JAVA CODING INTERVIEW

CODING INTERVIEW SOLUTION

Mr KOTIYANA

PREFACE

Java Coding Interview is here to help you through the INTERVIEW process, teaching you what you need to know and enabling you to perform at your very best. I've coached and interviewed hundreds of software engineers. The result is this book. These interview questions are real; they are not

pulled out of computer science textbooks. They reflect what's truly being asked at the top companies, so that you can be as prepared as possible. Cracking the Coding Interview makes a lot easier! it gives you the interview preparation you need to get the top software developer jobs. We are also sharing 20 java interview Programming questions; these questions are frequently asked

by the recruiters.

WHAT'S INSIDE?

- Programming Basics

-20 programming interview questions, ranging from the basics to the trickiest algorithm problems.

-Steps required to preparing for an interview at big companies like Google, Apple or Microsoft.

-Skills you must have to

become professional programmer.
-Important data structures and algorithms required for the interview.
-Learn how to become a great programmer!
-Coding interview tips.
-Programming Quotes!

ISBN: 9781520689197

Table of Contents

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 incohesive. 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.

STEPS TO PREPARE FOR A MICROSOFT, AMAZON, GOOGLE OR APPLE INTERVIEW.

Software engineer interview at any of these companies are quite standard and general, you can expect to have the similar kind of interviews as other big companies like Amazon, Facebook etc... Various skills are evaluated including general technical skill (data structure/algorithm), system design, testing, communication, analysis ability etc. and since the whole process is quite standard, certain ways of preparation

can definitely make your life easier.

What you need to prepare is case by case and I'll try to give some general tips, which you should always try to adjust to make them work for you.

STEP 1: DATA STRUCTURE AND ALGORITHMS PREPARATION

I would assume you already finish those basic courses at school so that you are not learning everything from scratch. Then this process may take one to several months.

The reason you should prepare well for data structure and algorithms first is that they are really the foundation of most software engineer interviews. A real interview question is like asking you to solve a problem with combination of skills you learnt from these basic knowledge, also you should be quite fast when analyzing time/space complexity, which is covered in this book as well.

STEP 2: BE FAMILIAR WITH GENERAL CODING QUESTIONS

At this step, you should be quite familiar with basic knowledge and concepts of computer science, it's better to practice with some real coding questions. This may take several months as well depending on your time and how familiar you are with data structure and algorithms.

The idea of this step is to teach you how to use what you learnt from those books to solve a real question and give you some ideas about what kind of questions is asked in a general interview. In fact I don't have much suggestion in this step instead of delving into those questions

and practice as much as you can.

STEP 3: SEARCH REAL QUESTIONS FROM THE COMPANY

Suppose you are preparing interviews for Facebook, then I'd suggest you do some Google search and it won't be hard for you to get tons of questions from Facebook interviews.

Since different company has different styles and focuses, this approach will help you be more familiar and prepared for that company's interview. Don't try to memorize questions and answers as

these companies usually avoid asking questions leaked on public, so it's quite unlikely to encounter the same question again.

STEP 4: KEEP PRACTICING

At this step, I expect you to have maybe 1 month left and you should be equipped with all you need for an interview except experiences.

Technical interview doesn't only evaluate your coding ability, but a variety of skills and abilities like communication skills, analysis ability etc.. Also many people will feel nervous solving a problem when someone is looking over his shoulder, thus he may even fail in the simplest questions.

I'd suggest you to find a friend who is also preparing for an interview, you guys can conduct mock interviews with each other and try to be familiar with this kind of intense atmosphere.

Conclusion:

It's highly recommended to make a preparation timeline and stick to it. Also spending half hour a day on preparation won't work normally. There's no better way than keep practicing and eventually you're going to crack the interview.

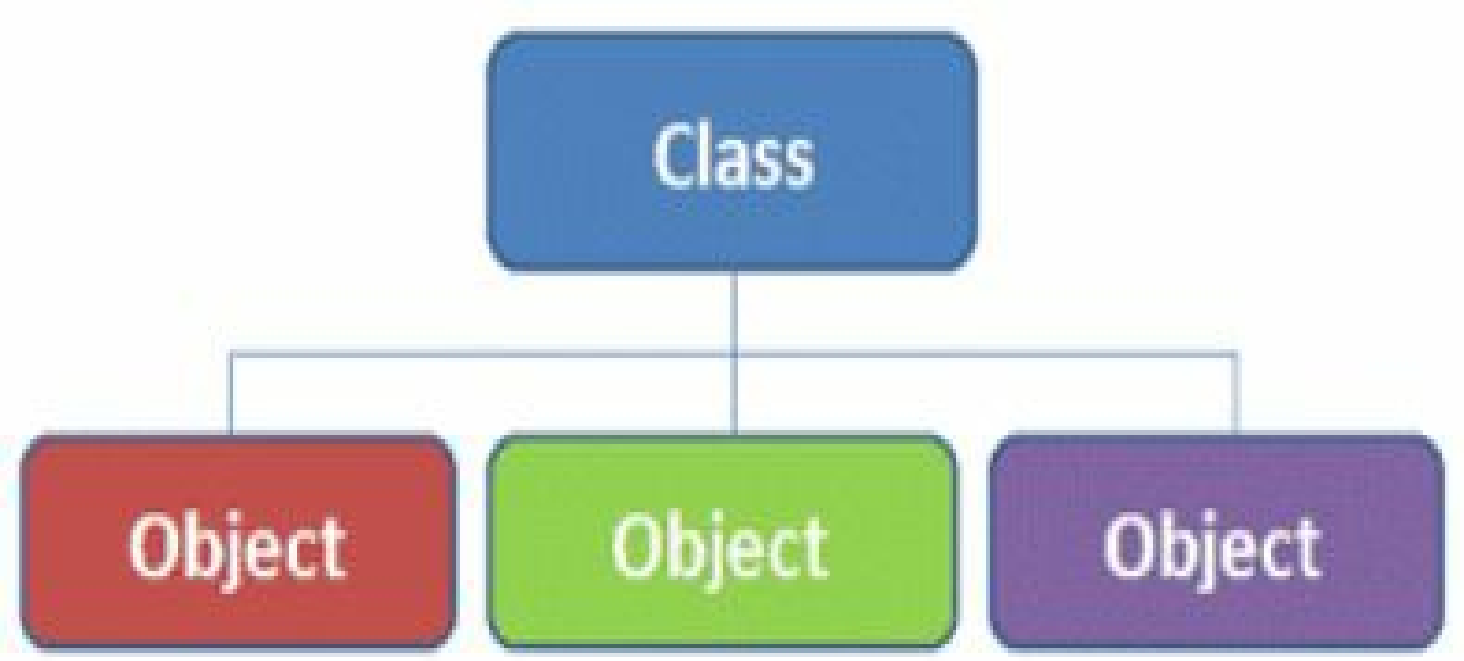
Chapter 2 | Programming Basics

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.

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.

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
{
}
```

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_hardToRemembe
```

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.

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
{
}
```

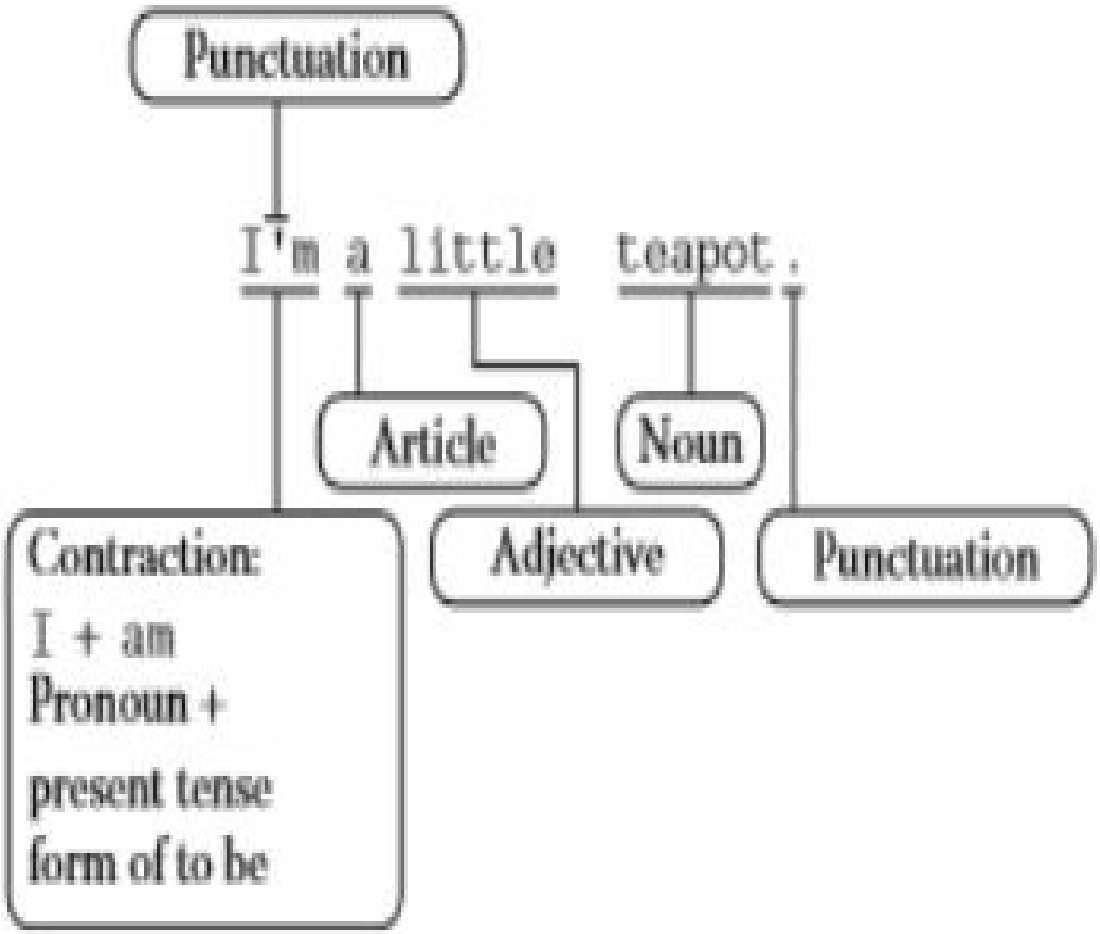
Every class needs a name; in the above example we named our class className, although we could have easily named the class Charles. When a new class is named the name becomes a new identifier. This also holds true for every variable's name, though scope limits how long and where the variable's identifier exists. We'll learn more about variables and scope soon. 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. Altogether in Java there are roughly 80 keywords you should be aware of.

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.

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.

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.

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.

```
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
```

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.

Class/Static Variables

Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor or a block.
There would only be one copy of each class variable per class, regardless of how many objects are created from it.
Static variables are rarely used other than being declared as constants.
Constants are variables that are declared as public/private, final and static.
Constant variables never change from

their initial value.
Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
Static variables are created when the program starts and destroyed when the program stops.
Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
Default values are same as instance variables.
For numbers, the default value is 0; for Booleans, it is false; and for
Object references, it is null. Values can be assigned during the declaration or

within the constructor. Additionally values can be assigned in special static initializer blocks.
Static variables can be accessed by calling with the class name. ClassName.VariableName.
When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

Example:

```
import java.io.*;
public class Employee{
// salary  variable is a private static
variable
private static double salary;
   // DEPARTMENT is a constant
public static final String DEPARTMENT
= "Development ";
   public static void main(String args[])
{
salary = 1000;
System.out.println(DEPARTMENT+"ave
```

```
salary:"+salary);
}
}
```

Output:

```
Development average salary:1000
```

Note: If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT

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.

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and insta
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 p
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 an
```

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

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

int a[]=
```

```
{33,3,4,5};//declaration, instantiation an

//printing array
for(int i=0;i<a.length;i++)//length is the
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.

```
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);
}

public static void main(String args[]){

int a[]={33,3,4,5};
min(a);//passing array to method

}}
```

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
```

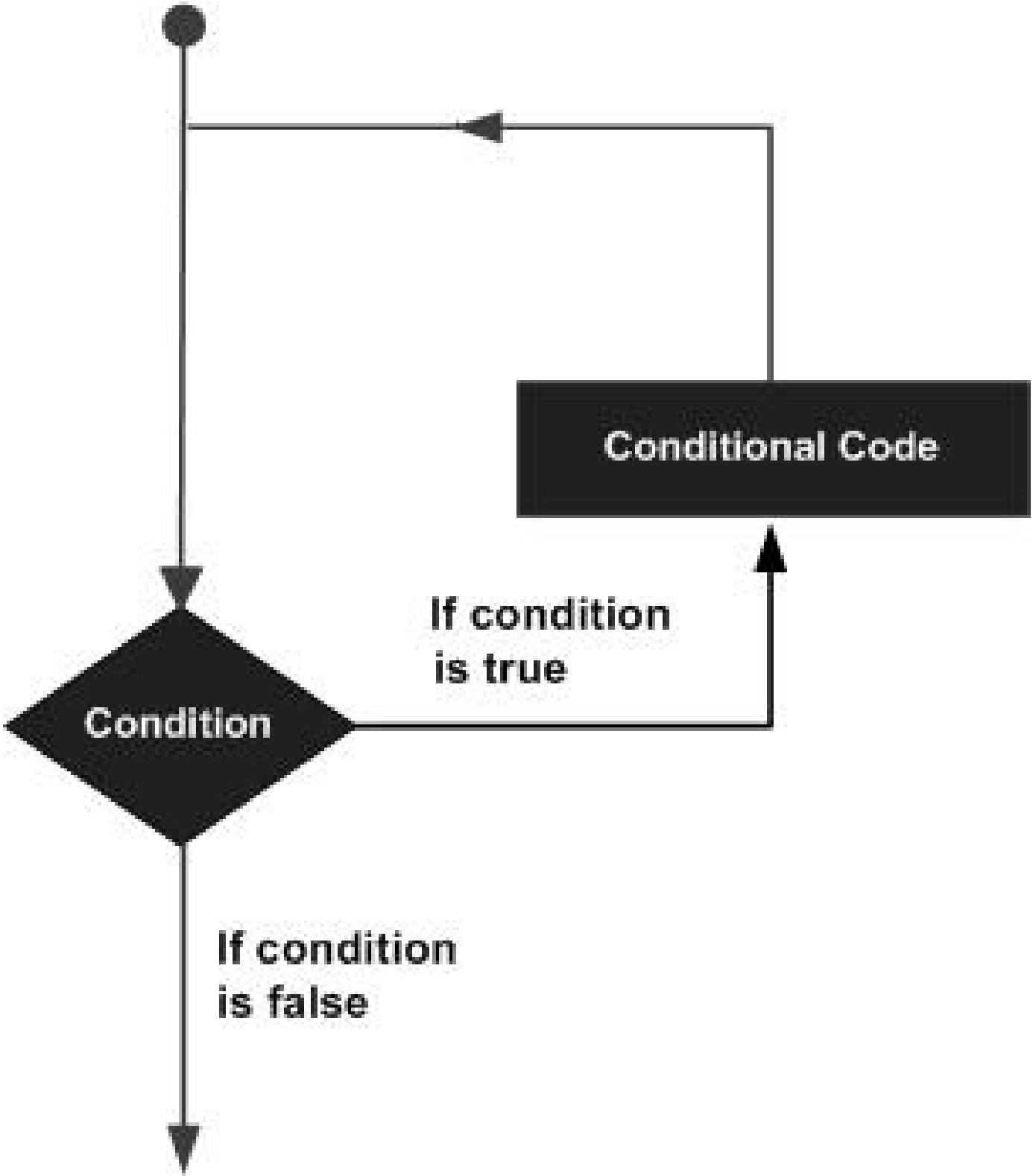
Loop

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
}
```

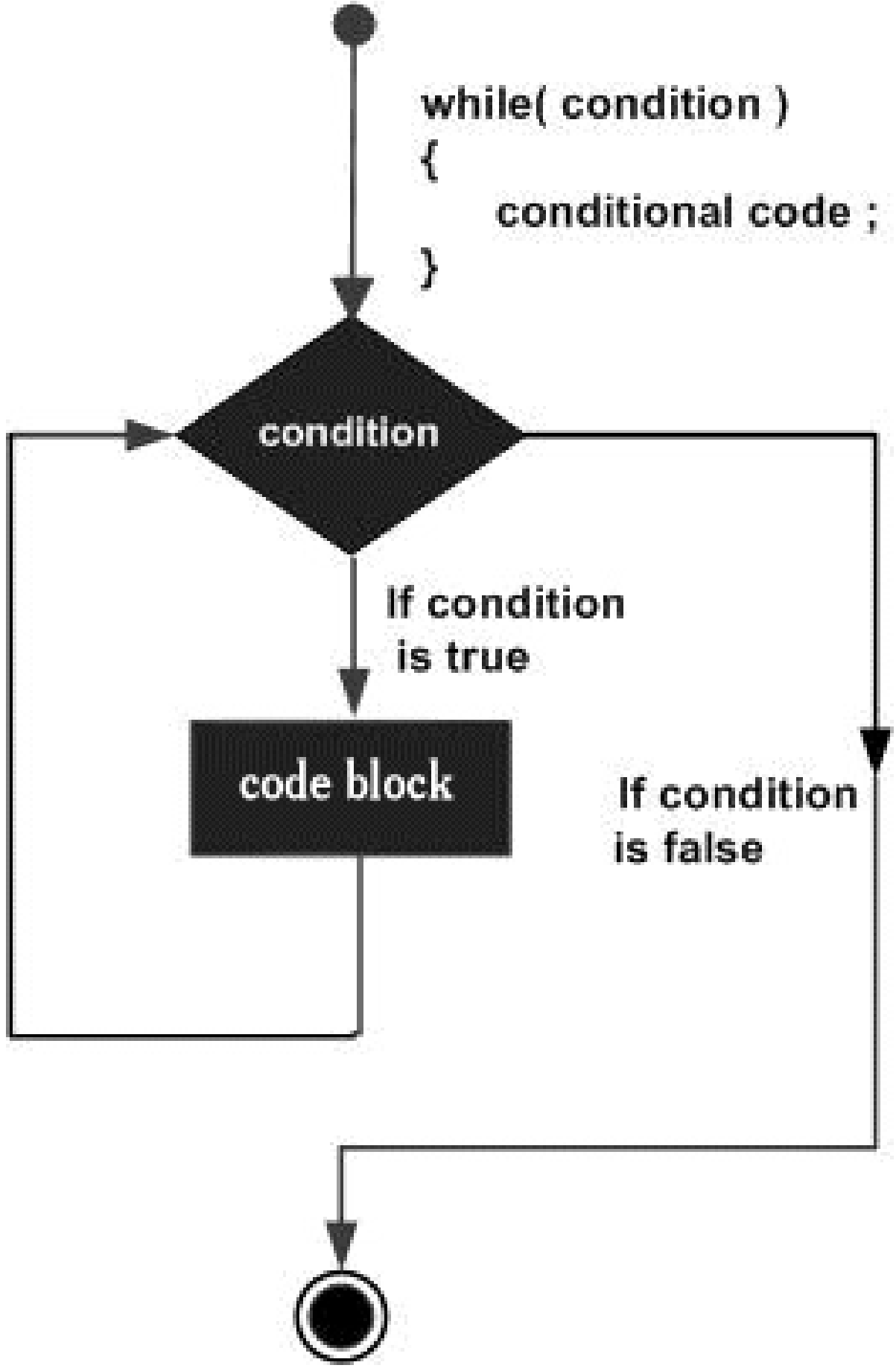
Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non zero value.

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:


while(condition)
{
conditional code ;
}
condition
If condition
is true
code block
If condition
is false

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:

```
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 –

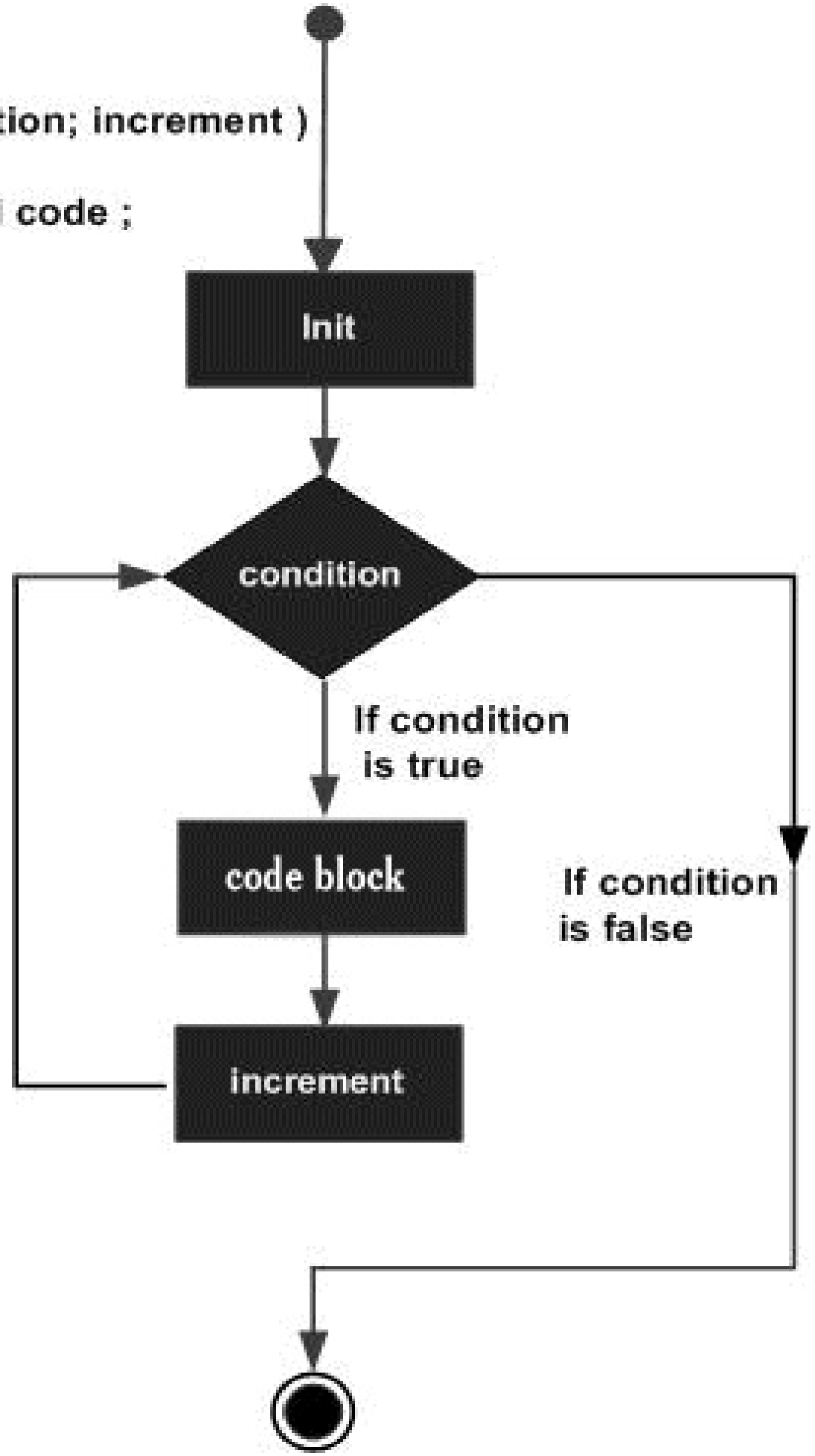
- The **initialization** step is executed first, and only once. This step allows you to declare and initialize any loop control variables and this step ends with a semi colon (;).
- Next, the **Boolean expression** is evaluated. If it is true, the body of the loop is executed. If it is false, the body 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:

```
for( init; condition; increment )
{
    conditional code ;
}
```



Example:

Following is an example code of the for loop in 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
```

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.

Java supports the following control statements.

- Break Statement
- Continue Statement

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

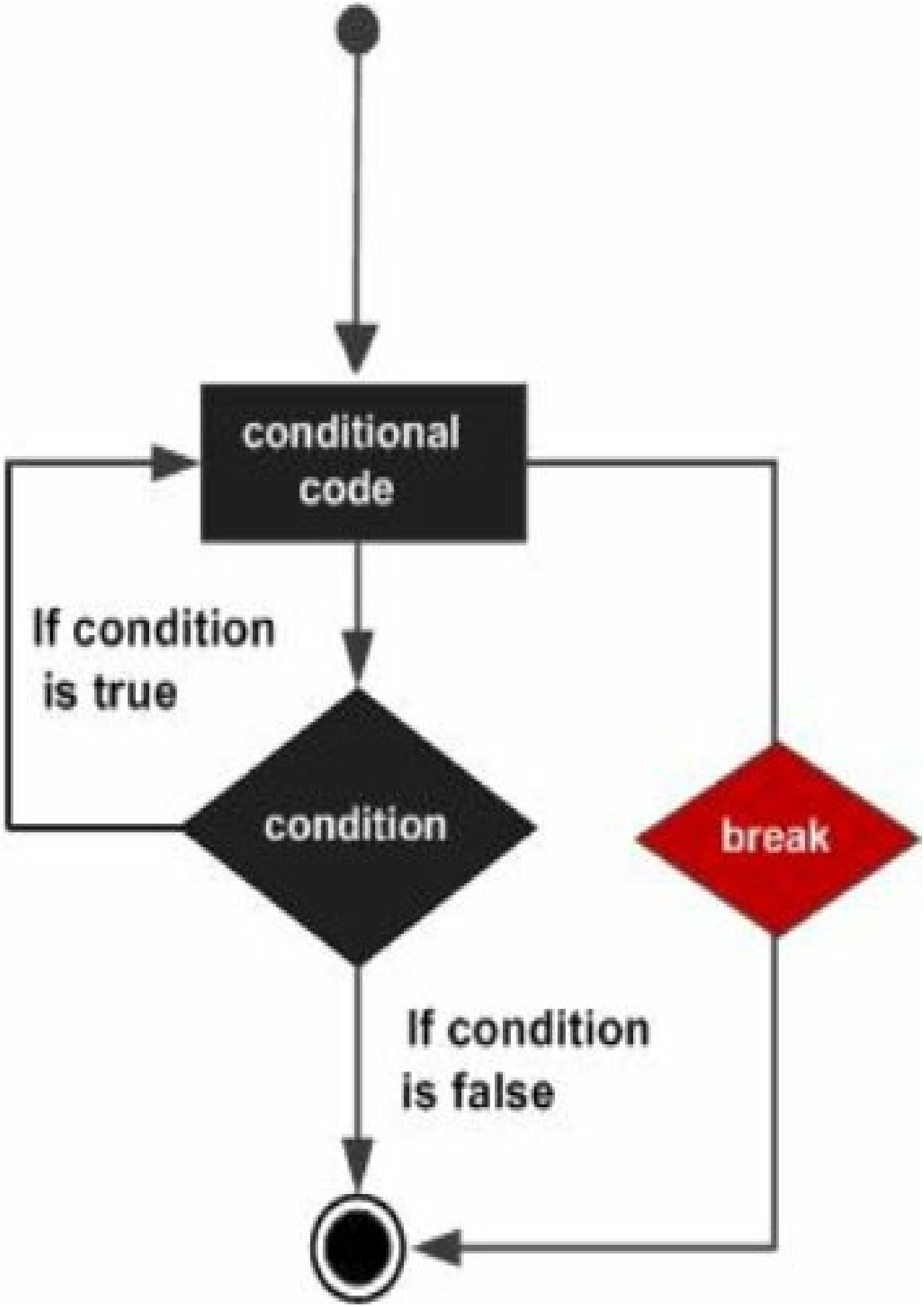
- When the **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

Syntax

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

```
break;
```

Flow Diagram:


conditional
code
If condition
is true
condition
break
If condition
is false

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.

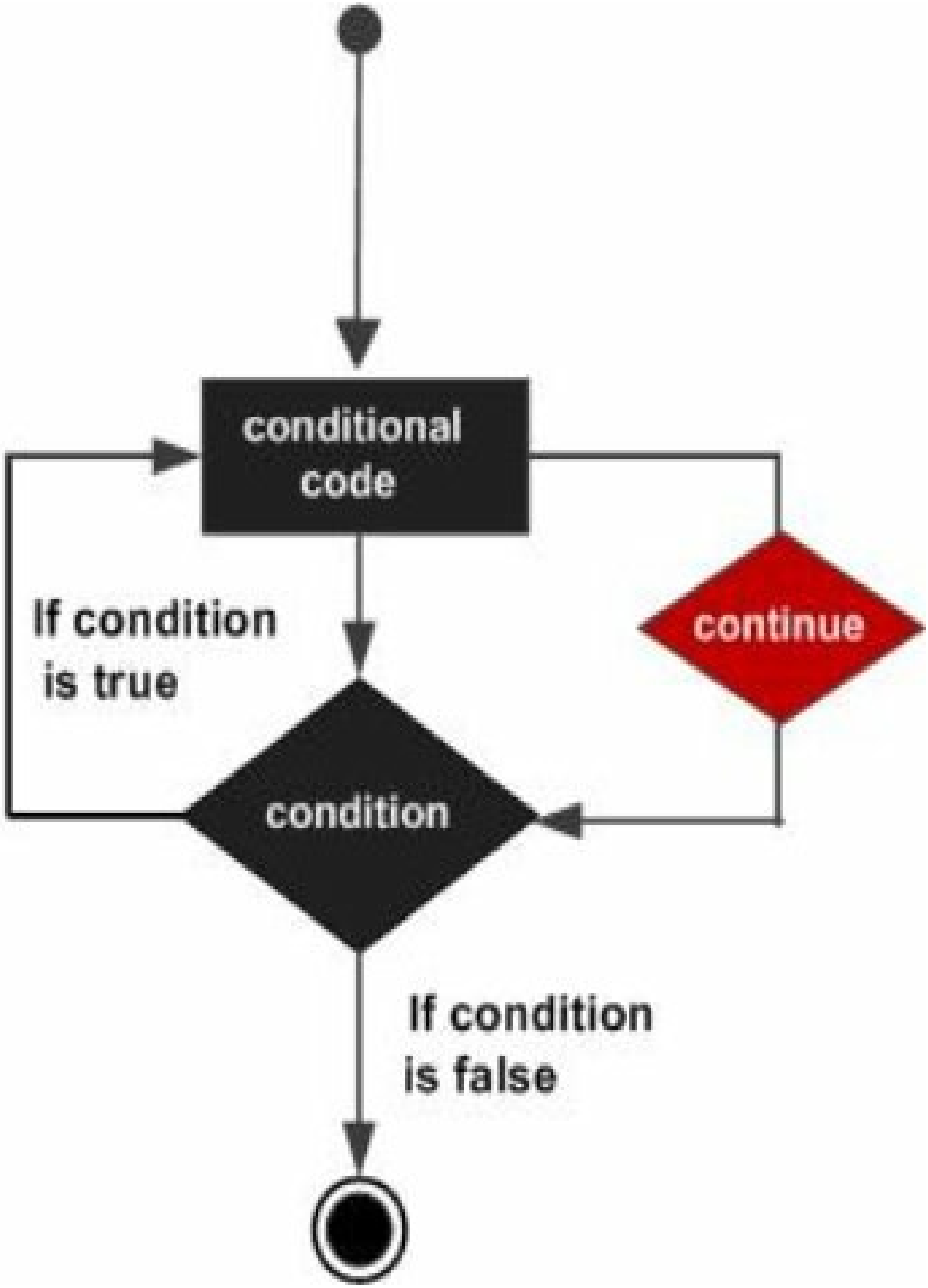
- In a for loop, the continue keyword causes control to immediately jump to the update statement.
- In a while loop or do/while loop, control immediately jumps to the Boolean expression.

Syntax

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

```
continue;
```

Flow Diagram:


conditional
code
If condition
is true
continue
condition
If condition
is false

Example:

```
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");
        }
    }
}
```

This will produce the following result :

Output

```
10
20
40
50
```

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. Even your morning routine could be

considered an algorithm! In fact, here's what your child's morning might look like written out as an algorithm:

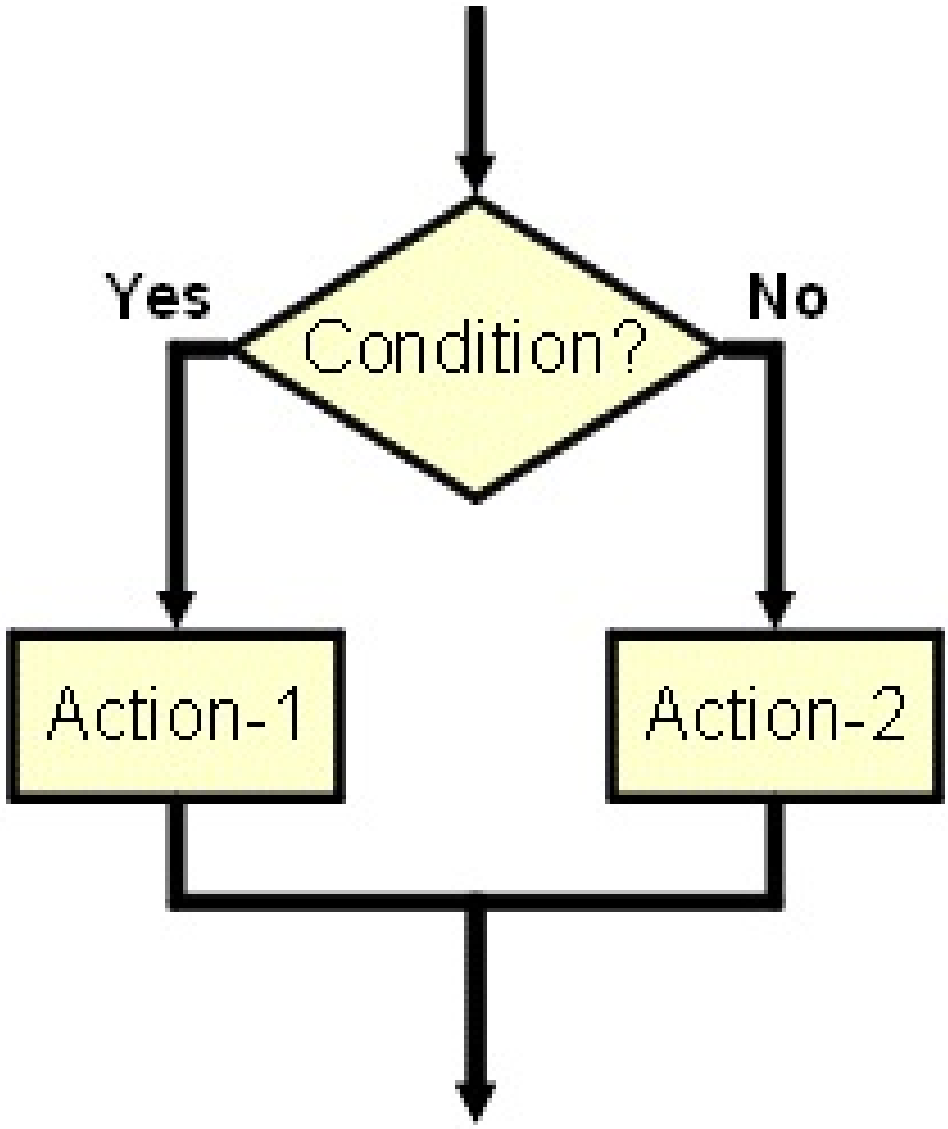

when Alarm Rings occurs
repeat 3
Hit Snooze Button
wait until Alarm Rings
Turn Off Alarm
Get Up
Shower
Get Dressed
Eat breakfast
if weather = sunny then
Walk to School
else
Ride to School with Parent

Statements and Expressions

When reading a book or story you extract meaning from an ordered chain of words.
In a similar way, computers extract commands from a chain of ordered instructions.
In English we call this a sentence; programmers call this a statement.

A statement is considered to be any chunk of code which accomplishes some sort of task separated by a semicolon. At the center of any given task is the

algorithm, not to be confused with a logarithm.
An algorithm is a systematic process that accomplishes something. In many ways you can think of it as a recipe, or rather a recipe is an algorithm for making food. It can take just one or two statements to accomplish a task, or it could take many hundreds of statements.


Yes
Condition?
No
Action-1
Action-2

This all depends on the difficulty of a given task. Each individual statement is a step toward a goal. This is the difference between spending a few minutes frying an egg, or spending an hour baking a souffle. Each step is usually fairly simple; it's the final result that matters.

Like sentences, statements have different forms. Statements can declare and assign values to variables. These are called declaration and assignment statements. These statements are used to set up various types of data and give them names.

Expressions:

The subjects of your statements are called identifiers. An assignment statement is used to give an identifier a value. When you read “Jack is a boy and Jill is a girl,” you’ve mentally assigned two subjects their genders.

In Java this might look more like:
gender Jack = male;
gender Jill = female;

Assignment statements often incorporate some sort of operation.
These are called expressive statements.
Different from expressing an emotion, expressions in code look more like “x +

y.” Expressions process data. After processing, the result of an expression can be assigned to a variable.

A collection of statements is called a code block, not like a roadblock which might stop your code, but more like a building block, anything that’s used to build.
When writing a story, we call a collection of sentences a paragraph. The statements in a block of code work with each other to accomplish a task.

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.

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.

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.

Chapter 3 | Data Structures & Interview Questions

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

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
{
    // 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

```
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;
```

}

```
    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

```
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;
                          }
```

```
            }
        }

    }

    public static void main(String[] args)
                                    int   arr[]  =
{3,60,35,2,45,320,5};

            System.out.println("Array Befo
             for(int i=0; i < arr.length; i++)
{
                    System.out.print(arr[i] + "
            }
            System.out.println();
```

```
            bubbleSort(arr);//sorting array

            System.out.println("Array Afte
            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

```
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];
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)
        {
            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 */
    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,
      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

```
public class SelectionSortExample {
    public static void selectionSort(int[] a
{
        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 low
```

```
            }
        }
        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 Selectio
    for(int i:arr1){
        System.out.print(i+" ");
    }
    System.out.println();

    selectionSort(arr1);//sorting array u
```

```
        System.out.println("After Selection
        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.

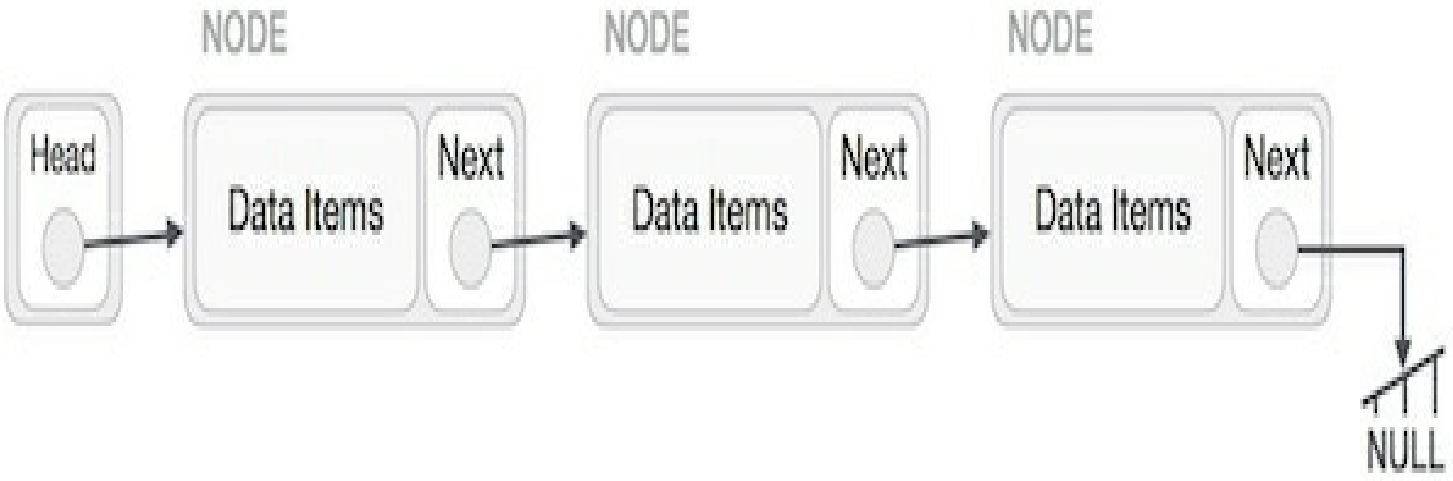
Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **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.

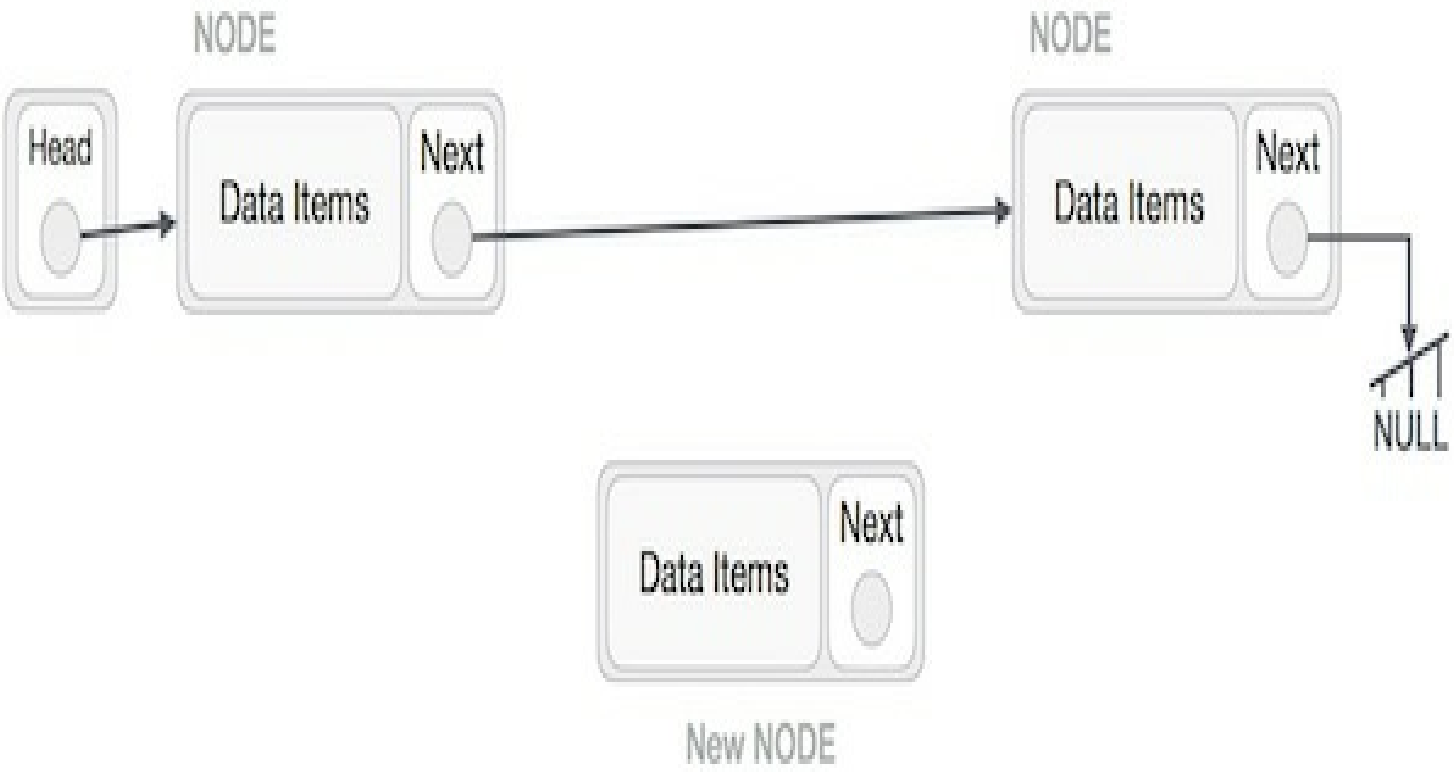
Basic Operations:

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.

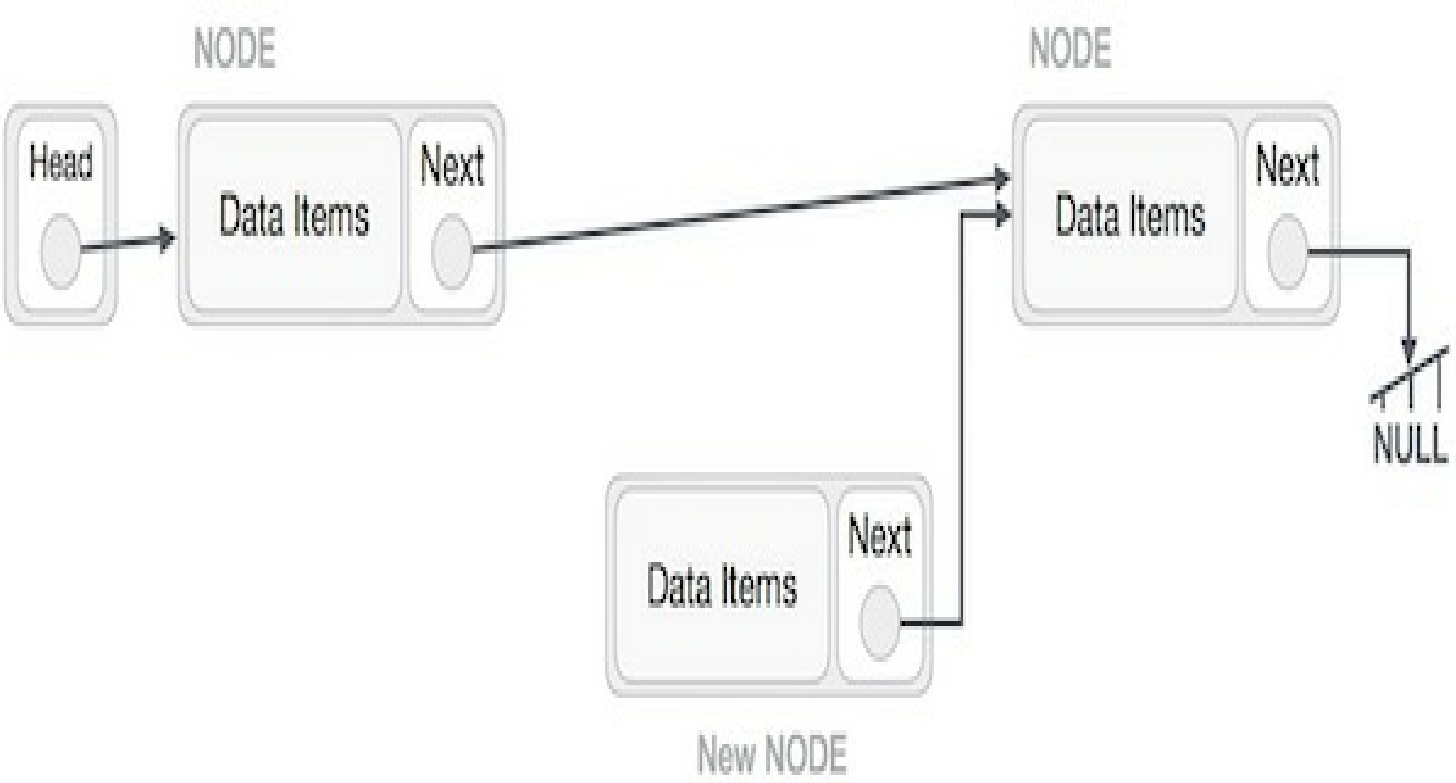


Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode)

and **C** (RightNode). Then point B.next to C −

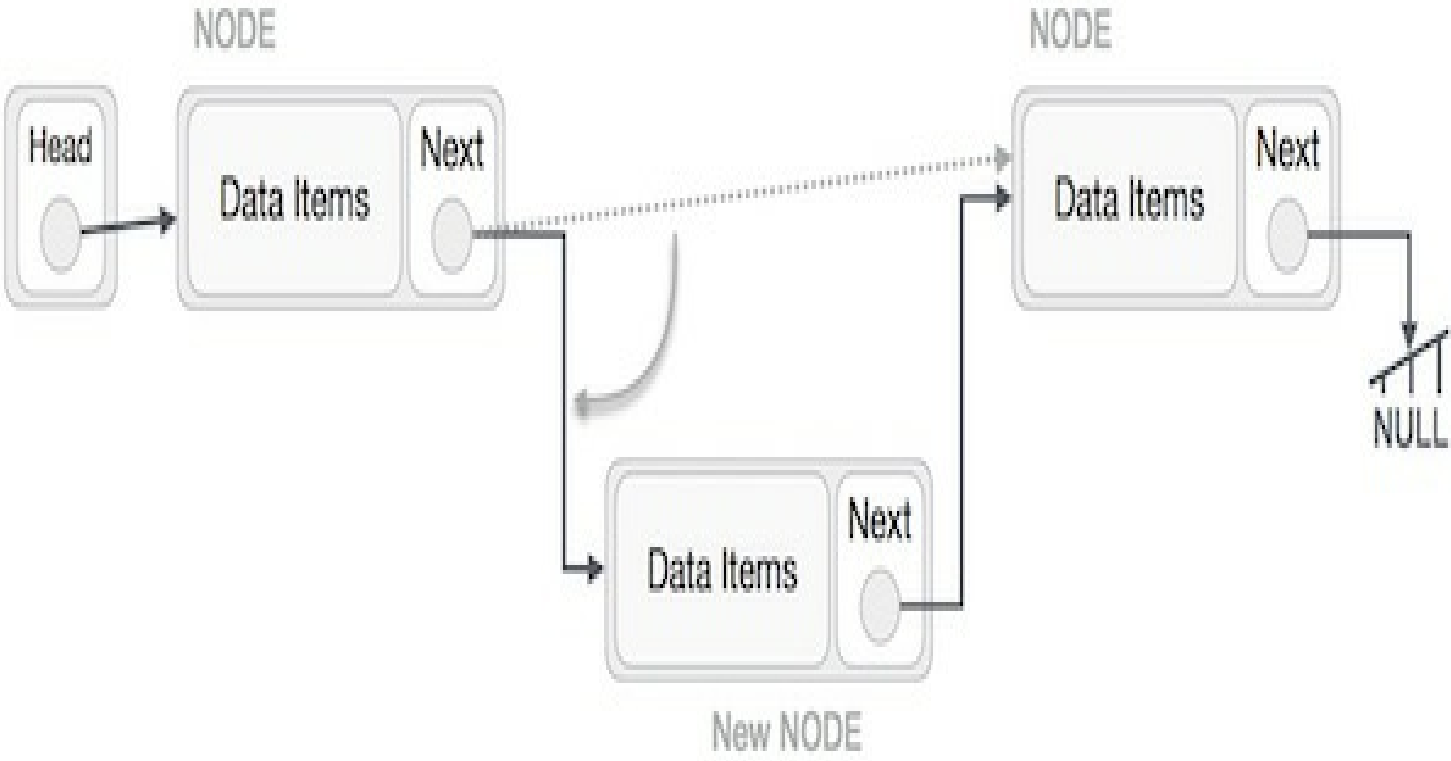
```
NewNode.next −> RightNode;
```

It should look like this −

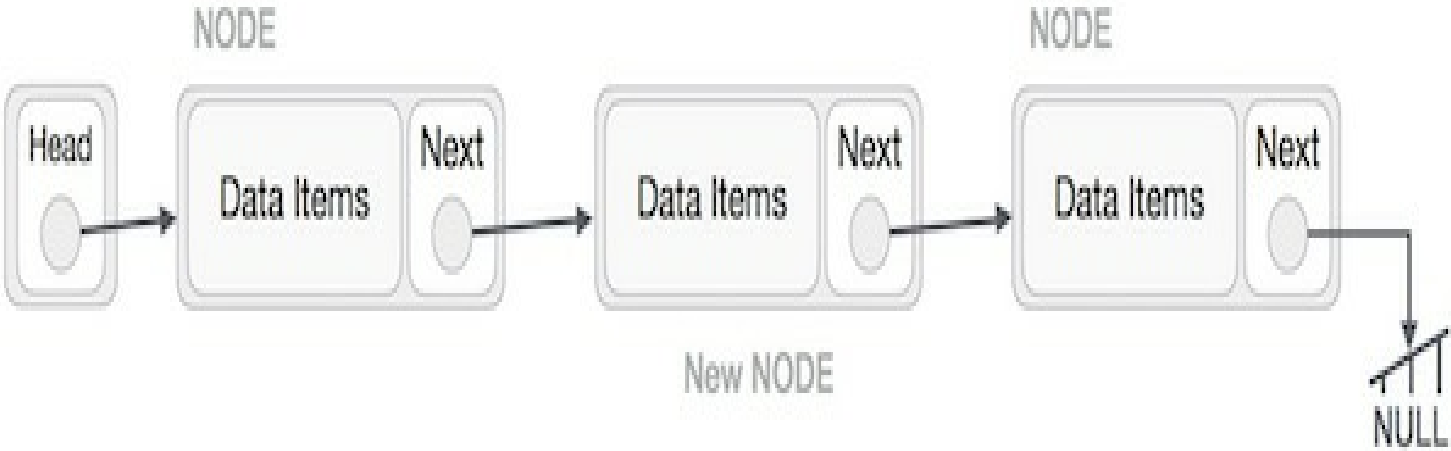


Now, the next node at the left should point to the new node.

```
LeftNode.next −> NewNode;
```



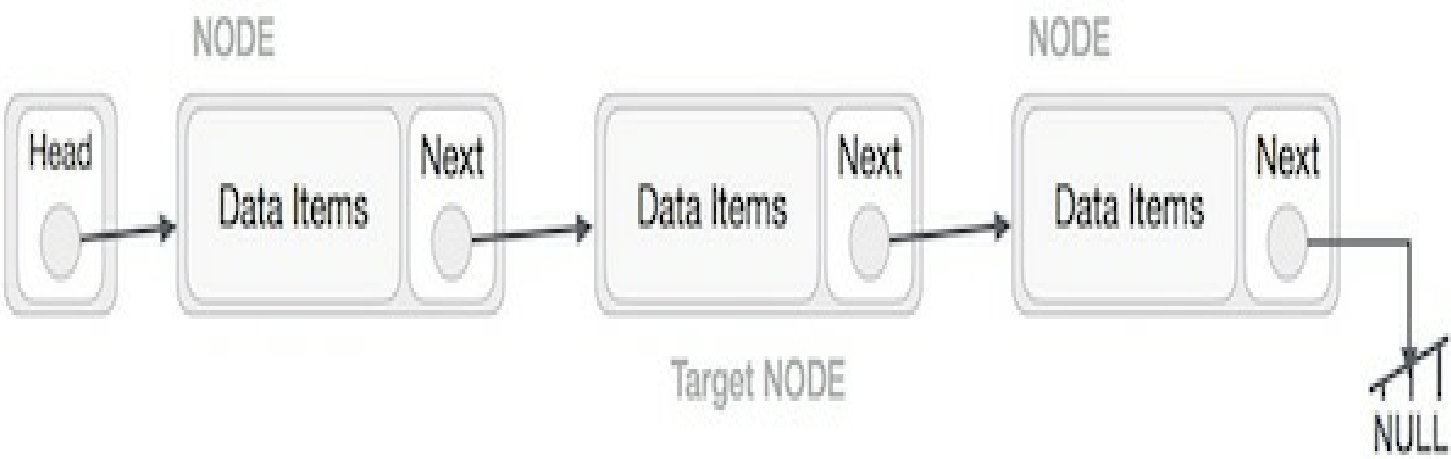
This will put the new node in the middle of the two. The new list should look like this −



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.

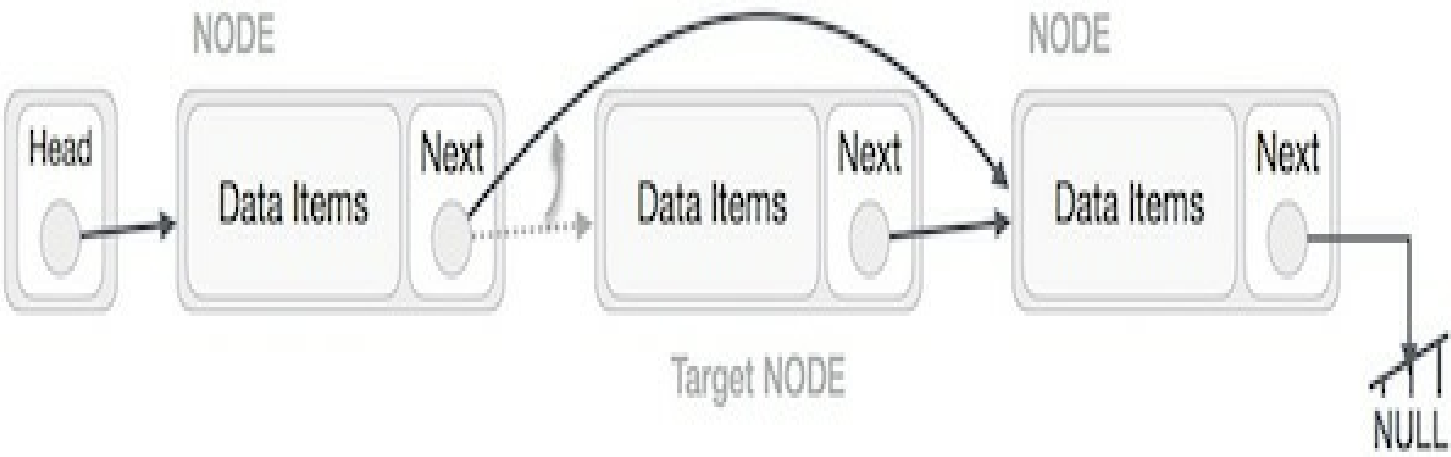
Deletion Operation:

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.



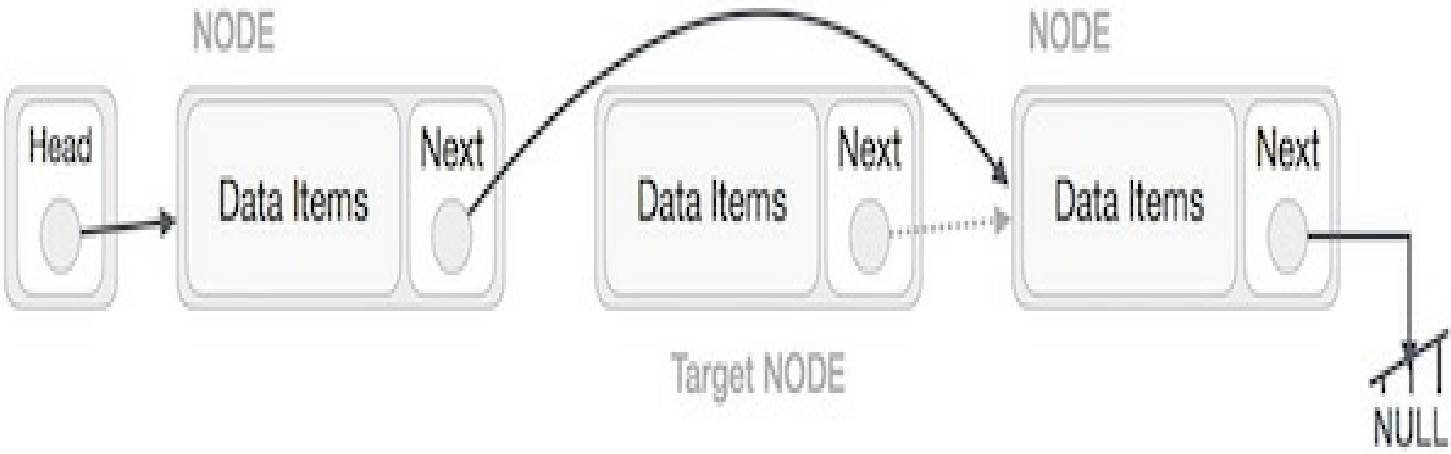
The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next -> TargetNode.next;
```

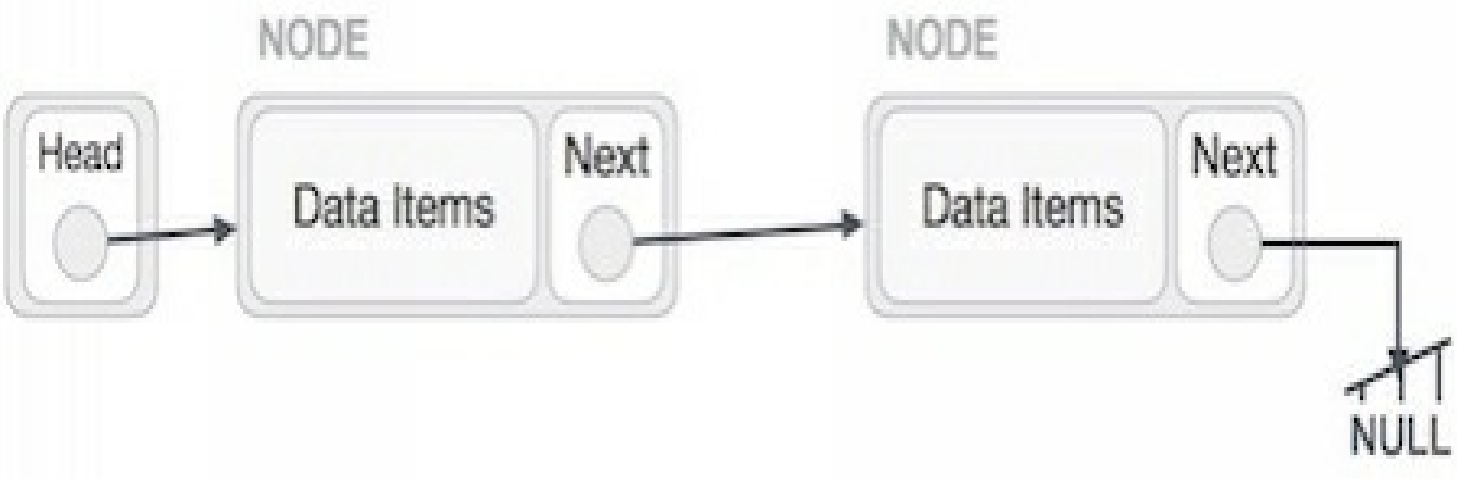


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```

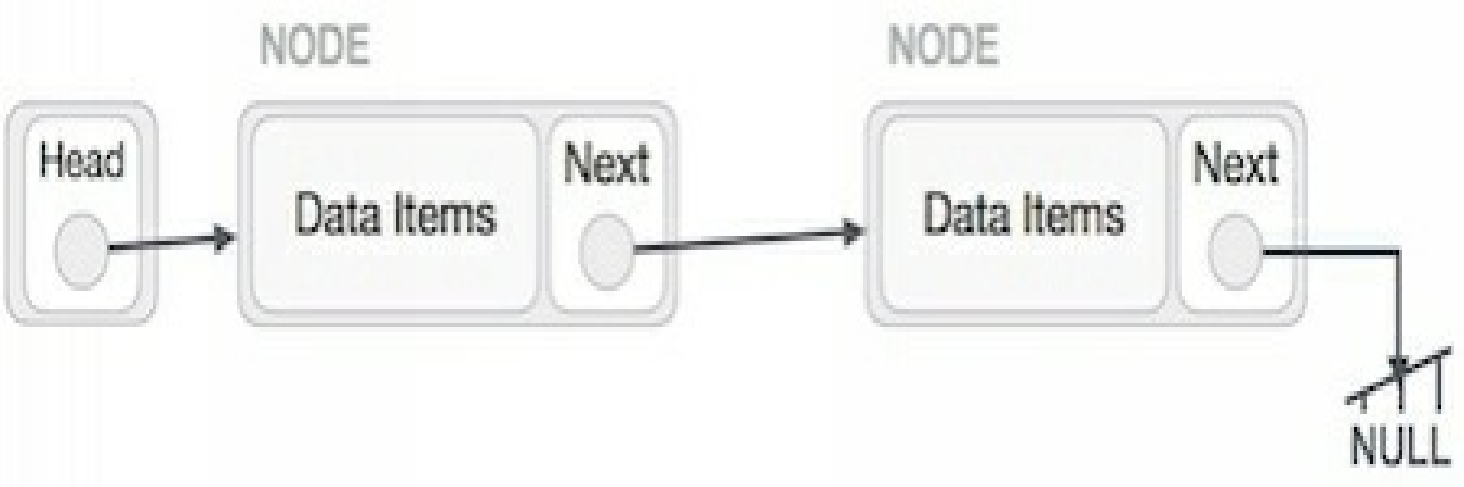


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

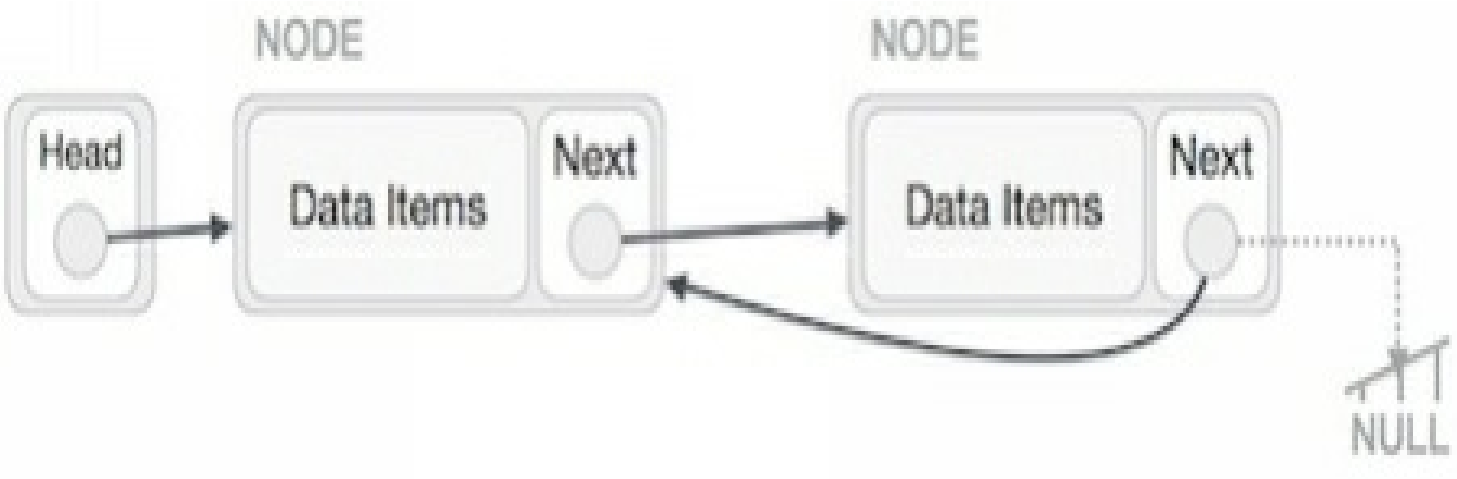


Reverse Operation:

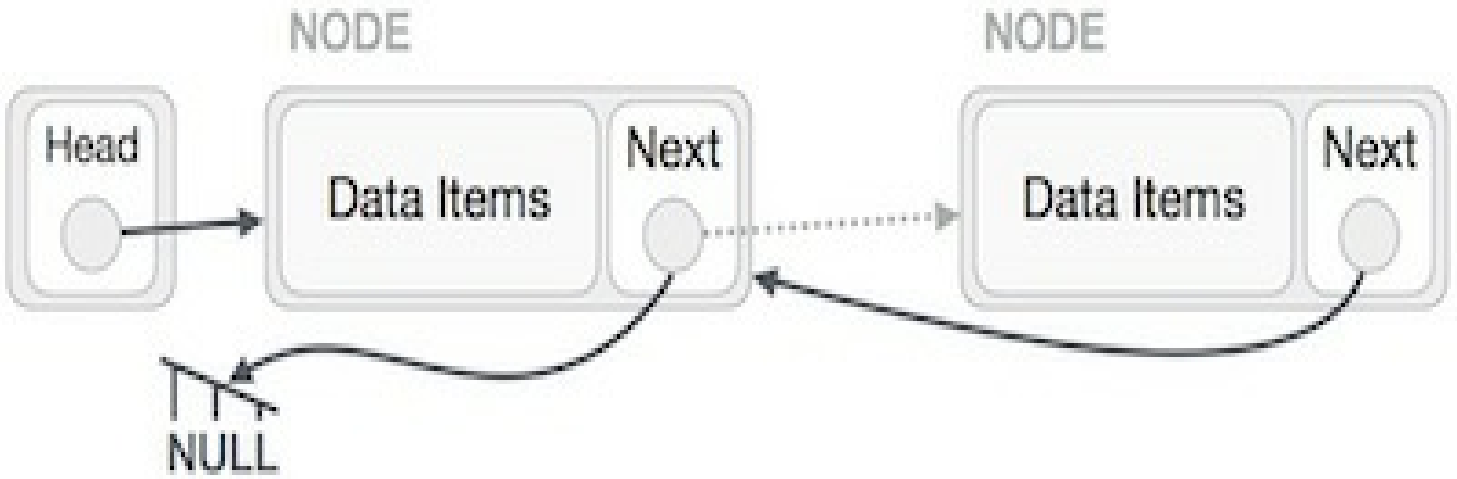
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



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 –

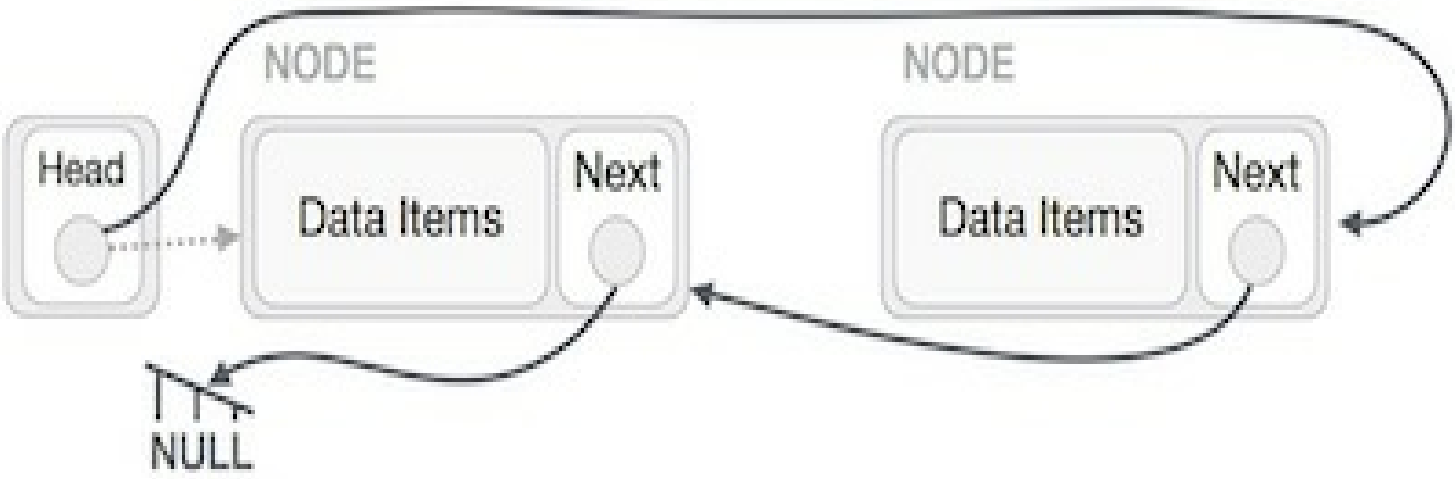


We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.

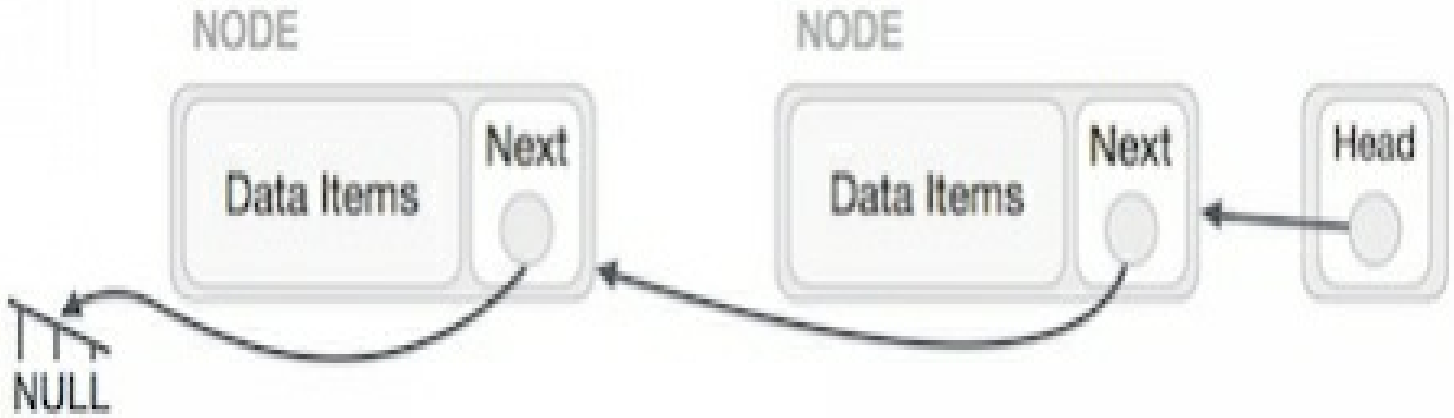


Except the node (first node) pointed by the head node, all nodes should point to

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

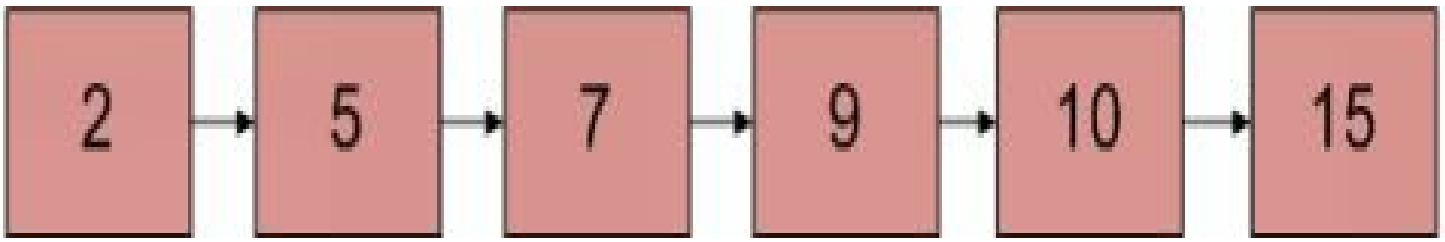
greater than the input node. The node just before GN is the appropriate

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



```
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;
        }
    }

                /*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;

        // 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.nex
= new Node(2);
        list.head.next.next.next.next.next.nex
= 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 | Basic Programming Interview Questions

Q.1) PROGRAM TO CHECK UNIQUE NUMBER IN JAVA

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

  public class IsUnique
{

public static boolean
isUniqueUsingHash(String word)
{
char[] chars = word.toCharArray();
```

```
Set<Character> set = new
HashSet<Character>();
for (char c : chars)

if (set.contains(c))
return false;
else
set.add(c);
return true;
}

public static boolean
isUniqueUsingSort(String word)
   {
   char[] chars = word.toCharArray();

if (chars.length <= 1)
```

```
return true;
Arrays.sort(chars);


char temp = chars[0];

for (int i = 1; i < chars.length; i++)
{
if (chars[i] == temp)
return false;
temp = chars[i];
}

return true;

}

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

```
throws IOException
{
System.out.println(isUniqueUsingHasl
? "Unique" : "Not Unique");
System.out.println(isUniqueUsingSort
? "Unique" : "Not Unique");

}
}
```

Output:

Q.2) PROGRAM TO FIND PERMUTATION OF A STRING

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

public class CheckPermutations
    {

    public static boolean
isPermutation(String    s1,
String s2)
        {
                char[]  a  =
s1.toCharArray();
```

```
 char[]
   b =
s2.toCharArray();

Arrays.sort(a);
Arrays.sort(b);
                  if
(a.length    !=
b.length)
           return
false;
          for (int
i  =  0;  i  <
a.length;
i++)
        {
```

```
        if (a[i] != b[i]) return false;

          }

          return true;

           }



        public static void main(String[]
args)
        {
        System.out.println(isPermutation("a
"cba") ? "It is a permutation" :
        "It is not a permutation");
```

```
System.out.println(isPermutation("test
"estt") ? "It is a permutation" : "It is
not a permutation");

System.out.println(isPermutation("test
"estt") ? "It is a permutation" : "It is
not a  permutation");

    }

    }
```

Output:

```
It is a permutation
It is a permutation
It is not a  permutation
```

Q.3) PROGRAM TO PUT HTML LINKS AROUND URLS STRINGS

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

  public class URLify
    {
public static char[] URLify(char[]
chars, int len)
{
  int spaces = countSpaces(chars,
len);

int end = len - 1 + spaces *2;
for (int i = len - 1; i >= 0; i-)
```

```
{
if (chars[i] == ' ')
{

chars[end - 2] ='%';
chars[end - 1] ='2';
chars[end] = '0';
end -= 3;

}
else
{
chars[end] =chars[i]; end--;
}

}
return chars;
}
```

```
static int countSpaces(char[] chars,
int len)
{
int count = 0;

for (int i = 0; i < len;i++)
if (chars[i] == ' ') count++;

return count;

}
```

```
public static void main(String[] args)
throws IOException
{
   char[] chars = "Mr John Smith
".toCharArray();
System.out.println(URLify(chars,
13));

}

}
```

Output:

Mr%20John%20Smith

Q.4) PROGRAM TO CHECK PALINDROME PERMUTATIONS OF A STRING

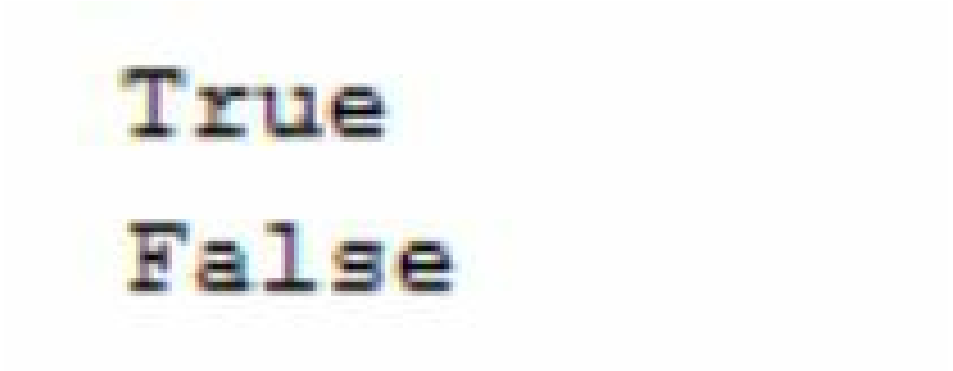
```
import java.util.*;


  public class PalindromePermutation
{

public static boolean
permuteHash(String str)
{
```

```
Map<Character, Integer> map = new
HashMap<Character, Integer>();
for (int i = 0; i < str.length(); i++) {

Character c =
Character.toLowerCase(str.charAt(i))

if (!Character.isLetter(c))
continue;

if (map.containsKey(c))
map.put(c, map.get(c) + 1);
else
map.put(c, 1);
}

int odd = 0;
```

```
for (Character key : map.keySet())
if (map.get(key) % 2 != 0)
odd++;



if (odd > 1)
return false;
else
return true;
}
public static void main(String[] args)
{
System.out.println(permuteHash("Tact
Coa") ? "True" : "False");
System.out.println(permuteHash("test'
```

```
? "True" : "False");
  }
```

Output:



Q.5) PROGRAM TO COMPRESS STRING

```
public class
StringCompression
{
public String
compress(String input)
{
char[] cs =
input.toCharArray();
```

```
char temp = cs[0];
int i = 0, j = 0, count = 0,
len = cs.length;
while(j < len)
{
cs[i++] = temp;
while (j < len && temp ==
cs[j])
{
j++;
count++;
}
if(j < len)
```

```
temp = cs[j++];
cs[i++] =
String.valueOf(count).charA
count = 1;
}
return new String(cs, 0, i);
}
```

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

StringCompression
compression = new
StringCompression();
System.out.println(compress
}
}
```

Output:


a2b3c3

Q.6) PROGRAM TO ROTATE MATRIX

```
import java.util.*;


public class RotateMatrix {

public static void rotate(int[][]
matrix)
{
int n = matrix.length;

for (int layer = 0; layer < n / 2;
layer++)
```

```
{
int first = layer;
int last = n - 1 - layer;
for (int i = first; i < last; i++)
{
int offset = i - first;
int top = matrix[first][i];

// left -> top

matrix[first][i] = matrix[last-offset]
[first];
// bottom -> left
matrix[last-offset][first] =
matrix[last][last-offset];
// right -> bottom matrix[last][last-
offset] = matrix[i][last];
// top -> right matrix[i][last] = top;
```

```
}

}

}
```

```
public static void main(String[] args)
{
int[][] arr = new int[][]
{

{1, 2, 3, 4, 5},
{6, 7, 8, 9, 10},
{11, 12, 13, 14, 15},
```

```
{16, 17, 18, 19, 20},
{21, 22, 23, 24, 25}

};

rotate(arr);

for (int[] a : arr)
System.out.println(Arrays.toString();

   }
```

Output:

```
[21, 16, 11, 6, 1]
[22, 17, 12, 7, 2]
[23, 18, 13, 8, 3]
[24, 19, 14, 9, 4]
[25, 20, 15, 10, 5]
```

Q.7) PROGRAM TO CONVERT ALL 1 INTO ZERO MATRIX.

```
import java.util.*;

  public class ZeroMatrix {

public static void zero(int[][] matrix)
{
int m = matrix.length;

int n = matrix[0].length;
boolean[] row = new boolean[m];
boolean[] col = new boolean[n];
```

```
for (int i = 0; i < m; i++) {
for (int j = 0; j < n; j++) {
if (matrix[i][j] == 0) {
row[i] = true;
col[j] = true;
}
}
}

for (int i = 0; i < m; i++) {
  for (int j = 0; j < n; j++) {
if (row[i] == true || col[j] == true)
matrix[i][j] = 0;
}
}
return;
```

```
}



public static void main(String[] args)
{
int[][] matrix = new int[][]
{
{0, 1, 1, 1, 0},
{1, 0, 1, 0, 1},
{1, 1, 1, 1, 1},
{1, 0, 1, 1, 1},
{1, 1, 1, 1, 1}
};

for (int i = 0; i < matrix.length; i++)
System.out.println(Arrays.toString(ma
```

```
zero(matrix);
System.out.println();

for (int i = 0; i < matrix.length; i++)
System.out.println(Arrays.toString(ma
}
}
```

Output:

```
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 0, 0, 0]
[0, 0, 1, 0, 0]
```

Q.8) PROGRAM TO ROTATE A STRING.

```
import java.util.*;

  public class StringRotation {

public static boolean
isRotation(String s1, String s2)
{
if (s1.length() != s2.length())
return false;
String s3 = s1 + s1;
return s3.contains(s2);

}
```

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

System.out.println(isRotation("waterbottl
"erbottlewat") ? "True" :
    "False");
    System.out.println(isRotation("waterb
    "erbottlewat") ? "True" : "False");
    System.out.println(isRotation("pandee
    "deeshpand") ? "True" : "False");
    }
```

Output:

```
True

False

False
```

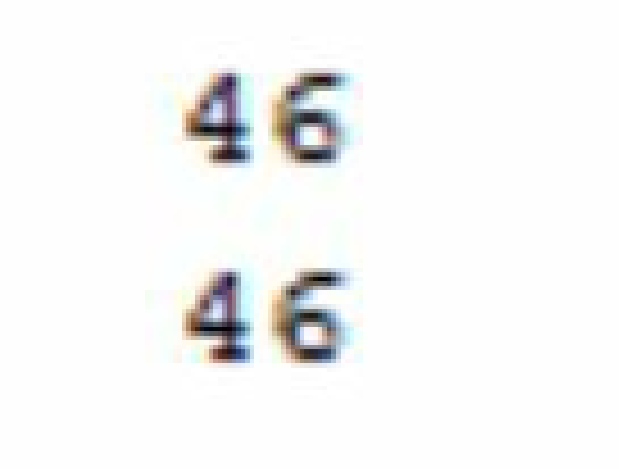
Q.9) PROGRAM TO ADD TWO NUMBERS WITHOUT USING PLUS ('+') SIGN

```
public class AddWithoutPlus
{
public static int add(int a, int b)
{
if (b > a)
{
int temp = b;
b = a;
a = temp;
}

int carry = 0;
```

```
while (b != 0)
{
carry = a & b;
a = a ^ b;
b = carry << 1;
}
return a;

}

public static int recursiveAdd(int a,
int b)
{
if (b == 0)

return a;
else
```

```
return recursiveAdd(a ^ b, (a & b) << 1);
}



public static void main(String[] args)
{
int a = 12;

int b = 34;
System.out.println(add(a, b));

System.out.println(recursiveAdd(b, a));

}
```

Output:



Q.10) PROGRAM TO REMOVE DUPLICATES CHARACTER FROM A STRING

```
import java.util.*;

public class RemoveDups {
public static class Node {
Node next;

char val;

public Node(char val)
{
this.val = val;
}
```

```
public String toString() {

StringBuilder sb = new
StringBuilder();
Node temp = this;

while (temp != null)
{
sb.append(temp.val);
temp = temp.next;
}

return sb.toString();

}
}
```

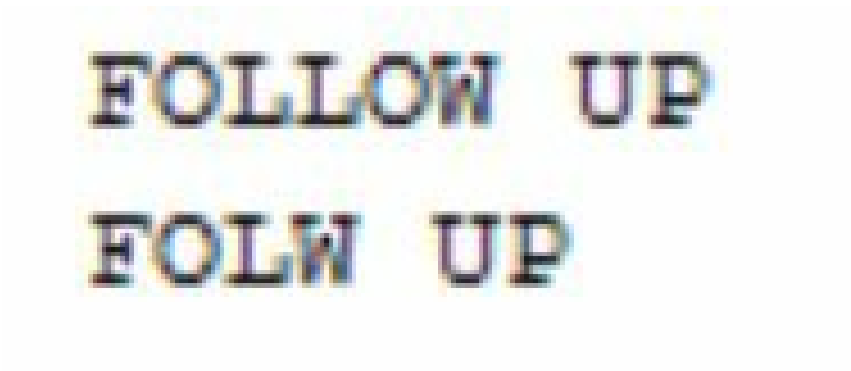
```
public static void
removeDupes(Node node)
{
Set<Character> set = new
HashSet<Character>();
set.add(node.val);

Node prev = node;
Node temp = node.next;

while (temp != null)
{
if (set.contains(temp.val))
{
```

```
prev.next = temp.next;
}

else
{
set.add(temp.val);

prev = temp;
}

temp = temp.next;

}

}
```

```
public static void main(String[] args)
{
Node a = new Node('F');
Node b = new Node('O');
Node c = new Node('L');
Node d = new Node('L');
Node e = new Node('O');
Node f = new Node('W');
Node g = new Node(' ');
Node h = new Node('U');
Node i = new Node('P');

a.next = b;
b.next = c;
c.next = d;
d.next = e;
e.next = f;
```

```
f.next = g;
g.next = h;
h.next = i;

System.out.println(a);
removeDupes(a);
System.out.println(a);
}
}
```

Output:



Q.11) PROGRAM TO RETURN A CHARACTER FROM THE STRING.

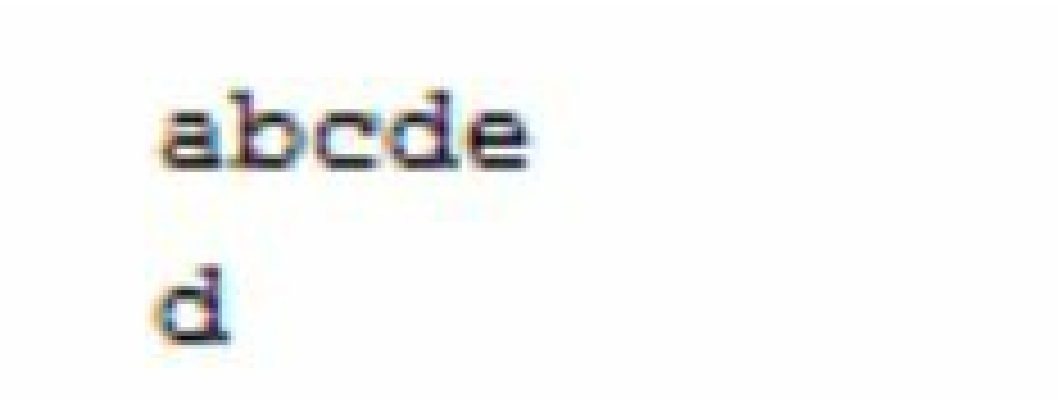
```
import java.util.*;


public class ReturnKth {
  public static class Node {
    Node next;
    char val;
    public Node(char val) {
      this.val = val;
    }
    public String toString() {
      StringBuilder sb = new
StringBuilder();
```

```
      Node temp = this;
      while (temp != null) {
        sb.append(temp.val);
        temp = temp.next;
      }
      return sb.toString();
    }
  }
  public static Node returnKth(Node
node, int k) {
    k--;
    Node first = node;
    Node last = node;
    for (int i = 0; i < k; i++)
      last = last.next;
    while (last.next != null) {
      last = last.next;
      first = first.next;
```

```
    }


    return first;
  }


  public static void main(String[]
args) {
    Node a = new Node('a');
    Node b = new Node('b');
    Node c = new Node('c');
    Node d = new Node('d');
    Node e = new Node('e');
    a.next = b;
    b.next = c;
    c.next = d;
    d.next = e;
```

```
    System.out.println(a);
    System.out.println(returnKth(a,
2).val);
  }
}
```

Output:



Q.12) PROGRAM TO REMOVE MIDDLE CHARATER FROM A STRING

```
import java.util.*;

public class DeleteMiddle
{
public static class Node
{
Node next;

char val;

public Node(char val)
{
```

```
this.val = val;
}

public String toString()
{
StringBuilder sb = new
StringBuilder();
Node temp = this;

while (temp != null)
{
sb.append(temp.val);
temp = temp.next;
}

return sb.toString();
}
```

```
public static boolean
deleteMiddle(Node node) {
if (node == null || node.next == null)
{
return false;
}
else
{
node.val = node.next.val;
node.next = node.next.next;
return true;
}
}
public static void main(String[] args
) {
  Node a = new Node('a');
   Node b = new Node('b');
```

```
    Node c = new Node('c');
    Node d = new Node('d');
     Node e = new Node('e');
  a.next = b;
  b.next = c;
  c.next = d;
  d.next = e;
  System.out.println(a);
  deleteMiddle(c);
  System.out.println(a);
  } }
```

Output:

```
abcde

abde
```

Q.13) WRITE A PROGRAM FOR BUBBLE SORT IN JAVA

```
public class MyBubbleSort {

    // logic to sort the elements
    public static void bubble_srt(int
array[])
{
        int n = array.length;
        int k;
        for (int m = n; m >= 0; m--) {
            for (int i = 0; i < n - 1; i++) {
                k = i + 1;
                if (array[i] > array[k]) {
```

```
                    swapNumbers(i, k,
array);
                }
            }
            printNumbers(array);
        }
    }

    private static void
swapNumbers(int i, int j, int[] array)
 {

        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
```

```
    private static void
printNumbers(int[] input) {

        for (int i = 0; i < input.length;
i++) {
            System.out.print(input[i] + ",
");
        }
        System.out.println("\n");
    }

    public static void main(String[]
args) {
        int[] input = { 4, 2, 9, 6, 23, 12,
34, 0, 1 };
```

```
        bubble_srt(input);

    }
}
```

Output:

2, 4, 6, 9, 12, 23, 0, 1, 34,

2, 4, 6, 9, 12, 0, 1, 23, 34,

2, 4, 6, 9, 0, 1, 12, 23, 34,

2, 4, 6, 0, 1, 9, 12, 23, 34,

2, 4, 0, 1, 6, 9, 12, 23, 34,

2, 0, 1, 4, 6, 9, 12, 23, 34,

0, 1, 2, 4, 6, 9, 12, 23, 34,

0, 1, 2, 4, 6, 9, 12, 23, 34,

0, 1, 2, 4, 6, 9, 12, 23, 34,

0, 1, 2, 4, 6, 9, 12, 23, 34,

Q.14) WRITE A PROGRAM FOR INSERTION SORT IN JAVA.

```
public class MyInsertionSort {

    public static void main(String[]
args)
{
        int[] input = { 4, 2, 9, 6, 23, 12,
34, 0, 1 };
        insertionSort(input);
    }

    private static void
```

```
printNumbers(int[] input)
{

        for (int i = 0; i < input.length;
i++) {
            System.out.print(input[i] + ",
");
        }
        System.out.println("\n");
    }

    public static void insertionSort(int
array[])
 {
        int n = array.length;
        for (int j = 1; j < n; j++) {
            int key = array[j];
            int i = j-1;
```

```
            while ( (i > -1) && ( array [i]
> key ) ) {
                array [i+1] = array [i];
                i--;
            }


            array[i+1] = key;
            printNumbers(array);
        }

}

}
```

Output:

2, 4, 9, 6, 23, 12, 34, 0, 1,

2, 4, 9, 6, 23, 12, 34, 0, 1,

2, 4, 6, 9, 23, 12, 34, 0, 1,

2, 4, 6, 9, 23, 12, 34, 0, 1,

2, 4, 6, 9, 12, 23, 34, 0, 1,

2, 4, 6, 9, 12, 23, 34, 0, 1,

0, 2, 4, 6, 9, 12, 23, 34, 1,

0, 1, 2, 4, 6, 9, 12, 23, 34,

Q.15) WRITE A PROGRAM TO IMPLEMENT HASHCODE AND EQUALS.

Description:

The hashcode of a Java Object is simply a number, it is 32-bit signed int, that allows an object to be managed by a hash-based data structure. We know that hash code is an unique id number allocated to an object by JVM. But actually

speaking, Hash code is not an unique number for an object.

If two objects are equals then these two objects should return same hash code. So we have to implement hashcode() method of a class in such way that if two objects are equals, ie compared by equal() method of that class, then those two objects must return same hash code. If you are overriding hashCode you need to override equals method also.

The below example shows how to override equals and hashcode methods. The class Price overrides equals and hashcode. If you notice the

hashcode implementation, it always generates unique hashcode for each object based on their state, ie if the object state is same, then you will get same hashcode. A HashMap is used in the example to store Price objects as keys. It shows though we generate different objects, but if state is same, still we can use this as key.

```
import java.util.HashMap;

public class MyHashcodeImpl {

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

```
        HashMap<Price, String> hm =
new HashMap<Price, String>();
        hm.put(new Price("Banana",
20), "Banana");
        hm.put(new Price("Apple", 40),
"Apple");
        hm.put(new Price("Orange",
30), "Orange");
        //creating new object to use as
key to get value
        Price key = new
Price("Banana", 20);
        System.out.println("Hashcode of
the key: "+key.hashCode());
        System.out.println("Value from
map: "+hm.get(key));
    }
}
```

```
class Price{
    private String item;
    private int price;
    public Price(String itm, int pr){
        this.item = itm;
        this.price = pr;
    }
    public int hashCode(){
        System.out.println("In
hashcode");
        int hashcode = 0;
        hashcode = price*20;
        hashcode += item.hashCode();
        return hashcode;
    }
```

```
 public boolean equals(Object obj){
        System.out.println("In equals");
        if (obj instanceof Price) {
            Price pp = (Price) obj;
            return
(pp.item.equals(this.item) &&
pp.price == this.price);
}
else {
            return false;
        }
    }

    public String getItem() {
        return item;
    }
```

```
    public void setItem(String item) {
        this.item = item;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    public String toString(){
        return "item: "+item+"  price:
"+price;
    }
}
```

Output:

```
In hashcode
In hashcode
In hashcode
In hashcode
Hashcode of the key: 1982479637
In hashcode
In equals
Value from map: Banana
```

Q.16) HOW TO GET DISTINCT ELEMENTS FROM AN ARRAY BY AVOIDING DUPLICATE ELEMENTS?

```
public class MyDisticntElements {
    public static void
printDistinctElements(int[] arr){

        for(int i=0;i<arr.length;i++){
            boolean isDistinct = false;
            for(int j=0;j<i;j++){
                if(arr[i] == arr[j]){
                    isDistinct = true;
                    break;
```

```
                }
            }
            if(!isDistinct){
                System.out.print(arr[i]+"
");
            }
        }
    }

    public static void main(String a[])
{
        int[] nums = {5,2,7,2,4,7,8,2,3};
        MyDisticntElements.printDistinc
    }
}
```

Output:

Q.17) WRITE A PROGRAM TO FIND THE SUM OF THE FIRST 1000 PRIME NUMBERS.

```
public class Main {
    public static void main(String
args[]){
        int number = 2;
        int count = 0;
        long sum = 0;
        while(count < 1000){
            if(isPrimeNumber(number)){
                sum += number;
                count++;
```

```
            }
            number++;
        }
        System.out.println(sum);
    }
    private static boolean
isPrimeNumber(int number){
        for(int i=2; i<=number/2; i++){
            if(number % i == 0){
                return false;
            }
        }
        return true;
    }
}
```

Output:

3682913

Q.18) WRITE A PROGRAM TO REMOVE DUPLICATES FROM SORTED ARRAY.

```
public class MyDuplicateElements
  {
    public static int[]
removeDuplicates(int[] input)
     {

        int j = 0;
        int i = 1;
        //return if the array length is less
than 2
        if(input.length < 2)
```

```
        {
         return input;
        }

        while(i < input.length)
          {
           if(input[i] == input[j])
           {
              i++;
           }else
           {
              input[++j] = input[i++];
           }
        }
        int[] output = new int[j+1];
        for(int k=0; k<output.length;
k++){
            output[k] = input[k];
```

```
        }

        return output;
    }

    public static void main(String a[])
{
        int[] input1 =
{2,3,6,6,8,9,10,10,10,12,12};
        int[] output =
removeDuplicates(input1);
        for(int i:output){
            System.out.print(i+" ");
        }
    }
}
```

Output:

```
2 3 6 8 9 10 12
```

Q.19) FIND LONGEST SUBSTRING WITHOUT REPEATING CHARACTERS.

```
import java.util.HashSet;
import java.util.Set;

public class MyLongestSubstr {

    private Set<String> subStrList =
new HashSet<String>();
    private int finalSubStrSize = 0;
```

```
    public Set<String>
getLongestSubstr(String input){
        //reset instance variables
        subStrList.clear();
        finalSubStrSize = 0;
        // have a boolean flag on each
character ascii value
        boolean[] flag = new
boolean[256];
        int j = 0;
        char[] inputCharArr =
input.toCharArray();
        for (int i = 0; i <
inputCharArr.length; i++) {
            char c = inputCharArr[i];
            if (flag[c]) {
                extractSubString(inputCharA
```

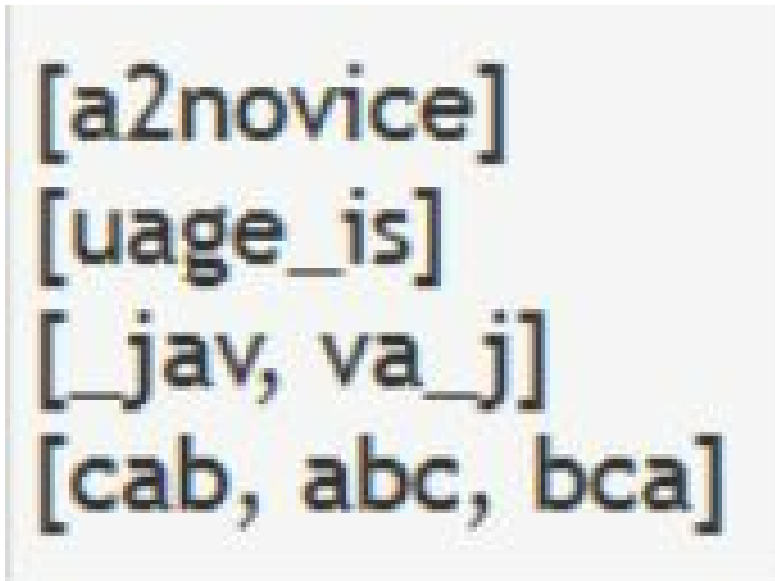
```
                for (int k = j; k < i; k++) {
                    if (inputCharArr[k] ==
c) {
                        j = k + 1;
                        break;
                    }
                    flag[inputCharArr[k]] =
false;
                }
            } else {
                flag[c] = true;
            }




}
```

```
            extractSubString(inputCharArr,j,
            return subStrList;
        }


        private String
extractSubString(char[] inputArr, int
start, int end){

            StringBuilder sb = new
StringBuilder();
            for(int i=start;i<end;i++){
                sb.append(inputArr[i]);
            }
            String subStr = sb.toString();
            if(subStr.length() >
finalSubStrSize){
```

```
            finalSubStrSize =
subStr.length();
            subStrList.clear();
            subStrList.add(subStr);
        } else if(subStr.length() ==
finalSubStrSize){
            subStrList.add(subStr);
        }

        return sb.toString();
    }

    public static void main(String a[])
{
        MyLongestSubstr mls = new
MyLongestSubstr();
        System.out.println(mls.getLonges
        Sytem.out.println(mls.getLongestS
```

```
        Sytem.out.println(mls.getLongest
        System.out.println(mls.getLonges
    }
}
```

Output:



Q.20) HOW TO SORT A STACK USING A TEMPORARY STACK?

```
import java.util.Stack;

public class StackSort
{

    public static Stack<Integer>
sortStack(Stack<Integer> input)
    {
        Stack<Integer> tmpStack = new
Stack<Integer>();
        System.out.println("==========
debug logs ==================");
        while(!input.isEmpty())
```

```
        {
            int tmp = input.pop();
            System.out.println("Element
taken out: "+tmp);
            while(!tmpStack.isEmpty()
&& tmpStack.peek() > tmp)
        {
                input.push(tmpStack.pop());
            }
            tmpStack.push(tmp);
            System.out.println("input:
"+input);
            System.out.println("tmpStack:
"+tmpStack);
        }
        System.out.println("==========
debug logs ended
==========");
```

```
        return tmpStack;
    }

    public static void main(String a[])
    {
        Stack<Integer> input = new
Stack<Integer>();
        input.add(34);
        input.add(3);
        input.add(31);
```

```
        input.add(98);
        input.add(92);
        input.add(23);
        System.out.println("input:
"+input);
        System.out.println("final sorted
list: "+sortStack(input));
    }
}
```

Output:

```
input: [34, 3, 31, 98, 92, 23]
========== debug logs
==========
```

Element taken out: 23
input: [34, 3, 31, 98, 92]
tmpStack: [23]
Element taken out: 92
input: [34, 3, 31, 98]
tmpStack: [23, 92]
Element taken out: 98
input: [34, 3, 31]
tmpStack: [23, 92, 98]
Element taken out: 31
input: [34, 3, 98, 92]
tmpStack: [23, 31]
Element taken out: 92
input: [34, 3, 98]
tmpStack: [23, 31, 92]
Element taken out: 98
input: [34, 3]
tmpStack: [23, 31, 92, 98]

Element taken out: 3
input: [34, 98, 92, 31, 23]
tmpStack: [3]
Element taken out: 23
input: [34, 98, 92, 31]
tmpStack: [3, 23]
Element taken out: 31
input: [34, 98, 92]
tmpStack: [3, 23, 31]
Element taken out: 92
input: [34, 98]
tmpStack: [3, 23, 31, 92]
Element taken out: 98
input: [34]
tmpStack: [3, 23, 31, 92, 98]
Element taken out: 34
input: [98, 92]
tmpStack: [3, 23, 31, 34]

Element taken out: 92
input: [98]
tmpStack: [3, 23, 31, 34, 92]

Element taken out: 98
input: []
tmpStack: [3, 23, 31, 34, 92, 98]
========== debug logs ended ==========
final sorted list: [3, 23, 31, 34, 92, 98]

5 SKILLS SELF-TAUGHT PROGRAMMERS COMMONLY LACK

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

WHAT ARE THE MOST IMPORTANT DATA STRUCTURE AND ALGORITHMS TO PREPARE FOR AN INTERVIEW?

Did you know that the top 10 data structures account for 99% of all data structure use in the real world? Probably not, because I just made those numbers up — but they're in the right ballpark. Yes, on occasion we ask a problem whose optimal solution requires a Bloom filter or suffix tree, but even those problems tend to have a near-optimal solution that uses a much more mundane data structure. The data

structures that are going to show up most frequently are:

- Array
- Stack / Queue
- Hashset / Hashmap / Hashtable / Dictionary
- Tree / binary tree
- Heap
- Graph

You should know these data structures inside and out. What are the insertion/deletion/lookup characteristics? (O(log n) for a balanced binary tree, for example.) What are the common caveats? (Hashing is tricky, and usually takes O(k) time when k is the size of the object being hashed.) What

algorithms tend to go along with each data structure? (Dijkstra's for a graph.) But when you understand these data structures, sometimes the solution to a problem will pop into your mind as soon as you even think about using the right one.

Stick to basics. I would classify the following data structures as ****must know****

. Linked List - Single and Doubly

. Stack

. Queues

. Binary Search Trees or general Binary Tree

. Heaps

. Basic Graph Traversal and Shortest Path

. Hashing

Following data structures may be asked. I would say that their probability of being asked is between **50 to 75% -**

. Tries

. Advance Graphs like flow and min-cut etc.

. Bit Manipulation

You will probably crack interviews with sufficient knowledge of above.

Following have very low probability of being asked (< 25%):

. Segment Trees / Binary Indexed Trees

. AVL Trees

. B+ Trees

Other hard data structures are absolutely unnecessary.

Following Algorithms / Tricks / Topics may also be important:

. Memory Management

. Basic Co-ordinate geometry - Manhattan Distance, Closest Point Pair

. Divide and Conquer

. Greedy

. Dynamic Programming - Extremely important

. Probability and basic Number Theory

. Sorting and Searching

Following topics is important for Knowledge / Experience based questions:

---

1. OS - Threads, Processes and Locks using Mutex, Semaphores
2. Scalability Issues, RPCs, Rate limiter, etc.
3. OOP Concepts
4. Databases - SQL, NoSQL, Writing simple Queries, Transactions, ACID
5. Linux Commands - sed, grep, ps, etc.

9 WAYS TO BECOME A GREAT PROGRAMMER!

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.

4 secrets of great programmers!

#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.

Resume Advice

Writing a resume is like exercising: You may not look forward to it, but you feel better once it's done. And like the results of a good workout, a well-presented resume can help you keep your career in shape.

But when writing a resume, what works and what doesn't? We thought we'd turn to Monster members like you for advice.

Here are some tips from both job seekers who write resumes and hiring professionals who read them for a living. Keep in mind that like resumes, opinions can vary -- what works for one person may not work for you.

TITLE AND OBJECTIVE:

A strong, descriptive resume title will help you stand out in a sea of resumes. “Titling your resume ‘Joe's do-it-all resume’ or ‘1975 hottie looking for a job resume’ gets your resume passed over by a busy recruiter,” says one Monster

member who should know -- he’s a recruiter himself. “Make the title useful. For instance, ‘Nursing Director, Pediatrics Labor and Delivery’ or ‘IT Telecom Project Manager, Microsoft and Cisco Certified’ or ‘Enterprise Software Sales Manager, Life Sciences’ -- enough with the stupid titles we dismiss and make fun of. This is your career we're talking about.”

And an objective must get an employer's attention quickly or it won't get any attention at all, says a district manager for a wireless company.

"I receive hundreds of resumes on a monthly basis," he says. "Two-thirds of the resumes are rejected due to the applicant having no clear objective in seeking employment with my company. Your resume must grab my attention within the first few words of the objective. It must be clearly written and relevant to the position you are applying

for. Take a little extra time and customize the objective to the position you are seeking…. If you cannot sell yourself with your resume, you might not have the opportunity to sell yourself at an interview."

LOOK AND FEEL:

As for typeface, you had definite opinions. "Don't use Times New Roman font," advises one seeker. "Your resume will look like everyone else's. Georgia and Tahoma are different, professional and pleasant to look at."

But another job seeker's font advice is more practical: "Use Times New Roman or Arial Narrow instead of other wider fonts to keep your resume to only one (or two) pages and Save paper."

Resume Expert Kim Isaacs recommends using a standard Microsoft Word-installed font so the layout will be consistent when an employer opens your resume. No matter what font you use, she suggests you stick with one per resume. "Also, the type should be large enough to be read on screen without causing eye fatigue," she says.

For the hard copy of your resume, make sure you invest in good paper stock, says one HR professional who has also composed and drafted resumes for professional clients.

“Before our prospective employer even takes one glance at our resume, there is something they do first, and that is FEEL it,” she says. “Having handled nearly hundreds of resumes each week, I think most people would be amazed how much notice you can get with a resume on good-quality paper.

Sometimes it is not even a conscious

thought, just as you shuffle stacks of resumes from here to there, making all the appropriate piles to serve your needs, you always tend to linger just a little longer over that one resume with paper that feels a little heavier, like the cotton/linen blends or the one that feels just slightly different than normal, like the parchments. You can double the effect if you choose good-quality paper in a professional color other than white."

LENGTH:

When President Lincoln was asked how long a man's legs should be, he said they should be able to reach from a man's body to the floor. Likewise, your resume should be long enough to sell you properly without overstating your accomplishments.

But of course, you had opinions on this, too. The consensus on resume length is simple: Keep it short. There are exceptions, though. "Never exceed one page, unless you have 15-plus years of

experience and are applying for a job in upper management," advises one job seeker. "Make sure that your resume remains one page and formatted properly, even when viewed in different formats and different views -- if someone opens your resume in a view other than the one you created it in and sees a hanging line, it looks unprofessional."

STYLE AND GRAMMAR:

Finally, it may seem like grade-school advice, but it bears repeating: "Although I try to counsel people on how to write a

raving resume and an awesome cover letter, I'm consistently shocked at how many resumes and cover letters I receive from people that are plagued with misspelled words, grammatical mistakes and basically little or no time spent proofreading prior to sending," says one Monster member who's been in the Staffing industry for 15-plus years. "In an era when competition seems to be one of an applicant's worst enemies; it seems that one would want to do everything possible to stand out in the crowd. Trust me: I won't give a second thought to deleting a resume and/or cover letter that is fraught with mistakes."

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!

5 CODING INTERVIEW TIPS!

ATTITUDE

Before you begin, you need to approach your practice sessions with the right attitude. Think of programming interviews as a form of standardized testing. Don't whine and think to yourself, *"but I'll never have to manually reverse a linked list in my job, so these questions are lame!"*

Ph.D. students are especially elitist because they somehow think that they are "above" preparing for petty programming interviews. That is a fine attitude if you are applying for pure research or academic jobs, but if you are interviewing at a company that uses programming interviews, then you've gotta prep!

As an analogy to high school standardized testing, I raised my SAT scores by 400 points (back when they were still out of 1600) through a few months of intense practice; there is no way that I could've gotten into MIT with

my original scores.

So before you even start practicing, you've gotta just view these interviews as yet another standardized test, another game that you need to play well and beat.

PRACTICING

I don't care how smart you are; there is simply ***no substitute*** for practicing a ton of problems. Work on problems for as long as you can before your brain explodes, then take a long break to reflect and internalize the lessons you learned through your struggle. And then repeat!

I practiced in front of the whiteboard for 1 to 2 hours at a time and did 1 to 3 practice sessions per day for two full

week's right before my interviews. That was around 40 hours of focused practice, which felt about right to me. You might need to practice more if you have less programming experience.

I used these two books as my main sources of practice problems:

- Programming Interviews Exposed
- Cracking the Coding Interview
- Stanford linked list problems (PDF)
- Stanford binary tree problems

Even if you are not familiar with the programming languages used in these

solutions, you can still code up solutions in your own language of choice and write tests to verify that they are correct.

Getting physical:

Buy your own whiteboard markers and practice using them. I personally like "MARKS-A-LOT low-odor markers", since the markers found in most office conference rooms make me nauseous.

- Always practice by writing code on a whiteboard. If you are in school, then there should be plenty of whiteboards around campus for you to use. If you are working in an

office, then you can use conference rooms after-hours.

- If you cannot practice in front of a whiteboard, then practice by writing code on blank pieces of white paper.

- After you write out your code by hand, type it into your computer to see if it actually compiles and runs correctly. This is an easy way to check for syntax or logical errors in your code. After you have practiced for a few weeks, you should be able to write error-free code on the whiteboard.

- After a week or two of intense practice, you should be able to hand-write legible, well-indented, well-aligned code on the whiteboard. If you cannot do that during your actual interview, then that will make a bad impression. Messiness is a turn-off.
- If you are doing a phone interview where you need to write code in Google Docs (or some other shared document), then practice writing code in that medium! Remember, you will never have a compiler during your interview, so you need to get good at writing compliable, runnable, and correct code even without a compiler handy.

AT THE INTERVIEW

When the interviewer presents a question to you, immediately sketch out a bunch of examples and ask a ton of clarifying questions to make sure you understand exactly what the interviewer is asking you to do.

Draw several examples and ask your interviewer questions of the form, *"for this case, you want the result to be X, right?"* Do not make any assumptions without first checking them over with your interviewer.

And whatever you do, don't flip out or try to jump straight to coding up an answer. Chances are, you either

- have no idea how to solve the problem, so you flip out and panic,
- or you think you have heard the problem before, so you want to jump the gun and sketch out a solution right away.

The former is obviously bad, but the latter might actually be worse, since you might have seen a similar problem that does not exactly match the problem you have been given. You will look like an

idiot if you try to solve the wrong problem by recalling it from memory!

COMMON PROGRAMMING INTERVIEW IDIOMS

Here are some common idioms and patterns that I have observed from doing hundreds of practice interview problems.

Strings:

- Get comfortable manipulating a string as an array of characters,

one character at a time (like C-style strings).

- Numerical arrays

Think about ***iterating backwards*** over the array elements as well as forwards. Backwards iteration is useful for, say, merging the contents of two arrays "in-place" (i.e., using O(1) outside storage).

-
  - Would the problem be easier if your array were sorted? If so, you can always tell the interviewer that you'd first do an O(n lg n) sort. Heapsort is an asymptotically optimal "in-place" sort. Once your array is sorted, think about

how you can use a variant of binary search to get O(lg n) performance rather than O(n) for an algorithm based on linear scanning.

Mappings and sets (hash tables) :

- Always think of mapping keys to values to make your life easier. If you can scan through your dataset and create an auxiliary hash table to map keys to values, then you can do O(1) lookups in a latter part of your algorithm.

- For some array-based problems, you might find it useful to create a "reverse mapping" between array elements and their indices. e.g., "the number 42 appears at index 6 in the array" is represented as a mapping of "42 -> 6".

- You can use a hash table as a set to do O(1) membership lookups. If you are being tested on low-level skillz, use bitsets and the proper bit-level operations to operate on them.

Linked lists:

- Linked list problems almost always involve singly-linked lists.

- If you are implementing an iterative algorithm to operate on a singly-linked list, chances are that you'll need to walk ***two pointers***, which I like to
- call cur and prev, down the list until one is null.
- Some problems require you to keep a gap of N elements between your cur and prev

pointers.

- Some problems requires your cur and prev pointers to advance at different speeds. e.g., moving prev up by one element while moving cur up by two elements.
- For most recursive algorithms, the base case is when the pointer is null.
- Sometimes you might need to keep a pointer to the final (tail) element of the list as well as to the head.

Binary trees:

- Remember that not all binary trees are *binary search trees*, and know the difference between the two.

- Know about the idea of a balanced binary search tree (e.g., AVL tree or red-black tree), but don't worry about being able to implement one during an interview.

Graphs:

- Whenever you need to represent binary relationships, think about using a graph. e.g., *X is friends with Y*, or *X has a crush on Y*, or *Task X needs to be done before task Y.*

- Now that you have a graph representation, what can you do with it? Chances are, you won't be asked to implement any sort of sophisticated graph algorithm since you simply don't have time in a 1-hour interview to do so.

- Definitely know how to implement a depth-first search using a stack data structure (or using recursion, which

implicitly uses your function call stack)

- Definitely know how to implement breadth-first search using a queue data structure.

- Draw a few examples of graphs for your particular problem and see what common structure arises, then tailor your algorithms to that type of structure. For example, you might find that the graphs for your problem are all a cyclic, or that they always have one unique source and sink node, or that they are bipartite (e.g., for 'matchmaking' problems), or that they are actually trees in disguise :)

- Know about the idea of topological sort.

Edge cases:

Always test your algorithm on edge-case examples. e.g., what if the user passes in an empty list or tree, or a list/tree with a single node.

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!