating a Package:

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

Example: Let us look at an example that creates a package called animals. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals* −

```java
/* File name : Animal.java */

package animals;

interface Animal {
   public void eat();

   public void travel();

}
```

Now, let us implement the above interface in the same package *animals* −

```java
package animals;

/* File name : MammalInt.java */

public class MammalInt implements Animal {

```

```java
public void eat() {

    System.out.println("Mammal eats");

}

public void travel() {

    System.out.println("Mammal travels");

}

public int noOfLegs() {

    return 0;

}

public static void main(String args[]) {

    MammalInt m = new MammalInt();

    m.eat();

    m.travel();

}



                                                            }
```

Now compile the java files as shown below −

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```

You can execute the class file within the package and get the result as shown below.

Mammal eats

Mammal travels

The import Keyword: If a class wants to use another class in the same package, the package name need not be used. Classes in the same package find each other without any special syntax.

Example: Here, a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```java
package payroll;
public class Boss {
  public void payEmployee(Employee e) {
    e.mailCheck();
  }

                                                }
```

What happens if the Employee class is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example −

```java
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example −

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example −

```
import payroll.Employee;
```

Note − A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

Access Modifiers

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, and method or data member. There are four types of access modifiers available in java:

1. Default – No keyword required

1. Private

1. Protected

1. Public

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Default Access Modifier - No Keyword: Default access modifier means we do not explicitly declare an access modifier for a class, field, method, *etc.*

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example: Variables and methods can be declared without any modifiers, as in the following examples −

```java
String version = "1.5.1";


boolean processOrder() {
   return true;


                                                }
```

Private Access Modifier – Private: Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.

Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Example: The following class uses private access control :

```java
public class Logger {
  private String format;
  public String getFormat() {
    return this.format;
  }
  public void setFormat(String format) {
    this.format = format;
  }
}
```

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So, to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format,

and *setFormat(String)*, which sets its value.

Public Access Modifier – Public: A class, method, constructor, interface, *etc.* declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example: The following function uses public access control −

```
public static void main(String[] arguments) {
  // ...

                                                 }
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Protected Access Modifier – Protected: Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example: The following parent class uses protected access control, to allow its child class override *openSpeaker()* method −

```java
class AudioPlayer {
   protected boolean openSpeaker(Speaker sp) {
      // implementation details
   }

                              }

class StreamingAudioPlayer {
   boolean openSpeaker(Speaker sp) {
      // implementation details
   }

                              }
```

Here, if we define openSpeaker() method as private, then it would not be accessible from any other class other

than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intention is to expose this method to its subclass only, that's why we have used protected modifier.

Access Control and Inheritance The following rules for inherited methods are enforced :

- Methods declared public in a superclass also must be public in all subclasses.

- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.

- Methods declared private are not inherited at all, so there is no rule for them.

CHAPTER 5 | VARIABLES, DATA TYPES & KEYWORDS

Variables

Variables are used to store information to be referenced and manipulated in a computer program. They also provide a way of labeling data with a descriptive name, so our programs can be understood more clearly by the reader and ourselves. It is helpful to think of variables as containers that hold information. Their sole purpose is to label and store data in memory. This data can then be used throughout your program.

A variable's name is called an **identifier**. For the most part an identifier is a unique word that a programmer, or in this case you, picks to name a variable. An identifier is always something that a programmer invented to describe a variable; it's like naming a new pet or a baby.

Identifiers: Identifiers, which are considered symbols or tokens, are words that you invent which you then assign a meaning. Identifiers can be as simple as the letter i or as complex as @OhHAICanIHasIdentifier01.
identifier is the word that's used to name any function, variable, or type of data you create. When the keyword class is used it's followed by another word called an identifier. After properly identifying a function, or other chunk of data, you can now refer to that data by its identifier. In other words, when you name some data Charles you access that data by the name Charles.

```
class MyNewClassImWriting

                              {

                              }
```

Variables are created using declarations. Declaration is defined as a formal statement or announcement.

Each set of words a programmer writes is called a statement. In English we'd use the word sentence, but programmers like to use their own vocabulary. Declaration statements for variables define both the type and the identifier for a variable.

```
public class Example

                              {

      int i;

                              }
```

Programmers use a semicolon (;) rather than a period to end the statement.

Therefore, if you want to sound like a programmer you can say you can "write a statement to declare a variable of type int with the identifier i." Or if you want to be overly dramatic you can proclaim "I declare a new variable of type int to be known as i!" and so it shall be.

Variable Names

It's important to know that variable identifiers and class identifiers can be pretty much anything. There are some rules to naming anything when programming. Here are some guidelines to help. Long names are more prone to typos, so keep identifiers short. A naming convention for variables should consider the following points.

The variable name should indicate what it's used for, or at least what you're going to do with it. This should be obvious, but a variable name shouldn't be misleading. Or rather, if you're using a variable named radius, you shouldn't be using it as a character's velocity. It's also helpful if you can pronounce the variable's name; otherwise you're likely to have issues when trying to explain your code to another programmer.

```
int someLong_complex_hardToRememberVariableName;
```

There is an advantage to keeping names as short as possible but still quite clear. Your computer screen, and most computers for that matter, can only fit so many characters on a single line. You could make the font smaller, but then you run into readability issues when letters get too small. Consider the following function, which requires more than one variable to work.

```
SomeCleverFunction(TopLeftCorner           –           SomeThickness         +
OffsetFromSomePosition,    BottomRightCorner    –    SomeThickness    +
OffsetFromSomePosition);
```

The code above uses many long variable names. Because of the length of each variable, the statement takes up multiple lines making a single statement harder to read. We could shorten the variable names, but it's easy to shorten them too much.

```
CleverFunc(TL–Thk+Ofst,LR–Thk+Ofst);
```

Variable names should be descriptive, so you know what you're going to be using them for: too short and they might lose their meaning.

```
int a;
```

While short and easy to remember, it's hard for anyone else coming in to read your code and know what you're using the variable a for. This becomes especially difficult when working with other programmers. Variable naming isn't completely without rules.

`int 8;`

A variable name can't be a number. This is bad; numbers have a special place in programming as much of it has other uses for them. IDE will try to help you spot problems. A squiggly red line will appear under any problems it spots. And speaking of variable names with numbers, you can use a number as part of a variable name.

`int varNumber2;`

The above name is perfectly valid, and can be useful, but conversely consider the following.

`int 13thInt;`

Variable names can't start with any numbers. To be perfectly honest, I'm not sure why this case breaks the compiler, but it does seem to be related to why numbers alone can't be used as variable names.

`int $; int this-that; int (^_^);`

Most special characters also have meanings, and are reserved for other uses. For instance, in JAVA a - is used for subtracting; in this case JAVA may think you're trying to subtract that from this. Keywords, you should remember,

are also invalid variable names as they already have a special meaning for JAVA.

In IDE (integrated Development Environment), you might notice that the word this is highlighted, indicating that it's a keyword. Spaces in the middle of a variable are also invalid.

```
int Spaces are bad;
```

Most likely, adding characters that aren't letters will break the compiler. Only the underscore and letters can be used for identifier names. As fun as it might be to use emoticons for a variable, it would be quite difficult to read when in use with the rest of the code.

```
int ADifferenceInCase; int adifferenceincase;
```

The two variables here are actually different. Case-sensitive languages like java do pay attention to the case of a character; this goes for everything else when calling things by name. Considering this: A is different from a.

As a programmer, you need to consider what a variable should be named. It must be clear to you and anyone else with whom you'll be sharing your work with. You'll also be typing your variable name many times, so they should be short and easy to remember and type. The last character we discuss here is the little strange @ or at. The @ can be used only if it's the first character in a variable's name.

```
int @home;
```

```
int noone@home;
```

In the second variable declared here we'll get an error. Some of these less regular characters are easy to spot in your code. When you have a long list of variables it's sometimes best to make them stand out visually. Some classically trained programmers like to use an underscore to indicate a class scope variable. The underscore is omitted in variables which exist only within a function. You would find the reason for the odd rule regarding @ when you use int, which is reserved as a keyword. You're allowed to use int @int, after which you can assign @int any integer value. However, many programmers tend to use MyInt, mInt, or _ int instead of @int based on their programming upbringing.

Good programmers will spend a great deal of time coming up with useful names for their variables and functions. Coming up with short descriptive names takes some getting used to, but here are some useful tips. Variables are often

named using nouns or adjectives as they describe an attribute related to what they're used for.

In the end once you start using the name of the variable throughout the rest of your code, it becomes harder to change it as it will need to be changed everywhere it's used. Doing a global change is called refactoring, and this happens so often that there is software available to help you "refactor" class, variable, and function names.

NOTE:

You may also notice the pattern in which uppercase and lowercase letters are used. This is referred to as either BumpyCase or CamelCase. Sometimes, the leading letter is lowercase, in which case it will look like headlessCamelCase rather than NormalCamelCase. Many long debates arise between programmers as to which is correct, but in the end either one will do. Because Java is case sensitive, you and anyone helping you should agree whether or not to use a leading uppercase letter. These differences usually come from where a person learned how to write software or who taught that person. The use of intermixed uppercase and lowercase is a part of programming style. Style also includes how white space is used.

Types of Variables

You must declare all variables before they can be used. Following is the basic form of a variable declaration:

data type variable [= value][, variable [= value] ...] ;

Here *data type* is one of Java's datatypes and *variable* is the name of the variable.   To declare more than one variable of the specified type, you can use a comma-separated list.

Example:

```
int  a, b, c;      // Declares three int  a, b, and c.

int  a = 10, b = 10;  // Example of initialization

byte  B = 22;  // initializes a byte type variable B.

double pi = 3.14159;  // declares and assigns a   value of PI.
```

Java Programming language defines mainly three kinds of variables.

1. Instance variables

2. Static Variables

3. Local Variables

1) Instance variables

Instance variables are variables that are declare inside a class but outside any method, constructor or block. Instance variable are also variable of object commonly known as field or property. They are referred as object variable. Each object has its own copy of each variable and thus, it doesn't affect the instance variable if one object changes the value of the variable. Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.

```
class Student
                                  {
  String name;
   int age;
                                  }
```

Here **name** and **age** are instance variable of   Student class.

2) Static variables
Static are class variables declared with static   keyword. Static variables are initialized only once. Static variables are also used in declaring constant along with final keyword.

class Student

{

String name;
int age;
 static int instituteCode=1101;

}

Here **instituteCode** is a static variable. Each object of Student class will share instituteCode property.

Additional points on static variable:

- static variable are also known as class variable.

- static means to remain constant.

- In Java, it means that it will be constant for all the instances created for that class.

- static variable need not be called from object.

- It is called by *classname.static variable name*

Note: A static variable can never be defined inside a method *i.e.* it can never be a local variable.

Example:

Suppose you make 2 objects of class Student and you change the value of static variable from one object. Now when you print it from other object, it will display the changed value. This is because it was declared static *i.e.* it is constant for every   object created.

```java
class Student{
    int a;
    static int id = 35;

    void change(){

        System.out.println(id);

    }

}

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

        Student o1 = new Student();
        Student o2 = new Student();

        o1.change();
        Student.id = 1;
          o2.change();

    }

}
```

Output:
        35
         1

3) Local variables

Local variables are declared in method, constructor or block. Local variables are initialized when method, constructor or block start and will be   destroyed once its end. Local variable reside in stack. Access modifiers are not used for local variable.

```
float getDiscount(int price)

                                {

 float discount;
 discount=price*(20/100);
  return discount;

                                }
```

Here **discount** is a local variable.

Additional points on Local variable:

- Local variables are declared in methods, constructors, or blocks.

- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block.

- Access modifiers cannot be used for local variables.

- Local variables are visible only within the declared method, constructor, or block.

- Local variables are implemented at stack level internally.

- There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

Example:

```java
public class Test {

  public void pupAge() {

    int age = 0;

    age = age + 7;

    System.out.println("Puppy age is : " + age);

  }

  public static void main(String args[]) {

    Test test = new Test();

    test.pupAge();

  }

}
```

Here, *age* is a local variable. This is defined   inside *pupAge()* method and its scope is limited to only this method.

This will produce the following result: Output Puppy age is: 7

Data Types in Java

Data types represent the different values to be stored in the variable. In java, there are two types of data types:

- Primitive data types
- Non-primitive data types
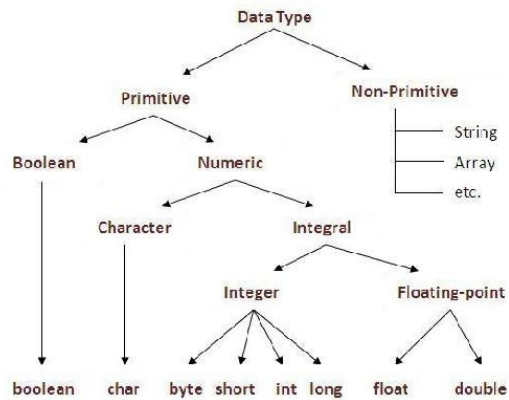
Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	**0.0f**	4 byte
double	**0.0d**	8 byte

Why char uses 2 byte in java and what is \u0000 ?
It is because java uses Unicode system than ASCII code system. The \u0000 is the lowest range of Unicode system. To get detail explanation about Unicode visit next page.

Types

When we assign a variable a value such as int i = 1; we're making an interesting assumption. We assume using 1 automatically infers a value type that matches the variable type. Computers use a wide variety of different types of data for storing numbers. So far we've been seeing the word int being thrown around as though you know what an int is. In our everyday lives we hardly think there's any difference between numbers 1 and 1.0 or even the word one.

As humans we can easily conceptualize numbers as units of measure and converting between them. The conversion between the word one and the number 1 isn't so easy for a computer. The keyword int is short for integer. Integers are whole number values. The integer is a Java built-in type. Basically, this means that integers are a fundamental part of java. Other built-in types include float, double, string, bool, and object. Less commonly used built-in types are char, short, and byte. Altogether, there are 15 different built-in types.

All of these, except object and string, are value types. Every type of data you're able to build must be based on all of these built-in types. The system which creates all of these floats, doubles, and bools is rooted in the origin of computing. Remember the punch tapes and pieces of paper with holes in them? Those were records of 1s and 0s which were fed into computers for storing in the form of mechanical switches, either on or off. The patterns represented numbers or instructions and logic.

Today, we still use the same system of 1s and 0s, or binary, only we don't need to punch holes in paper to tell the computer what to do. The methods of creating and storing the instructions have become many times more complex, but nonetheless the principle of the 1 and 0 are the same. These are called bits; one possible origin for that name is the little bits of paper that were left over from punching all of the holes in the paper cards.

Value and Reference Types:

The int, float, double, and bool are commonly used value types. The basics of this usage pattern relate to the system that stores and organizes the computer's memory. When you talk about computer storage you use the word megabyte or gigabyte. Value types are stored with a very specific purpose in mind: to keep track of a numeric value.

NOTE:
The word mega refers to how many millions of bytes a component in your computer can store. Giga indicates how every many billions of bytes are being stored. Have you ever thought about what a byte actually is? A byte is what is called an 8-bit unsigned integer, or a system of using 8 bits to form a number. What this means is that it's a whole number like 0, 7, or 32,767. The word unsigned indicates that the number cannot be negative, like −512. A 1 or 0 in computer terms is called a bit. We won't go into detail on how computers use bits to count, and it's a rather fun thing to learn on your own.

However, it's important to know that a byte has a limited range, from 0 to 255. I'll just leave you with the idea that you could count up to 1023 on your 10 fingers if you used binary rather than decimal. Your fingers can be used to represent a 10-bit unsigned integer. The computer's calculation capabilities used to be far more limited than they are today. An 8-bit game console made in the 1980s had a limited number of colors, 256 to be exact.
This limitation was based on the number of bits that the processor could handle. The processor had a limited number of transistors which could be used at any one time. Shortly after, floating point coprocessors were introduced, which had a much larger numeric range allowed, by having

closer to 32 bits to work with. In all of these cases, when you use a value type, that number is actually stored as 1s and 0s in your computer's RAM. This should be considered fairly remarkable.

In the past you'd have to go through flaming hoops to store a value. Now you just type in float f = 3.1415926535; and you can measure the circumference of the universe accurately to within the width of a single atom. What all of these types have in common is that they are stored as a single element. Large values like double gigawatts = 1210000000.0; and double mint = 2.0; use the same amount of memory.

Your computer doesn't assign one double or the other more space in memory. These types are referred to as primitive types. The only difference between an int and a float is the number of bits they use at a time, 8 and 32 respectively; doubles use 64 bits.

The string and object types differ a bit in how they are stored. These types are a composite of any number smaller elements. The bigger a string or object the bigger chunk of memory the computer opens up to place that object or string. These are called reference types or sometimes called nullable types because Java doesn't look to a single element in memory to get data from it.

Strong Typing:

This has nothing to do with how hard you type on the keyboard!

Java sees a difference between numbers in how they are written. A whole number like 1 is different from 1.0, even though we can think of them as having the same value. Some of these differences can be added by adding a letter. Therefore, 1.0 and 1.0f are actually different numbers as well. Java is a strongly typed programming language. When a programming language uses strong types you're expected to keep the different types separated.

This means that a 1 + 1.0f can create a problem. Because the two numbers are actually different types of numbers we have to convert one of them to match the other before the operation is allowed to take place. To convert from one type to another we have to tell Java that we intend to make that conversion from a 1 to a 1.0 by using what's called a cast operator. Aside from changing how memory is used, types also limit any problems which might come about when working in a team of programmers

Understanding Floating Points

Floating point numbers have been a focus of computers for many years. Gaining floating point accuracy was the goal of many early computer scientists. Because there are an infinite possibilities of how many numbers can follow a decimal point, it's impossible to truly represent any fraction completely using binary computing.

A common example is π, which we call pi. It's assumed that there are an endless number of digits following 3.14. Even today computers are set loose to calculate the hundreds of billions of digits beyond the decimal point.

Computers set aside some of the bits of a floating point number aside to represent where the decimal appears in the number, but even this is limited. The first bit is usually the sign bit, setting the negative or positive range of the number. The following 8 bits is the exponent for the number called a mantissa. The remaining bits are the rest of the number appearing around the decimal point. A float value can move the decimal 38 digits in either direction, but it's limited to the values it's able to store.

Without special considerations, computers are not able to handle arbitrarily large numbers. To cast a float into an int, you need to be more explicit.

That's why Java requires you to use the cast and you need to add the (int) in

```
int Zint = (int)Zmove;
```

The (int) is a cast operator; or rather (type) acts as a converter from the type on the right to the type needed on the left.

Keywords

Keywords are special words, or symbols, that tell the compiler to do specific things. For instance, the keyword class at the beginning of a line tells the compiler you are creating a class. A class declaration is the line of code that tells the compiler you're about to create a new class. The order in which words appear is important. English requires sentences and grammar to properly convey our thoughts to the reader.

In code, Java or otherwise, programming requires statements and syntax to properly convey instructions to the computer.

class className
{
// code goes here
}

Every class needs a name; in the above example we named our class className, although we could have easily named the class Charles. When a new class is named the name becomes a new identifier. This also holds true for every variable's name, though scope limits how long and where the variable's identifier exists.

We have learned about variables and scope.
You can't use keywords for anything other than what Java expects them to be used for. There are exceptions, but in general, keywords provide you with specific commands.
In the Java programming language, a **keyword** is one of **53** reserved words that have a predefined meaning in the language; because of this, programmers cannot use keywords as names for variables, methods, classes, or as

any other identifier. Due to their special functions in the language, most integrated development environments for Java use syntax highlighting to display keywords in a different color for easy identification.

List of Keywords:

abstract

Abstract is used to implement the abstraction in java. A method which doesn't have method definition must be declared as abstract and the class containing it must be declared as abstract. You can't instantiate abstract classes. Abstract methods must be implemented in the sub classes. You can't use abstract keyword with variables and constructors **assert (added in J2SE 1.4)** Assert describes a predicate (a true–false statement) placed in a java-program to indicate that the developer thinks that the predicate is always true at that place. If an assertion evaluates to false at run-time, an assertion failure results, which typically causes execution to abort. Optionally enable by ClassLoader method.

boolean

Defines a boolean variable for the values "true" or "false" only.

break

Used to end the execution in the current loop body.

byte

The byte keyword is used to declare a field that can hold an 8-bit signed two's complement integer. This keyword is also used to declare that a method returns a value of the primitive type byte.

case

A statement in the `switch` block can be labeled with one or more `case` or `default` labels. The `switch` statement evaluates its expression, then executes all statements that follow the matching `case` label;

catch

Used in conjunction with a try block and an optional finally block. The statements in the catch block specify what to do if a specific type of exception is thrown by the try block.

char

Defines a character variable capable of holding any character of the java source file's character set.

class

A type that defines the implementation of a particular kind of object. A class definition defines instance and class fields, methods, and inner classes as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass is implicitly Object. The class keyword can also be used in the form Class**.class** to get a Class object without needing an instance of that class.

For example, **String.class**can be used instead of doing **new String().getClass()**.

continue

Used to resume program execution at the end of the current loop body. If followed by a label, continue resumes execution at the end of the enclosing labeled loop body.

default

The default keyword can optionally be used in a switch statement to label a block of statements to be executed if no case matches the specified value; see *switch*. Alternatively, the default keyword can also be used to declare default values in a Java annotation. From Java 8 onwards, the default keyword is also used to specify that a method in an interface provides the default implementation of a method.

do

The do keyword is used in conjunction with while to create a do-while loop, which executes a block of statements associated with the loop and then tests a boolean expression associated with the while. If the expression evaluates to true, the block is executed again; this continues until the expression evaluates to false.

double

The double keyword is used to declare a variable that can hold a 64-bit double precision IEEE 754 floating-point number. This keyword is also used to declare that a method returns a value of the primitive type double.

else

The else keyword is used in conjunction with if to create an if-else statement, which tests a boolean expression; if the expression evaluates to true, the block of statements associated with the if are evaluated; if it evaluates to false, the block of statements associated with the else are evaluated.

enum (added in J2SE 5.0) A Java keyword used to declare an enumerated type. Enumerations extend the base class Enum.

extends

Used in a class declaration to specify the superclass; used in an interface declaration to specify one or more superinterfaces. Class X extends class Y to add functionality, either by adding fields or methods to class Y, or by overriding methods of class Y. An interface Z extends one or more interfaces by adding methods. Class X is said to be a subclass of class Y; Interface Z is said to be a subinterface of the interfaces it extends.

Also used to specify an upper bound on a type parameter in Generics.

final

Define an entity once that cannot be changed nor derived from later. More specifically: a final class cannot be subclassed, a final method cannot be overridden, and a final variable can occur at most once as a left-hand expression on an executed command. All methods in a final class are implicitly final.

finally

Used to define a block of statements for a block defined previously by the try keyword. The finally block is executed after execution exits the try block and any associated catch clauses regardless of whether an exception was thrown or caught, or execution left method

in the middle of the `try` or `catch` blocks using the `return` keyword.

float

The float keyword is used to declare a variable that can hold a 32-bit single precision IEEE 754 floating-point number. This keyword is also used to declare that a method returns a value of the primitive type float.

for

The for keyword is used to create a for loop, which specifies a variable initialization, a boolean expression, and an incrementation. The variable initialization is performed first, and then the boolean expression is evaluated. If the expression evaluates to true, the block of statements associated with the loop are executed, and then the incrementation is performed. The boolean expression is then evaluated again; this continues until the expression evaluates to false.

As of J2SE 5.0, the for keyword can also be used to create a so-called "enhanced for loop", which specifies an arrayor Iterable object; each iteration of the loop executes the associated block of statements using a different element in the array or Iterable.

if

The if keyword is used to create an if statement, which tests a boolean expression; if the expression evaluates to true, the block of statements associated with the if statement is executed. This keyword can also be used to create an if-else statement; see *else*.

implements

Included in a class declaration to specify one or more interfaces that are implemented by the current class. A class inherits the types and abstract methods declared by the interfaces.

import

Used at the beginning of a source file to specify classes or entire Java packages to be referred to later without including their package names in the reference. Since J2SE 5.0, import statements can import static members of a class.

instanceof

A binary operator that takes an object reference as its first operand and a class or interface as its second operand and produces a boolean result. The instanceof operator evaluates to true if and only if the runtime type of the object is assignment compatible with the class or interface.

int

The int keyword is used to declare a variable that can hold a 32-bit signed two's complement integer. This keyword is also used to declare that a method returns a value of the primitive type int.

interface

Used to declare a special type of class that only contains abstract or default methods, constant (static final) fields and static interfaces. It can later be implemented by classes that declare the interface with the implements keyword.

long

The `long` keyword is used to declare a variable that can hold a 64-bit signed two's complement integer. This keyword is also used to declare that a method returns a value of the primitive type `long`.

native

Used in method declarations to specify that the method is not implemented in the same Java source file, but rather in another language.

new

Used to create an instance of a class or array object. Using keyword for this end is not completely necessary

(as exemplified by [Scala](#)), though it serves two purposes: it enables the existence of different namespace for methods and class names, it defines statically and locally that a fresh object is indeed created, and of what runtime type it is (arguably introducing dependency into the code).

package

A group of types. Packages are declared with the `package` keyword.

private

The private keyword is used in the declaration of a method, field, or inner class; private members can only be accessed by other members of their own class.

protected

The protected keyword is used in the declaration of a method, field, or inner class; protected members can only be accessed by members of their own class, that class's subclasses or classes from the same package.

public

The `public` keyword is used in the declaration of a class, method, or field; public classes, methods, and fields can be accessed by the members of any class.

return

Used to finish the execution of a method. It can be followed by a value required by the method definition that is returned to the caller.

short

The short keyword is used to declare a field that can hold a 16-bit signed two's complement integer. This keyword is also used to declare that a method returns a value of the primitive type short.

static

Used to declare a field, method, or inner class as a class field. Classes maintain one copy of class fields regardless of how many instances exist of that class. static also is used to define a method as a class method. Class methods are bound to the class instead of to a specific instance, and can only operate on class fields. (Classes and interfaces declared as static members of another class or interface are actually top-level classes and are *not* inner classes.) **strictfp (added in J2SE 1.2)** A Java keyword used to restrict the precision and rounding of floating point calculations to ensure portability.

super

Used to access members of a class inherited by the class in which it appears. Allows a subclass to access overridden methods and hidden members of its superclass. The super keyword is also used to forward a call from a constructor to a constructor in the superclass.

Also used to specify a lower bound on a type parameter in Generics.

switch

The switch keyword is used in conjunction with case and default to create a switch statement, which evaluates a variable, matches its value to a specific case, and executes the block of statements associated with that case. If no case matches the value, the optional block labelled by default is executed, if included.

synchronized Used in the declaration of a method or code block to acquire the mutex lock for an object while the current thread executes the code. For static methods, the object locked is the class's Class. Guarantees that at most one thread at a time operating on the same object executes that code. The mutex lock is automatically released when execution exits the synchronized code. Fields, classes and interfaces cannot be declared as *synchronized*.

this

Used to represent an instance of the class in which it appears. this can be used to access class members and as a reference to the current instance. The this keyword is also used to forward a call from one constructor in a class to another constructor in the same class.

throw

Causes the declared exception instance to be thrown. This causes execution to continue with the first enclosing exception handler declared by the `catch` keyword to handle an assignment compatible exception type. If no such exception handler is found in the current method, then the method returns and the process is repeated in the calling method. If no exception handler is found in any method call on the stack, then the exception is passed to the thread's uncaught exception handler.

throws

Used in method declarations to specify which exceptions are not handled within the method but rather passed to the next higher level of the program. All uncaught exceptions in a method that are not instances of RuntimeException must be declared using the throws keyword.

transient

Declares that an instance field is not part of the default serialized form of an object. When an object is serialized, only the values of its non-transient instance fields are included in the default serial representation. When an object is deserialized, transient fields are initialized only to their default value. If the default form is not used, *e.g.* when a *serialPersistentFields*table is declared in the class hierarchy, all transient keywords are ignored.

try

Defines a block of statements that have exception handling. If an exception is thrown inside the try block, an optional catch block can handle declared exception types. Also, an optional finally block can be declared that will be executed when execution exits the try block and catch clauses, regardless of whether an exception is thrown or not. A try block must have at least one catch clause or a finally block.

void

The `void` keyword is used to declare that a method does not return any value.

volatile

Used in field declarations to specify that the variable is modified asynchronously by concurrently running threads. Methods, classes and interfaces thus cannot be declared *volatile*, nor can local variables or parameters.

while

The while keyword is used to create a while loop, which tests a boolean expression and executes the block of statements associated with the loop if the expression evaluates to true; this continues until the expression evaluates to false. This keyword can also be used to create a do-while loop; **Reserved words for literal values:**

true

A boolean literal value.

null

A reference literal value.

false

A boolean literal value.

Return Keyword

We need to love the return keyword. This keyword turns a function into data. There are a couple of conditions that need to be met before this will work. So far, we've been using the keyword void to declare the return type of a function. This looks like the following code fragment.

```
void MyFunction()

                                        {

//code here...

                                        }
```

In this case, using return will be pretty simple.

```
void MyFunction()

                                        {

//code here ...
return;

                                        }
```

This function returns void. This statement has a deeper meaning. Returning a value makes a lot more sense when a real value, something other than a void, is actually returned. Let's take a look at a function that has more meaning.

The keyword void at the beginning of the function declaration means that this function does not have a return type. If we change the declaration, we need to ensure that there is a returned value that matches the declaration.

This can be as simple as the following code fragment.

```
int MyFunction()
```

```
                                  {
```

```
//code here...
return 1;
//1 is an int
```

```
                                  }
```

This function returns int 1.

Declaring a function with a return value requires that the return type and the declaration match. When the function is used, it should be treated like a data type that matches the function's return type.

Are Errors Bad?

Don't be afraid to make mistakes. The code you write cannot break IDE or your computer. At least nothing you'll learn from this book will enable you to do so. As a matter of fact, it is often good to produce errors. The errors tell us what we can and cannot do. Programmers often ask themselves, "Will the compiler let me do this?" which may produce an error. If not, then the programmer will continue his thought with

"huh, I guess it will, so that means I can …"

Creating and fixing errors tells us what the compiler expects from us. It is a sort of a conversation between the programmer and the computer.

for example, the programmer asking questions in the form of a code statement and the computer replying with an error to the programmer who asks for things the computer can't do.

Compile Time and Run Time Errors

At compile time, when the code does not comply with the Java syntactic and semantics rules as described in Java Language Specification (JLS), compile-time errors will occurs. The goal of the compiler is to ensure the code is compliant with these rules. Any rule-violations detected at this stage are reported as compilation errors.

The best way to get to know those rules is to go through all the sections in the JLS containing the key words "compile-time error".

In general, these rules include syntax checking: declarations, expressions, lexical parsing, file-naming conventions etc;

exception handling: for checked exceptions;

accessibility, type-compatibility, name resolution: checking to see all named entities - variables,

classes, method calls *etc.* are reachable through at least one of the declared path;

etc.

The following are some common compile time errors:

- a class tries to extend more than one class

Overloading or overriding is not implemented correctly

- attempt to refer to a variable that is not in the scope of the current block
- an inner class has the same name as one of one of its enclosing classes
- a class contains one or more abstract methods and the class itself is not declared "abstract"
- a class tries to reference a private member of another class
- trying to create an instance of an abstract class

- trying to change the value of an already initialized constant (final member)
- declare two (class or instance) members with the same name

When the code compiles without any error, there is still chance that the code will fail at run time.

The errors only occurs at run time are call run time errors. Run time errors are those that passed compiler's checking, but fails when the code gets executed. There are a lot of causes may result in runtime errors, such as incompatible type-casting, referencing an invalid index in an array, using an null-object, resource problems like unavailable file-handles, out of memory situations, thread dead-locks, infinite loops(not detected!), *etc.*

The following are some common runtime errors:

- trying to invoke a method on an uninitialized variable (NullPointerException)
- ran out memory (memory leaks...) (OutOfMemoryError)
- trying to open a file that doesn't exist (FileNotFoundException)
- trying to pass arguments to a method which are not within the accepted bounds (IllegalArgumentException)
- trying to invoke the start() method on a dead thread (IllegalThreadStateException)

CHAPTER 6 | FUNCTIONS AND OPERATORS

What Are Functions?

Functions, sometimes called methods, contain statements which can process data. The statements can or cannot process data. Methods can be accessed by other statements. This action is referred to as calling a function or making a function call.

Functions may look different in different programming languages, but the way they work is mostly the same. The usual pattern is taking in data and using logic to manipulate that data. Functions may also be referred to by other names, for example, methods. The major differences come from the different ways the languages use syntax. Syntax is basically the use of spaces or tabs, operators, or keywords.

In the end, all you're doing is telling the compiler how to convert your instructions into computer-interpreted commands. Variables and functions make up the bulk of programming. Any bit of data you want to remember is stored in a variable. Variables are manipulated by your functions. In general, when you group variables and functions together in one place, you call that a class.

Example:
public void PrintNum ()

```
{
     System.out.println (anotherNum);
}
```

When writing a new function, it's good practice to fill in the entirety of the function's layout before continuing on to another task. This puts the compiler at ease; leaving a function in the form void MyFunction and then moving on to another function leaves the compiler confused as to what you're planning on doing. The integrated development environment, in this case MonoDevelop, is constantly reading and interpreting what you are writing, somewhat like a spell checker in a word processor. When it comes across a statement that has no conclusive form, like a variable, function, or class definition, its interpretation of the code you're writing will raise a warning or an error. MonoDevelop might seem a bit fussy, but it's doing its best to help.

Writing a Function:

A function consists of a declaration and a body. Some programmers like to call these methods, but semantics aside, a function is basically a container for a collection of statements.

Let's continue with the example:

```
void MyFunction ()

{

}
```

Here is a basic function called MyFunction. We can add in additional keywords to modify the function's visibility. One common modifier we'll be seeing is public.

```
public void MyFunction ()

{

}
```

The public keyword needs to appear before the return type of the function. In this case, it's void, which means that the function doesn't return anything. but functions can act as a value in a few different ways. For reference, a function that returns an int would look like this. A return statement of some kind must always be present in a function that has a return type.

public int MyFunction ()

{

```
    return 1;
```

```
                                }
```

The public modifier isn't always necessary, unless you need to make this function available to other classes. If this point doesn't make sense, it will soon. The last part of the function that is always required is the parameter list. It's valid to leave it empty, but to get a feeling for what an arg, or argument in the parameter list, looks like, move on to the next **example.**

```
public void MyFunction (int i)
```

```
                                {
```

```
                                }
```

For the moment, we'll hold off on using the parameter list, but it's important to know what you're looking at later on so it doesn't come as a surprise. Parameter lists are used to pass information from outside of the function to the inside of the function. This is how classes are able to send information from one to another.

Function declaration is similar to class declaration. Functions are the meat of where logic is done to handle variables.

```
public int MyFunction ()

                                {

        return 1;

                                }
```

The public modifier isn't always necessary, unless you need to make this function available to other classes. If this point doesn't make sense, it will soon. The last part of the function that is always required is the parameter list. It's valid to leave it empty, but to get a feeling for what an arg, or argument in the parameter list, looks like, move on to the next **example.**

```
public void MyFunction (int i)

                                {

                                }
```

For the moment, we'll hold off on using the parameter list, but it's important to know what you're looking at later on so it doesn't come as a surprise. Parameter lists are used to pass information from outside of the function to the inside of the function. This is how classes are able to send information from one to another.

Function declaration is similar to class declaration. Functions are the meat of where logic is done to handle variables.

public int MyFunction ()

```
                          {
```

```
        return 1;
```

```
                          }
```

The public modifier isn't always necessary, unless you need to make this function available to other classes. If this point doesn't make sense, it will soon. The last part of the function that is always required is the parameter list. It's valid to leave it empty, but to get a feeling for what an arg, or argument in the parameter list, looks like, move on to the next **example.**

```
public void MyFunction (int i)
```

```
                          {
```

```
                          }
```

For the moment, we'll hold off on using the parameter list, but it's important to know what you're looking at later on so it doesn't come as a surprise. Parameter lists are used to pass information from outside of the function to the inside of the function. This is how classes are able to send information from one to another.

Function declaration is similar to class declaration. Functions are the meat of where logic is done to handle variables.

public int MyFunction ()

{

```
    return 1;
```

```
                                  }
```

The public modifier isn't always necessary, unless you need to make this function available to other classes. If this point doesn't make sense, it will soon. The last part of the function that is always required is the parameter list. It's valid to leave it empty, but to get a feeling for what an arg, or argument in the parameter list, looks like, move on to the next **example.**

```
public void MyFunction (int i)
```

```
                                  {
```

```
                                  }
```

For the moment, we'll hold off on using the parameter list, but it's important to know what you're looking at later on so it doesn't come as a surprise. Parameter lists are used to pass information from outside of the function to the inside of the function. This is how classes are able to send information from one to another.

Function declaration is similar to class declaration. Functions are the meat of where logic is done to handle variables.

Function Scope:

Variables often live an ephemeral life. Some variables exist only over a few lines of code. Variables may come into existence only for the moment a function starts and then disappear when the function is done. Variables in the class scope exist for as long as the class exists. The life of the variable depends on where it's created.

Example:

```
public class Classscopm{
   int ClassInt;
void first()
```

```
                                  {
```

```
      int firsttint;
```

```
                                  }
```

```
void second()
```

```
                                  {
```

```
      int secondint;
```

```
                                  }
```

```
                                  }
```

If we look at the above figure we can visualize how scope is divided. The outer box represents who can see ClassInt. Within the first () function we have a firsttint that only exists within the first () function.

The same is repeated for the secondint, found only in the second () function. This means that first () can use both ClassInt and firsttint but not secondint. Likewise, second () can see ClassInt and secondint but not firsttint.

Parameter Lists:

When a function is called, its identifier followed by its parameter list is added to code. Parameters are like little mail slots on the front door of a house. There might be a slot for integers, floats, and arrays. Each slot on the door, or argument in the parameter list, is a type followed by an identifier.

If we look at a basic example, we'll see how all this works.

```
public class Example {
    int a = 0;
void SetA (int i) {
    a = i;
                                          }
                                          }
```

void SetA (int i) takes in one parameter of type int. This is a simple example of a value parameter; there are other types of parameters which we'll go into next. The value parameter declares the type and an identifier whose scope is limited to the contents of the function it's declared with. When you run the above program and give value 3 to the parameter of the function SetA(3) you'll see a 0 followed by a 3. We can do this any number of times, and each time we're allowed to change what value we set A to. Of course, it's easier to use a = 3 in the class, but then we wouldn't be learning anything.

Side Effects: When a function directly sets a variable in the class the function lives in, programmers like to call this a side effect. Side effects are usually things that programmers try to avoid, but it's not always practical. Writing functions with side effects tend to prevent the function from being useful in other classes. We'll find other systems that allow us to avoid using side effects, but in some cases, it's necessary.

```
Class MySideEffect {

        int a = 0;

void SetA ()

                                {

        a = 5;

                                }

                                }
```

The above SetA() function sets int a to 5. The SetA() function does have a clear purpose, setting A, but there's nothing in the function that indicates where the variable it's setting lives. If the variable lived in another class which MySideEffect inherited from, then you wouldn't even see the variable in this class. Once the complexity in a class grows, where and when a variable gets changed or read becomes more obscure. If a function remains self-contained, testing and fixing that function becomes far easier.

Reading what a function does should not involve jumping around to find what variable it's making changes to. This brings us back to the topic of scope. Limiting the scope or reach of a function helps limit the number of places where things can go wrong.

```
class MySideEffect {

    int a = 0;

void SetA ()

                                        {

        a = 5;

                                        }

void SetAgain ()

                                        {

            a = 10;

                                        }

                                        }
```

The more functions that have a side effect, the more strange behaviors might occur. If Main () function is expecting one value but gets something else entirely, you can start to get unexpected behaviors. Worse yet, if SetAgain() is called from another class and you were expecting a to be 5, you're going to run into more strange behaviors.

Multiple Arguments:

There aren't any limits to the number of parameters that a function can accept. Some languages limit you to no more than 16 arguments, which seems acceptable. If your logic requires more than 16 parameters, it's probably going to be easier to separate your function into different parts. However, if we're going to be doing something simple, we might want to use more than one parameter anyway.

```
int a = 0;

void SetA (int i, int j) {
    a = i * j;

                                    }
```

We can add a second parameter to SetA. A comma token tells the function to accept another variable in the parameter list. For this example, we're using two ints to multiply against one another and assign a value to a. There isn't anything limiting the types we're allowed to use in the parameter list.

```
int a = 0;
float b = 0;

void SetA (int i, float j) {
    a = i;
    b = j;

                                    }
```

The function with multiple parameters can do more. Value parameters are helpful ways to get data into a function to accomplish some sort of simple task.

Code Blocks

Blocks are collections of statements grouped together when they're evaluated. This evaluation of code is usually controlled by an opening and closing curly brace. Java programmers often use tabs to indent different blocks of code. This makes the different blocks more readable. Some languages like python rely on the tabs to indicate separate blocks of code. Getting used to how tabs are used is essential for writing readable code. Next, we'll use a logic control statement followed by a block of code. The block for this if statement is indented to the right to indicate it's confined within the logic control statement.

```
        if (true)
```

```
                                    {
```

```
    //code goes here
                        //im also a part of the code
```

```
                                    }
```

If we take a look at a more complex section of code we can see why it's important to differentiate code by using tabs. When looking at code it might not be apparent, but programmers are very fastidious about text layout. Each programmer has a preferred style, but it's safe to say that most, if not all, programmers hate seeing badly formatted code.

```
int i = 7;
int j = 13;
    if (i < 13){
  int j = 7;
```

```
                                    }
```

In the above example we can clearly see int j being set twice. What isn't obvious is why it's declared twice. All of the lines start at the same position and look crowded. Programming tools offer many text layout options, so white space is used to separate different blocks of code.

```
int i = 7;

int j = 13;

if (i < 13)

                              {

        int j = 7;

                              }
```

By adding in some white space we can make the if statement more visible. Adding white space around our blocks of code is important to maintain readability. There are also blocks of code that can live within another block of code. In general it's all related to the same block. Blocks inside of a block are called nested code blocks. White space is used to separate the nested code from the block it's found in.

```
if(true)

                              {

 //all of the code
        //im also a part of the code
        if(true)

                              {

                //im another block of code
```

```
        //living inside of a bigger
        //block of code
```

```
                                }
```

```
        //yep more code here too
```

```
                                }
```

The block of code following the second if statement is considered to be nested inside of the first block of code. This is because of the placement of the first opening curly brace { and the related closing curly brace }. You form hierarchies by placing the different curly braces within one another.

```
if(true)
```

```
                                {
```

```
//all of the code
//im also a part of the code
if(true)
```

```
                                {
```

```
//im another block of code
//living inside of another
//block of code
```

```
                                }
```

```
//yep more code here too
```

```
                                }
```

Poorly indented curly braces make for confusing code; the above code is still valid, but it's less readable compared to the previous version.

Most code-editing tools will automatically add in the proper number of indents for you. It's something you'll have to get used to when writing your own code. We'll go into further detail as to what this all means later. Just gather from this discussion that a bunch of statements appearing between a pair of curly braces is considered a block of code.

Much of programming involves proper use of style and a great deal of white space to accomplish that style. So far we've just been covering vocabulary, but without that vocabulary we can't have a proper conversation about programming. This too includes talking to other programmer friends, who will have a hard time understanding you if you can't speak in their language.

Logic and Operators

Logic allows you to control what part of a function is evaluated based on changes to variables. Using logic, you'll be able to change which statements in your code will run. Simply put, everything you write must cover each situation you plan to cover. Logic is controlled through a few simple systems, primarily the if keyword and variations of if.

Booleans: In Java booleans, or bools for short, are either true or false. It's easiest to think of these as switches either in on or in off position. To declare a var as a bool, you use something like the following.

```
public class Example
{
    public bool SomeBool;
}
```

Equality Operators:

Equality operators create boolean conditions. There are many ways to set a boolean variable. For instance, comparisons between values are a useful means to set variables. The most basic method to determine equality is using the following operator: ==. There's a difference between use of a single and a double equals to symbol. = is used to assign a value whereas == is used to compare values.

When you need to compare two values you can use the following concept. You'll need to remember that these operators are called equality operators, if you need to talk to a programmer.

The syntax here may look a bit confusing at first, but there are ways around that.

```
void Func ()

                          {

      SomeBool = (1 == 1);

                          }
```

There are other operators to be aware of. You will be introduced to the other logical operators later in the chapter. In this case, we are asking if two number values are the same.

To make this a more clear, we can break out the code into more lines. Now, we're looking at a versus b. Clearly, they don't look the same; they are different letters after all. However, they do contain the same integer value, and that's what's really being compared here.

```
void Func ()

                              {

        int a = 1;
    int b = 1;
        SomeBool = (a == b);

                              }
```

Evaluations have a left and a right side. The single equal to operator (=) separates the different sides. The left side of the = is calculated and looks to the value to the right to get its assignment. Because 1 == 1, that is to say, 1 is equivalent to 1, the final result of the statement is that SomeBool is true.

Relational Operators : Bool values can also be set by comparing values. The operators used to compare two different values are called relational operators. We use ==, the is equal symbol, to check if values are the same; we can also use !=, or not equal, to check of two variables are different.

This works similarly to the ! in the previous section. Programmers more often check if one value is greater or lesser than another value. They do this by using >, or greater than, and <, or less than. We can also use > =, greater or equal to, and < =, less than or equal to. Let's see a few examples of how this is used.

```java
public class Test {

  public static void main(String args[]) {

    int a = 10;

    int b = 20;

    System.out.println("a == b = " + (a == b) );

    System.out.println("a != b = " + (a != b) );

    System.out.println("a > b = " + (a > b) );

    System.out.println("a < b = " + (a < b) );

    System.out.println("b >= a = " + (b >= a) );

    System.out.println("b <= a = " + (b <= a) );  }


  }
```

```
Output:
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false
```

CHAPTER 7 | CONTROLING EXECUTION, LOOPS & ARRAYS

Controlling Execution

The control flow statements in Java allow you to run or skip blocks of code when special conditions are met.

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code.

Java uses all of C's execution control statements, so if you've programmed with C or C++, then most of what you see will be familiar. Most procedural programming languages have some kind of control statements, and there is often overlap among languages.

In Java, the keywords include **if-else**, **while**, **do-while**, **for**, **return**, **break**, and a selection statement called **switch**.

Java does not, however, support the much-maligned goto (which can still be the most expedient way to solve certain types of problems). You can still do a goto-like jump, but it is much more constrained than a typical goto.

True and false:
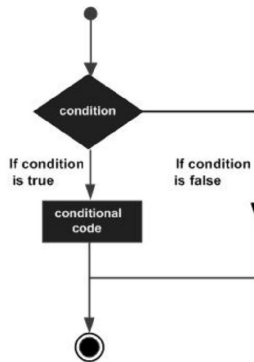
All conditional statements use the truth or falsehood of a conditional expression to determine the execution path.

An example of a conditional expression is a == b.

This uses the conditional operator == to see if the value of a is equivalent to the value of b.

The expression returns true or false.

Any of the relational operators you've seen in the previous chapter can be used to produce a conditional statement.

Note that Java doesn't allow you to use a number as a Boolean, even though it's allowed in C and C++ (where truth is nonzero and falsehood is zero).

If you want to use a non-Boolean in a Boolean test, such as if(a), you must first convert it to a boolean value by using a conditional expression, such as if(a != 0).

if-else statement:
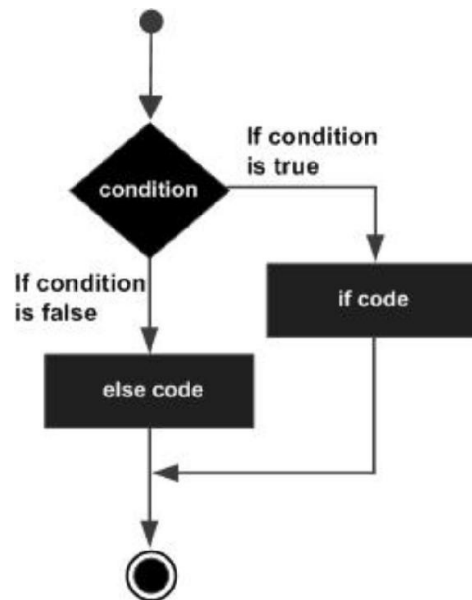
The if-else statement is the most basic way to control program flow. The else is optional, so you can use if in two forms:

```
if(Boolean-expression)     {
     //statement

                                }
```

or

```
if(Boolean-expression)    {
    //statement

                               }

else {
    //state ment

                               }
```

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If not, the first set of code after the end of the if statement (after the closing curly brace) will be executed.

The Boolean-expression must produce a boolean result. The statement is either a simple statement terminated by a semicolon, or a compound statement, which is a group of simple statements enclosed in braces.

Whenever the word "statement" is used, it always implies that the statement can be simple or compound.

Here is an example for the if-else statement:

```
public class Test {

   public static void main(String args[]) {
      int x = 10;

      if( x < 20 ) {
         System.out.print("This is if statement");
      }
   }
}
```

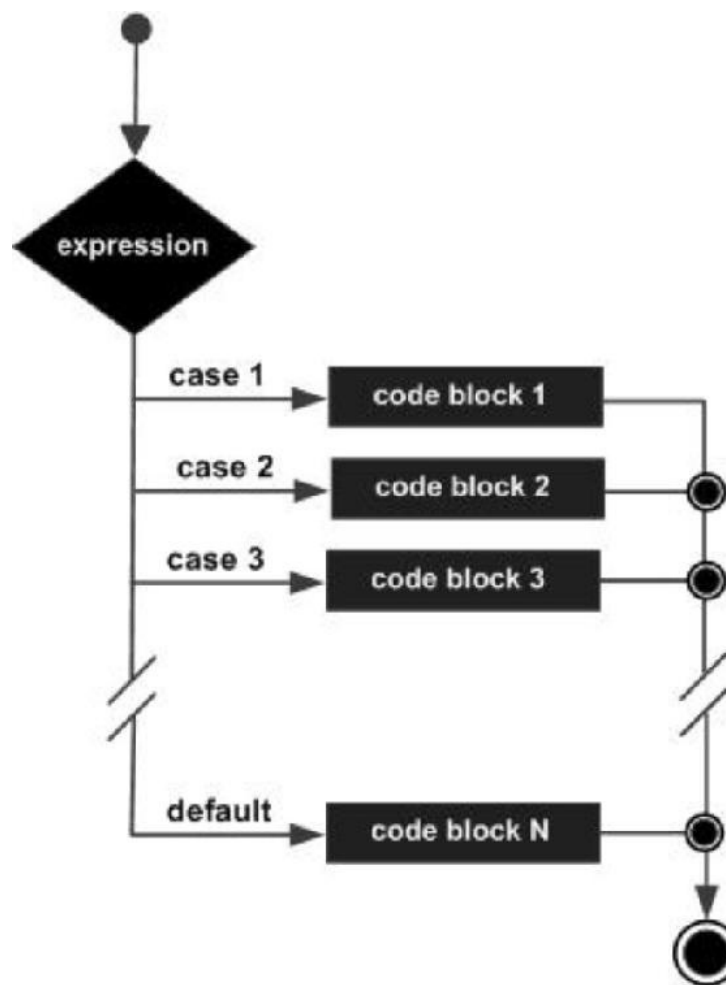This will produce the following result:

Output:

```
This is if statement.
```

Switch case Statement:

The switch is sometimes called a selection statement. The switch statement selects from among pieces of code based on the value of an integral expression. A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Its general form is:
switch(integral-selector) {
```
case integral-value1 : statement; break;
case integral-value2 : statement; break;
case integral-value3 : statement; break;
```

```
case integral-value4 : statement; break;
case integral-value5 : statement; break;
```

```
// ...
```

```
default: statement;
```

```
}
```

Integral-selector is an expression that produces an integral value. The switch compares the result of integral-selector to each integral-value. If it finds a match, the corresponding statement (a single statement or multiple statements; braces are not required) executes. If no match occurs, the default statement executes.

You will notice in the preceding definition that each case ends with a **break**, which causes execution to jump to the end of the switch body. This is the conventional way to build a switch statement, but the break is optional. If it is missing, the code for the following case statements executes until a break is encountered. Although you don't usually want this kind of behavior, it can be useful to an experienced programmer.

Note that the last statement, following the default, doesn't have a break because the execution just falls through to where the break would have taken it anyway. You could put a break at the end of the default statement with no harm if you considered it important for style's sake.

The switch statement is a clean way to implement multiway selection (i.e., selecting from among a number of different execution paths), but it requires a selector that evaluates to an integral value, such as int or char. If you want to use, for example, a string or a floating point number as a selector, it won't work in a switch statement. For non-integral types, you must use a series of if statements. At the end of the next chapter, you'll see that Java SE5's new enum feature helps ease this restriction, as enums are designed to work nicely with switch.

Example:

```java
public class Test {

  public static void main(String args[]) {
    // char grade = args[0].charAt(0);
    char grade = 'C';

    switch(grade) {
      case 'A' :
            System.out.println("Excellent!");
        break;
      case 'B' :
      case 'C' :
        System.out.println("Well done");
        break;
      case 'D' :
        System.out.println("You passed");
      case 'F' :
        System.out.println("Better try again");
        break;
      default :
        System.out.println("Invalid grade");

    }

    System.out.println("Your grade is " + grade);

  }

}
```

This will produce the following result:

Output:

Well done

Your grade is C

Loop

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths.

A **loop** statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages −
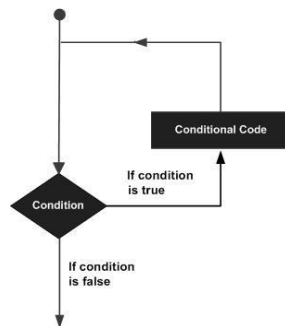
Java programming language provides the following types of loop to handle looping requirements.

While loop: If we want to run a specific statement more than once in a loop, we need another method. To do this, java comes with another couple of keywords, while and for. The while statement is somewhat easier to use. It needs only one bool argument to determine if it should continue to execute. This statement is somewhat like the if statement, only that it returns to the top of the while statement when it's done with its evaluations.

The syntax of a while loop is :

```
while(Boolean_expression) {
   // Statements

                                  }
```

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

When executing, if the *boolean_expression* result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram:
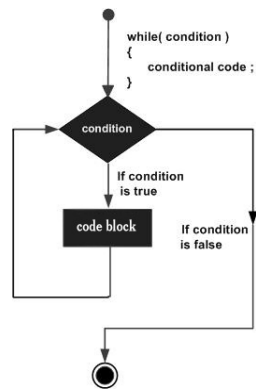
Here, key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```java
public class Test {

  public static void main(String args[]) {

    int x = 10; while( x < 20 ) {

      System.out.print("value of x : " + x ); x++; System.out.print("\n"); }

  }

}
```

This will produce the following result: Output:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

For Loop: To gain a bit more control over the execution of a loop, we have another option. The for loop requires three different statements to operate. The first statement is called an initialization, the second is a condition, and the third is an operation.

Syntax:

```
for(initialization; Boolean_expression; update) {
  // Statements }
```

Here is the flow of control in a **for** loop −

- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).

- Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop will not be executed and control jumps to the next statement past the for loop.

- After the **body** of the for loop gets executed, the control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank with a semicolon at the end.

- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats (body of loop, then update step, then Boolean expression). After the Boolean expression is false, the for loop terminates.
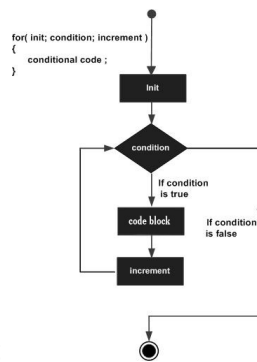
Flow Diagram:

Example: Following is an example code of the for loop in Java.

```java
public class Test {

  public static void main(String args[]) {

    for(int x = 10; x < 20; x = x + 1) {

      System.out.print("value of x : " + x ); System.out.print("\n"); }

  }

}
```

This will produce the following result: Output:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

<u>Loop Control Statements: Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.</u>

Java supports the following control statements.

- Break Statement
- Continue Statement

The **break** statement in Java programming language has the following two usages −

- When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

Syntax The syntax of a break is a single statement inside any loop: break;

Flow Diagram:
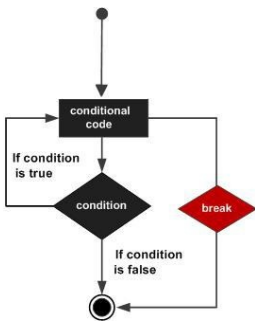
Example:

```
public class Test {

   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};

      for(int x : numbers ) {
         if( x == 30 ) {
            break;
         }
         System.out.print( x );
```

```
        System.out.print("\n"); }

  }
                                    }
```

This will produce the following result: Output

```
10
```

20

The **continue** keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- In a for loop, the continue keyword causes control to immediately jump to the update statement.

- In a while loop or do/while loop, control immediately jumps to the Boolean expression.

Syntax: The syntax of a continue is a single statement inside any loop −

```
continue;
```

Flow Diagram:
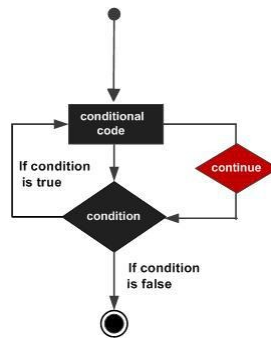
Example:

```java
public class Test {

  public static void main(String args[]) {
    int [] numbers = {10, 20, 30, 40, 50};

    for(int x : numbers ) {
      if( x == 30 ) {
        continue;
      }
      System.out.print( x );
      System.out.print("\n");
```

```
        }

    }

}
```

Output:

```
10
20
40
50
```

Arrays

Arrays are nicely organized lists of data. Think of a numbered list that starts at zero and extends one line every time you add something to the list. Arrays are useful for any number of situations because they're treated as a single hunk of data. For instance, if you wanted to store a bunch of high scores, you'd want to do that with an array. Initially, you might want to have a list of 10 items. You could in theory use the following code to store each score.

```
int score1;
int score2;
int score3;
int score4;
int score5;
int score6;
int score7;
int score8;
int score9;
int score10;
```

To make matters worse, if you needed to process each value, you'd need to create a set of code that deals with each variable by name. To check if score2 is higher than score1, you'd need to write a function specifically to check those two variables before switching them. Thank goodness for arrays.

There are two types of array:

- Single Dimensional Array

- Multidimensional Array

Single Dimensional Array in java: Syntax to Declare an Array in java: dataType[] arr; (or)

```
dataType []arr; (or)
dataType arr[];
```

Instantiation of an Array in java:
arrayRefVar=**new** datatype[size];

Example of single dimensional java array: Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```java
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10; //initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//printing array
for(int i=0;i<a.length;i++)//length is the property                    of
array
System.out.println(a[i]);

                                        }

                                        }
```

Output: 10
    20
      70
      40
      50

Declaration, Instantiation and Initialization of Java Array: We can declare, instantiate and initialize the java array together by:
int   a[]={33,3,4,5};//declaration,   instantiation and   initialization

Let's see the simple example to print this array.

```java
class Testarray1{
public static void main(String args[]){

int                    a[]={33,3,4,5};//declaration,                    instantiation
and                          initialization

//printing array
for(int i=0;i<a.length;i++)//length is the property                    of
System.out.println(a[i]);
```

```
}}
```

Output:

33

```
     3
     4
     5
```

Passing Array to method in java:
We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

```java
class Testarray2{
static void min(int arr[]){
int min=arr[0];

for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];

System.out.println(min);
```
```java
}
```
```java
 public static void main(String args[]){
  int a[]={33,3,4,5};
min(a);   //passing array to method
```
```java
}
```
```java
}
```

Output: 3

Multidimensional array in java:

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java:

    dataType[][] arrayRefVar; (or)
    dataType [][]arrayRefVar; (or)
    dataType arrayRefVar[][]; (or)
    dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in java:
int[][] arr=new int[3][3]; //3 row and 3 column

Example to initialize Multidimensional Array in java
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;

Example of Multidimensional java array:

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
class Testarray3{
public static void main(String args[]){

//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};

//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");

}

System.out.println();

}

}

}
```

Output: 1 2 3
       2 4 5
       4 4 5

CHAPTER 8 | OBJECT ORIENTED PROGRAMMING

Classes

The most important part of any Java program is the class. Classes here are not the sort that students go to; rather, they are classes in terms of classification. Classes in Java appear in several different forms. Some are called partial classes, and some classes can even appear inside of another class; these classes are called nested classes. A class is a collection of instructions and data. The instructions can be referred to as functions or methods. The data stored in a class is called a field or property. All of the naming aside, what's important is what classes do and how they're used.

Classes are flexible and easily changed, but this flexibility is not without some complexity. As we've seen before, the class declaration can contain a lot more than the following example. However, much of the complexity comes later in this book, when needed.

When you approach a game or any sort of programming task it's important to first collect data, and then figure out how to use the data. In most programming paradigms it's best to start off with variables at the beginning of your new class. For instance, in this pseudo code we might want to know what day it is before going out:
Time day = today;

```
if (day == Friday)

                          {

PartyTonight = true;

                          }
```

We start off with getting a day, and then deciding what to do tonight based on the day. We should refrain from partying and then checking what the day is unless we want to show up for work late.

A class is a blueprint from which individual objects are created.
Following is a sample of a class.
Example:
public class Dog {
```
  String breed;
```

```
   int age;
   String color;

   void barking() {

                                         }


   void hungry() {

                                         }


   void sleeping() {

                                         }

                                         }
```

A class can contain any of the following variable types.

Local variables: Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

Instance variables: Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

Class variables: Class variables are variables declared within a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kinds of methods.

In the above example, barking(), hungry() and sleeping() are methods.

Following are some of the important topics that need to be discussed when looking into classes of the Java Language.

Constructors: When discussing about classes, one of the most important sub topic would be constructors.

the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class.

Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Following is an example of a constructor: Example
public class Puppy {

```
  public Puppy() {

                                                      }


  public Puppy(String name) {
     // This constructor has one parameter, name.

                                                      }
                                                      }
```

Java also supports [Singleton Classes](#) where you would be able to create only one instance of a class.

Introduction to Objects

Object oriented programming (OOP) was invented to associate data and logic into nicely packaged bundles of code. When you write a class in Java you're creating blueprints and instructions for a new object. Any number of objects can then be constructed based on your blueprints. Programmers use the term instantiate when talking about creating an object from the class blueprint. Each instantiated object is an instance of the class from which it was constructed. The class is not itself one of the instances created by it. The newly created object contains all of the functions and data written into the class.

All programming languages provide abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. By "kind" I mean, "What is it that you are abstracting?" Assembly language is a small abstraction of the underlying machine. Many so-called "imperative" languages that followed (such as FORTRAN, BASIC, and C) were abstractions of assembly language. These languages are big improvements over assembly language, but their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve.

The programmer must establish the association between the machine model (in the "solution space," which is the place where you're implementing that solution, such as a computer) and the model of the problem that is actually being solved (in the "problem space," which is the place where the problem exists, such as a business). The effort required to perform this mapping, and the fact that it is extrinsic to the programming language, produces programs that are difficult to write and expensive to maintain, and as a side effect created the entire "programming methods" industry.

The alternative to modeling the machine is to model the problem you're trying to solve. Early languages such as LISP and APL chose particular views of the world ("All problems are ultimately lists" or "All problems are algorithmic," respectively). Prolog casts all problems into chains of decisions. Languages have been created for constraint-based programming and for programming exclusively by manipulating graphical symbols. (The latter proved to be too restrictive.)

Each of these approaches may be a good solution to the particular class of problem they're designed to solve, but when you step outside of that domain they become awkward.

The object-oriented approach goes a step further by providing tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. We refer to the elements in the problem space and their representations in the solution space as "objects." (You will also need other objects that don't have problem-space analogs.)

The idea is that the program is allowed to adapt itself to the lingo of the problem by adding new types of objects, so when you read the code describing the solution, you're reading words that also express the problem. This is a more flexible and powerful language abstraction than what we've had before.1 Thus, OOP allows you to describe the problem in terms of the problem, rather than in terms of the computer where the solution will run. There's still a connection back to the computer: Each object looks quite a bit like a little computer—it has a state, and it has operations that you can ask it to perform. However, this doesn't seem like such a bad analogy to objects in the real world—they all have characteristics and behaviors.

To simply put, OOP is the process of building objects, instancing them, and then using them. Using objects involves reading data, using logic on the data, and then carrying out tasks based on the logic.

Creating an Object: As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class: **Declaration**: A variable declaration with a variable name with an object type.

Instantiation: The **'new'** keyword is used to create the object.

Initialization: The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Following is an example of creating an object:
public class Puppy {

```
  public Puppy(String name) {

    // This constructor has one parameter, name.

    System.out.println("Passed Name is :" + name );

                                }

  public static void main(String []args) {

    // Following statement would create an object myPuppy

    Puppy myPuppy = new Puppy( "tommy" );

                             } }
```

Output:
Passed Name is :tommy

Accessing Instance Variables and Methods: Instance variables and methods are accessed via created objects. To access an instance variable, following is the fully qualified path:

/* First create an object */

ObjectReference = new Constructor();

/* Now call a variable as follows */

ObjectReference.variableName;

/* Now you can call a class method as follows */

ObjectReference.MethodName();

Example: This example explains how to access instance variables and methods of a class.

```java
public class Puppy {

  int puppyAge;

  public Puppy(String name) {

    // This constructor has one parameter, name.

    System.out.println("Name chosen is :" + name );

                                }

  public void setAge( int age ) {

    puppyAge = age;

                                }

  public int getAge( ) {

    System.out.println("Puppy's age is :" + puppyAge );
```

```
      return puppyAge;

                                    }


  public static void main(String []args) {

      /* Object creation */

      Puppy myPuppy = new Puppy( "tommy" );


      /* Call class method to set puppy's age */

      myPuppy.setAge( 2 );


      /* Call another class method to get puppy's age */

      myPuppy.getAge( );


      /* You can access instance variable as follows as well */

      System.out.println("Variable Value :" + myPuppy.puppyAge );

                                    }

                                    }
```

If we compile and run the above program, then it will produce the following result: **Output**:

Name chosen is :tommy

Puppy's age is :2

Variable Value :2

Characteristics of object-oriented programming: These characteristics represent a pure approach to object-oriented programming: **1. Everything is an object.** Think of an object as a fancy variable; it stores data, but you can "make requests" to that object, asking it to perform operations on itself. In theory, you can take any conceptual component in the problem you're trying to solve (dogs, buildings, services, etc.) and represent it as an object in your program.

2. A program is a bunch of objects telling each other what to do by sending messages. To make a request of an object, you "send a message" to that object. More concretely, you can think of a message as a request to call a method that belongs to a particular object.

3. Each object has its own memory made up of other objects. Put another way, you create a new kind of object by making a package containing existing objects. Thus, you can build complexity into a program while hiding it behind the simplicity of objects.

4. Every object has a type. Using the parlance, each object is an instance of a class, in which "class" is synonymous with "type." The most important distinguishing characteristic of a class is "What messages can you send to it?"

5. All objects of a particular type can receive the same messages. This is actually a loaded statement, as you will see later. Because an object of type "circle" is also an object of type "shape," a circle is guaranteed to accept shape messages. This means you can write code that talks to shapes and automatically handle anything that fits the description of a shape. This substitutability is one of the powerful concepts in OOP.

Even more succinct description of an object is: An object has state, behavior and identity.

This means that an object can have internal data (which gives it state), methods (to produce behavior), and each object can be uniquely distinguished from every other object—to put this in a concrete sense, each object has a unique address in memory.

An object has an interface: So, although what we really do in object-oriented programming is create new data types, virtually all object-oriented programming languages use the "**class**" keyword. When you see the word "**type**" think "**class**" and vice versa.

Since a class describes a set of objects that have identical characteristics (data elements) and behaviors (functionality), a class is really a data type because a floating point number, for example, also has a set of characteristics and behaviors. The difference is that a programmer defines a class to fit a problem rather than being forced to use an existing data type that was designed to represent a unit of storage in a machine. You extend the programming language by adding new data types specific to your needs. The programming system welcomes the new classes and gives them all the care and type checking that it gives to built-in types.

The object-oriented approach is not limited to building simulations. Whether or not you agree that any program is a simulation of the system you're designing, the use of OOP techniques can easily reduce a large set of problems to a simple solution.

Once a class is established, you can make as many objects of that class as you like, and then manipulate those objects as if they are the elements that exist in the problem you are trying to solve. Indeed, one of the challenges of object-oriented programming is to create a one-to one mapping between the elements in the problem space and objects in the solution space.

But how do you get an object to do useful work for you? There needs to be a way to make a request of the object so that it will do something, such as complete a transaction, draw something on the screen, or turn on a switch. And each object can satisfy only certain requests. The requests you can make of an object are defined by its interface, and the type is what determines the interface. A simple example might be a representation of a light bulb:
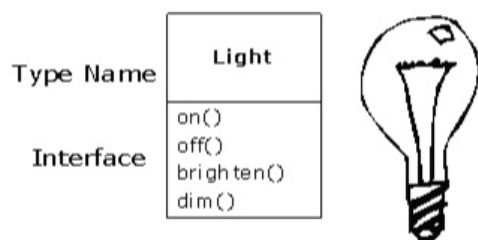

```
Light lt = new Light();
```

lt.on();

The interface determines the requests that you can make for a particular object. However, there must be code somewhere to satisfy that request. This, along with the hidden data, comprises the implementation. From a procedural programming standpoint, it's not that complicated.

A type has a method associated with each possible request, and when you make a particular request to an object, that method is called. This process is usually summarized by saying that you "send a message" (make a request) to an object, and the object figures out what to do with that message (it executes code).

Here, the name of the type/class is Light, the name of this particular Light object is lt, and the requests that you can make of a Light object are to turn it on, turn it off, make it brighter, or make it dimmer. You create a Light object by defining a "reference" (lt) for that object and calling new to request a new object of that type. To send a message to the object, you state the name of the object and connect it to the message request with a period (dot). From the standpoint of the user of a predefined class, that's pretty much all there is to programming with objects.

The preceding diagram follows the format of the Unified Modeling Language (UML). Each class is represented by a box, with the type name in the top portion of the box, any data members that you care to describe in the middle portion of the box, and the methods (the functions that belong to this object, which receive any messages you send to that object) in the bottom portion of the box. Often, only the name of the class and the public methods are shown in UML design diagrams, so the middle portion is not shown, as in this case. If you're interested only in the class name, then the bottom portion doesn't need to be shown, either.

Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.

- All of the methods in an interface are abstract.

- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.

- An interface is not extended by a class; it is implemented by a class.

- An interface can extend multiple interfaces.

Declaring Interfaces: The interface keyword is used to declare an interface. Here is a simple example to declare an interface −

Example Following is an example of an interface:

```
/* File name : NameOfInterface.java */
import java.lang.*;
// Any number of import statements


public interface NameOfInterface {
   // Any number of final, static fields
   // Any number of abstract method declarations\

                              }
```

Interfaces have the following properties:

- An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example:

```
/* File name : Animal.java */
interface Animal {
   public void eat();
   public void travel();
```

```
                                 }
```

Implementing Interfaces: When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example:

```
/* File name : MammalInt.java */
public class MammalInt implements Animal {

   public void eat() {
      System.out.println("Mammal eats"); }

   public void travel() {
      System.out.println("Mammal travels"); }

   public int noOfLegs() {
```

```
      return 0; }

   public static void main(String args[]) {

   MammalInt m = new MammalInt(); m.eat(); m.travel(); }

                                    }
```

This will produce the following result: Output:

Mammal eats

Mammal travels

When overriding methods defined in interfaces, there are several rules to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.

- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.

- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules:

- A class can implement more than one interface at a time.

- A class can extend only one class, but implement many interfaces.

- An interface can extend another interface, in a similar way as a class can extend another class.

An object provides services: While you're trying to develop or understand a program design, one of the best ways to think about objects is as "**service providers**." Your program itself will provide services to the user, and it will accomplish this by using the services offered by other objects. Your goal is to produce (or even better, locate in existing code libraries) a set of objects that provide the ideal services to solve your problem.

A way to start doing this is to ask, "If I could magically pull them out of a hat, what objects would solve my problem right away?" For example, suppose you are creating a bookkeeping program. You might imagine some objects that contain pre-defined bookkeeping input screens, another set of objects that perform bookkeeping calculations, and an object that handles printing of checks and invoices on all different kinds of printers. Maybe some of these objects already exist, and for the ones that don't, what would they look like? What services would those objects provide, and what objects would they need to fulfill their obligations? If you keep doing this, you will eventually reach a point where you can say either, "That object seems simple enough to sit down and write" or "I'm sure that object must exist already." This is a reasonable way to decompose a problem into a set of objects.

Thinking of an object as a service provider has an additional benefit: It helps to improve the cohesiveness of the object. High cohesion is a fundamental quality of software design: It means that the various aspects of a software component (such as an object, although this could also apply to a method or a library of objects) "fit together" well. One problem people have when designing objects is cramming too much functionality into one object. For example, in your check printing module, you may decide you need an object that knows all about formatting and printing. You'll probably discover that this is too much for one object, and that what you need is three or more objects. One object might be a catalog of all the possible check layouts, which can be queried for information about how to print a check. One object or set of objects can be a generic printing interface that knows all about different kinds of printers (but nothing about bookkeeping—this one is a candidate for buying rather than writing yourself). And a third object could use the services of the other two to accomplish the task. Thus, each object has a cohesive set of services it offers. In a good object-oriented design, each object does one thing well, but doesn't try to do too much. This not only allows the discovery of objects that might be purchased (the printer interface object), but it also produces new objects that might be reused somewhere else (the catalog of check layouts).

Treating objects as service providers is a great simplifying tool. This is useful not only during the design process, but also when someone else is trying to understand your code or reuse an object. If they can see the value of the object based on what service it provides, it makes it much easier to fit it into the design.

The hidden implementation:

It is helpful to break up the playing field into class creators (those who create new data types) and client programmers4 (the class consumers who use the data types in their applications). The goal of the client programmer is to collect a toolbox full of classes to use for rapid application development. The goal of the class creator is to build a class that exposes only what's necessary to the client programmer and keeps everything else hidden. Why? Because if it's hidden, the client programmer can't access it, which means that the class creator can change the hidden portion at will without worrying about the impact on anyone else. The hidden portion usually represents the tender insides of an object that could easily be corrupted by a careless or uninformed client programmer, so hiding the implementation reduces program bugs.

In any relationship it's important to have boundaries that are respected by all parties involved. When you create a library, you establish a relationship with the client programmer, who is also a programmer, but one who is putting together an application by using your library, possibly to build a bigger library. If all the members of a class are available to everyone, then the client programmer can do anything with that class and there's no way to enforce rules. Even though you might really prefer that the client programmer not directly

manipulate some of the members of your class, without access control there's no way to prevent it. Everything's naked to the world.

So the first reason for access control is to keep client programmers' hands off portions they shouldn't touch—parts that are necessary for the internal operation of the data type but not part of the interface that users need in order to solve their particular problems. This is actually a service to client programmers because they can easily see what's important to them and what they can ignore.

The second reason for access control is to allow the library designer to change the internal workings of the class without worrying about how it will affect the client programmer. For example, you might implement a particular class in a simple fashion to ease development, and then later discover that you need to rewrite it in order to make it run faster. If the interface and implementation are clearly separated and protected, you can accomplish this easily.

Java uses three explicit keywords to set the boundaries in a class: public, private, and protected. These access specifiers determine who can use the definitions that follow. public means the following element is available to everyone. The private

keyword, on the other hand, means that no one can access that element except you, the creator of the type, inside methods of that type. private is a brick wall between you and the client programmer. Someone who tries to access a private member will get a compile-time error. The protected keyword acts like private, with the exception that an inheriting class has access to protected members, but not private members. Inheritance will be introduced shortly.

Java also has a "**default**" access, which comes into play if you don't use one of the aforementioned specifiers. This is usually called package access because classes can access the members of other classes in the same package (library component), but outside of the package those same members appear to be private.

Encapsulation: Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

To achieve encapsulation in Java:

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

Example: Following is an example that demonstrates how to achieve Encapsulation in Java:

```java
public class EncapTest {
  private String name; private String idNum; private int age;
  public int getAge() {
    return age; }


  public String getName() {
    return name; }


  public String getIdNum() {
    return idNum; }


  public void setAge( int newAge) {
    age = newAge; }
```

```java
    public void setName(String newName) {

      name = newName; }


    public void setIdNum( String newId) {

      idNum = newId; }
```

                                }

The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be accessed using the following program:

```java
public class RunEncap {

  public static void main(String args[]) {

        EncapTest encap = new EncapTest(); encap.setName("James");
encap.setAge(20); encap.setIdNum("12343ms");

        System.out.print("Name : " + encap.getName() + " Age : " +
encap.getAge()); }
```

                                }

 This will produce the following result: Output: Name: James Age : 20

Benefits of Encapsulation:

- The fields of a class can be made read-only or write-only.

- A class can have total control over what is stored in its fields.

- The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

Reusing the implementation:

Once a class has been created and tested, it should (ideally) represent a useful unit of code. It turns out that this reusability is not nearly so easy to achieve as many would hope; it takes experience and insight to produce a reusable object design. But once you have such a design, it begs to be reused. Code reuse is one of the greatest advantages that object-oriented programming languages provide.

The simplest way to reuse a class is to just use an object of that class directly, but you can also place an object of that class inside a new class. We call this "creating a member object." Your new class can be made up of any number and type of other objects, in any combination that you need to achieve the functionality desired in your new class. Because you are composing a new class from existing classes, this concept is called composition (if the composition happens dynamically, it's usually called aggregation). Composition is often referred to as a "has-a" relationship, as in "A car has an engine."

(This UML diagram indicates composition with the filled diamond, which states there is one car. I will typically use a simpler form: just a line, without the diamond, to indicate an association.5)

Composition comes with a great deal of flexibility. The member objects of your new class are typically private, making them inaccessible to the client programmers who are using the class. This allows you to change those members without disturbing existing client code. You can also change the member objects at run time, to dynamically change the behavior of your program. Inheritance, which is described next, does not have this flexibility since the compiler must place compile-time restrictions on classes created with inheritance.

Because inheritance is so important in object-oriented programming, it is often highly emphasized, and the new programmer can get the idea that inheritance should be used everywhere. This can result in awkward and overly complicated designs. Instead, you should first look to composition when creating new classes, since it is simpler and more flexible. If you take this approach, your designs will be cleaner. Once you've had some experience, it will be reasonably obvious when you need inheritance.

Inheritance: By itself, the idea of an object is a convenient tool. It allows you to package data and functionality together by concept, so you can represent an appropriate problem-space idea rather than being forced to use the idioms of the underlying machine. These concepts are expressed as fundamental units in the programming language by using the class keyword.

It seems a pity, however, to go to all the trouble to create a class and then be forced to create a brand new one that might have similar functionality. It's nicer if we can take the existing class, clone it, and then make additions and modifications to the clone. This is effectively what you get with inheritance, with the exception that if the original class (called the base class or superclass or parent class) is changed, the modified "clone" (called the derived class or inherited class or subclass or child class) also reflects those changes.
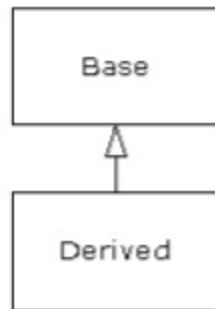
(The arrow in this UML diagram points from the derived class to the base class. As you will see, there is commonly more than one derived class.) A type does more than describe the constraints on a set of objects; it also has a relationship with other types. Two types can have characteristics and behaviors in common, but one type may contain more characteristics than another and may also handle more messages (or handle them differently). Inheritance expresses this similarity between types by using the concept of base types and derived types. A base type contains all of the characteristics and behaviors that are shared among the types derived from it. You create a base type to represent the core of your ideas about some objects in your system. From the base type, you derive other types to express the different ways that this core can be realized.

For example, a trash-recycling machine sorts pieces of trash. The base type is "trash", and each piece of trash has a weight, a value, and so on, and can be shredded, melted, or decomposed. From this, more specific types of trash are derived that may have additional characteristics (a bottle has a color) or behaviors (an aluminum can may be crushed, a steel can is magnetic). In

addition, some behaviors may be different (the value of paper depends on its type and condition). Using inheritance, you can build a type hierarchy that expresses the problem you're trying to solve in terms of its types.

A second example is the classic "shape" example, perhaps used in a computer-aided design system or game simulation. The base type is "shape," and each shape has a size, a color, a position, and so on. Each shape can be drawn, erased, moved, colored, *etc*. From this, specific types of shapes are derived (inherited)— circle, square, triangle, and so on—each of which may have additional characteristics and behaviors. Certain shapes can be flipped, for example. Some behaviors may be different, such as when you want to calculate the area of a shape. The type hierarchy embodies both the similarities and differences between the shapes.
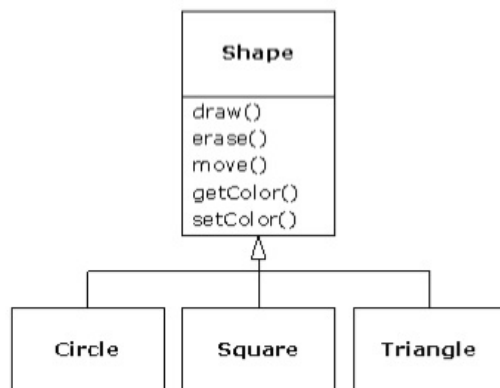
Casting the solution in the same terms as the problem is very useful because you don't need a lot of intermediate models to get from a description of the problem to a description of the solution. With objects, the type hierarchy is the primary model, so you go directly from the description of the system in the real world to the description of the system in code. Indeed, one of the difficulties people have with object-oriented design is that it's too simple to get from the beginning to the end. A mind trained to look for complex solutions can initially be stumped by this simplicity.

When you inherit from an existing type, you create a new type. This new type contains not only all the members of the existing type (although the private ones are hidden away and inaccessible), but more importantly it duplicates the interface of the base class. That is, all the messages you can send to objects of the base class you can also send to objects of the derived class. Since we know the type of a class by the messages we can send to it, this means that the derived

class is the same type as the base class. In the previous example, "A circle is a shape." This type equivalence via inheritance is one of the fundamental gateways in understanding the meaning of object-oriented programming.

Since both the base class and derived class have the same fundamental interface, there must be some implementation to go along with that interface. That is, there must be some code to execute when an object receives a particular message. If you simply inherit a class and don't do anything else, the methods from the base-class interface come right along into the derived class. That means objects of the derived class have not only the same type, they also have the same behavior, which isn't particularly interesting.

You have two ways to differentiate your new derived class from the original base class. The first is quite straightforward: You simply add brand new methods to the derived class. These new methods are not part of the base-class interface. This means that the base class simply didn't do as much as you wanted it to, so you added more methods. This simple and primitive use for inheritance is, at times, the perfect solution to your problem. However, you should look closely for the possibility that your base class might also need these additional methods. This process of discovery and iteration of your design happens regularly in object-oriented programming.

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword: extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

class Super {

.....

.....

}

```
                                                        }

class Sub extends Super {
    .....
    .....

                                                }
```

Example:

```java
class Calculation {
  int z;
  public void addition(int x, int y) {
    z = x + y;
    System.out.println("The sum of the given numbers:"+z);
  }
  public void Subtraction(int x, int y) {
    z = x - y;
    System.out.println("The difference between the given numbers:"+z);
  }
}
```

```java
}

public class My_Calculation extends Calculation {
  public void multiplication(int x, int y) {
    z = x * y;
    System.out.println("The product of the given numbers:"+z);
  }
  public static void main(String args[]) {
    int a = 20, b = 10;
    My_Calculation demo = new My_Calculation();
    demo.addition(a, b);
    demo.Subtraction(a, b);
```

```
      demo.multiplication(a, b);

   }
```

```
                              }
```

Compile and execute the above code as shown below.

```
javac My_Calculation.java
java My_Calculation
```

After executing the program, it will produce the following result: Output:

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.

The super keyword: The super keyword is similar to this keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.

- It is used to **invoke the superclass** constructor from subclass.

Differentiating the Members: If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable super.method();
```

Example:

```java
class Super_class {

  int num = 20;


  // display method of superclass

  public void display() {

    System.out.println("This is the display method of superclass");

  }
```

```java
}



public class Sub_class extends Super_class {

  int num = 10;


  // display method of sub class

  public void display() {

    System.out.println("This is the display method of subclass");

  }


  public void my_method() {

    // Instantiating subclass

    Sub_class sub = new Sub_class();


    // Invoking the display() method of sub class
```

```
      sub.display();


      // Invoking the display() method of superclass
      super.display();


      // printing the value of variable num of subclass
       System.out.println("value of the variable named num in sub class:"+
sub.num);


      // printing the value of variable num of superclass
         System.out.println("value of the variable named num in super
class:"+ super.num);

   }


   public static void main(String args[]) {
      Sub_class obj = new Sub_class();
      obj.my_method();
   }
```

```
                                     }
```

Compile and execute the above code using the following syntax.

```
javac Super_Demo
```

```
java Super
```

On executing the program, you will get the following result: Output
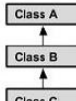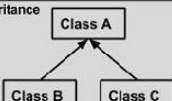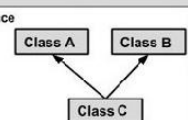
```
This is the display method of subclass

This is the display method of superclass

value of the variable named num in sub class:10

value of the variable named num in super class:20
```

Types of Inheritance:

There are various types of inheritance as demonstrated below.

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal −

Example: public class extends Animal, Mammal{}

However, a class can implement one or more interfaces, which has helped Java get rid of the impossibility of multiple inheritance.

Interchangeable objects with polymorphism: When dealing with type hierarchies, you often want to treat an object not as the specific type that it is, but instead as its base type. This allows you to write code that doesn't depend on specific types. In the shape example, methods manipulate generic shapes, unconcerned about whether they're circles, squares, triangles, or some shape that hasn't even been defined yet. All shapes can be drawn, erased, and moved, so these methods simply send a message to a shape object; they don't worry about how the object copes with the message.

Such code is unaffected by the addition of new types, and adding new types is the most common way to extend an object-oriented program to handle new situations. For example, you can derive a new subtype of shape called pentagon without modifying the methods that deal only with generic shapes. This ability to easily extend a design by deriving new subtypes is one of the essential ways to encapsulate change. This greatly improves designs while reducing the cost of software maintenance.

There's a problem, however, with attempting to treat derived-type objects as their generic base types (circles as shapes, bicycles as vehicles, cormorants as birds, etc.). If a method is going to tell a generic shape to draw itself, or a generic vehicle to steer, or a generic bird to move, the compiler cannot know at compile time precisely what piece of code will be executed. That's the whole point—when the message is sent, the programmer doesn't want to know what piece of code will be executed; the draw method can be applied equally to a circle, a square, or a triangle, and the object will execute the proper code depending on its specific type.

If you don't have to know what piece of code will be executed, then when you add a new subtype, the code it executes can be different without requiring changes to the method that calls it. Therefore, the compiler cannot know precisely what piece of code is executed, so what does it do?

For example, in the following diagram the **BirdController** object just works with generic Bird objects and does not know what exact type they are. This is convenient from **BirdController's** perspective because it doesn't have to write special code to determine the exact type of Bird it's working with or that Bird's behavior. So how does it happen that, when move( ) is called while ignoring the specific type of Bird, the right behavior will occur (a Goose walks, flies, or swims, and a Penguin walks or swims)?
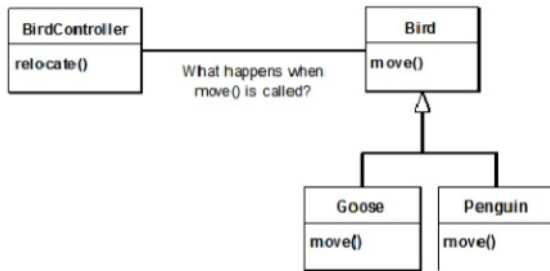
The answer is the primary twist in object-oriented programming: The compiler cannot make a function call in the traditional sense. The function call generated by a non-OOP compiler causes what is called early binding, a term you may not have heard before because you've never thought about it any other way. It means the compiler generates a call to a specific function name, and the runtime system resolves this call to the absolute address of the code to be executed. In OOP, the program cannot determine the address of the code until run time, so some other scheme is necessary when a message is sent to a generic object.

To solve the problem, object-oriented languages use the concept of late binding. When you send a message to an object, the code being called isn't determined until run time. The compiler does ensure that the method exists and performs type checking on the arguments and return value, but it doesn't know the exact code to execute.

To perform late binding, Java uses a special bit of code in lieu of the absolute call. This code calculates the address of the method body, using information stored in the object (this process is covered in great detail in the Polymorphism chapter). Thus, each object can behave differently according to the contents of that special bit of code. When you send a message to an object, the object actually does figure out what to do with that message.

In some languages you must explicitly state that you want a method to have the flexibility of late binding properties (C++ uses the virtual keyword to do this). In these languages, by default, methods are not dynamically bound. In Java, dynamic binding is the default behavior and you don't need to remember to add any extra keywords in order to get   polymorphism.

CHAPTER 9 | EXCEPTION HANDLING

Error Handling With Exception

The ideal time to catch an error is at compile time, before you even try to run the program. However, not all errors can be detected at compile time. The rest of the problems must be handled at run time through some formality that allows the originator of the error to pass appropriate information to a recipient who will know how to handle the difficulty properly.

Improved error recovery is one of the most powerful ways that you can increase the robustness of your code. Error recovery is a fundamental concern for every program you write, but it's especially important in Java, where one of the primary goals is to create program components for others to use. To create a robust system, each component must be robust. By providing a consistent error-reporting model using exceptions, Java allows components to reliably communicate problems to client code.

The goals for exception handling in Java are to simplify the creation of large, reliable programs using less code than currently possible, and to do so with more confidence that your application doesn't have an unhandled error. Exceptions are not terribly difficult to learn, and are one of those features that provide immediate and significant benefits to your project.

Because exception handling is the only official way that Java reports errors, and it is enforced by the Java compiler, there are only so many examples that can be written in this book without learning about exception handling. This chapter introduces you to the code that you need to write to properly handle exceptions, and shows how you can generate your own exceptions if one of your methods gets into trouble.

Concepts:

C and other earlier languages often had multiple error-handling schemes, and these were generally established by convention and not as part of the programming language. Typically, you returned a special value or set a flag, and the recipient was supposed to look at the value or the flag and determine that something was amiss. However, as the years passed, it was discovered that programmers who use a library tend to think of themselves as invincible—as in "Yes, errors might happen to others, but not in my code." So, not too surprisingly, they wouldn't check for the error conditions (and sometimes the error conditions were too silly to check for1). If you were thorough enough to

check for an error every time you called a method, your code could turn into an unreadable nightmare. Because programmers could still coax systems out of these languages, they were resistant to admitting the truth: that this approach to handling errors was a major limitation to creating large, robust, maintainable programs.

The solution is to take the casual nature out of error handling and to enforce formality. This actually has a long history, because implementations of exception handling go back to operating systems in the 1960s, and even to BASIC'S "on error goto." But C++ exception handling was based on Ada, and Java's is based primarily on C++ (although it looks more like Object Pascal).

The word "exception" is meant in the sense of "I take exception to that." At the point where the problem occurs, you might not know what to do with it, but you do know that you can't just continue on merrily; you must stop, and somebody, somewhere, must figure out what to do. But you don't have enough information in the current context to fix the problem. So you hand the problem out to a higher context where someone is qualified to make the proper decision.

The other rather significant benefit of exceptions is that they tend to reduce the complexity of error-handling code. Without exceptions, you must check for a particular error and deal with it at multiple places in your program. With exceptions, you no longer need to check for errors at the point of the method call, since the exception will guarantee that someone catches it. You only need to handle the problem in one place, in the so-called exception handler. This saves you code, and it separates the code that describes what you want to do during normal execution from the code that is executed when things go awry. In general, reading, writing, and debugging code becomes much clearer with exceptions than when using the old way of error handling.

Basic exceptions:

An exceptional condition is a problem that prevents the continuation of the current method or scope. It's important to distinguish an exceptional condition from a normal problem, in which you have enough information in the current context to somehow cope with the difficulty. With an exceptional condition, you cannot continue processing because you don't have the information necessary to deal with the problem in the current context. All you can do is jump out of the current context and relegate that problem to a higher context. This is what happens when you throw an exception.

Division is a simple example. If you're about to divide by zero, it's worth checking for that condition. But what does it mean that the denominator is zero? Maybe you know, in the context of the problem you're trying to solve in that particular method, how to deal with a zero denominator. But if it's an unexpected value, you can't deal with it and so must throw an exception rather than continuing along that execution path.

When you throw an exception, several things happen. First, the exception object is created in the same way that any Java object is created: on the heap, with new. Then the current path of execution (the one you couldn't continue) is stopped and the reference for the exception object is ejected from the current context. At this point the exception-handling mechanism takes over and begins to look for an appropriate place to continue executing the program. This appropriate place is the exception handler, whose job is to recover from the problem so the program can either try another tack or just continue.

As a simple example of throwing an exception, consider an object reference called t. It's possible that you might be passed a reference that hasn't been initialized, so you might want to check before trying to call a method using that object reference. You can send information about the error into a larger context by creating an object representing your information and "throwing" it out of your current context. This is called throwing an exception. Here's what it looks like:

```
if(t == null)
    throw new NullPointerException();
```

This throws the exception, which allows you in the current context to abdicate responsibility for thinking about the issue further. It's just magically handled somewhere else. Precisely where will be shown shortly.

Exceptions allow you to think of everything that you do as a transaction, and the exceptions guard those transactions: "...the fundamental premise of transactions is that we needed exception handling in distributed computations. Transactions are the computer equivalent of contract law. If anything goes wrong, we'll just blow away the whole computation."2 You can also think about exceptions as a built-in undo system, because (with some care) you can have various recovery points in your program. If a part of the program fails, the exception will "undo" back to a known stable point in the program.

One of the most important aspects of exceptions is that if something bad happens, they don't allow a program to continue along its ordinary path. This has been a real problem in languages like C and C++; especially C, which had no way to force a program to stop going down a path if a problem occurred, so it was possible to ignore problems for a long time and get into a completely inappropriate state. Exceptions allow you to (if nothing else) force the program to stop and tell you what went wrong, or (ideally) force the program to deal with the problem and return to a stable state.

Exception arguments:

As with any object in Java, you always create exceptions on the heap using new, which allocates storage and calls a constructor. There are two constructors in all standard exceptions: The first is the default constructor, and the second takes a string argument so that you can place pertinent information in the exception:

```
throw new NullPointerException("t = null");
```

This string can later be extracted using various methods, as you'll see.

The keyword throw produces a number of interesting results. After creating an exception object with new, you give the resulting reference to throw. The object is, in effect, "returned" from the method, even though that object type isn't normally what the method is designed to return. A simplistic way to think about exception handling is as a different kind of return mechanism, although you get into trouble if you take that analogy too far. You can also exit from ordinary scopes by throwing an exception. In either case, an exception object is returned, and the method or scope exits.

Any similarity to an ordinary return from a method ends here, because where you return is someplace completely different from where you return for a normal method call. (You end up in an appropriate exception handler that might be far away—many levels on the call stack— from where the exception was thrown.)

In addition, you can throw any type of Throwable, which is the exception root class. Typically, you'll throw a different class of exception for each different type of error. The information about the error is represented both inside the exception object and implicitly in the name of the exception class, so someone in the bigger context can figure out what to do with your exception. (Often, the only information is the type of exception, and nothing meaningful is stored within the exception object.)

Catching an exception:

To see how an exception is caught, you must first understand the concept of a guarded region. This is a section of code that might produce exceptions and is followed by the code to handle those exceptions.

The try block:

If you're inside a method and you throw an exception (or another method that you call within this method throws an exception), that method will exit in the process of throwing. If you don't want a throw to exit the method, you can set up a special block within that method to capture the exception. This is called the try block because you "try" your various method calls there. The try block is an ordinary scope preceded by the keyword try:

```
try {
    // Code that might generate exceptions

                                        }
```

If you were checking for errors carefully in a programming language that didn't support exception handling, you'd have to surround every method call with setup and error-testing code, even if you call the same method several times. With exception handling, you put everything in a try block and capture all the exceptions in one place. This means your code is much easier to write and read because the goal of the code is not confused with the error checking.

Exception handlers:

Of course, the thrown exception must end up someplace. This "place" is the exception handler, and there's one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
// Code that might generate exceptions
} catch(Type1 id1)|{
// Handle exceptions of Type1
} catch(Type2 id2) {
// Handle exceptions of Type2
} catch(Type3 id3) {
// Handle exceptions of Type3

                                          }
```

Each catch clause (exception handler) is like a little method that takes one and only one argument of a particular type.

The identifier (id1, id2, and so on) can be used inside the handler, just like a method argument. Sometimes you never use the identifier because the type of the exception gives you enough information to deal with the exception, but the identifier must still be there.

The handlers must appear directly after the try block. If an exception is thrown, the exception-handling mechanism goes hunting for the first handler with an argument that matches the type of the exception. Then it enters that catch clause, and the exception is considered handled. The search for handlers stops once the catch clause is finished. Only the matching catch clause executes; it's not like a switch statement in which you need a break after each case to prevent the remaining ones from executing. Note that within the try block, a number of different method calls might generate the same exception, but you need only one handler.

Termination vs. resumption:

There are two basic models in exception-handling theory. Java supportst termination,3 in which you assume that the error is so critical that there's no way

to get back to where the exception occurred. Whoever threw the exception decided that there was no way to salvage the situation, and they don't want to come back.

The alternative is called resumption. It means that the exception handler is expected to do something to rectify the situation, and then the faulting method is retried, presuming success the second time. If you want resumption, it means you still hope to continue execution after the exception is handled.

If you want resumption-like behavior in Java, don't throw an exception when you encounter an error. Instead, call a method that fixes the problem. Alternatively, place your try block inside a while loop that keeps reentering the try block until the result is satisfactory.

Historically, programmers using operating systems that supported resumptive exception handling eventually ended up using termination-like code and skipping resumption. So although resumption sounds attractive at first, it isn't quite so useful in practice. The dominant reason is probably the coupling that results: A resumptive handler would need to be aware of where the exception is thrown, and contain non-generic code specific to the throwing location. This makes the code difficult to write and maintain, especially for large systems where the exception can be generated from many points.

Catching any exception:

It is possible to create a handler that catches any type of exception. You do this by catching the base-class exception type Exception (there are other types of base exceptions, but Exception is the base that's pertinent to virtually all programming activities):

```
catch(Exception e) {
System.out.println("Caught an exception");

                                        }
```

This will catch any exception, so if you use it you'll want to put it at the end of your list of handlers to avoid preempting any exception handlers that might otherwise follow it.

Since the Exception class is the base of all the exception classes that are important to the programmer, you don't get much specific information about the exception, but you can call the methods that come from its base type Throwable:

```
String getMessage( )
String getLocalizedMessage( )
```
Gets the detail message, or a message adjusted for this particular locale.

```
String toString( )
```

Returns a short description of the Throwable, including the detail message if there is one.

```
void printStackTrace( )
voidprintStackTrace(PrintStream)  voidprintStackTrace(java.io.PrintWriter)
```

Prints the Throwable and the Throwable's call stack trace. The call stack shows the sequence of method calls that brought you to the point at which the exception was thrown. The first version prints to standard error, the second and third print to a stream of your choice (in the I/O chapter, you'll understand why there are two types of streams).

Throwable fillInStackTrace( )

Records information within this Throwable object about the current state of the stack frames. Useful when an application is rethrowing an error or exception (more about this shortly).

In addition, you get some other methods from Throwable's base type Object (everybody's base type). The one that might come in handy for exceptions is getClass( ), which returns an object representing the class of this object. You can in turn query this Class object for its name with getName( ), which includes package information, or getSimpleName( ), which produces the class name alone.

Here's an example that shows the use of the basic Exception methods:

```java
public class ExceptionMethods {
public static void main(String[] args) {
try {
throw new Exception("My Exception");
} catch(Exception e) {
print("Caught Exception");     print("getMessage():" +
e.getMessage());     print("getLocalizedMessage():"
+      e.getLocalizedMessage());
print("toString():" +
e);     print("printStackTrace():");     e.printStackTrace(System.out);

}

}

}
```

Output:

```
Caught Exception

getMessage() : My Exception
getLocalizedMessage() : My Exception
toString(): java.lang.Exception : My Exception
printStackTrace():
java.lang.Exception: My Exception
at ExceptionMethods.main(ExceptionMethods.java:8)
```

You can see that the methods provide successively more information—each is effectively a superset of the previous one.

CHAPTER 10 | Algorithms & the Big O Notation

Thinking in Algorithms

An algorithm is a detailed step-by-step instruction set or formula for solving a problem or completing a task. In computing, programmers write algorithms that instruct the computer how to perform a task.

When you think of an algorithm in the most general way (not just in regards to computing), algorithms are everywhere. A recipe for making food is an algorithm, the method you use to solve addition or long division problems is an algorithm, and the process of folding a shirt or a pair of pants is an algorithm.

So, what is a programming algorithm?

You can think of a programming algorithm as a recipe that describes the exact steps needed for the computer to solve a problem or reach a goal. We've all seen food recipes - they list the ingredients needed and a set of steps for how to make the described meal. Well, an algorithm is just like that. In computer lingo, the word for a recipe is a procedure, and the ingredients are called inputs. Your computer looks at your procedure, follows it to the letter, and you get to see the results, which are called outputs.

A programming algorithm describes how to do something, and your computer will do it exactly that way every time. Well, it will once you convert your algorithm into a language it understands!

However, it's important to note that a programming algorithm is not computer code. It's written in simple English (or whatever the programmer speaks). It doesn't beat around the bush--it has a start, a middle, and an end. In fact, you will probably label the first step 'start' and the last step 'end.' It includes only what you need to carry out the task. It does not include anything unclear, often called ambiguous in computer lingo, that someone reading it might wonder about.

It always leads to a solution and tries to be the most efficient solution we can think up. It's often a good idea to number the steps, but you don't have to. Instead of numbered steps, some folks use indentation and write in **pseudocode**, which is a semi-programming language used to describe the steps in an algorithm. But, we won't use that here since simplicity is the main thing.

Qualities of a good algorithm:
   1. Inputs and outputs should be defined precisely.

2. Each step in algorithm should be clear and unambiguous.

3. Algorithm should be most effective among many different ways to solve a problem.

4. An algorithm shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.

Examples of Algorithms In Programming: **Write an algorithm to add two numbers entered by user.**

Step 1: Start
Step 2: Declare variables num1, num2 and sum.

Step 3: Read values num1 and num2.

Step 4: Add num1 and num2 and assign the result to sum.

   $$sum \leftarrow num1+num2$$

Step 5: Display sum

Step 6: Stop

Write an algorithm to find the largest among three different numbers entered by user.

Step 1: Start

Step 2: Declare variables a, b and c.

Step 3: Read variables a, b and c.

Step 4: If a>b
    If a>c
      Display a is the largest number.
    Else
      Display c is the largest number.
   Else
    If b>c
      Display b is the largest number.
    Else
        Display c is the greatest number.

Step 5: Stop

Write an algorithm to find all roots of a quadratic equation $ax^2+bx+c=0$.

Step 1: Start
Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;
Step 3: Calculate discriminant D ← b2-4ac
Step 4: If D≥0
      r1 ← (-b+√D)/2a
       r2 ← (-b-√D)/2a
      Display r1 and r2 as roots.
    Else
      Calculate real part and imaginary part rp ← b/2a
ip ← √(-D)/2a Display rp+j(ip) and rp-j(ip) as roots
Step 5: Stop

Write an algorithm to find the factorial of a number entered by user.

Step 1: Start Step 2: Declare variables n,factorial and i.

Step 3: Initialize variables factorial ← 1

　　　i ← 1

Step 4: Read value of n Step 5: Repeat the steps until i=n 5.1: factorial ← factorial*i 5.2: i ← i+1

Step 6: Display factorial Step 7: Stop

Write an algorithm to check whether a number entered by user is prime or not.

Step 1: Start

Step 2: Declare variables n,i,flag.

Step 3: Initialize variables
    flag ← 1
     i ← 2

Step 4: Read n from user.

Step 5: Repeat the steps until i<(n/2)
    5.1 If remainder of n÷i equals 0
        flag ← 0
        Go to step 6
    5.2 i ← i+1

Step 6: If flag=0
        Display n is not prime
      else
        Display n is prime

Step 7: Stop

Write an algorithm to find the Fibonacci series till term≤1000.

Step 1: Start

Step 2: Declare variables first_term,second_term and temp.

Step 3: Initialize variables first_term ← 0 second_term ← 1

Step 4: Display first_term and second_term

Step 5: Repeat the steps until second_term≤1000

    5.1: temp ← second_term

    5.2: second_term ← second_term+first term

    5.3: first_term ← temp

    5.4: Display second_term

Step 6: Stop

What kinds of problems are solved by   algorithms?

Sorting is by no means the only computational problem for which algorithms have been developed. Practical applications of algorithms are ubiquitous and include the following examples:

1. The Human Genome Project has made great progress toward the goals of identifying all the 100,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms. Although the solutions to the various problems involved are beyond the scope of this book, many methods to solve these biological problems use ideas from several of the chapters in this book, thereby enabling scientists to accomplish tasks while using resources efficiently. The savings are in time, both human and machine, and in money, as more information can be extracted from laboratory techniques.

2. The Internet enables people all around the world to quickly access and retrieve large amounts of information. With the aid of clever algorithms, sites on the Internet are able to manage and manipulate this large volume of data. Examples of problems that make essential use of algorithms include finding good routes on which the data will travel, and using a search engine to quickly find pages on which particular information resides.

3. Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit card numbers, passwords, and bank statements. The core technologies used in electronic commerce include public-key cryptography and digital signatures, which are based on numerical algorithms and number theory.

4. Manufacturing and other commercial enterprises often need to allocate scarce resources in the most beneficial way. An oil company may wish to know where to place its wells in order to maximize its expected profit. A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election. An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that

government regulations regarding crew scheduling are met. An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively. All of these are examples of problems that can be solved using linear programming.

Although some of the details of these examples are beyond the scope of this book, we do give underlying techniques that apply to these problems and problem areas.

Algorithms as a technology

Suppose computers were infinitely fast and computer memory was free. Would you have any reason to study algorithms? The answer is yes, if for no other reason than that you would still like to demonstrate that your solution method terminates and does so with the correct answer.

If computers were infinitely fast, any correct method for solving a problem would do.

You would probably want your implementation to be within the bounds of good software engineering practice (for example, your implementation should be well designed and documented), but you would most often use whichever method was the easiest to implement. Of course, computers may be fast, but they are not infinitely fast. And memory may be inexpensive, but it is not free.

Computing time is therefore a bounded resource, and so is space in memory. You should use these resources wisely, and algorithms that are efficient in terms of time or space will help you do so.

It's useful to visualize some aspects of how things are laid out while the program is running— in particular how memory is arranged. There are five different places to store data: **1. Registers.** This is the fastest storage because it exists in a place different from that of other storage: inside the processor. However, the number of registers is severely limited, so registers are allocated as they are needed. You don't have direct control, nor do you see any evidence in your programs that registers even exist (C & C++, on the other hand, allow you to suggest register allocation to the compiler).

2. The stack. This lives in the general random-access memory (RAM) area, but has direct support from the processor via its stack pointer. The stack pointer is moved down to create new memory and moved up to release that memory. This is an extremely fast and efficient way to allocate storage, second only to registers. The Java system must know, while it is creating the program, the exact lifetime of all the items that are stored on the stack. This constraint places limits on the flexibility of your programs, so while some Java storage exists on the stack—in particular, object references—Java objects themselves are not placed on the stack.

3. The heap. This is a general-purpose pool of memory (also in the RAM area) where all Java objects live. The nice thing about the heap is that, unlike the stack, the compiler doesn't need to know how long that storage must stay on the heap. Thus, there's a great deal of flexibility in using storage on the heap. Whenever you need an object, you simply write the code to create it by using new, and the storage is allocated on the heap when that code is executed. Of course there's a price you pay for this flexibility: It may take more time to allocate and clean up heap storage than stack storage (if you even could create objects on the stack in Java, as you can in C++).

4. Constant storage. Constant values are often placed directly in the program code, which is safe since they can never change. Sometimes constants are cordoned off by themselves so that they can be optionally placed in read-only memory (ROM), in embedded systems.2

5. Non-RAM storage. If data lives completely outside a program, it can exist while the program is not running, outside the control of the program. The two primary examples of this are streamed objects, in which objects are turned into streams of bytes, generally to be sent to another machine, and persistent objects, in which the objects are placed on disk so they will hold their state even when

the program is terminated. The trick with these types of storage is turning the objects into something that can exist on the other medium, and yet can be resurrected into a regular RAMbased object when necessary. Java provides support for lightweight persistence, and mechanisms such as JDBC and Hibernate provide more sophisticated support for storing and retrieving object information in databases.

Big O Notation

Big O notation is the language we use for articulating how long an algorithm takes to run. It's how we compare the efficiency of different approaches to a problem.

Imagine you have a list of 10 objects, and you want to sort them in order. There's a whole bunch of algorithms you can use to make that happen, but not all algorithms are built equal. Some are quicker than others but more importantly the speed of an algorithm can vary depending on how many items it's dealing with. Big O is a way of measuring how an algorithm scales. Big O references how **complex** an algorithm is.

Big O is represented using something like O(n). The O simply denoted we're talking about big O and you can ignore it (at least for the purpose of the interview). *n* is the thing the complexity is in relation to; for programming interview questions this is almost always the size of a collection. The complexity will increase or decrease in accordance with the size of the data store.

Below is a list of the Big O complexities in order of how well they scale relative to the dataset.

O(1)/Constant Complexity: Constant. This means irrelevant of the size of the data set the algorithm will always take a constant time. *1 item takes 1 second, 10 items takes 1 second, 100 items takes 1 second.* It always takes the same amount of time.

O(log n)/Logarithmic Complexity: Not as good as constant, but still pretty good. The time taken increases with the size of the data set, but not proportionately so. This means the algorithm takes longer per item on smaller datasets relative to larger ones. *1 item takes 1 second, 10 items takes 2 seconds, 100 items takes 3 seconds.* If your dataset has 10 items, each item causes 0.2 seconds latency. If your dataset has 100, it only takes 0.03 seconds extra per item. This makes log n algorithms very scalable.

O(n)/Linear Complexity: The larger the data set, the time taken grows proportionately. *1 item takes 1 second, 10 items takes 10 seconds, 100 items takes 100 seconds.*

O(n log n): A nice combination of the previous two. Normally there's 2 parts to the sort, the first loop is O(n), the second is O(log n), combining to form O(n log n) *1 item takes 2 seconds, 10 items takes 12 seconds, 100 items takes 103 seconds.*

O(n^2)/Quadratic Complexity: Things are getting extra slow. *1 item takes 1 second, 10 items takes 100, 100 items takes 10000.*

O(2^n): Exponential Growth! The algorithm takes twice as long for every new element added. *1 item takes 1 second, 10 items takes 1024 seconds, 100 items takes* 1267650600228229401496703205376 seconds.

Examples:

Hopefully you're with me so far, but let's dive into some example algorithms for sorting and searching. The important thing is to be able to explain what complexity an algorithm is. Interviewers love to get candidates to design algorithms and then ask what the complexity of it is.

1. O(1)

Irrelevant of the size, it will always return at constant speed. The javadoc for Queue states that it is *"constant time for the retrieval methods (peek, element, and size)"*. It's pretty clear why this is the case. For peek, we are always returning the first element which we always have a reference to; it doesn't matter how many elements follow it. The size of the list is updated upon element addition/removal, and referencing this number is just one operation to access no matter what the size of the list is.

2. O(log n)

The classic example is a Binary search. You're a massive geek so you've obviously alphabetised your movie collection. To find your copy of "Back To The Future", you first go to the middle of the list. You discover the middle film is "Meet The Fockers", so you then head to the movie in between the start and this film. You discover this is "Children of Men". You repeat this again and you've found back to the future.

Although adding more elements will increase the amount of time it takes to search, it doesn't do so proportionally. Therefore it is O(log n).

3. O(n)

As discussed in my post on collections, LinkedLists are not so good (relatively speaking) when it comes to retrieval. It actually has a Big O of O(n) as in the **worst case**, to find an element T which is the last element in the list, it is necessary to navigate the entire list of n elements to find T. As the number of elements increases so does the access time in proportion.

4. O(n log n)

The best example of O(n log n) is a **merge sort**. This is a divide and conquer

algorithm. Imagine you have a list of integers. We divide the list in two again and again until we are left with with a number of lists with 1 item in: each of these lists is therefore sorted.We then merge each list with it's neighbour (comparing the first elements of each every time). We repeat this with the new composite list until we have our sorted result.

How to figure out Big O in an interview?

This is not an exhaustive list of Big O. Much as you can $O(n^2)$, you can also have $O(n^{\wedge3})$ (imagine throwing an extra loop into your bubble sort for no apparent reason). What the list on this page should allow you to do is have a stab in the dark at figuring out what the big O of an algorithm is. If someone is asking you this during an interview they probably want to see how you try and figure it out. Break down the loops and processing.

- Does it have to go through the entire list? There will be an *n* in there somewhere.
- Does the algorithms processing time increase at a slower rate than the size of the data set? Then there's probably a *log n* in there.
- Are there nested loops? You're probably looking at $n^{\wedge2}$ or $n^{\wedge3}$.
- Is access time constant irrelevant of the size of the dataset?? *O(1)*

Sample Questions:

I have an array of the numbers 1 to 100 in a random number. One of the numbers is missing. Write an algorithm to figure out what the number is and what position is missing.

There are many variations of this question all of which are very popular. To calculate the missing number we can simply add up all the numbers we do have, and subtract this from the expected answer of the sum of all numbers between 1 and 100. To do this we have to iterate the list once. Whilst doing this we can also note which spot has the gap.

public void findTheBlank(int[] theNumbers) {

```
public void findTheBlank(int[] theNumbers) {

int sumOfAllNumbers = 0;

int sumOfNumbersPresent = 0;

int blankSpace = 0;


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

sumOfAllNumbers += i + 1;

sumOfNumbersPresent += theNumbers[i];

if (theNumbers[i] == 0)

blankSpace = i;

                              }

System.out.println("Missing number = " + (sumOfAllNumbers -
sumOfNumbersPresent) + " at location " + blankSpace +" of the array");

                              }

new BubbleSort().findTheBlank(new int[]{7,6,0,1,3,2,4});

//Missing number = 5 at location 2 of the array
```
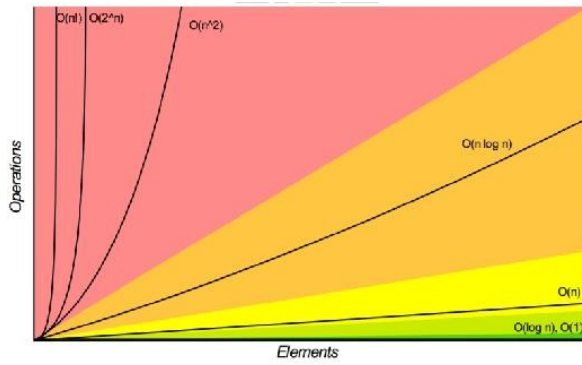
What is the big O for this algorithm?

Our algorithm iterates through our list once, so it's simply **O(n).**

Big-O Complexity Chart:

O(n!) O(2^n)   O(n^2)

Operations

O(n log n)

O(n)

O(log n), O(1)

Elements

Horrible Bad Fair Good Excellent

Array Sorting Algorithms complexity:

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	$\Omega(n\ \log(n))$	$\Theta(n\ \log(n))$	$O(n^2)$	$O(\log(n))$
Mergesort	$\Omega(n\ \log(n))$	$\Theta(n\ \log(n))$	$O(n\ \log(n))$	$O(n)$
Timsort	$\Omega(n)$	$\Theta(n\ \log(n))$	$O(n\ \log(n))$	$O(n)$
Heapsort	$\Omega(n\ \log(n))$	$\Theta(n\ \log(n))$	$O(n\ \log(n))$	$O(1)$
Bubble Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
Tree Sort	$\Omega(n\ \log(n))$	$\Theta(n\ \log(n))$	$O(n^2)$	$O(n)$
Shell Sort	$\Omega(n\ \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
Bucket Sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$

CHAPTER 11 | DATA STRUCTURE

Even though computers can perform literally millions of mathematical computations per second, when a problem gets large and complicated, performance can nonetheless be an important consideration. One of the most crucial aspects to how quickly a problem can be solved is how the data is stored in memory.

To illustrate this point, consider going to the local library to find a book about a specific subject matter. Most likely, you will be able to use some kind of electronic reference or, in the worst case, a card catalog, to determine the title and author of the book you want. Since the books are typically shelved by category, and within each category sorted by author's name, it is a fairly straightforward and painless process to then physically select your book from the shelves.

Now, suppose instead you came to the library in search of a particular book, but instead of organized shelves, were greeted with large garbage bags lining both sides of the room, each arbitrarily filled with books that may or may not have anything to do with one another. It would take hours, or even days, to find the book you needed, a comparative eternity. This is how software runs when data is not stored in an efficient format appropriate to the application.

Simple Data Structure:

The simplest data structures are primitive variables. They hold a single value, and beyond that, are of limited use. When many related values need to be stored, an array is used. It is assumed that the reader of this article has a solid understanding of variables and arrays.

A somewhat more difficult concept, though equally primitive, are pointers. Pointers, instead of holding an actual value, simply hold a memory address that, in theory, contains some useful piece of data. Most seasoned C++ coders have a solid understanding of how to use pointers, and many of the caveats, while fledgling programmers may find themselves a bit spoiled by more modern "managed" languages which, for better or worse, handle pointers implicitly.

Either way, it should suffice to know that pointers "point" somewhere in memory, and do not actually store data themselves.

A less abstract way to think about pointers is in how the human mind remembers (or cannot remember) certain things. Many times, a good engineer may not necessarily know a particular formula/constant/equation, but when asked, they could tell you exactly which reference to check.

Arrays:
Arrays are a very simple data structure, and may be thought of as a list of a fixed length. Arrays are nice because of their simplicity, and are well suited for situations where the number of data items is known (or can be programmatically determined). Suppose you need a piece of code to calculate the average of several numbers.

An array is a perfect data structure to hold the individual values, since they have no specific order, and the required computations do not require any special handling other than to iterate through all of the values. The other big strength of arrays is that they can be accessed randomly, by index. For instance, if you have an array containing a list of names of students seated in a classroom, where each seat is numbered 1 through n, then studentName[i] is a trivial way to read or store the name of the student in seat i.

An array might also be thought of as a pre-bound pad of paper. It has a fixed number of pages, each page holds information, and is in a predefined location that never changes.

Sorting Techniques: Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

- **Telephone Directory** − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

- **Dictionary** − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

In-place Sorting and Not-in-place Sorting: Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within

the array itself. This is called **in-place sorting**. Bubble sort is an example of in-place sorting.

However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called **not-in-place sorting**. Merge-sort is an example of not-in-place sorting.

Stable and Not Stable Sorting: If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.
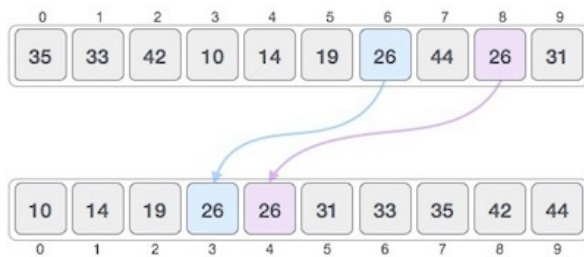
If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.
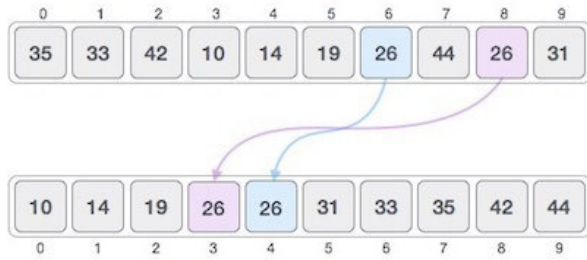
Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Adaptive and Non-Adaptive Sorting Algorithm:

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Sorting and Searching

Binary Search

Binary search is one of the fundamental algorithms in computer science. In order to explore it, we'll first build up a theoretical backbone, then use that to implement the algorithm properly and avoid those nasty off-by-one errors everyone's been talking about.

Finding a value in a sorted sequence

In its simplest form, binary search is used to quickly find a value in a sorted sequence (consider a sequence an ordinary array for now). We'll call the sought value the *target* value for clarity. Binary search maintains a contiguous subsequence of the starting sequence where the target value is surely located. This is called the *search space*. The search space is initially the entire sequence. At each step, the algorithm compares the median value in the search space to the target value. Based on the comparison and because the sequence is sorted, it can then eliminate half of the search space. By doing this repeatedly, it will eventually be left with a search space consisting of a single element, the target value.

For example, consider the following sequence of integers sorted in ascending order and say we are looking for the number 55:

0	5	13	19	22	41	55	68	72	81	98

We are interested in the location of the target value in the sequence so we will represent the search space as indices into the sequence. Initially, the search space contains indices 1 through 11. Since the search space is really an interval, it suffices to store just two numbers, the low and high indices. As described above, we now choose the median value, which is the value at index 6 (the midpoint between 1 and 11): this value is 41 and it is smaller than the target value. From this we conclude not only that the element at index 6 is not the target value, but also that no element at indices between 1 and 5 can be the target value, because all elements at these indices are smaller than 41, which is smaller than the target value. This brings the search space down to indices 7 through 11:

55	68	72	81	98

Proceeding in a similar fashion, we chop off the second half of the search space and are left with:

55	68

Depending on how we choose the median of an even number of elements we will either find 55 in the next step or chop off 68 to get a search space of only one element.

Either way, we conclude that the index where the target value is located is 7.

If the target value was not present in the sequence, binary search would empty the search space entirely. This condition is easy to check and handle. Here is some code to go with the description:

```
binary_search(A, target): lo = 1, hi = size(A) while lo <= hi: mid = lo + (hi-lo)/2

    if A[mid] == target:
        return mid

     else if A[mid] < target:

      lo = mid+1

    else: hi = mid-1


  // target was not found
```

Complexity:

Since each comparison binary search uses halves the search space, we can assert and easily prove that binary search will never use more than (in big-oh notation) $O(\log N)$ comparisons to find the target value.

The logarithm is an awfully slowly growing function. In case you're not aware of just how efficient binary search is, consider looking up a name in a phone book containing a million names. Binary search lets you systematically find any given name using at most 21 comparisons. If you could manage a list containing all the people in the world sorted by name, you could find any person in less than 35 steps.

Program: Java implementation of recursive Binary Search

class BinarySearch

```java
{
    // Returns index of x if it is present in arr[l..r], else // return -1
    int binarySearch(int arr[], int l, int r, int x)

    {
        if (r>=l)

        {
            int mid = l + (r - l)/2;

            // If the element is present at the middle itself
            if (arr[mid] == x)
                return mid;

            // If element is smaller than mid, then it can only // be present in left subarray
            if (arr[mid] > x)
                return binarySearch(arr, l, mid-1, x);

            // Else the element can only be present in right
            // subarray
            return binarySearch(arr, mid+1, r, x);

        }

        // We reach here when element is not present in array
        return -1;

    }

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

```
{
```

```
BinarySearch ob = new BinarySearch();
int arr[] = {2,3,4,10,40};
```

```
        int n = arr.length;
        int x = 10;
        int result = ob.binarySearch(arr,0,n-1,x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index "+result);
    }
}
```

Output:

```
Element is present at index 3
```

Program: Iterative implementation of Binary Search

```java
class BinarySearch
{
    // Returns index of x if it is present in arr[], else
    // return -1
    int binarySearch(int arr[], int x)
    {
        int l = 0, r = arr.length - 1;
        while (l <= r)
        {
            int m = l + (r-l)/2;

            // Check if x is present at mid
            if (arr[m] == x)
                return m;

            // If x greater, ignore left half
            if (arr[m] < x)
                l = m + 1;

            // If x is smaller, ignore right half
            else
                r = m - 1;
        }

        // if we reach here, then element was not present
        return -1;
    }
}
```

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

{

BinarySearch ob = new BinarySearch();
int arr[] = {2, 3, 4, 10, 40};
int n = arr.length;
```

```
        int x = 10;
        int result = ob.binarySearch(arr, x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index "+result);
    }
}
```

Output:

```
 Element is present at index 3
```

Bubble Sort

Bubble sort algorithm is known as the simplest sorting algorithm.

In bubble sort algorithm, array is traversed from first element to last element. Here, current element is compared with the next element. If current element is greater than the next element, it is swapped.

Algorithm: We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort(list) for all elements of list if list[i] > list[i+1]

      swap(list[i], list[i+1]) end if end for return list end BubbleSort
```

Program: Bubble sort program in java

```java
public class BubbleSortExample {
    static void bubbleSort(int[] arr) {
        int n = arr.length;
        int temp = 0;
        for(int i=0; i < n; i++){
            for(int j=1; j < (n-i); j++){
                if(arr[j-1] > arr[j]){
                    //swap elements
                    temp = arr[j-1];
                    arr[j-1] = arr[j];
                    arr[j] = temp;


                }


                }


            }


        }
```

```java
public static void main(String[] args) {
        int arr[] ={3,60,35,2,45,320,5};
System.out.println("Array Before Bubble Sort");
        for(int i=0; i < arr.length; i++){
            System.out.print(arr[i] + " ");

        }

    System.out.println();

    bubleSort(arr);//sorting array elements using bubble sort

    System.out.println("Array After Bubble Sort");
    for(int i=0; i < arr.length; i++){
        System.out.print(arr[i] + " ");

        }


        }

        }
```

Output:

Array Before Bubble Sort

3 60 35 2 45 320 5

Array After Bubble Sort

2 3 5 35 45 60 320

Insertion sort

Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.

Algorithm

Step 1 − If it is the first element, it is already sorted.
return 1;
Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

Program: Java program for implementation of Insertion Sort

```java
class InsertionSort
{
    /*Function to sort array using insertion sort*/
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i=1; i<n; ++i)
        {
            int key = arr[i];
            int j = i-1;
            /* Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position */
            while (j>=0 && arr[j] > key)
            {
                arr[j+1] = arr[j];
                j = j-1;
            }
            arr[j+1] = key;
        }
    }

    /* A utility function to print array of size n*/
    static void printArray(int arr[])
    {
```

```
    int n = arr.length;
    for (int i=0; i<n; ++i)
       System.out.print(arr[i] + " ");

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

```
    public static void main(String args[])
      {
        int arr[] = {12, 11, 13, 5, 6};
          InsertionSort ob = new InsertionSort();
        ob.sort(arr);
        printArray(arr);
      }
}
```

Output:

```
 5 6 11 12 13
```

Mergesort

The *Mergesort* algorithm can be used to sort a collection of objects. *Mergesort* is a so called *divide and conquer* algorithm. *Divide and conquer* algorithms divide the original data into smaller sets of data to solve the problem.

Algorithm:

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

Program: Java program for Merge Sort

class MergeSort

```
{

    // Merges two subarrays of arr[].
    // First subarray is arr[l..m]
    // Second subarray is arr[m+1..r]
    void merge(int arr[], int l, int m, int r)

    {

        // Find sizes of two subarrays to be merged
        int n1 = m - l + 1;
        int n2 = r - m;

        /* Create temp arrays */
        int L[] = new int [n1];
```

```java
    int R[] = new int [n2];

    /*Copy data to temp arrays*/
    for (int i=0; i<n1; ++i)
        L[i] = arr[l + i];
    for (int j=0; j<n2; ++j)
        R[j] = arr[m + 1+ j];




    /* Merge the temp arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarry array
    int k = l;
    while (i < n1 && j < n2)

                                {

        if (L[i] <= R[j])

                                {

            arr[k] = L[i];
            i++;

                                }

    else

                                {

            arr[k] = R[j];
            j++;

                                }
```

```
        }

    k++;

                                                }


    /* Copy remaining elements of L[] if any */
    while (i < n1)
```

```c
        {
            arr[k] = L[i];
            i++;
            k++;
        }

    /* Copy remaining elements of L[] if any */
    while (j < n2)
        {
            arr[k] = R[j];
            j++;
            k++;
        }
    }

// Main function that sorts arr[l..r] using // merge()
void sort(int arr[], int l, int r)
    {
    if (l < r)
        {
        // Find the middle point
        int m = (l+r)/2;

        // Sort first and second halves
        sort(arr, l, m);
```

```
        sort(arr , m+1, r);

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}


    /* A utility function to print array of size n */
```

```java
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}

public static void main(String args[])
{
    int arr[] = {12, 11, 13, 5, 6, 7};

    System.out.println("Given Array");
    printArray(arr);

    MergeSort ob = new MergeSort();
    ob.sort(arr, 0, arr.length-1);

    System.out.println("\nSorted array");
    printArray(arr);
}
}
```

Output:

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

Quicksort

Sort algorithms order the elements of an array according to a predefined order. Quicksort is a divide and conquer algorithm. In a divide and conquer sorting algorithm the original data is separated into two parts "divide" which are individually sorted and "conquered" and then combined Algorithm:

If the array contains only one element or zero elements than the array is sorted.

If the array contains more than one element than:

- Select an element from the array. This element is called the "pivot element". For example select the element in the middle of the array.
- All elements which are smaller then the pivot element are placed in one array and all elements which are larger are placed in another array.
- Sort both arrays by recursively applying Quicksort to them.
- Combine the arrays.

Quicksort can be implemented to sort "in-place". This means that the sorting takes place in the array and that no additional array needs to be created.

Program: Java program for implementation of QuickSort

```
class QuickSort

{

    /* This function takes last element as pivot, places the pivot element at its
correct position in sorted array, and places all smaller (smaller than pivot) to left
of pivot and all greater elements to right of pivot */

    int partition(int arr[], int low, int high)

    {

        int pivot = arr[high];
        int i = (low-1); // index of smaller element
        for (int j=low; j<=high-1; j++)

        {

            // If current element is smaller than or // equal to pivot
            if (arr[j] <= pivot)
```

```
                                    {

        i++;

        // swap arr[i] and arr[j]

        int temp = arr[i];

        arr[i] = arr[j];

        arr[j] = temp;

                                    }

                                    }

    // swap arr[i+1] and arr[high] (or pivot)
    int temp = arr[i+1];

    arr[i+1] = arr[high];

    arr[high] = temp;


    return i+1;

                                    }

  /* The main function that implements QuickSort()

    arr[] --> Array to be sorted, low  --> Starting index,
```

```java
    high  --> Ending index

                                   */

  void sort(int arr[], int low, int high)

                                   {

    if (low < high)

                                   {

      /* pi is partitioning index, arr[pi] is

        now at right place */
      int pi = partition(arr, low, high);

      // Recursively sort elements before

      // partition and after partition
      sort(arr, low, pi-1);

      sort(arr, pi+1, high);

                                   }

                                   }

  /* A utility function to print array of size n */

  static void printArray(int arr[])

                                   {

    int n = arr.length;

    for (int i=0; i<n; ++i)

      System.out.print(arr[i]+" ");

    System.out.println();

                                   }
```

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

{

    int arr[] = {10, 7, 8, 9, 1, 5};

    int n = arr.length;

    QuickSort ob = new QuickSort();

    ob.sort(arr, 0, n-1);

    System.out.println("sorted array");

    printArray(arr);

}

}
```

Output:

Sorted array:

1 5 7 8 9 10

Selection sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

Algorithm:

Step 1 − **Set MIN to location 0**

Step 2 − **Search the minimum element in the list**

Step 3 − **Swap with value at location MIN**

Step 4 − **Increment MIN to point to next element**

Step 5 − **Repeat until list is sorted**

Program: Selection Sort in Java

```java
public class SelectionSortExample {
  public static void selectionSort(int[] arr){
    for (int i = 0; i < arr.length - 1; i++)

    {

      int index = i;
      for (int j = i + 1; j < arr.length; j++){
        if (arr[j] < arr[index]){
          index = j;//searching for lowest index

        }

      }

      int smallerNumber = arr[index];
      arr[index] = arr[i];
      arr[i] = smallerNumber;

    }

  }

  public static void main(String a[]){
    int[] arr1 = {9,14,3,2,43,11,58,22};
    System.out.println("Before Selection Sort");
    for(int i:arr1){
      System.out.print(i+" ");

    }
```

```
System.out.println();

selectionSort(arr1);//sorting array using selection sort
```

```java
        System.out.println("After Selection Sort");
        for(int i:arr1){
            System.out.print(i+" ");
        }
    }
}
```

Output:

Before Selection Sort
9 14 3 2 43 11 58 22
After Selection Sort
2 3 9 11 14 22 43 58

Linked List

A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.

- **Next** − Each link of a linked list contains a link to the next link called Next.

- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation:

Linked list can be visualized as a chain of nodes, where every node points to the next node.
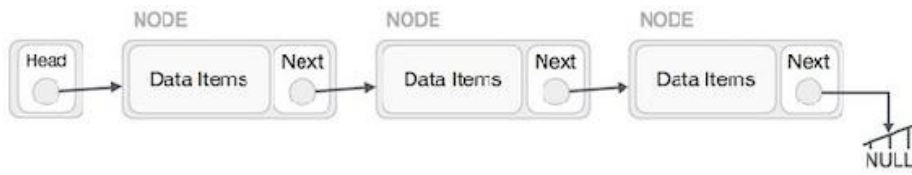
As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.

- Each link carries a data field(s) and a link field called next.

- Each link is linked with its next link using its next link.

- Last link carries a link as null to mark the end of the list.

Types of Linked List: Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.

- **Doubly Linked List** − Items can be navigated forward and backward.

- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations: Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.

- **Deletion** − Deletes an element at the beginning of the list.

- **Display** − Displays the complete list.

- **Search** − Searches an element using the given key.

- **Delete** − Deletes an element using the given key.

Insertion Operation:

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.
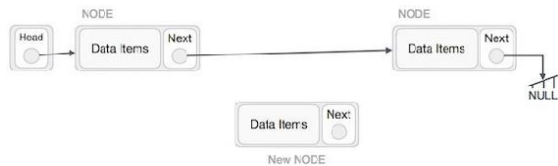
Imagine that we are inserting a node **B** (NewNode),
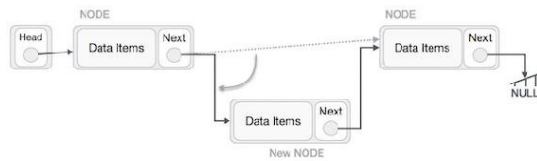between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −
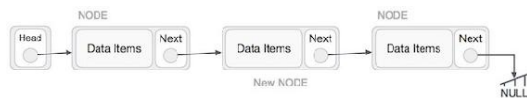
NewNode.next −> RightNode;

It should look like this −
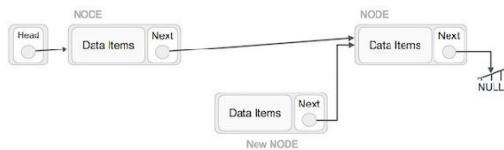
Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;

This will put the new node in the middle of the two. The new list should look like this −


Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.
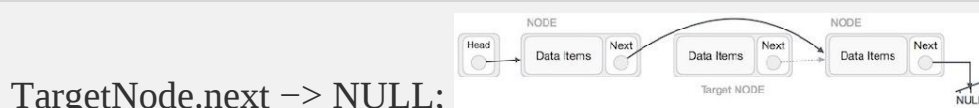
Deletion Operation:

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

The left (previous) node of the target node now should point to the next node of the target node – LeftNode.next -> TargetNode.next;
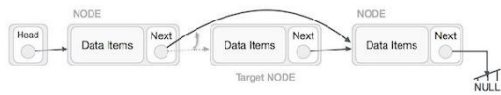
This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.
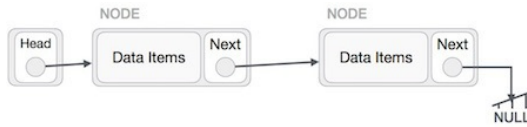
TargetNode.next -> NULL;

We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.
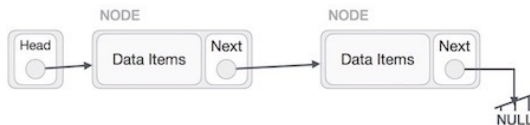
Reverse Operation:

This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.

First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −
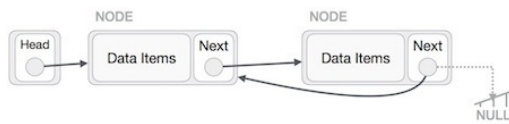
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.
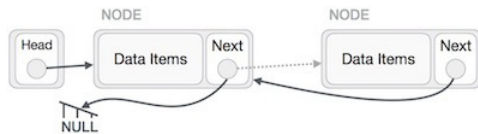
Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.
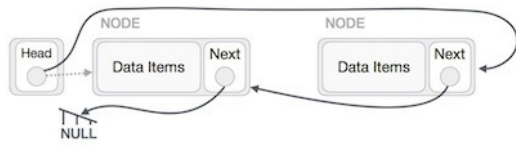
We'll make the head node point to the new first node by using the temp node.
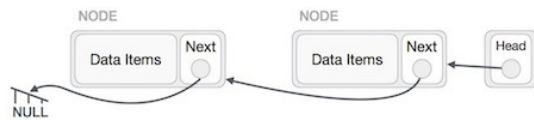
The linked list is now reversed.

Program: **Given a linked list which is sorted, how will you insert in sorted way?**

Algorithm:
Let input linked list is sorted in increasing order.

1) If Linked list is empty then make the node as head and return it.

2) If value of the node to be inserted is smaller than value of head node then insert the node at start and make it head.

3) In a loop, find the appropriate node after which the input node (let 9) is to be inserted. To find the appropriate node start from head, keep moving until you reach a node GN (10 in the below diagram) who's value is greater than the input node. The node just before GN is the appropriate node (7).

4) Insert the node (9) after the appropriate node (7) found in step 3.

Initial Linked List

Linked List after insertion of 9
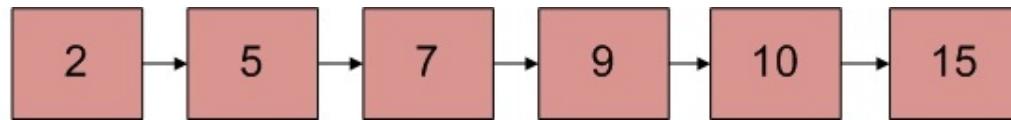
```java
class LinkedList

{

  Node head;  // head of list

  /* Linked list Node*/
  class Node

  {

    int data;
    Node next;
    Node(int d) {data = d; next = null; }

  }


  /* function to insert a new_node in a list. */
  void sortedInsert(Node new_node)

  {

    Node current;
     /* Special case for head node */
    if (head == null || head.data >= new_node.data)

    {

      new_node.next = head;
      head = new_node;

    }

    else {
       /* Locate the node before point of insertion. */
      current = head;
```

```
    while (current.next != null &&
        current.next.data < new_node.data)
      current = current.next;
 new_node.next = current.next;
current.next = new_node;
```

```java
        }

        }

        /*Utility functions*/
        /* Function to create a node */
        Node newNode(int data)

        {

            Node x = new Node(data);
            return x;

        }


        /* Function to print linked list */
        void printList()

        {

            Node temp = head;
            while (temp != null)

            {

                System.out.print(temp.data+" ");
                temp = temp.next;

            }

        }


        /* Drier function to test above methods */
        public static void main(String args[])

        {
```

```
LinkedList llist = new LinkedList();
Node new_node;
new_node = llist.newNode(5);
llist.sortedInsert(new_node);
new_node = llist.newNode(10);
llist.sortedInsert(new_node);
new_node = llist.newNode(7);
llist.sortedInsert(new_node);
new_node = llist.newNode(3);
llist.sortedInsert(new_node);
new_node = llist.newNode(1);
```

```
    llist.sortedInsert(new_node);
    new_node = llist.newNode(9);
    llist.sortedInsert(new_node);
    System.out.println("Created Linked List");
    llist.printList();
  }
}
```

Output:

```
Created Linked List

1 3 5 7 9 10
```

Program: Delete a given node in Linked List under given constraints Given a Singly Linked List, write a function to delete a given node. Your function must follow following constraints:
1) It must accept pointer to the start node as first parameter and node to be deleted as second parameter i.e., pointer to head node is not global.
2) It should not return pointer to the head node.
3) It should not accept pointer to pointer to head node.

You may assume that the Linked List never becomes empty.
Let the function name be deleteNode(). In a straightforward implementation, the function needs to modify head pointer when the node to be deleted is first node.

```
class LinkedList {

    static Node head;

    static class Node {

        int data;
        Node next;

        Node(int d) {
            data = d;
            next = null;

        }

    }

    void deleteNode(Node node, Node n) {

        // When node to be deleted is head node
        if (node == n) {
            if (node.next == null) {
                System.out.println("There is only one node. The list "
                        + "can't be made empty ");
                return;
```

```
                                    }
```

```
    /* Copy the data of next node to head */
    node.data = node.next.data;

    // store address of next node
    n = node.next;

    // Remove the link of next node
    node.next = node.next.next;

    // free memory
    System.gc();
     return;
```

```
                                    }
```

```
 // When not first node, follow the normal deletion process
// find the previous node
Node prev = node;
while (prev.next != null && prev.next != n) {
   prev = prev.next;
```

```
                                    }
```

```
// Check if node really exists in Linked List
if (prev.next == null) {
   System.out.println("Given node is not present in Linked List");
   return;
```

```
                                    }
```

```
// Remove node from Linked List
prev.next = prev.next.next;
```

```java
        // Free memory
        System.gc();

        return;

    }


    /* Utility function to print a linked list */
    void printList(Node head) {
        while (head != null) {
            System.out.print(head.data + " ");
            head = head.next;

        }

        System.out.println("");

    }


    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.head = new Node(12);
        list.head.next = new Node(15);
        list.head.next.next = new Node(10);
        list.head.next.next.next = new Node(11);
        list.head.next.next.next.next = new Node(5);
        list.head.next.next.next.next.next = new Node(6);
        list.head.next.next.next.next.next.next = new Node(2);
        list.head.next.next.next.next.next.next.next = new Node(3);

        System.out.println("Given Linked List :");
        list.printList(head);
        System.out.println("");

        // Let us delete the node with value 10
```

```
        System.out.println("Deleting node :" + head.next.next.data);
        list.deleteNode(head, head.next.next);

        System.out.println("Modified Linked list :");
        list.printList(head);
        System.out.println("");

        // Lets delete the first node
        System.out.println("Deleting first Node");
        list.deleteNode(head, head);
        System.out.println("Modified Linked List");
        list.printList(head);
```

```
    }
```

```
}
```

Output:

```
Given Linked List: 12 15 10 11 5 6 2 3


Deleting node 10:

Modified Linked List: 12 15 11 5 6 2 3
```

Deleting first node

Modified Linked List: 15 11 5 6 2 3

CHAPTER 4 | NETWORKING

What is network programming?

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols –

1. **TCP** − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

2. **UDP** − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This chapter gives a good understanding on the following two subjects−

1. **Socket Programming** − This is the most widely used concept in Networking and it has been explained in very detail.
2. **URL Processing** − URL stands for Uniform Resource Locator and represents a resource on the World Wide Web, such as a Web page or FTP directory

Socket Programming:

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets −

The server instantiates a ServerSocket object, denoting which port number communication is to occur on.

The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.

After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.

The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.

On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.

ServerSocket Class Methods:

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests.

The ServerSocket class has four constructors –

Sr.No.	Method & Description
1	**public ServerSocket(int port) throws IOException** Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	**public ServerSocket(int port, int backlog) throws IOException** Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.
3	**public ServerSocket(int port, int backlog, InetAddress address) throws IOException** Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on.
4	**public ServerSocket() throws IOException** Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket.

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Following are some of the common methods of the ServerSocket class −

Sr.No.	Method & Description
1	**public int getLocalPort()** Returns the port that the server socket is listening on. This method is

1	useful if you passed in 0 as the port number in a constructor and let the server find a port for you.
2	**public Socket accept() throws IOException** Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the timeout value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely.
3	**public void setSoTimeout(int timeout)** Sets the timeout value for how long the server socket waits for a client during the accept().
4	**public void bind(SocketAddress host, int backlog)** Binds the socket to the specified server and port in the SocketAddress object. Use this method if you have instantiated the ServerSocket using the no-argument constructor.

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and the server, and communication can begin.

Socket Class Methods: The **java.net.Socket** class represents the socket that both the client and the server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.

The Socket class has five constructors that a client uses to connect to a server –

Sr.No.	Method & Description
1	**public Socket(String host, int port) throws UnknownHostException, IOException.** This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
2	**public Socket(InetAddress host, int port) throws IOException This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.**
3	**public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.** Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4	**public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String.
5	**public Socket()** Creates an unconnected socket. Use the connect() method to connect this socket to a server.

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port. Some methods of interest in the Socket class are listed here. Notice that both the client and the server have a Socket object, so these methods can be invoked by both the client and the server.

Sr.No.	Method & Description
1	**public void connect(SocketAddress host, int timeout) throws IOException This method connects the socket to the specified host. This method is needed only when you instantiate the Socket using the no-argument constructor.**
2	**public InetAddress getInetAddress() This method returns the address of the other computer that this socket is connected to.**
3	**public int getPort() Returns the port the socket is bound to on the remote machine.**
4	**public int getLocalPort() Returns the port the socket is bound to on the local machine.**
5	**public SocketAddress getRemoteSocketAddress() Returns the address of the remote socket.**
6	**public InputStream getInputStream() throws IOException Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.**
7	**public OutputStream getOutputStream() throws IOException Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket.**
8	**public void close() throws IOException Closes the socket, which makes this Socket object no longer capable of connecting again to any server.**

InetAddress Class Methods: This class represents an Internet Protocol (IP) address. Here are following usefull methods which you would need while doing socket programming –

Sr.No.Method & Description	
1	**static InetAddress getByAddress(byte[] addr) Returns an InetAddress object given the raw IP address.**
2	**static InetAddress getByAddress(String host, byte[] addr) Creates an InetAddress based on the provided host name and IP address.**
3	**static InetAddress getByName(String host) Determines the IP address of a host, given the host's name.**
4	**String getHostAddress() Returns the IP address string in textual presentation.**
5	**String getHostName() Gets the host name for this IP address.**
6	**static InetAddress InetAddress getLocalHost() Returns the local host.**
7	**String toString() Converts this IP address to a String.**

Socket Client Example The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a

response.

Example // File Name GreetingClient.java import java.net.*; import java.io.*;

```java
public class GreetingClient {

   public static void main(String [] args) {

      String serverName = args[0]; int port = Integer.parseInt(args[1]);
   try {

    System.out.println("Connecting to " + serverName + " on port " + port);
Socket client = new Socket(serverName, port); System.out.println("Just
connected to " + client.getRemoteSocketAddress());
OutputStream outToServer = client.getOutputStream(); DataOutputStream out =
new DataOutputStream(outToServer);
         out.writeUTF("Hello from " + client.getLocalSocketAddress());

InputStream inFromServer = client.getInputStream(); DataInputStream in = new
DataInputStream(inFromServer);
         System.out.println("Server says " + in.readUTF()); client.close();
}catch(IOException e) {
         e.printStackTrace(); }
   }

}
```

Socket Server Example The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument −

Example: // File Name GreetingServer.java import java.net.*; import java.io.*;

```java
public class GreetingServer extends Thread {
   private ServerSocket serverSocket;
   public GreetingServer(int port) throws IOException {
      serverSocket = new ServerSocket(port); serverSocket.setSoTimeout(10000);
}

   public void run() {
      while(true) {
         try {
            System.out.println("Waiting for client on port " +
               serverSocket.getLocalPort() + "..."); Socket server =
serverSocket.accept();
            System.out.println("Just connected to " +
server.getRemoteSocketAddress()); DataInputStream in = new
DataInputStream(server.getInputStream());
            System.out.println(in.readUTF()); DataOutputStream out = new
DataOutputStream(server.getOutputStream()); out.writeUTF("Thank you for
connecting to " + server.getLocalSocketAddress() + "\nGoodbye!");
server.close();
         }catch(SocketTimeoutException s) {
            System.out.println("Socket timed out!"); break; }catch(IOException e) {
            e.printStackTrace(); break; }
      }
   }

   public static void main(String [] args) {
      int port = Integer.parseInt(args[0]); try {
         Thread t = new GreetingServer(port); t.start(); }catch(IOException e) {
         e.printStackTrace(); }
   }
```

<div align="center">}</div>

Compile the client and the server and then start the server as follows −

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

Check the client program as follows −

Output:

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

URL Processing: URL stands for Uniform Resource Locator and represents a resource on the World Wide Web, such as a Web page or FTP directory.

This section shows you how to write Java programs that communicate with a URL. A URL can be broken down into parts, as follows –

protocol://host:port/path?query#ref

Examples of protocols include HTTP, HTTPS, FTP, and File. The path is also referred to as the filename, and the host is also called the authority.
The following is a URL to a web page whose protocol is HTTP –
https://www.amrood.com/index.htm?language=en#j2se

Notice that this URL does not specify a port, in which case the default port for the protocol is used. With HTTP, the default port is 80.

URL Class Methods The **java.net.URL** class represents a URL and has a complete set of methods to manipulate URL in Java.

The URL class has several constructors for creating URLs, including the following –

Sr.No.	Method & Description
1	**public URL(String protocol, String host, int port, String file) throws MalformedURLException Creates a URL by putting together the given parts.**
2	**public URL(String protocol, String host, String file) throws MalformedURLException Identical to the previous constructor, except that the default port for the given protocol is used.**
3	**public URL(String url) throws MalformedURLException Creates a URL from the given String.**
4	**public URL(URL context, String url) throws MalformedURLException Creates a URL by parsing together the URL and String arguments.**

The URL class contains many methods for accessing the various parts of the URL being represented. Some of the methods in the URL class include the following –

Sr.No.	Method & Description
1	**public String getPath() Returns the path of the URL.**
2	**public String getQuery() Returns the query part of the URL.**

3	**public String getAuthority() Returns the authority of the URL.**
4	**public int getPort() Returns the port of the URL.**
5	**public int getDefaultPort() Returns the default port for the protocol of the URL.**
6	**public String getProtocol() Returns the protocol of the URL.**
7	**public String getHost() Returns the host of the URL.**
8	**public String getHost() Returns the host of the URL.**
9	**public String getFile() Returns the filename of the URL.**
10	**public String getRef() Returns the reference part of the URL.**
11	**public URLConnection openConnection() throws IOException Opens a connection to the URL, allowing a client to communicate with the resource.**

Example:

The following URLDemo program demonstrates the various parts of a URL. A URL is entered on the command line, and the URLDemo program outputs each part of the given URL.

```java
// File Name : URLDemo.java import java.net.*; import java.io.*;
public class URLDemo {

  public static void main(String [] args) {
    try {
      URL url = new URL("https://www.amrood.com/index.htm?language=en#j2se");
      System.out.println("URL is " + url.toString());
System.out.println("protocol is " + url.getProtocol());
System.out.println("authority is " + url.getAuthority()); System.out.println("file
name is " + url.getFile()); System.out.println("host is " + url.getHost());
System.out.println("path is " + url.getPath()); System.out.println("port is " +
url.getPort()); System.out.println("default port is " + url.getDefaultPort());
System.out.println("query is " + url.getQuery()); System.out.println("ref is " +
url.getRef()); }catch(IOException e) {
      e.printStackTrace(); }
  }

                                      }
```

**Output URL is https://www.amrood.com/index.htm?language=en#j2se
protocol is http authority is www.amrood.com file name is /index.htm?
language=en host is www.amrood.com path is /index.htm port is -1**
default port is 80
query is language=en ref is j2se

URLConnections Class Methods: The openConnection() method returns a **java.net.URLConnection**, an abstract class whose subclasses represent the various types of URL connections.

For example:

If you connect to a URL whose protocol is HTTP, the openConnection() method returns an HttpURLConnection object.

If you connect to a URL that represents a JAR file, the openConnection() method returns a JarURLConnection object, *etc.*

The URLConnection class has many methods for setting or determining information about the connection, including the following:

Sr.No.	Method & Description
1	**Object getContent()** Retrieves the contents of this URL connection.
2	**Object getContent(Class[] classes) Retrieves the contents of this URL connection.**
3	**String getContentEncoding() Returns the value of the content-encoding header field.**
4	**int getContentLength()** Returns the value of the content-length header field.
5	**String getContentType()** Returns the value of the content-type header field.

6	**int getLastModified()** Returns the value of the last-modified header field.
7	**long getExpiration()** Returns the value of the expired header field.
8	**long getIfModifiedSince() Returns the value of this object's ifModifiedSince field.**
9	**public void setDoInput(boolean input) Passes in true to denote that the connection will be used for input. The default value is true because clients typically read from a URLConnection.**
10	**public void setDoOutput(boolean output) Passes in true to denote that the connection will be used for output. The default value is false because many types of URLs do not support being written to.**
11	**public InputStream getInputStream() throws IOException Returns the input stream of the URL connection for reading from the resource.**
12	**public OutputStream getOutputStream() throws IOException Returns the output stream of the URL connection for writing to the resource.**
13	**public URL getURL()** Returns the URL that this URLConnection object is connected to.

Example: The following URLConnectionDemo program connects to a URL entered from the command line.

If the URL represents an HTTP resource, the connection is cast to HttpURLConnection, and the data in the resource is read one line at a time.

```java
// File Name : URLConnDemo.java import java.net.*; import java.io.*;
public class URLConnDemo {

   public static void main(String [] args) {
      try {
         URL url = new URL("https://www.amrood.com"); URLConnection
urlConnection = url.openConnection(); HttpURLConnection connection = null;
if(urlConnection instanceof HttpURLConnection) {
            connection = (HttpURLConnection) urlConnection; }else {
            System.out.println("Please enter an HTTP URL."); return; }

         BufferedReader in = new BufferedReader(
            new InputStreamReader(connection.getInputStream())); String urlString
= ""; String current;
         while((current = in.readLine()) != null) {
            urlString += current; }
         System.out.println(urlString); }catch(IOException e) {
         e.printStackTrace(); }
   }

}
```

Output: $ java URLConnDemo

.....a complete HTML content of home page of amrood.com.....

Learning To Copy & Paste

"A good artist creates, a great artist steals." There are code examples for you to copy and paste into your project. The content of this book and all downloadable content are no different. This means that you'll have to understand what the code is doing to interpret it to fit your needs.

Every programmer does this habitually.

Copy/Paste

It is important to learn how to do this since it is something that you actually need to do in many cases not only to learn but also to get any unfamiliar task done. Programming is a constant learning process. It is a language to command computers.

Anytime you learn a new language, there will be plenty of words which you'll have to look up.

Add to this the fact that every programmer gets to make up new words, and you've got a language that you'll always need a dictionary for.

When some code is shown, you'll be expected to copy that code into your project. With your fingers ready on the keyboard, you'll want to get in the habit of typing.

There is a reason why programmers are usually fast typists. This is also cause for programmers to be picky about what keyboard they prefer. In most cases, the projects in this book will be in some state where you can read text that is already in place.

Most of the projects are in a more complete state, where you'll be able to run them and see an intended result. As a programmer I've gotten used to searching the Internet for example code.

Once I've discovered something that looks useful, I copy that code and paste it into a simple test case. After I've wrapped my head around what the code is doing, I rewrite it in a form that is more suited for my specific case.

Even if the code involves some fun trick, I'll learn from that code.

As I learn new tricks, I grow as a programmer. The only way to learn new tricks is to find the necessity to solve a new problem for which I haven't already figured out a solution. Finding solutions to problems is a fundamental part of being a programmer.

1. ALGORITHMS

This is classic computer science right here. Programming without knowledge of algorithms is like carpentry with just one kind of saw: you can get the job done, but it's going to take a lot longer.

You can look at an [algorithm](#) as "discipline". When you learn to write them, what you're doing is solving a problem with discipline; using structure, patterns, and logical steps.

When you don't know how to discipline your mind, you don't know how to write algorithms.

Not only can you not write algorithms unless you've studied them, you don't know how many algorithms others have written, too.

I spent four days trying to figure out how to do a permutation. I was so proud of myself when I figured it out. Right up until I discovered that [B.R. Heaps had figured it out in 1963](#).

2. DESIGN PATTERNS

This comes with education and/or experience. There's more than one way to structure your code, and there's a right time and a wrong time for each. You either need to make mistakes along the way and learn when to use each pattern, or learn from someone else who's already made the mistake (a teacher).

3. PROGRAMMING PARADIGMS

[Object-Oriented Programming](#) is not The Way. Neither is [Functional Programming](#). Nor [Reactive](#) [Programming. It is *A Way*](#).

There are different ways to program, and they each have a purpose. Not only that, some languages are naturally better-suited for one paradigm or another.

"If all you have is a hammer, everything looks like a nail."

Take that into self-taught programming and you'll find yourself hammering in nails, screws, staples, and thumb tacks.

I remember a self-taught .NET programmer actually telling me once, "well, it's not programming unless it's object-oriented. And that's why I don't consider JavaScript a programming language." That's a very, *very* flawed train of thought.

4. DATA STRUCTURES

Granted, your language can give you a basic idea of what the different data structures are. But again, that's a *basic idea*.

Self -taught programmers can have a tendency to only stick to data structures that work within their Favorite language. Just because it's not a primitive, or even a common structure in your language, that doesn't mean it can't exist. Of course, that also means that maybe it *shouldn't exist,* either.

The world is very small if it all can fit inside of an array.

5. TESTING

Maybe it's just me, but there's a lot of ways to test your code before pushing it to an environment. Learn how to do unit testing.

More importantly, learn *test-driven development*.

There's a difference between testing your code, and writing testable code.

SELF-TAUGHT PROGRAMMERS I HAVE INTERVIEWED OFTEN LACKED KNOWLEDGE IN THESE AREAS:

- FORMAL VOCABULARY. You have to know the correct names of [data structures](#) and other things by heart to by able to have an effective conversation related to a project in software development.

- TESTING. Most of the autodidacts lack knowledge or generally do not understand the importance of the testing process.

- PROGRAM PARADIGMS (and corresponding language). *"If you only have a hammer, every problem looks like a nail"*. I've had many of these hammer-types. They often just do not understand why not every problem can be solved the same way/with the same methodology.

- MACHINE RELATED MATHEMATICAL PROBLEMS. They lack numerical mathematics skills and do not understand why floating points arithmetic can fail you if you don't watch out

1. PRACTICE

Asides from following tutorials, you should work on your own projects.

"The most fundamental thing is that you actually go and code. I've heard it recommended that by the time you finish college, you should have written 100,000 lines at minimum."
— Andrei Thorp from Evernote.

But how do I get start, you may ask.

"I always tell people to find something they're doing more than once a week and to try to automate it. Ignore if anyone else has solved the problem before, and just make a tool/utility for yourself that fixes a common issue in your life."
— Kasra Rahjerdi, Mobile Lead at Stack Overflow.

"Like any other skill, it takes practice – deliberate practice, stepping outside of your comfort zone and learning the nuance and subtleties – that set apart great from good."
— Derick Bailey, the creator of WatchMeCode.net.

Derick is a top 0.42% StackOverflow user, and has also contributed to open source frameworks such as has MarionetteJS and BackboneJS.

It's OK to fail. Coding is all about failing and fixing things, and about learning how to do things better. If you don't build things and work on areas that you know you are weak on, you'll never get better.

If you ever need to receive advice on how to improve and what you're not doing so well on, feel free to ask an experienced developer to help your get straightened up by either reviewing your code or walking you through concepts you are having trouble understanding.

2. BE PETEINT

No-brainer here, but it's easy to get frustrated by your lack of progress and forget that you're not alone.

"Becoming a good programmer takes a long, long time and a lot of tedious evenings. Before you can write good code, you have to write hundreds of thousands of lines."

— Mike Arpaia, a former Etsy dev who now builds information security software for Facebook.

Mike stresses that beginners should give up on the assumption that one can become an excellent developer quickly.

But… what if you're not even past the tutorial stages yet? What if you're still banging your head against the wall and wondering perhaps you're just not cut out for programming? Before you leap to conclusions,

know that **everyone has a different learning style**. Author of the Ruby on Rails Tutorial, Michael Hartl, points out that beginners should try lots of different resources (books, videos, etc.) to see what 'clicks'.

In fact, Craig Coffman, the CTO of Reserve, has personally learned through a lot of trial-and-error and by picking projects that were personal and interesting. However, since all the interesting challenges are big ones, he suggests beginners to start with biting off reasonably-sized pieces.

That way, when you lose interest or get stuck, you still have a feeling of progress and accomplishment.

3. STAY INTERESTED

If you're bored by the project you're working on, you should probably reconsider any lofty goals of learning to code. Or, maybe you're just working on the wrong project or learning through the wrong resource. ***Always keep yourself motivated by working on personal projects that excites you.***

Coraline Ada Ehmke, founder of LGBTech and contributor to high-profile open source projects such as [Rails](#) and [RSpec](#), started coding at a young age out of interest. However, her first class in college as a Computer Science major made her doubt her passion.

> I remember our semester-long project was to write software for an ATM. I was so bored and not challenged, I decided that if that's what life as an engineer was like, I didn't want any part of it, so I dropped out soon after.

However, she continued to work on projects she found interesting. By 1993 she was online and building web sites, and has been developing web apps ever since.

4. LOVE THE ERROR

As a beginner you'll likely be mired in bugs. If you feel intimidated by all the red, you're not alone. Ross Chapman, a UX Engineer who coded for [Zendesk](#) and now working at [ScienceExchange](#), admits to being a scared developer when he first started out.

> I didn't have the patience because I wasn't ready to love the challenge of fixing things. But that's pretty much where all my really bad habits come from.

With that said, Ross urges beginners to embrace errors as crucial learning moments. Since you'll be debugging for life, you should get used to errors and learn to recognize the error messages.

> *"Being able to quickly parse and understand error messages will save you a lot of time and get you a long way. The fact you've tried will be very much appreciated by the person you're asking for help."*
> — Jack Franklin, author of "Beginning jQuery"

Jack also recommends beginners to make an attempt at fixing problems on their own at first. When hitting a wall when debugging, Lee Byron, co-contributor to React, personally attempts to understand what's going on by making ample usage of the debugger tools.

Once I understand exactly what is happening – step by step, then I can compare that to what I expected to happen and isolate the surprising parts and see where my assumptions were wrong or how some code led to the surprising situation.

Errors aren't limited to bugs, however. Sometimes, you make bad decisions such as using the wrong data structure. According to Mike, getting burned by those bad decisions will eventually help you learn when it's reasonable to use certain design patterns

5. UNDSERSTAND HOW THINGS WORK

> *"No matter what level you're at I'll say this: never ever write a line of code without knowing why it works, to the metal. Like, be obsessively curious. Be the Indiana Jones of source. Curiosity is one constant among engineers. I don't think you could make it in this biz without looking into the monitor with wonder. Both childlike, and ruthlessly academic."*
> — Ross Chapman

Suffice to say, interest is not enough. You have to **strive** to understand how things work if you're aiming to become a professional developer of some decency.

> *"You can start out understanding the tools you use by sifting through StackOverflow questions. I've learned a thing or two from them. [The top AngularJS questions are] really interesting to read through as Angular is such a big framework."*
>> — Todd Motto, an AngularJS conference speaker and Developer Expert at Google Rohan Singh, a senior infrastructure engineer at Spotify, stresses the importance of working towards understanding the layer one level of the stack **beneath** what you're working on right now. "Everything we do as software engineers involves working at some level of abstraction," Rohan says. In other words, if you use some sort of database, you can take away the internals of the database and expect it to "just work".

Furthermore, **to really understand how things work, you should be able to explain why certain technical choices are better than others, and be able to troubleshoot problems when things *don't* work the way the do.** Rohan achieves this in practice by trying, a little bit at a time, to learn about and understand the fundamentals of whatever platform or system he uses — whether that's Python or Go or the Linux operating system. According to him, this eventually helps you generate a mental of model of how things work under the covers, and broadens your base of understanding.

Ultimately, you'll grow as engineer, and as a bonus you'd be able to debug more efficiently by learning how to do more "lean back" debugging as opposed to "lean in" debugging. In other words, you'd lean back and think hard about how things work under the covers to figure out what the problem might be. "This can be a lot faster and involve a lot less flailing than 'lean in' debugging with an interactive debugger or other tools," Rohan says.

In fact, Andrei Thorp from Evernote thinks everyone should learn basic C early on.

Because it's minimal and doesn't do much for you, it forces you to understand how computers really work on a lower level. For example, C makes you manage the memory you use yourself – which means that later, when you use something like Python, you actually understand what Python is doing for you. Then, when you see some strange bug, you have this toolkit in your mind to understand what the problem could be.

6. KEEP LEARNING NEW THINGS

Nothing will kill your career/craft trajectory more than working at some shitty mundane programming job. Go somewhere where you are encouraged/forced to constantly learn new tricks, Says Jonathan Henson, who currently works at Amazon Web Services. Jonathan also tries to learn a new programming language, paradigm, or stack every year. He then puts himself on projects where he would have the opportunity to apply those skills.

"I think the most important skill to learn is meta-learning," says Kasra. "That's what separates engineers and programmers to me. There's something to be said about spending 12 weeks at a course learning one framework really well, but I really respect (and like to hire) devs that are able to learn whatever they need, on the spot, to do a task."

So, what's the best way to learn new skills?

Reading about what you want to do is a start. Steve Klabnik, who's a Rust core team member and ranked #37 on the all-time Rails contributors list, seeks out any established research on the topic and also tries to figure out how people who are good at the thing he wants to do achieve their results.

The most important thing is to just do it.

1. Try to do the thing, probably do it poorly.
2. Figure out where I'm going wrong, and what i need to improve 3. Work on what I've identified.
4. Repeat."

CTO of Bellhops, Adam Haney, says his favorite trick to learning new languages is to reimplement a previous project using the paradigms of that new language. For example, he would take something he wrote as object oriented code in C++ and then reimplement it in a functional language.

I feel like this kind practice has prepared me to evaluate new technologies because I understand the underlying Computer Science principles even if I don't know the intricate details of the language or framework.

If you struggle with memorization, Andrei recommends building a memory palace. The general idea is that you use your brain's powerful visual memory, and map that to more technical data, like numbers. He also strongly believe in techniques like The Seinfeld Calendar, which is based on Jerry Seinfeld's idea that you don't need to work hard every day — you only need to progress a little bit every day. "So with his calendar, you just check off whether you worked on the project today or not," Andrei says. "There are some nice apps that will help you with this.

On Android, I use HabitBull. As your streak gets longer, you feel more motivated to keep it running

7. LEARN HOW TO WORK WITH OTHERS

Another way you can learn new things is to work on projects with other people.

"The legend of the lone coder is a myth," Adam says. "Almost all substantial projects require teamwork." This means you'll need to learn the skills of breaking a problem down into multiple parts, build good interfaces between parts of the codebase, and collaborate on architecture.

"Working with a group of like-minded engineers who challenge you will definitely put you on the fast-track," Craig says. "Working in isolation makes it hard to catch yourself making silly choices and to learn new things."

Everyone makes mistakes – that's just how programming is. Beginners should strive to hang around great engineers and receive feedback. "Don't be sensitive about your mistakes," Jonathan stresses. "That's how you improve. Admit your mistakes and learn from them."

Getting your code reviewed will also force you into thinking about why you did something and understand code better. "My favorite engineers to work with are the ones who don't let you off the hook about the code you write," says Ross. "I remember when I first was challenged, and it freaked the shit out of me. But that night I went home and studied till I knew I could at least attempt a confident explanation of how to pass this around closures."

So, where do you find mentors or peers who can pair up with you and help you out?

According to Jack, local meetups often have sessions where free coaching is offered to anyone who would like it. Other free resources include Twitter groups, Slack groups, and iRC channels.

8. DON'T JUST CODE – BUILD SOLUTIONS

"A lot of programming isn't about code; it's about understanding other disciplines or standards.

You can't solve someone's problem with code if you don't understand their problem. *Working on projects exposed me to the way that small businesses, marketers, brokers and other professionals approach the world. When you understand how they currently solve problems, you can work with them to come up with new and better solutions."*

—Adam Haney, CTO of Bellhops One thing Ross wishes he had done earlier in his career was to better appreciate the discipline and history of

Software Engineering itself.

These days it's easy to dive into the vocation and only focus on the "coding." Especially as browser coders or web app coders in a booming market working with huge dynamic programming languages (Ruby, JavaScript) and vastly quirky "computer" languages and formats (HTML, CSS), we might be tempted to spend all our time racing to master the myriad tools, frameworks and APIs so we can crush interviews or level up at the job. But building a product on a team is always a social exercise, and a particular one at that with a unique set of challenges that are mostly non-technical. Like, turns out the hardest thing in software engineering is deciding what to build, not how; though maybe this is less true in as the JavaScript mycelium rhizomes dramatically.

Ross said it took him a while to understand that most of software engineering happens in your head first. "Coding will likely become the easy part soon. But a dope engineer can draw a solution with boxes, circles, and lines—and I know that's a learned skill because I've been doing it more and it I'm getting better at it." The realization that coding was much more "chin in hand and white boarding" was actually so resonant for him, he wrote a blog post about it.

"Remember that this is about human beings and our lives, not just about the technology that you're coding with," says Derick. "Learn how humans think, interact and deal with each other. Then represent what you've learned in your software architecture and design.

9. DON'T RE-INVENT THE WHEEL

Finally, no matter how good you get, your code will never be 100% original, as many problems have been solved in your language of choice already. "Absolutely ain't no shame in keeping the wheel as is," Ross assures. "However, when it's time to commit, you better be damn sure you can defend that code to your team with Dwayne Johnson-like charm and confidence."

"Don't re-invent the wheel just because you don't understand an abstraction," Mike reminds us.

However, this isn't to say there is absolutely no value to re-inventing the wheel. Matthew Zeiler, CEO of Clarifai, encourages people to build things that already have existing solutions if that's what interests them. Building a tool from scratch will help you learn more about software engineering, system design, scalability, and more.

Conclusion

Frustrations abound when learning how to develop apps or projects, but hopefully the tips above made you feel more confident in your quest to become a developer.

#1.REGARDING CAREER

- Write code that can be read, understood, and manipulated by others. This allows you to hand-off and take on new challenges.

- If you horde your knowledge, you'll be the only care taker of it --- or in an engineering/business minded organization, a risk.

- Stop taking pride in code or hackatons survived, means nothing in a permanent team. Execution and collaboration will serve you greater.

#2.REGARDING SOLUTIONS AND CODING

- If you can't explain it to a non-programmer, you might be over-complicating or over-optimizing.

- If you can't draw a architecture diagram, you also might be over-complicating it.

- Don't show off by writing "compact code"

#3.REGARDING PERSONAL IMPROVEMENT

- Don't be a Java or C/C++ (or other) fan-boy/girl. You'll be learning 10+ languages in a long-term career. Treat them as tools, not bandwagons.

- There will always be a better programmer and they are hard to identify. Learn from them all.

- You are not defined by the quality of your code. Don't judge yourself that way.

#4.REGARDING PHILOSOPHY

- Programming is the art of enabling non-programmers to do more than what they can do alone.

- Computer science is a catalyst to nearly every field of study or industry in the world. CS enables humanity to do more, solve more, be better.

- Computer science != programming. If your college only taught you how to program, go ask for your money back.

DIFFERENCE BETWEEN A PROGRAMMER, A GOOD PROGRAMMER AND A GREAT PROGRAMMER.

- **Programmer**: anyone who can write working programs to solve problems, given a sufficiently detailed problem statement.

- **Good programmer**: a programmer who collaborates with others to create maintainable, elegant programs suitable for use by the customer, on time and with low defect rates, with little or no interpersonal drama.

- **Great programmer**: a good programmer who understands algorithms and architectures intuitively, can build self-consistent large systems with little supervision, can invent new algorithms, can refactor live systems without breaking them, can communicate effectively and cogently with non-technical staff on technical and non-technical issues, understands how to keep his or her ego in check, and can teach his or her skills to others.

The path of becoming a great programmer is to start by being a programmer, and then develop the skills needed to be a good programmer, then practice those skills until you master them, then develop the skills needed to be a great programmer, and then practice those skills until you master them.

The amount of time this takes depends on your personal skills, personality, and training. It also depends on the experience and opportunities that you have during your career, and how you react to them.

4 REASONS WHY YOUR PROGRAM CRASHES!

There may be 4 reasons why your program may crash:

- Your program may depend on some element of randomness: user input, randomly generated number, time, *etc.*

- If Your program is using an uninitialized variable, it could be accessing data it isn't supposed to (same with accessing something outside of an arrays indices)

- Your program may be using an external library that crashes all the time.

- Stack overflows!

Programming Quotes!

"Talk is cheap. Show me the code."
— **Linus Torvalds**

"Programs must be written for people to read, and only incidentally for machines to execute."
— **Harold Abelson**, **Structure and Interpretation of Computer Programs**

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."
— **Rick Cook**, **The Wizardry Compiled**

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live"
— **John Woods**

"That's the thing about people who think they hate computers. What they really hate is lousy programmers."
— **Larry Niven** "The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct."
— **Donald Ervin Knuth**, **Selected Papers on Computer Science**

"I'm not a great programmer; I'm just a good programmer with great habits."
— **Kent Beck**

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"
— **Brian W. Kernighan**

"A language that doesn't affect the way you think about programming is not worth knowing."
— **Alan J. Perlis**

"The computer programmer is a creator of universes for which he alone is the lawgiver. No playwright, no stage director, no emperor, however powerful, has ever exercised such absolute authority to arrange a stage or field of battle and to command such unswervingly dutiful actors or troops."
— **Joseph Weizenbaum**

"Walking on water and developing software from a specification are easy if both are frozen."
— **Edward Berard**

"Perl – The only language that looks the same before and after RSA encryption."
— **Keith Bostic**

"The most disastrous thing that you can ever learn is your first programming language."
— **Alan Kay**

"A computer is like a violin. You can imagine a novice trying first a phonograph and then a violin. The latter, he says, sounds terrible. That is the argument we have heard from our humanists and most of our computer scientists. Computer programs are good, they say, for particular purposes, but they aren't flexible. Neither is a violin, or a typewriter, until you learn how to use it."
— **Marvin Minsky** "The most important property of a program is whether it accomplishes the intention of its user."
— **C.A.R. Hoare**

"Object-oriented programming offers a sustainable way to write spaghetti code. It lets you accrete programs as a series of patches."
— **Paul Graham**, **Hackers & Painters: Big Ideas from the Computer Age**

"At forty, I was too old to work as a programmer myself anymore; writing code is a young person's job."
— **Michael Crichton**, **Prey**

"Some of the best programming is done on paper, really. Putting it into the computer is just a minor detail."
— **Max Kanat-Alexander**, **Code Simplicity: The Fundamentals of Software**

"Programmers are not to be measured by their ingenuity and their logic but by the completeness of their case analysis."
— **Alan J. Perlis**

"Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code. "
— **Edsger W. Dijkstra**

"Don't gloss over a routine or piece of code involved in the bug because you "know" it works. Prove it. Prove it in this context, with this data, with these boundary conditions."
— **Andrew Hunt**, **The Pragmatic Programmer: From Journeyman to Master**

"Remember that code is really the language in which we ultimately express the requirements. We may create languages that are closer to the requirements. We may create tools that help us parse and assemble those requirements into formal structures. But we will never eliminate necessary precision—so there will always be code."
— **Robert C. Martin**

"Software testing is a sport like hunting, it's bughunting."
— **Amit Kalantri**

"Programming, it turns out, is hard. The fundamental rules are typically simple and clear. But programs built on top of these rules tend to become complex enough to introduce their own rules and complexity. You're building your own maze, in a way, and you might just get lost in it."
— **Marijn Haverbeke**

Thank You!