International Journal of Trend in Scientific Research and Development (IJTSRD)
Volume 4 Issue 4, June 2020 Available Online: www.ijtsrd.com e-ISSN: 2456 – 6470

# Java (A Language which is Vast in Itself)

**Yash Arora, Raghav Arya**

Student, Dronacharya College of Engineering, Gurugaon, Harayana, India

**ABSTRACT**

In this paper, I have done the audit of the Java programming language for the students. I will show four models and completely help to students in study. This paper surveys recent research on programming languages and development various models. Enhancements in wherever handling over late conditions has engaged architects to make structures that assistance message in the classroom. Learning includes two methods which are understanding data and changing that learning. We likewise exhibit a different layered Student Model which underpins versatile coaching by gathering the issue particular information state from understudy arrangements. This Research Work is tied in with getting the hang of programming instead of the capable practice. The circumstances are expected for master programming engineers. This paper is an attempt to contemplate how students take in the Java programming Language and make secure use of using it. An understudy in what way can find the weakness of Java application.

***How to cite this paper:*** Yash Arora | Raghav Arya "Java (A Language which is Vast in Itself)" Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-4 | Issue-4, June 2020, pp.1325-1332, URL: www.ijtsrd.com/papers/ijtsrd31179.pdf

**IJTSRD31179**



## 1. INTRODUCTION TO JAVA

### THE BEGINNING OF JAVA

Java is an object-oriented programming language developed by Sun Microsystems in 1991 and released in 1995 as a core component of Sun's Java platform. The language derives much of its syntax from C and C++, but has a simpler object model and fewer low-level facilities. The Java language was designed to be small, simple, and portable across platforms and operating systems. The Java language was developed as part of a research project to develop software for consumer electronics devices—television sets, VCRs, toasters, and the other sorts of machines we can buy at any department store. Java's goals at that time were to be small, fast, efficient, and easily portable to a wide range of hardware devices.

### THE FAMILY HISTORY OF JAVA

Before going on to study Java, let's take a brief look, through quotes, at the language on which Java was based, travelling back over 30years to do so.

### WHERE IT STARTS: C

The earliest precursor of Java is C: a language developed by Ken Thompson at Bell Labs in the early 1970s. C was used as a system programming language. C began achieving its widespread popularity when Bell's UNIX operating system was rewritten in C. Unix was the first operating system written in high level language; it was distributed to universities for free, where it become popular. Linux is currently a popular variant of UNIX. "C is a general-purpose programming language which feature economy of expression, modern control flow and data structure, and a rich set of operators. C is not a "very high level" language, nor a "big" one, and is not specialized to any particular area of application.

### FROM C TO C++

"A programming language serves two related purpose: it provides a vehicle for the programmers to specify action to be executed, and it provides a set of concepts for the programmers to use when thinking about what can be done. The first aspect ideally requires a language that is "close to the machine," so that all important aspect of the machine handled simply and efficiently in a way that is reasonably obvious to the programmer. C language was primarily designed with this in mind. The second aspect ideally requires a language that is "close to the problem to be solved," so that the concept of a solution can be expressed directly and concisely. The facilities added to C to create C++ were primarily designed with this in mind."

### JAVA AS A SUCCESSOR TO C++

"The Java programming language is a class-based, general purpose, concurrent, object-oriented language. It is designed to be simple enough that many programmers can achieve fluency in the language. The Java programming language is related to C and C++ but it is organized rather differently, with a number of aspects of C and C++ omitted and a few ideas from other languages included. It is intended to be production language, not a research language, and so, as C.A.R. Hoare suggested in his classic paper on language design, the design has avoided including new and untested features. The Java programming language is a relatively high-

level language, in that detail of the machine representation are not available through the language. It includes automatic storage management, typically using a garbage collector, to avoid the safety problems. High performance garbage collected implementation can have bounded pauses to support system programming and real time application. The language does not include any unsafe construct, such as array accesses without checking, since such unsafe constructs would cause a program to behave in an unspecified way."

**What is the Java Technology?**

- A programming language: - We can use java as a programming language. By using this programming language, we can develop different type of application for e.g.: Console Application (CUI), Window Application (GUI) and Web Applications
- A development environment: - It provides a development environment in which we can develop, check, debug and execute the applications.
- An application environment: - It provides an environment in which application can run or execute.
- A deployment environment: - It provides a deployment environment in which we can check application after deployment how application will react
- It is similar in syntax to C++.

**Features of Java are as follows:**

1. Compiled and Interpreted
2. Platform Independent and portable
3. Object- oriented
4. Robust and secure
5. Distributed
6. Familiar, simple and small
7. Multithreaded and Interactive

**1. Compiled and Interpreted**

Basically, a computer language is either compiled or interpreted. Java comes together both these approaches thus making Java a two-stage system. Java compiler translates Java code to Byte-code instructions and Java Interpreter generate machine code that can be directly executed by machine that is running the Java program.

**2. Platform Independent and portable**

Java supports the feature portability. Java programs can be easily moved from one computer system to another and anywhere. Changes and upgrades in operating systems, processors and system resources will not force any alteration in Java programs. This is reason why Java has become a trendy language for programming on Internet which interconnects different kind of systems worldwide. Java certifies portability in two ways. First way is, Java compiler generates the bytecode and that can be executed on any machine. Second way is, size of primitive data types is machine independent.

**3. Object- oriented**

Java is truly object-oriented language. In Java, almost everything is an Object. All program code and data exist in objects and classes. Java comes with an extensive set of classes; organize in packages that can be used in program by Inheritance. The object model in Java is trouble-free and easy to enlarge.

**4. Robust and secure**

Java is a most strong language which provides many securities to make certain reliable code. It is design as garbage – collected language, which helps the programmers virtually from all memory management problems. Java also includes the concept of exception handling, which detain serious errors and reduces all kind of threat of crashing the system. Security is an important feature of Java and this is the strong reason that programmer use this language for programming on Internet. The absence of pointers in Java ensures that programs cannot get right of entry to memory location without proper approval.

**5. Distributed**

Java is called as Distributed language for construct applications on networks which can contribute both data and programs. Java applications can open and access remote objects on Internet easily. That means multiple programmers at multiple remote locations to work together on single task.

**6. Multithreaded and Interactive**

Multithreaded means managing multiple tasks simultaneously. Java maintains multithreaded programs. That means we need not wait for the application to complete one task before starting next task. This feature is helpful for graphic applications.

**JAVA LANGUAGE FUNDAMENTAL**

**1. Data Type**

Data type specifies the size and type of values that can be stored in an identifier. The Java language is rich in its data types. Different data types allow we to select the type appropriate to the needs of the application. Data types in Java are classified into two types:

A. Primitive—which include Integer, Character, Boolean, and Floating Point.
B. Non-primitive—which include Classes, Interfaces, and Arrays.

**A. Primitive Data Types Integer**

Integer types can hold whole numbers such as 123 and −96. The size of the values that can be stored depends on the integer type that we choose.

9,223,372,036,854,775,808to9,223,372,036,854,755,807

The range of values is calculated as - $(2^{n-1})$ to $(2^{n-1})$ −1; where n is the number of bits required. For example, the byte data type requires 1 byte = 8 bits. Therefore, the range of values that can be stored in the byte data type is $-(2^{8-1})$ to $(2^{8-1})-1$
= $-2^7$ to $(2^7)$ -1
= −128 to 127

**B. Character**

It stores character constants in the memory. It assumes a size of 2 bytes, but basically it can hold only a single character because char stores Unicode character sets. It has a minimum value of 'u0000' (or 0) and a maximum value of 'uffff' (or 65,535, inclusive).

**Boolean**
Boolean data types are used to store values with two states: true or false.

### C. Java Tokens
A token is the smallest element in a program that is meaningful to the compiler. These tokens define the structure of the language. The Java token set can be divided into five categories: Identifiers, Keywords, Literals, Operators, and Separators.

### D. Identifiers
Identifiers are names provided by we. These can be assigned to variables, methods, functions, classes etc. to uniquely identify them to the compiler.

### E. Keywords
Keywords are reserved words that have a specific meaning for the compiler. They cannot be used as identifiers. Java has a rich set of keywords. Some examples are: Boolean, char, if, protected, new, this, try, catch, null, thread safe etc.

### F. Literals
Literals are variables whose values remain constant throughout the program. They are also called Constants. Literals can be of four types. They are:

I. String Literals: - String Literals are always enclosed in double quotes and are implemented using the java.lang.String class. Enclosing a character string within double quotes will automatically create a new String object. For example, String s = "this is a string"; String objects are immutable, which means that once created, their values cannot be changed.
II. Character Literals: - These are enclosed in single quotes and contain only one character.
III. Boolean Literals: - They can only have the values true or false. These values do not correspond to 1 or 0 as in C or C++.
IV. Numeric Literals: - Numeric Literals can contain integer or floating-point values.

### G. Operators
An operator is a symbol that operates on one or more operands to produce a result.

### H. Separators
Separators are symbols that indicate the division and arrangement of groups of code. The structure and function of code is generally defined by the separators. The separators used in Java are as follows:

Parentheses ( ): - Used to define

precedence in expressions, to

enclose parameters in method

definitions, and enclosing cast
types. Braces { }: - Used to define a block of code and to hold the values of arrays. Brackets [ ]: - Used to declare array types.

Semicolon ; :- Used to separate statements.

Comma,:- Used to separate identifiers in a variable declaration and in the for statement.

Period.: - Used to separate package names from classes and subclasses and to separate a variable or a method from a reference variable.

**Variables**
There are different types of variables in Java. They are as follows:

### I. Instance Variables (Non-Static Fields)
Objects store their individual states in "non-static fields", that is, fields declared without the static keyword. Non-static fields are also known as instance variables because their values are unique to each instance of a class. For example, the current Speed of one bicycle is independent from the current Speed of another.

### J. Class Variables (Static Fields)
A class variable is any field declared with the static modifier; this tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated. A field defining the number of gears for a particular kind of bicycle could be marked as static since, conceptually, the same number of gears will apply to all instances. The code static int num Gears = 6; would create such a static field.

### K. Local Variables
A method stores its temporary state in local variables. The syntax for declaring a local variable is similar to declaring a field (for example, int count = 0;). There is no special keyword designating a variable as local; that determination comes entirely from the location in which the variable is declared—between the opening and closing braces of a method. As such, local variables are only visible to the methods in which they are declared; they are not accessible from the rest of the class.

**Parameters**
They are the variables that are passed to the methods of a class.

### L. Variable Declaration
Identifiers are the names of variables. They must be composed of only letters, numbers, the underscore, and the dollar sign ($). They cannot contain white spaces. Identifiers may only begin with a letter, the underscore, or the dollar sign. A variable cannot begin with a number. All variable names are case sensitive.

**OOP'S CONCEPTS**

### 1. INHERITANCE
The process of obtaining the data members and methods from one class to another class is known as **inheritance**. It is one of the fundamental features of object-oriented programming. A class that is declared with abstract keyword, is known as **abstract class**. An abstract class is one which is containing some defined method and some undefined method. In java programming undefined methods are known as unImplemented or abstract method. The process of obtaining the data members and methods from one class to another class is known as **inheritance**. It is one of the fundamental features of object-oriented programming.

**TYPES OF INHERITANCE**

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance

**Why use Inheritance?**

- For Method Overriding (used for Runtime Polymorphism).
- Its main uses are to enable polymorphism and to be able to reuse code for different classes by putting it in a common super class
- For code Re-usability

**2. Method Overloading**

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**.

**Why method Overloading?**

Suppose we have to perform addition of given number but there can be any number of arguments, if we write method such as a(int, int)for two arguments, b(int, int, int) for three arguments then it is very difficult for you and other programmer to understand purpose or behaviors of method they cannot identify purpose of method. So, we use method overloading to easily figure out the program. For example, above two methods we can write sum (int, int) and sum (int, int, int) using method overloading concept.

**DIFFERENT WAYS TO OVERLOAD THE METHOD**

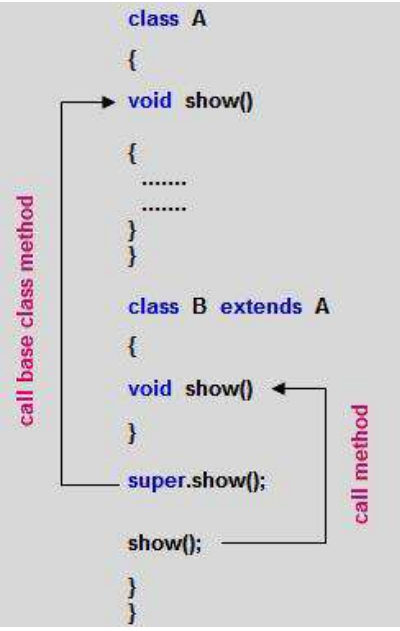
- By changing number of arguments or parameters
- By changing the data type
- By changing the order of arguments.

**METHOD OVERRIDING**

Whenever same method name is existing in both base class and derived class with same types of parameters or same order of parameters is known as **method Overriding**.

**ADVANTAGE OF JAVA METHOD OVERRIDING**

- Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method Overriding is used for Runtime Polymorphism




**Interface**

**Interface** is similar to class which is collection of public static final variables (constants) and abstract methods. The interface is a mechanism to achieve fully abstraction in java. There can be only abstract methods in the interface. It is used to achieve fully abstraction and multiple inheritance in Java.

**Why we use Interface?**

- It is used to achieve fully abstraction.
- By using Interface, you can achieve multiple inheritance in java.

**WHEN WE USE ABSTRACT AND WHEN INTERFACE**

If we do not know about any things about implementation just, we have requirement specification then we should be going for **Interface**

If we are talking about implementation but not completely (partially implemented) then we should be going for **abstract**

**ABSTRACTION**

**Abstraction** is the concept of exposing only the required essential characteristics and behavior with respect to a context.

Hiding of data is known as **data abstraction**. In object-oriented programming language this is implemented automatically while writing the code in the form of class and object.

**REAL LIFE EXAMPLE OF ABSTRACTION**

Abstraction shows only important things to the user and hides the internal details for example when we ride a bike, we only know about how to ride bike but cannot know about how it works? and also, we do not know internal functionality of bike.

**ENCAPSULATION**

**Encapsulation** is a process of wrapping of data and methods in a single unit is called encapsulation. Encapsulation is achieved in java language by class concept. Combining of state and behavior in a single container is known as encapsulation. In java language encapsulation can be achieve using **class** keyword, state represents declaration of variables on attributes and behavior represents operations in terms of method.

**BENEFITS OF ENCAPSULATION**

- Provides abstraction between an object and its clients.
- Protects an object from unwanted access by clients.
- Example: A bank application forbids (restrict) a client to change an Account's balance

**POLYMORPHISM**

The process of representing one form in multiple forms is known as **Polymorphism**. Here original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.

Polymorphism is not a programming concept but it is one of the principals of OOPs. For many objects-oriented programming language polymorphism principles is common

but whose implementations are varying from one objects-oriented programming language to another object-oriented programming language.

Polymorphism principal is divided into two sub principal they are:

- Static or Compile time polymorphism
- Dynamic or Runtime polymorphism

**EXCEPTION HANDLING**

An exception is an event, which occurs during the execution of the program, that an interrupt the normal flow of the program 's instruction. In other words, Exceptions are generated when a recognized condition, usually an error condition, arises during the execution of a method. Java includes a system for running exceptions, by tracking the potential for each method to throw specific exceptions. For each method that could throw an exception, our code must report to the Java compiler that it could throw that exact exception. The compiler marks that method as potentially throwing that exception, and then need any code calling the method to handle the possible exception. Exception handling is basically use five keywords as follows:

- Try
- Catch
- Throw

**Overview**

Exceptions are generated when an error condition occur during the execution of a method. It is possible that a statement might throw more than one kind of exception. Exception can be generated by Java-runtime system or they can be manually generated by code. Error Handling becomes a necessary while developing an application to account for exceptional situations that may occur during the program execution.

Exceptions are generated when a recognized an error condition during the execution of a program. Java includes a system for running exceptions, by tracking the potential for each method to throw specific exceptions

- for each method that could throw an exception, our code must report to the Java compiler that it could throw that exact exception.
- the compiler marks that method as potentially throwing that exception, and then need any code calling the method to handle the possible exception.

There are two ways to handle an exception:

- we can try the "risky" code, catch the exception, and do something about it, after which the transmission of the exception come to an end.
- we can mark that this method throws that exception; in which case the Java runtime engine will throw the exception back to the method.

So, if we use a method in our code that is marked as throwing a particular exception, the compiler will not allow that code unless we handle the exception. If the exception occurs in a try block, the JVM looks to the catch block(s) that follow to see if any of them equivalent the exception type. The first one that matches will be executed. If none match, then this method ends, and execution jumps to the method that called this one, at the point the call was made.

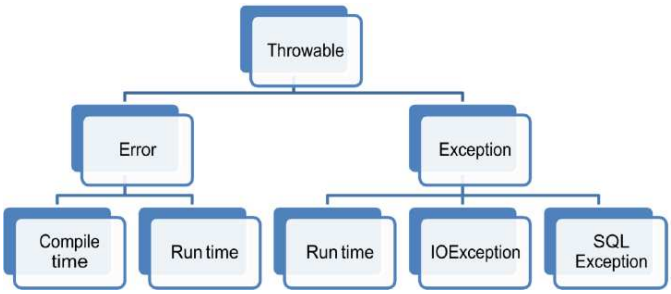



**Fig: a partial view of the Throw able family**

**Try…catch**

If a method is going to resolve potential exception internally, the line of code that could generate the exception is placed inside a try block. There may be other code inside the try block, before and/or after the risky line(s) - any code that depends upon the risky code's success should be in the try block, since it will automatically be skipped if the exception occurs.

code risky/unsafe code code that depends on the risky code succeeding

There is usually at least one catch block immediately after the try block. A catch block must specify what type of exception it will catch.

Syntax: catch (ExceptionClassNameexceptionObjectName) {code using methods fromexceptionObjectName}

There can be more than one catch block, each one marked for a correct exception class. The exception class that is caught can be any class in the exception hierarchy, either a general (base) class, or a very correct (derived) class. The catch block(s) must handle all checked exceptions that the try block is known to throw unless we want to throw that exception back to the method. It is possible to have a try block without any catch blocks if we have a finally block but any checked exceptions still need to be caught, or the method needs to declare that it throws them. If an exception occurs within a try block, execution jumps to the first catch block whose exception class matches the exception that occurred. Any steps remaining in the try block are skipped. If no exception occurs, then the catch blocks are skipped If declare a variable within a try block, it will not exist outside the try block, since the curly braces define the scope of the variable. We will often need that variable later, if nowhere else other than the catch or finally blocks, so we would need to declare the variable before the try. If we declare but don't initialize a variable before a try block, and the only place we set a value for that variable is in the try block, then it is possible when execution leaves the try. catch structure that the variable never received a value. So, we would get a "possibly uninitialized value" error message from the compiler, since it actually keeps track of that sort of thing.

**THROW**

We can throw an exception explicitly using the throw statement. For example, we need to throw an exception when a user enters a wrong student ID or password. The throws clause is used to list the types of exception that can be thrown in the execution of a method in a program. The throw statement causes termination of the normal flow of control of the java code and prevents the execution of the subsequent statements. The throw clause conveys the

control to the nearest catch block handling the type of exception object throws. If no such catch block exists, the program terminates. The throw statement accepts a single argument, which is an object of the Exception class.

**THROWS**

The throws statement is used by a method to specify the types of exceptions the method throws. If a method is capable of raising an exception that it does not handle, the method must specify that the exception have to be handled by the calling method. This is done using the throws statement. The throws clause lists the types of exceptions that a method might throw.

**FINALLY**

To guarantee that a line of code runs, whether an exception occurs or not, use a finally block after the try and catch blocks. The code in the finally block will almost always execute, even if an unhandled exception occurs; in fact, even if a return statement is encountered. If an exception causes a catch block to execute, the finally block will be executed after the catch block or if an uncaught exception occurs, the finally block executes, and then execution exits this method and the exception is thrown to the method that called this method.

**MULTI-THREADING**

**THE JAVA THREAD MODEL**

Thread is a sequential path of execution of a program. In java we can create multiple threads for the full utilization of the processor time. Java environment has been built around the multithreading model. In fact, all java class libraries have been designed keeping multithreading in mind. If a thread goes off to sleep for some time, the rest program does not get affected by this. Similarly, an animation loop can be fired that will not stop the working of rest of the system. At a point of time a thread can be in any one of the following states: new, ready, running, inactive and finished. A thread enters the new state as soon as it is created. When it is started, it is ready to run. The start () method in turn calls the run () method which makes the thread enter the running state. While running, a thread might get blocked because some resource that it requires is not available, or it could be suspended on purpose for some reason. In such a case the thread enters the state of being inactive. A thread can also be stopped purposely because its time has been expired, then it enters the state of ready to run once again. A thread that is in running state can be stopped once its job has finished. A thread that is in inactive state can either be resumed, in which case it enters the ready state again, or it can be stopped in which case it enters the finished state.

**MULTITHREADING**

Multithreading is a process of executing multiple threads simultaneously. So, at this point we will ask ourselves what a thread is. A thread is a lightweight subprocess, a smallest unit of processing. It is a separate path of execution. It shares the memory area of process. So, in short, Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system. Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes. I need to define another term related to threads: process: A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. In Java, the Java Virtual Machine (JVM) allows an application to have multiple threads of execution running concurrently. It allows a program to be more responsible to the user.

**THREAD PRIORITIES**

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled. Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5). Threads with higher priority are more important to a program and should be allocated processor time before lower - priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

**STREAMS**

A stream is a path of communication between the source of some information and its destination. That information, an un-interpreted byte stream, can come from any "pipe source," the computer's memory, or even from the Internet. In fact, the source and destination of a stream are completely arbitrary producers and consumers of bytes, respectively. Therein lies the power of the abstraction. You don't need to know about the source of the information when reading from a stream, and you don't need to know about the final destination when writing to one. General-purpose methods that can read from any source accept a stream argument to specify that source; general methods for writing accept a stream to specify the destination. Arbitrary processors (or filters) of data have two stream arguments. They read from the first, process the data, and write the results to the second. These processors have no idea of either the source or the destination of the data they are processing. Sources and destinations can vary widely: from two memory buffers on the same local computer, to the ELF transmissions to and from a submarine at sea, to the Realtime data streams of a NASA probe in deep space. By decoupling the consuming, processing, or producing of data from the sources and destinations of that data, you can mix and match any combination of them at will as you write your program. In the future, when new, previously nonexistent forms of source or destination (or consumer, processor, or producer) appear, they can be used within the same framework, with no changes to your classes. New stream abstractions, supporting higher levels of interpretation "on top of" the bytes, can be written completely independently of the underlying transport mechanisms for the bytes themselves. At the pinnacle of this stream framework are the two abstract classes, Input Stream and Output Stream. If you turn briefly to the diagram for java.io in Appendix B, these classes are a virtual cornucopia of categorized classes, demonstrating the wide range of streams in the system, but also demonstrating an extremely well-designed hierarchy of relationships between these streams, one well worth

learning from. Let's begin with the parents and then work our way down this bushy tree. There are two types of Streams in java, Input Stream and Output Stream.

**INPUT STREAMS**

All the methods you will explore today are declared to throw IOExceptions. This new subclass of Exception conceptually embodies all the possible I/O errors that might occur while using streams. Several subclasses of it define a few, more specific exceptions that can be thrown as well. For now, it is enough to know that you must either catch an IOException, or be in a method that can "pass it along," to be a well-behaved user of streams.

**READ ()**

The most important method to the consumer of an input stream is the one that reads bytes from the source. This method, read (), comes in many flavors, and each is demonstrated in an example in today's lesson. Each of these read () methods is defined to "block" (wait) until all the input requested becomes available.

Don't worry about this limitation; because of multithreading, you can do as many other things as you like while this one thread is waiting for input. In fact, it is a common idiom to assign a thread to each stream of input (and for each stream of output) that is solely responsible for reading from it (or writing to it). These input threads might then "hand off" the information to other threads for processing. This naturally overlaps the I/O time of your program with its compute time.

This form of read()attempts to fill the entire buffer given. If it cannot (usually due to reaching the end of the input stream), it returns the actual number of bytes that were read into the buffer. After that, any further calls to read()return -1, indicating that you are at the end of the stream. Note that the if statement still works even in this case, because -1! = 1024(this corresponds to an input stream with no bytes in it all).

**SKIP()**

What if you want to skip over some of the bytes in a stream, or start reading a stream from other than its beginning?

MARK() AND RESET()

Some streams support the notion of marking a position in the stream, and then later resetting the stream to that position to reread the bytes there. Clearly, the stream would have to "remember" all those bytes, so there is a limitation on how far apart in a stream the mark and its subsequent reset can occur. There's also a method that asks whether or not the stream supports the notion of marking at all.

When marking a stream, you specify the maximum number of bytes you intend to allow to pass before resetting it. This allows the stream to limit the size of its byte "memory." If this number of bytes goes by and you have not yet reset(), the mark becomes invalid, and attempting to reset()will throw an exception. Marking and resetting a stream is most valuable when you are attempting to identify the type of the stream (or the next part of the stream), but to do so, you must consume a significant piece of it in the process.

Often, this is because you have several blackbox parsers that you can hand the stream to, but they will consume some (unknown to you) number of bytes before making up their mind about whether the stream is of their type. Set a large size for the read limit above, and let each parser run until it either throws an error or completes a successful parse. If an error is thrown, reset()and try the next parser.

**close()**

Because you don't know what 531sources an open stream rep531sents, nor how to deal with them properly when you're finished reading the stream, you must usually explicitly close down a stream so that it can release these 531sources. Of course, garbage collection and a finalization method can do this for you, but what if you need to reopen that stream or those resources before they have been freed by this asynchronous p5ocess? At best, this is annoying or confusing; at worst, it introduces an unexpected, obscure, and difficult-totrack-down bug. Because you're interacting with the outside world of external 531sources, it's safer to be explicit about when you're finished using them:

**OUTPUT STREAMS**

Output streams are, in almost every case, paired with a "brother" Input Stream that you've already learned. If an Input Stream performs a certain operation, the "brother" Output Stream performs the inverse operation.

**write()**

The most important method to the producer of an output stream is the one that writes bytes to the destination. This method, write(), comes in many flavors.

**FLUSH()**

Because you don't know what an output stream is connected to, you might be required to "flush" your output through some buffered cache to get it to be written (in a timely manner, or at all). OutputStream's version of this method does nothing, but it is expected that subclasses that require flushing (for example, BufferedOutputStream and PrintStream) will override this version to
do something nontrivial.

**CLOSE()**

Just like for an Input Stream, you should (usually) explicitly close down an Output Stream so that it can release any resources it may have reserved on your behalf.

(All the same notes and examples from InputStream's close() method apply here, with the prefix In replaced everywhere by Out.) All output streams descend from the abstract class Output Stream. All share the previous few methods in common

- mean something that alternates).
- Do not use the word "essentially" to mean "approximately" or "effectively".
- In your paper title, if the words "that uses" can accurately replace the word "using", capitalize the "u"; if not, keep using lower-cased.
- Be aware of the different meanings of the homophones "affect" and "effect", "complement" and "compliment", "discreet" and "discrete", "principal" and "principle".
- Do not confuse "imply" and "infer".

- The prefix "non" is not a word; it should be joined to the word it modifies, usually without a hyphen.
- There is no period after the "et" in the Latin abbreviation "et al."
- The abbreviation "i.e." means "that is", and the abbreviation "e.g." means "for example".

**CONCLUSION**

Nothing is perfect in the world. So I am also no exception. I have tried my best to build this project with efficient information. I do not permit the project to be 100% accurate. This project help the all customer which have multiple bank account in different banks. Due to this the time customer save own time and they easly login on this and make transaction easily. The main focus of this project is to save the customer time which have multiple bank account in different banks.

The maintenance of the records is made efficient, as all the records are stored in it, through which data can be retrieved easily.

We finally conclude that using this research we provided is of a great interface between the user and the programming environment, thus satisfying the requirements of multiple users. And finally the users will be satisfied with our service.

**REFERENCES**

[1] Javapoint.com.

[2] Wc3.com.

[3] Google.com.

[4] Training.com.

[5] Study material from NIIT.