

DANIEL UGOCHUKWU ONOVO
UB81898CO91115

BACHELOR IN COMPUTER SCIENCE
COURSE: JAVA PROGRAMMING

ATLANTIC INTERNATIONAL UNIVERSITY
HONOLULU, HAWAII U.S.A

SEPTEMBER, 2024

TABLE OF CONTENTS

CONTENTS	PAGE
1. Introduction to JAVA Programming	3
2. History of Java Releases	3
3. Advantages of Java programming	5
4. Disadvantages of Java Programming	6
5. Java Program Structure	8
6. Primitive Data Type in Java	10
7. Printing Output in Java	14
8. Accepting Input from User in Java	17
9. Java Option Panes	21
10. Variable Declaration and Initialization in Java	24
11. Java Array	28
12. Using Operators in Java	32
13. Control Structure in Java	37
14. Object Oriented Programming in Java	43
15. Conclusion	51
16. Bibliography	52

1.0 INTRODUCTION TO JAVA PROGRAMMING

Java is an object-oriented language for general-purpose business applications and for interactive, Web-based Internet applications developed by James Gosling from Sun Microsystems in 1995 but later owned by Oracle corporation. The core objective was to provide a platform-independent alternative to C++. In other terms, it is architecturally neutral, i.e you can use Java to write a program that will run on any platform or device (operating system). Java program can run on a wide variety of computers because it does not execute instructions on a computer directly. Instead, it runs on a Java Virtual Machine (JVM).

Java is a general-purpose programming language that's used in virtually all industries for almost any type of application. By being language is object-oriented (OO), Java allows you to easily relate program constructs to real world objects.

Java is used for:

- Mobile applications (specially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection
- And much, much more!

1.1 History of Java Releases

Java Version	Release Date	Important Features/Code Name
JDK 1.0	Jan 1996	Initial release
JDK 1.1	Feb 1997	Reflection, JDBC, Inner classes, RMI
J2SE 1.2	Dec 1998	Collection, JIT, String memory map
J2SE 1.3	May 2000	Java Sound, Java Indexing, JNDI
J2SE 1.4	Feb 2002	Assert, regex, exception chaining,
J2SE 5.0	Sept 2004	Generics, autoboxing, enums
Java SE 6.0	Dec 2006	JDBC 4.0, java compiler API, Annotations

Java SE 7.0	July 2011	String in switch-case, Java nio, exception handling new way
Java 8	March 2014	A major release with key features like Lambda expressions, the Stream API, and the new Date and Time API (java.time).
Java 9	Sept 2017	Introduced the module system (Project Jigsaw), JShell (the Java Shell), and other enhancements like improved performance and new APIs
Java 10	March 2018	Featured local variable type inference with the var keyword and other minor improvements.
Java 11	Sept 2018	As a Long-Term Support (LTS) release, it included features like the HTTP Client API, local-variable syntax for lambda parameters, and the removal of some deprecated features.
Java 12	March 2019	Introduced features such as the Shenandoah garbage collector and the switch expressions preview feature.
Java 13	Sept 2019	Continued with incremental improvements including text blocks as a preview feature and enhancements to the garbage collector.
Java 14	March 2020	Added features like records (as a preview feature), instanceof pattern matching, and the new foreign-memory access API (also as a preview).
Java 15	Sept 2020	Included features like text blocks, sealed classes, and the ZGC (Garbage Collector) improvements.
Java 16	March 2021	Introduced features such as the JEP 394 (pattern matching for instanceof), JEP 395 (records), and JEP 396 (strong encapsulation of JDK internals).
Java 17	Sept 2021	An LTS release, bringing features like pattern matching for switch (preview), sealed classes, and new enhancements in the language and JVM.
Java 18	March 2022	Included features like the new UTF-8 by default, and various performance improvements and updates.
Java 19	Sept 2022	Added features such as the continuation of pattern matching for switch, virtual threads (preview), and improvements in the Project Loom.
Java 20	March 2023	Focused on further enhancements in pattern matching, virtual threads, and other ongoing features from previous versions.
Java 21	Sept 2023	Another LTS release, continuing to improve upon the features

		introduced in earlier versions and adding new enhancements, including updates to pattern matching, record types, and other performance optimizations.
--	--	---

1.2 Advantages of Java programming language

- Java is Object Oriented (OO).
- It has Built-in support for socket communication and memory management (automatic garbage collection).
- It has better portability than other languages across operating systems.
- **Strongly Typed:** Java enforces strict type checking at compile time, which helps catch errors early in the development process. This can lead to more robust and reliable code.
- **Automatic Memory Management:** Java includes built-in garbage collection, which automatically manages memory allocation and deallocation. This reduces the likelihood of memory leaks and other related issues.
- **Rich Standard Library:** Java provides a comprehensive standard library that includes utilities for tasks such as data manipulation, networking, file I/O, and user interface development. This extensive library accelerates development and reduces the need for external libraries.
- **Multi-threading Support:** Java has built-in support for multi-threading, allowing developers to write highly responsive and concurrent applications. The `java.util.concurrent` package offers powerful tools for managing threads and synchronization.
- **Security:** Java has a strong security model that includes features such as the sandbox environment for applets, bytecode verification, and runtime security checks. These features help protect against various types of security threats.
- **Community and Ecosystem:** Java boasts a large and active community, which contributes to a rich ecosystem of frameworks, libraries, and tools. This community support helps with troubleshooting, learning, and expanding your development capabilities.
- **Performance:** While Java was initially criticized for being slower than languages like C++, modern JVMs and Just-In-Time (JIT) compilers have significantly improved Java's performance, making it suitable for a wide range of applications.
- **Scalability:** Java is well-suited for large-scale systems and enterprise applications. Its architecture and frameworks (like Spring) support scalable and maintainable solutions for complex business needs.
- **Cross-Platform GUI:** Java's Swing and JavaFX libraries provide tools for creating graphical user interfaces that work across different platforms, making it easier to build cross-platform desktop applications.

- Java maintains a strong commitment to backward compatibility, meaning that code written in older versions of Java will typically run on newer versions without modification. This helps in maintaining and upgrading existing systems with minimal risk.
- And last but not least, Java supports Web-based applications (Applet, Servlet, and JSP), distributed applications (sockets, RMI, EJB etc.) and network protocols (HTTP, JRMP etc.) with the help of extensive standardized APIs (Application Programming Interfaces).

1.3 Disadvantages of Java Programming Language

Java is a versatile and widely-used programming language, but it does have some disadvantages:

- **Performance Overheads:** Java applications typically run on the Java Virtual Machine (JVM), which introduces some performance overhead compared to native languages like C or C++. The Just-In-Time (JIT) compilation and garbage collection processes can also impact performance.
- **Memory Consumption:** Java applications can consume more memory than applications written in some other languages. The JVM itself requires a significant amount of memory, and Java's automatic garbage collection can lead to higher memory usage.
- **Verbose Syntax:** Java code can be quite verbose compared to languages like Python or Ruby. This verbosity can lead to more boilerplate code and longer development times for simple tasks.
- **Lack of Modern Language Features:** While Java has been evolving, it sometimes lags behind other languages in adopting certain modern programming features. For example, features like pattern matching, records, and local-variable type inference have been added only recently.
- **GUI Development:** Java's Swing and AWT libraries for building graphical user interfaces are often criticized for being outdated and less modern compared to other UI frameworks.
- **Slow Startup Time:** Java applications, particularly large ones, can have a slower startup time due to the JVM's initialization and class loading processes.
- **Complex Configuration and Tooling:** Setting up a Java development environment can be complex, with various tools, configurations, and dependencies to manage. While tools like Maven and Gradle help with build automation, they also add another layer of complexity.
- **Verbosity in Error Handling:** Java's exception handling can be verbose, with a lot of boilerplate code needed to handle and propagate exceptions.

- **Limited Low-Level Programming:** Java is designed to be a high-level language with a focus on portability and safety, which limits its ability to perform low-level system programming tasks that languages like C or C++ can handle.

1.4 THE JAVA PROGRAM LIFE CYCLE

Java requires the source code of your program to be compiled first. It gets converted to either machine-specific code or a byte code that is understood by some run-time engine or a java virtual machine.

Not only will the program be checked for syntax errors by a Java compiler, but some other libraries of Java code can be added (linked) to your program after the compilation is complete (deployment stage).

Step1 : Create a source document using any editor and save file as .java (e.g. aiu.java)

Step2 : Compile the aiu.java file using “javac” command or eclipse will compile it automatically.

Step3 : Byte Code (aiu.class) will be generated on disk.

Step4 : This byte code can run on any platform or device having JVM (java.exe convert byte code in machine language)

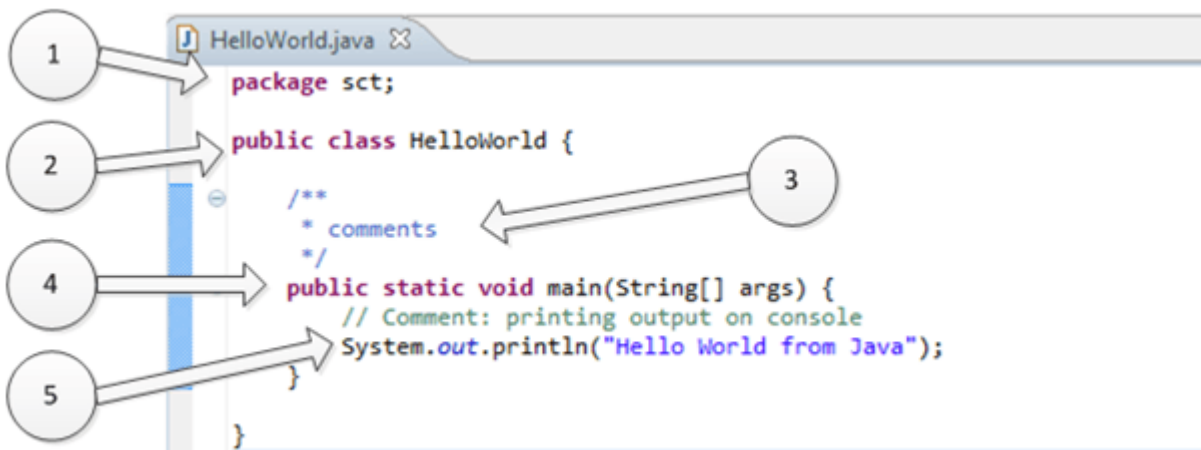
Core Java terminologies.

- **JDK (Java Development Kit):** JDK contains JRE along with various development tools like Java source compilers, Java libraries, Java debuggers, bundling and deployment tools
- **JVM (Java Virtual Machine):** ‘JVM’ is software that can be ported to various hardware platforms. JVM will become an instance of JRE at runtime of java program. Byte codes are the machine language for the JVM. Like a real computing machine, JVM has an instruction set which manipulates various memory areas at run time. Thus for different hardware platforms, one has corresponding the implementation of JVM available as vendor supplied JREs.
- **JRE (Java Runtime Environment):** It is also called JVM implementation. It is part of JDK but can be used independently to run any compiled java program (byte code).
- **Java API (Application Programming Interface) :** These are Set of classes written using Java programming language which runs on JVM. These classes will help programmers by providing

standard methods like reading from the console, writing to the console, saving objects in data structure etc.

1.5 JAVA PROGRAM STRUCTURE

Let's use the example of 'HelloWorld' Java program (as shown in the screenshot picture below) to understand structure and features of the class. This program is written on few lines, and its only task is to print "Hello World from Java" on the screen



1. "package sct":

It is package declaration statement. The package statement defines a namespace in which classes are stored. The package is used to organize the classes based on functionality. If you omit the package statement, the class names are put into the default package, which has no name. Package statement cannot appear anywhere in the program. It must be the first line of your program or you can omit it.

2. "public class HelloWorld":

This line has various aspects of java programming.

a. public: This is access modifier keyword which tells compiler access to class. Various values of access modifiers can be public, protected, private or default (no value).

b. class: This keyword used to declare a class. Name of class (HelloWorld) followed by this keyword.

3. Comments section:

We can write comments in java in two ways.

- a. Line comments: It starts with two forward slashes (//) and continues to the end of the current line. Line comments do not require an ending symbol.
- b. Block comments start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*).Block comments can also extend across as many lines as needed.

4. “public static void main (String [] args)”:

Its method (Function) named main with string array as an argument.

- a. public: Access Modifier
- b. static: static is a reserved keyword which means that a method is accessible and usable even though no objects of the class exist.
- c. void: This keyword declares nothing would be returned from the method. The method can return any primitive or object.
- d. Method content inside curly braces. { } asdf;sd

5. System.out.println("Hello World from Java") :

- a. System: It is the name of Java utility class.
- b. out:It is an object which belongs to System class.
- c. println: It is utility method name which is used to send any String to the console.
- d. “Hello World from Java”: It is String literal set as argument to println method.

1.6 COMMON PROGRAMMING GUIDELINES:

- Java identifiers must start with a letter, a currency character (\$), or a connecting character such as the underscore (_). Identifiers cannot start with a number. After first character identifiers can contain any combination of letters,currency characters, connecting characters, or numbers. For example,
 - int variable1 = 10 ; //This is valid
 - int 4var =10 ; // this is invalid, identifier can't start with digit.
- Identifiers, method names, class names are case-sensitive; var and Var are two different identifiers.
- You can't use Java keywords as identifiers, Below table shows a list of Java keywords.

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while
assert	enum				

- **Classes and interfaces:** The first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "camelCase").
- **Methods:** The first letter should be lowercase, and then normal camelCase rules should be used. For example:
 - getBalance
 - doCalculation
 - setCustomerName
- **Variables:** Like methods, the camelCase format should be used, starting with a lowercase letter. Sun recommends short, meaningful names, which sounds good to us. Some examples:
 - buttonWidth
 - accountBalance
 - empName
- **Constants:** Java constants are created by marking variables static and final. They should be named using uppercase letters with underscore characters as separators:
 - MIN_HEIGHT
- There can be only one public class per source code file.
- Comments can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here.
- If there is a public class in a file, the name of the file must match the name of the public class. For example, a class declared as "public class Dog { }" must be in a source code file named Dog.java.

2.0 PRIMITIVE DATA TYPE IN JAVA

Not everything in Java is an object. There is a special group of data types (also known as primitive types) that will be used quite often in programming. Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types which can be put in four groups

- **Integers:** This group includes byte, short, int, and long, which are for whole-valued signed numbers.
- **Floating-point numbers:** This group includes float and double, which represent numbers with fractional precision.
- **Characters:** This group includes char, which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes boolean, which is a special type for representing true/false values.

Let's discuss each in details:

byte

The smallest integer type is byte. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. Byte variables are declared by use of the byte keyword. For example, the following declares and initialize byte variables called b:

```
byte b =100;
```

short:

The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters. Following example declares and initialize short variable called s:

```
short s =123;
```

int:

The most commonly used integer type is int. It is a signed 32-bit type that has a range from -2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use long instead.

```
int v = 123543;  
int calc = -9876345;
```

long:

long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use of this data type might be in banking application when large amount is to be calculated and stored.

```
long amountVal = 1234567891;
```

float:

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example interest rate calculation or calculating square root. The float data type is a single-precision 32-bit IEEE 754 floating point. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision. The declaration and initialization syntax for float variables given below, please note “f” after value initialization.

```
float intrestRate = 12.25f;
```

double:

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as `sin()`, `cos()`, and `sqrt()`, return double values. The declaration and initialization syntax for double variables given below, please note “d” after value initialization.

```
double sineVal = 12345.234d;
```

boolean:

The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This is the type returned by all relational operators, as in the case of `a < b`. boolean is also the type required by the conditional expressions that govern the control statements such as if or while.

```
boolean flag = true;  
booleanval = false;
```

char:

In Java, the data type used to store characters is char. The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive). There are no negative chars.

```
char ch1 = 88; // code for X  
char ch2 = 'Y';
```

Primitive Variables can be of two types:

(1) Class level (instance) variable:

It's not mandatory to initialize Class level (instance) variable. If we do not initialize instance variable compiler will assign default value to it. Generally speaking, this default will be zero or null, depending on the data type. Relying on such default values, however, is generally considered bad coding practice. The following chart summarizes the default values for the above data types.

Primitive Data Type	Default Value 3.6
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

(2) Method local variable:

Method local variables have to be initialized before using it. The compiler never assigns a default value to an un-initialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an un-initialized local variable will result in a compile-time error.

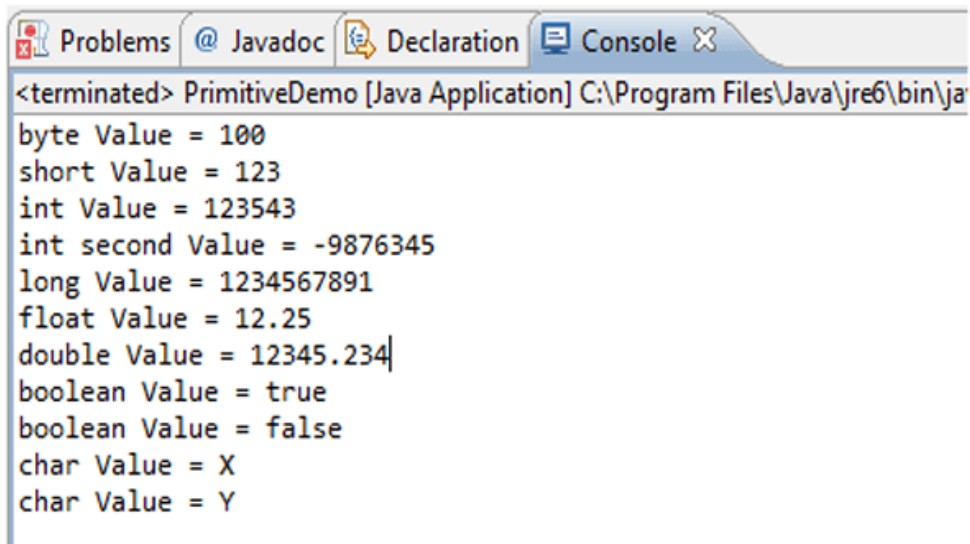
Let's See simple java program which declares, initialize and print all of primitive types.

Java class PrimitiveDemo as below:

```
package primitive;
public class PrimitiveDemo {
    public static void main(String[] args) {
        byte b =100;
        short s =123;
        int v = 123543;
        int calc = -9876345;
        long amountVal = 1234567891;
        float intrestRate = 12.25f;
        double sineVal = 12345.234d;
        boolean flag = true;
        boolean val = false;
        char ch1 = 88; // code for X
        char ch2 = 'Y';
        System.out.println("byte Value = "+ b);
        System.out.println("short Value = "+ s);
        System.out.println("int Value = "+ v);
        System.out.println("int second Value = "+ calc);
        System.out.println("long Value = "+ amountVal);
        System.out.println("float Value = "+ intrestRate);
    }
}
```

```
        System.out.println("double Value = "+ sineVal);  
        System.out.println("boolean Value = "+ flag);  
        System.out.println("boolean Value = "+ val);  
        System.out.println("char Value = "+ ch1);  
        System.out.println("char Value = "+ ch2);  
    }  
}
```

Output of above PrimitiveDemo class as below:



```
<terminated> PrimitiveDemo [Java Application] C:\Program Files\Java\jre6\bin\ja  
byte Value = 100  
short Value = 123  
int Value = 123543  
int second Value = -9876345  
long Value = 1234567891  
float Value = 12.25  
double Value = 12345.234  
boolean Value = true  
boolean Value = false  
char Value = X  
char Value = Y
```

3.0 PRINTING OUTPUT IN JAVA

Printing output in Java is a fundamental aspect of programming that allows developers to display information to the user, debug applications, and provide feedback. In Java, there are several ways to print output, with varying levels of complexity and features. Here's an extensive look at the various methods available for printing output in Java.

1. Using System.out

The most common method for printing output in Java is through the System.out object. This object is an instance of `PrintStream`, and it provides several methods to output data to the console.

Basic Methods

- **print():** This method prints the specified data to the console without adding a newline character at the end.

```
System.out.print("Welcome to, AIU!"); // Output: Welcome to,  
AIU!
```

- **println():** This method prints the specified data followed by a newline character, moving the cursor to the next line.

```
System.out.println("Welcome to, AIU!"); // Output: Welcome to,  
AIU!
```

- **printf():** This method allows for formatted output, similar to C's `printf`. It takes a format string followed by arguments.

```
int number = 10;  
System.out.printf("The number is: %d\n", number); // Output:  
The number is: 10
```

2. Formatting Output

Java provides a robust way to format strings, especially with `String.format()` and the `printf()` method.

Using `String.format()`

You can create a formatted string and then print it using `System.out.println()`.

```
String formattedString = String.format("The value of pi is  
approximately %.2f", Math.PI);  
System.out.println(formattedString); // Output: The value of pi is  
approximately 3.14
```

3. Using the `java.util.Formatter` Class

For more complex formatting, Java has a `Formatter` class that provides various methods to format data.

```
import java.util.Formatter;
```

```
Formatter formatter = new Formatter();  
formatter.format("The value of e is approximately %.2f", Math.E);  
System.out.println(formatter); // Output: The value of e is  
approximately 2.72  
formatter.close();
```

4. Printing to Files

Java also provides ways to print output to files using the `PrintWriter` and `FileWriter` classes.

```
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

try (PrintWriter writer = new PrintWriter(new
FileWriter("output.txt"))) {
    writer.println("Hello, File!");
} catch (IOException e) {
    e.printStackTrace();
}
```

5. Logging

For more advanced output requirements, especially in production applications, you might want to use logging frameworks such as `java.util.logging`, `Log4j`, or `SLF4J`. These frameworks provide more control over output, allowing for different logging levels (info, debug, error) and output destinations (console, files, remote servers).

Example Using `java.util.logging`

```
import java.util.logging.Logger;
import java.util.logging.Level;

public class LoggingExample {
    private static final Logger logger =
Logger.getLogger(LoggingExample.class.getName());

    public static void main(String[] args) {
        logger.info("This is an info message.");
        logger.warning("This is a warning message.");
    }
}
```

6. Console Output

Java also provides `System.console()`, which can be useful for reading user input and printing output in console applications. However, this method may return null if the Java application is not connected to a console.

```
import java.io.Console;
```



```
public class ConsoleExample {
    public static void main(String[] args) {
        Console console = System.console();
        if (console == null) {
            System.out.println("No console available");
            return;
        }
        console.printf("Hello, Console!%n");
    }
}
```

7. Special Characters

When printing output, you can use special escape sequences to format the text further.

- `\n`: New line
- `\t`: Tab
- `\\`: Backslash
- `\"`: Double quote

Example

```
System.out.println("Hello,\nWorld!"); // Output: Hello,
//                                     World!
```

4.0 ACCEPTING INPUT FROM USER IN JAVA

Java's vast code libraries are one of its advantages for developers. The Scanner class is one extremely helpful class that manages user input. The `java.util` library contains the Scanner class. Include a reference to the Scanner class in your code to use it. The keyword `import` is used to accomplish this.

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

The import statement needs to go just above the Class statement:

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

```
public class StringVariables {
```

```
}
```

This instructs Java that you wish to use the Scanner class, which is part of the java.util library, and that you want to use it in that specific library.

Creating an object from the Scanner class is the next step. (A class consists just of code. Until you construct a new object out of it, it accomplishes nothing.)

To create a new Scanner object the code is this:

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

Thus, a **Scanner** variable will be created instead of an int or a String variable; which will be named **user_input**. The keyword **new** comes after the equals symbol. Using this, new objects from a class can be created. We are building an object that belongs to the Scanner class. We must inform Java that this will be System Input (System.in) in between round brackets.

To get the user input, you can call into action one of the many methods available to your new Scanner object. One of these methods is called **next**. This gets the next string of text that a user types on the keyboard:

```
String first_name;  
first_name = user_input.next( );
```

After typing a dot following our **user_input** object, a popup will appear showing the available methods. Double-click on "**next**" and then type a semicolon to end the line. Additionally, we can print a message to prompt the user:

```
String first_name;  
System.out.print("Enter your first name: ");  
first_name = user_input.next( );
```

Notice that **print** is being used instead of **println** as in the previous instance. The distinction between the two is that **println** advances the cursor to a new line after displaying the output, while **print** keeps it on the same line.

Additionally, a prompt for a family name will be included:

```
String family_name;  
System.out.print("Enter your family name: ");  
family_name = user_input.next( );
```

This is the same code, except that java will now store whatever the user types into our **family_name** variable instead of our **first_name** variable.

To print out the input, the following can be added:

```
String full_name;  
full_name = first_name + " " + family_name;
```

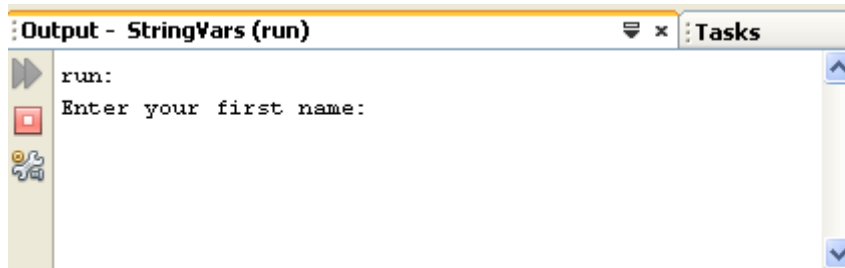
```
System.out.println("You are " + full_name);
```

Another String variable, **full_name** will be set up. Whatever is in the two variables **first_name** and **family_name** will be stored. In between the two, we've added a space. The final line prints it all out in the Output window.

So adapt your code so that it matches that in the image below:

```
package stringvars;  
  
import java.util.Scanner;  
  
public class StringVariables {  
  
    public static void main(String[] args) {  
  
        Scanner user_input = new Scanner(System.in);  
  
        String first_name;  
        System.out.print("Enter your first name: ");  
        first_name = user_input.next();  
  
        String family_name;  
        System.out.print("Enter your family name: ");  
        family_name = user_input.next();  
  
        String full_name;  
        full_name = first_name + " " + family_name;  
  
        System.out.println("You are " + full_name);  
  
    }  
}
```

On running the programme the Output window displays the following:

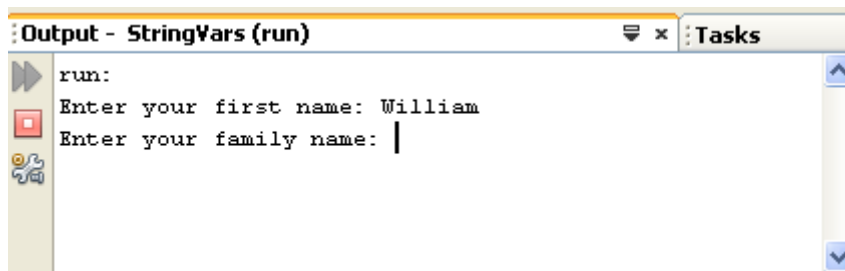


```
run:
Enter your first name:
```

Java is now pausing until something is entered on your keyboard. It won't progress until you hit the enter key. So left click after "Enter your first name:" and you'll see your cursor flashing away. Type a first name, and then hit the enter key on your keyboard.

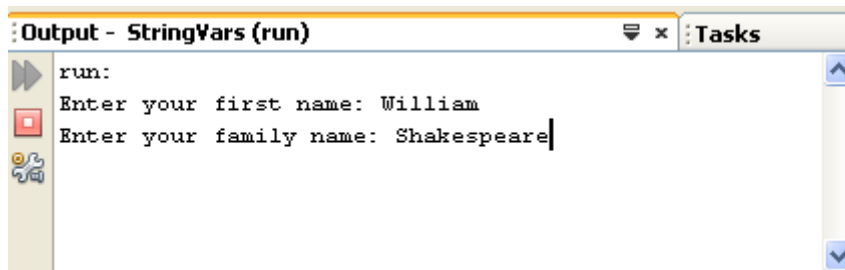
After the enter key is hit, java will take whatever was typed and store it in the variable name to the left of the equals sign. For us, this was the variable called `first_name`.

The programme then moves on to the next line of code:



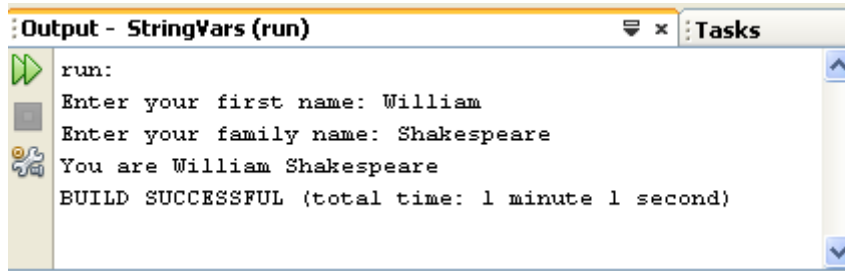
```
run:
Enter your first name: William
Enter your family name: |
```

Type a family name, and hit the enter key again:



```
run:
Enter your first name: William
Enter your family name: Shakespeare|
```

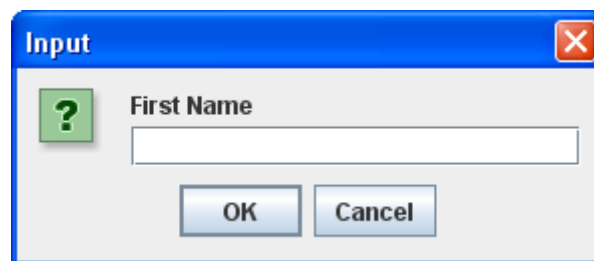
The user input has now finished, and the rest of the programme executes. This is the output of the two names. The final result should like this:



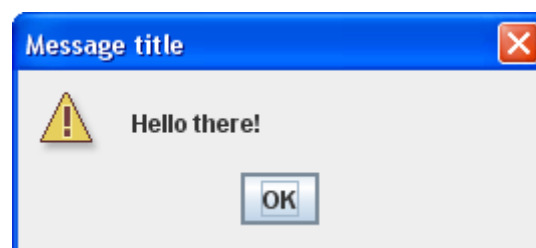
So the Scanner class is used to get input from a user. Whatever was typed was stored in variables. The result was then printed to the Output window.

5.0 JAVA OPTION PANES

Another useful class for accepting user input, and displaying results, is the JOptionPane class. This is located in the javax.swing library. The JOptionPane class allows you to have input boxes like this one:



And message boxes like this:



The first thing to do is to reference the library we want to use:

```
import javax.swing.JOptionPane;
```

This tells java that we want to use the JOptionPane class, located in the javax.swing library.

Add the import line to your new project, and your code window should look like something like this:

```
package userInput;
import javax.swing.JOptionPane;

public class InputBoxes {

    public static void main(String[] args) {

    }

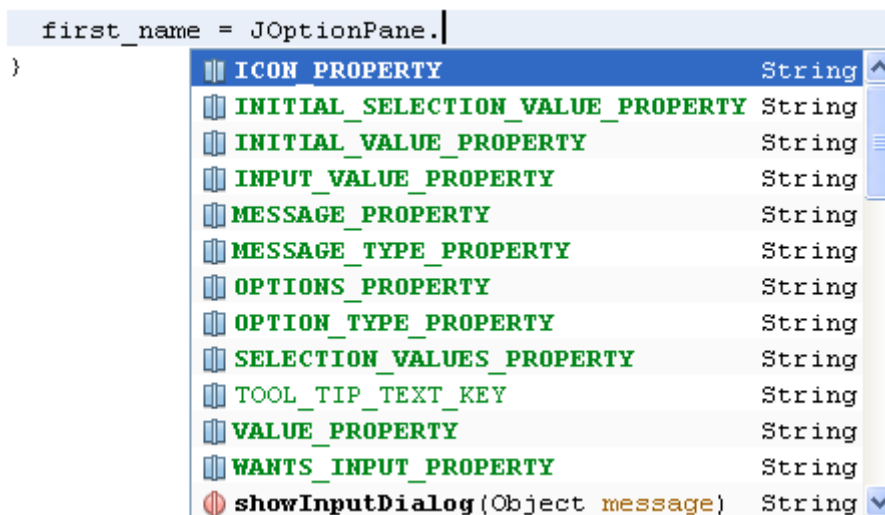
}
```

(The reason for the wavy underline is that we haven't used the JOptionPane class yet. It will go away once we do.)

To get an input box that the user can type into, we can use the **showInputDialog** method of JOptionPane. We'll store the input straight into a first name variable again, as done previously. So add the following line to your main method:

```
String first_name;
first_name = JOptionPane.showInputDialog("First Name");
```

As soon as full stop is typed after JOptionPane the following popup list will appear:



Double click **showInputDialog**. In between the round brackets of showInputDialog type the message that you want displayed above the input text box. We've typed "First name". Like all strings, it needs to go between double quotes.

Add the following code so that we can get the user's family name:

```
String family_name;
family_name = JOptionPane.showInputDialog("Family Name");
```

Join the two together, and add some text:

```
String full_name;  
full_name = "You are " + first_name + " " + family_name;
```

To display the result in a message box, add the following:

```
JOptionPane.showMessageDialog( null, full_name );
```

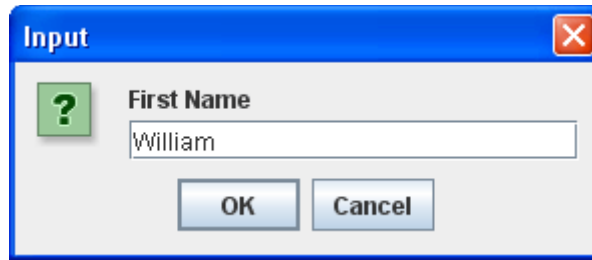
This time, `showMessageDialog` from the popup list should be used. Inside the parentheses, the keyword `null` is placed first, signifying that the message box isn't associated with any other part of the program. After a comma, the text to be displayed in the message box is added. The complete code should look like this:

```
package userinput;  
import javax.swing.JOptionPane;  
  
public class InputBoxes {  
  
    public static void main(String[] args) {  
  
        String first_name;  
        first_name = JOptionPane.showInputDialog("First Name");  
  
        String family_name;  
        family_name = JOptionPane.showInputDialog("Family Name");  
  
        String full_name;  
        full_name = "You are " + first_name + " " + family_name;  
  
        JOptionPane.showMessageDialog(null, full_name);  
        System.exit(0);  
  
    }  
}
```

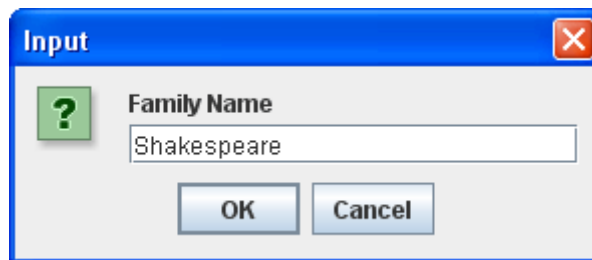
The line at the bottom of the code: **System.exit(0);** ensures that the programme exits. It also help in removing all the created objects from memory.

When the code is run the First Name input box will be shown. Type something into it, then click OK:

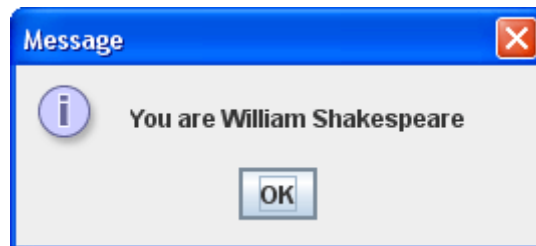
TIP: (Another quick way to run your programme in NetBeans is by right-clicking anywhere inside of the coding window. From the menu that appears, select **Run File.**)



When the Family Name input box appears, type a family name and click OK:



After you click OK, the message box will display:



Click OK to end the programme.

6.0 VARIABLE DECLARATION AND INITIALIZATION IN JAVA

Java is a statically typed language, which means you must specify the type of each variable explicitly. In Java, you need to declare the type of the variable before you use it.

Syntax for declaring a variable:

```
type variableName;
```

Syntax for initializing a variable:

```
variableName = value;
```

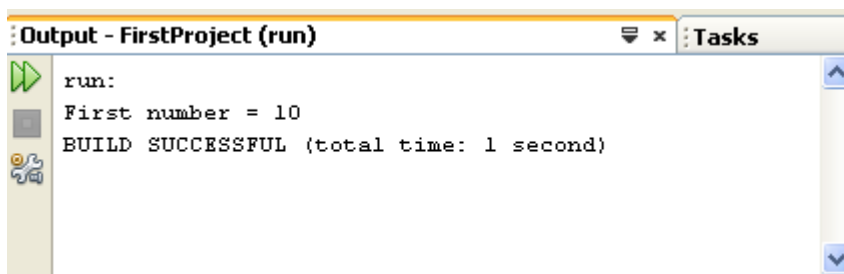
Declaration and initialization in one line:


```
type variableName = value;
```

Example:

```
int age = 25;  
float temperature = 98.6f;  
boolean isJavaFun = true;  
char initial = 'J';
```

Number used in the last code line above joining the string, first number and variable, first number
i.e The output for running the above code is shown in the image below



Let's create a simple Java code that can add up the value of two variables

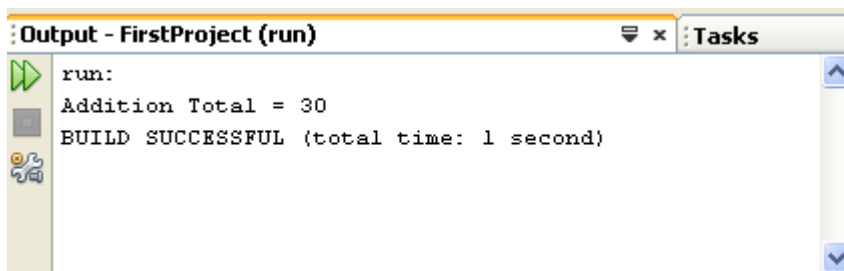
Add up these two first

`answer = first_number + second_number;`

store the answer here

`answer = first_number + second_number;`

```
public static void main(String[] args) {  
  
    int first_number, second_number, answer;  
  
    first_number = 10;  
    second_number = 20;  
    answer = first_number + second_number;  
  
    System.out.println("Addition Total = " + answer );  
}
```



```
Output - FirstProject (run) x Tasks  
run:  
Addition Total = 30  
BUILD SUCCESSFUL (total time: 1 second)
```

Reference data types

Reference data types refer to objects and arrays. They are used to store references to data rather than the data itself.

Example:

```
String name = "John Doe";  
int[] numbers = {1, 2, 3, 4, 5};
```

Variable Scope

Variables in Java have different scopes depending on where they are declared:

- **Local Variables:** Declared inside a method or block. They must be initialized before use.

- **Instance Variables:** Declared inside a class but outside any method. They are initialized to default values (e.g., 0 for integers, null for objects).
- **Class Variables (Static Variables):** Declared with the `static` keyword. Shared among all instances of a class.

Example:

```
public class MyClass {  
    // Class variable  
    static int classVariable = 10;  
  
    // Instance variable  
    int instanceVariable;  
  
    public void myMethod() {  
        // Local variable  
        int localVariable = 5;  
        System.out.println(localVariable);  
    }  
}
```

Constants

Constants are variables whose value cannot be changed once assigned. Constants can be defined using the *final* keyword.

Example:

```
public class Constants {  
    public static final double PI = 3.14159;  
}
```

Type Casting

Sometimes you need to convert a variable from one type to another. This process is known as type casting.

Implicit Casting (Widening Conversion):

```
int intValue = 10;  
double doubleValue = intValue; // Automatically done
```

Explicit Casting (Narrowing Conversion):

```
double doubleValue = 10.5;  
int intValue = (int) doubleValue; // Manual casting
```

String Manipulation

In Java, strings are objects of the `String` class. They are immutable, meaning their values cannot be changed once created.

Example:

```
String greeting = "Hello";  
String name = "World";  
String message = greeting + " " + name; // Concatenation  
System.out.println(message); // Output: Hello World
```

Basic rules in working with Variables

- **Use meaningful variable names:** Names should describe the purpose of the variable.
- **Follow naming conventions:** For instance, use camelCase for variable names (`myVariable`).
- **Initialize variables:** Ensure variables are initialized before use to avoid unpredictable behavior.

7.0 JAVA ARRAY:

An array is a group of similar typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. The array is a simple type of data structure which can store primitive variable or objects. For example, imagine if you had to store the result of six subjects we can do it using an array. To create an array value in Java, you use the `new` keyword, just as you do to create an object.

7.1 Defining and constructing one dimensional array

```
int resultArray[] = new int[6];
```

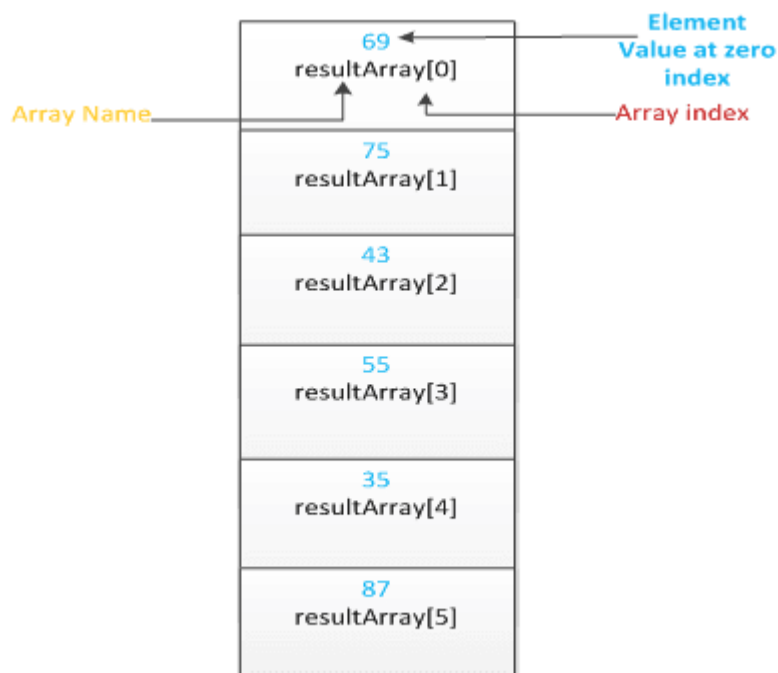


Here, type specifies the type of variables (`int`, `boolean`, `char`, `float` etc) being stored, size specifies the number of elements in the array, and arrayname is the variable name that is the reference to the array. Array size must be specified while creating an array. If you are creating a `int[]`, for example, you must specify how many `int` values you want it to hold (in above statement `resultArray[]` is having size 6 `int` values). Once an array is created, it can never grow or shrink.

7.2 Initializing array: You can initialize specific element in the array by specifying its index within square brackets. All array indexes start at zero.

```
resultArray[0]=69;
```

This will initialize first element (index zero) of resultArray[] with integer value 69. Array elements can be initialized/accessed in any order. In memory, it will create a structure similar to below figure.



7.3 Array Literals

The null literal used to represent the absence of an object can also be used to represent the absence of an array. For example:

```
String [] name = null;
```

In addition to the null literal, Java also defines a special syntax that allows you to specify array values literally in your programs. This syntax can be used only when declaring a variable of array type. It combines the creation of the array object with the initialization of the array elements:

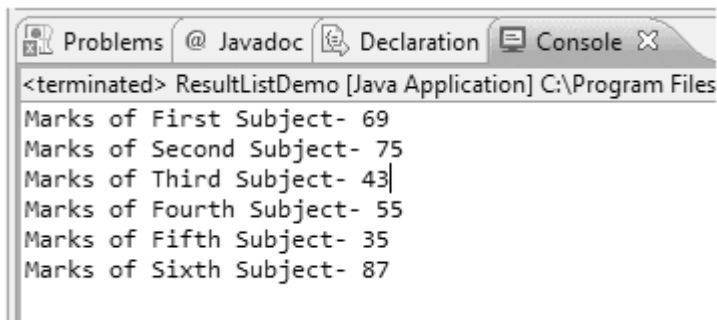
```
String[] daysOfWeek = {"Sunday", "Monday", "Tuesday",  
"Wednesday", "Thursday", "Friday", "Saturday"};
```

This creates an array that contains the seven string element representing days of the week within the curly braces. Note that we don't use the new keyword or specify the type of the array in this array literal syntax. The type is implicit in the variable declaration of which the initializer is a part. Also, the array length is not specified explicitly with this syntax; it is determined implicitly by counting the number of elements listed between the curly braces.

Let's see sample java program to demonstrate this concept. This program will help to understand initializing and accessing specific array elements.

```
package arrayDemo;
import java.util.Arrays;
public class ResultListDemo {
    public static void main(String[] args) {
        //Array Declaration
        int resultArray[] = new int[6];
        //Array Initialization
            resultArray[0]=69;
            resultArray[1]=75;
            resultArray[2]=43;
            resultArray[3]=55;
            resultArray[4]=35;
            resultArray[5]=87;
        //Array elements access
        System.out.println("Marks of First Subject- "+ resultArray[0]);
        System.out.println("Marks of Second Subject- "+ resultArray[1]);
        System.out.println("Marks of Third Subject- "+ resultArray[2]);
        System.out.println("Marks of Fourth Subject- "+ resultArray[3]);
        System.out.println("Marks of Fifth Subject- "+ resultArray[4]);
        System.out.println("Marks of Sixth Subject- "+ resultArray[5]);
    }
}
```

Output:



```
<terminated> ResultListDemo [Java Application] C:\Program Files
Marks of First Subject- 69
Marks of Second Subject- 75
Marks of Third Subject- 43
Marks of Fourth Subject- 55
Marks of Fifth Subject- 35
Marks of Sixth Subject- 87
```

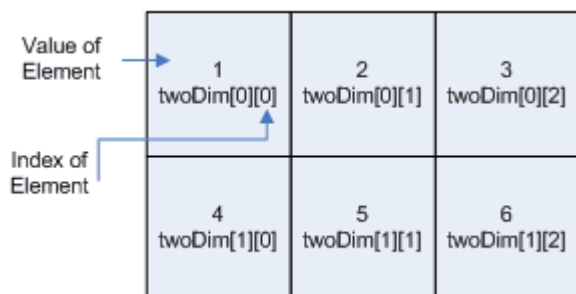
Alternative syntax for declaring, initializing of array in the same statement

```
int [] resultArray = {69,75,43,55,35,87};
```

7.4 Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoDim. This will create a matrix of the size 2x3 in memory.

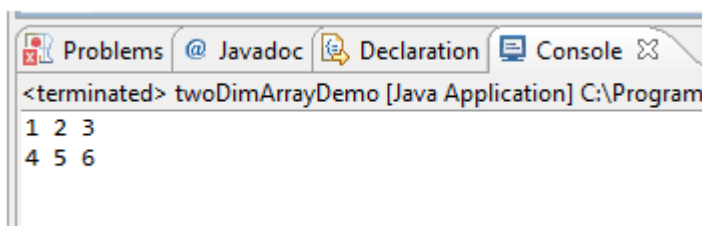
```
int twoDim[][] = new int[2][3];
```



Let's have look at below program to understand 2-dimentional array

```
package arrayDemo;
public class twoDimArrayDemo {
    public static void main (String []args){
        int twoDim [][] = new int [2][3];
        twoDim[0][0]=1;
        twoDim[0][1]=2;
        twoDim[0][2]=3;
        twoDim[1][0]=4;
        twoDim[1][1]=5;
        twoDim[1][2]=6;
        System.out.println(twoDim[0][0] + " " + twoDim[0][1] + " " +
twoDim[0][2]);
        System.out.println(twoDim[1][0] + " " + twoDim[1][1] + " " +
twoDim[1][2]);
    }
}
```

Output:



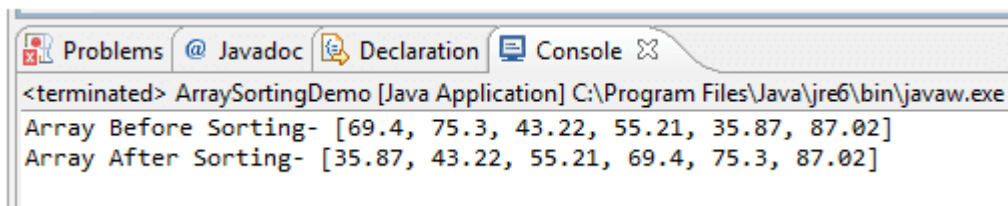
7.5 Inbuilt Helper Class (java.util.Arrays) for Arrays Manipulation:

Java provides very important helper class (java.util.Arrays) for array manipulation. This class has many utility methods like array sorting, printing values of all array elements, searching of an element, copy one array into another array etc. Let's see sample program to understand this class for better programming. In below program float array has been declared. We are printing the array elements before sorting and after sorting.

```
package arrayDemo;
import java.util.Arrays;
```

```
public class ArraySortingDemo {
    public static void main(String[] args) {
        //Declaring array of float elements
        float [] resultArray = {69.4f,75.3f,43.22f,55.21f,35.87f,87.02f};
        System.out.println("Array Before Sorting- " +
Arrays.toString(resultArray));
        //below line will sort the array in ascending order
        Arrays.sort(resultArray);
        System.out.println("Array After Sorting- " +
Arrays.toString(resultArray));
    }
}
```

Output:



```
<terminated> ArraySortingDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Array Before Sorting- [69.4, 75.3, 43.22, 55.21, 35.87, 87.02]
Array After Sorting- [35.87, 43.22, 55.21, 69.4, 75.3, 87.02]
```

Similar to “java.util.Arrays” System class also has a functionality of efficiently copying data from one array to another. Syntax as below,

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

The two Object arguments specify the array to copy from and the array to copy to. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

Important points:

- You will get “ArrayIndexOutOfBoundsException” if you try to access an array with an illegal index, that is with a negative number or with a number greater than or equal to its size.
- Arrays are widely used with loops (for loops, while loops). There will be more sample program of arrays with loops tutorial.

8.0 USING OPERATORS IN JAVA:

Before delving into control structure in programming it’s important to look at in Java. Java has the following operators:

1. Conditional or Relational Operators
2. Arithmetic Operators

3. Logical Operators

Assignment Operators

8.1 Assignment Operators

Operator	Name	Example	Equivalent
=	Assignment	X=10	A holds 10
+=	Addition assignment	i+=5;	i=i+5
-=	Subtraction assignment	j-=10;	j=j-10;
=	Multiplication assignment	k=2;	k=k*2;
/=	Division assignment	x/=10;	x=x/10;
%=	Remainder assignment	a%=4;	a=a%4;

Below is the sample program explaining assignment operators:

```
public class AssignmentOperatorDemo {
    public static void main(String[] args) {
        //Literal Primitive Assignment
        byte b = 25;
        System.out.println("Primitive byte Assignment- "+ b);
        //Assigning one primitive to another primitive
        byte c =b;
        System.out.println("Primitive byte Assignment from another byte
variable- "+ c);
        //Literal assignment based on arithmetic operation
        int a = 23+b;
        System.out.println("Primitive int Assignment from arithmetic
operation- "+ a);
        //Implicit Casting of variable x and y
        short s = 45;
        int x = b;
        int y = s;
        System.out.println("Implicit Casting of byte to int- "+ x);
        System.out.println("Implicit Casting of short to int- "+ y);
        //Short-Hand Assignment Operators
        int i = 10;
        i+=10;
        System.out.println("Addition Oprator- "+ i);
        i-=10;
        System.out.println("Subtraction Oprator- "+ i);
        i*=10;
        System.out.println("Multiplication Operator- " + i);
        i/=10;
        System.out.println("Division Operator- " + i);
        i%=3;
        System.out.println("Reminder Operator- " + i);
    }
}
```

Output:

```

Problems @ Javadoc Declaration Console X
<terminated> AssignmentOperatorDemo [Java Application] C:\Program Files\Java
Primitive byte Assignment- 25
Primitive byte Assignment from another byte variable- 25
Primitive int Assignment from arithmetic operation- 48
Implicit Casting of byte to int- 25
Implicit Casting of short to int- 45
Addition Operator- 20
Subtraction Operator- 10
Multiplication Operator- 100
Division Operator- 10
Reminder Operator- 1
    
```

8.2 Arithmetic Operators

Operator	Use	Description	Example
+	$x + y$	Adds x and y	float num = 23.4 + 1.6; // num=25
-	$x - y$	Subtracts y from x	long n = 12.456 - 2.456; //n=10
	-x	Arithmetically negates x	int i = 10; -i; // i = -10
*	$x * y$	Multiplies x by y	int m = 10*2; // m=20
/	x / y	Divides x by y	float div = 20/100 ; // div = 0.2
%	$x \% y$	Computes the remainder of dividing x by y (Modulus)	int rm = 20/3; // rm = 2

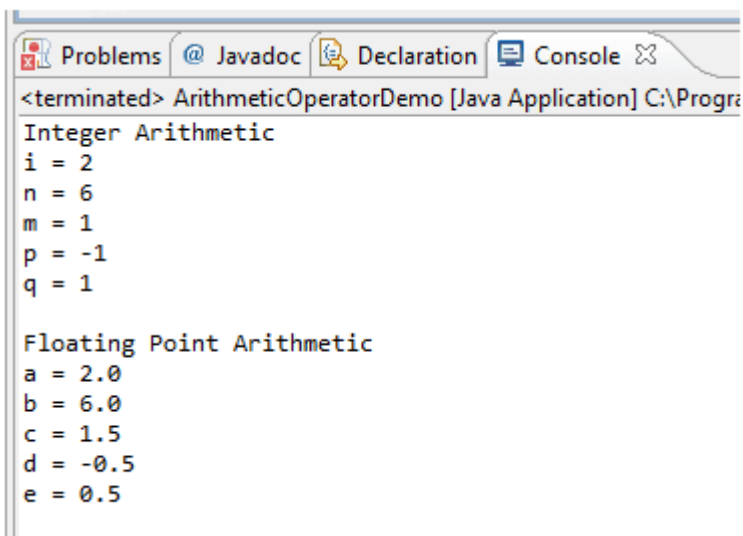
The following simple program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

```

public class ArithmeticOperatorDemo {
    // Demonstrate the basic arithmetic operators.
    public static void main(String args[]) {
        // arithmetic using integers
        System.out.println("Integer Arithmetic");
        int i = 1 + 1;
        int n = i * 3;
        int m = n / 4;
        int p = m - i;
        int q = -p;
        System.out.println("i = " + i);
        System.out.println("n = " + n);
        System.out.println("m = " + m);
        System.out.println("p = " + p);
        System.out.println("q = " + q);
        // arithmetic using doubles
        System.out.println("\nFloating Point Arithmetic");
        double a = 1 + 1;
        double b = a * 3;
    }
}
    
```

```
        double c = b / 4;  
        double d = c - a;  
        double e = -d;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
        System.out.println("d = " + d);  
        System.out.println("e = " + e);  
    }  
}
```

Output:



```
<terminated> ArithmeticOperatorDemo [Java Application] C:\Progra  
Integer Arithmetic  
i = 2  
n = 6  
m = 1  
p = -1  
q = 1  
  
Floating Point Arithmetic  
a = 2.0  
b = 6.0  
c = 1.5  
d = -0.5  
e = 0.5
```

The program below demonstrate use of modulus:

```
public class RemainderDemo {  
    public static void main (String [] args) {  
        int x = 15;  
        int int_remainder = x % 10;  
        System.out.println("The result of 15 % 10 is the "  
            + "remainder of 15 divided by 10. The remainder is "  
            + int_remainder);  
        double d = 15.25;  
        double double_remainder= d % 10;  
        System.out.println("The result of 15.25 % 10 is the "  
            + "remainder of 15.25 divided by 10. The remainder  
is " + double_remainder);  
    }  
}
```

Output:

```

Problems @ Javadoc Declaration Console
<terminated> RemainderDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (24-Mar-2013 10:32:48 AM)
The result of 15 % 10 is the remainder of 15 divided by 10. The remainder is 5
The result of 15.25 % 10 is the remainder of 15.25 divided by 10. The remainder is 5.25
    
```

Also, there are a couple of quirks to keep in mind regarding division by 0:

- A non-zero floating-point value divided by 0 results in a signed Infinity. 0.0/0.0 results is NaN.
- A non-zero integer value divided by integer 0 will result in ArithmeticException at runtime

8.3 Conditional or relational Operators

Operator	Description	Example (a=10, b=15)	Result
==	Equality operator	a==b	false
!=	Not Equal to operator	a!=b	true
>	Greater than	a>b	false
<	Less than	a<b	true
>=	Greater than or equal to	a>=b	false
<=	Less than or equal to	a<=b	true

The program below demonstrates the use of conditional operators

```

public class IfelseLadderDemo {
    public static void main(String[] args) {
        int a =120;
        if(a< 100){
            System.out.println("Variable is less than 100");
        }
        else if(a==100)
        {
            System.out.println("Variable is equal to 100");
        }
        else if (a>100)
        {
            System.out.println("Variable is greater than 100");
        }
    }
}
    
```

8.4 Logical OperatorsS/NO	OPERATOR	DESCRIPTION
1		Returns <code>true</code> if both operands are <code>true</code> .
2		Returns <code>true</code> if at least one of the operands is <code>true</code> .
3		
4		Inverts the value of a boolean expression

Here's a simple example demonstrating logical operators in a conditional statement:

```
public class LogicalOperatorsExample {
    public static void main(String[] args) {
        int age = 20;
        boolean hasLicense = true;

        if (age >= 18 && hasLicense) {
            System.out.println("You can drive.");
        } else {
            System.out.println("You cannot drive.");
        }
    }
}
```

9.0 CONTROL STRUCTURE IN JAVA

Control structures in programming generally are fundamental constructs that dictate the flow of execution of code. They enable programmers to create complex logic by controlling how statements are executed based on certain conditions or iterations. The main types of control structures are:

1. **Sequential Control:** The default mode where statements are executed in the order they appear.
2. **Selection (Conditional) Control:** Allows for decision-making in code. Common structures include:
 - **if** statements: Execute a block of code if a condition is true.
 - **else** statements: Provide an alternative block when the condition is false.
 - **switch** statements: Selects one of many code blocks to execute based on the value of a variable.
3. **Repetition (Looping) Control:** Executes a block of code multiple times. Common types include:
 - **for** loops: Repeat a block a specific number of times.
 - **while** loops: Continue executing as long as a condition remains true.
 - **do-while** loops: Similar to while, but the block is executed at least once.

Java as a programming language is not exempted in utilizing these concepts of control structure in defining the flow of code execution in a program. In all the codes that has been used so far,

sequential control structure has been used. At this point lets look at Conditional and Repetition Control Structure.

9.1 Java IF Statement (Conditional)

/condition

*Output/Display Message*Here's a simple example of If Statement in use:

Here is an expansion of the above code with an

```
package conditionallogic;

public class IFStatements {

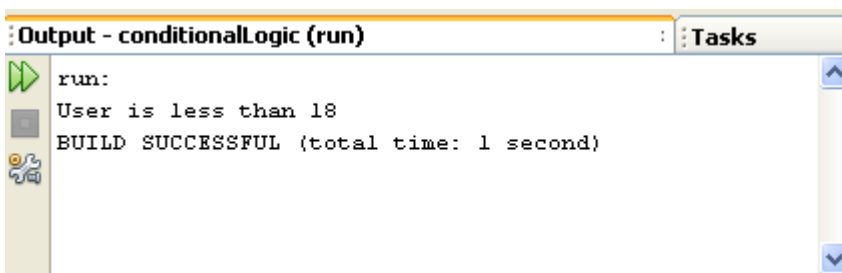
    public static void main(String[] args) {

        int user = 17;

        if (user < 18) {
            System.out.println("User is less than 18");
        }

    }
}
```

Here's the output on running the above code:



```
Output - conditionalLogic (run) | Tasks
run:
User is less than 18
BUILD SUCCESSFUL (total time: 1 second)
```

9.2 JAVA IF ... ELSE Statement:

The if...else statement in Java is used for decision-making in your code. It allows you to execute a block of code based on whether a condition is true or false

Here's the basic syntax:

```
if (condition) {  
    // Code to execute if the condition is true  
} else {  
    // Code to execute if the condition is false  
}
```

Let's look at an example where we check whether a number is even or odd:

```
import java.util.Scanner;  
  
public class EvenOddChecker {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
  
        System.out.print("Enter a number: ");  
        int number = scanner.nextInt();  
  
        if (number % 2 == 0) {  
            System.out.println(number + " is even.");  
        } else {  
            System.out.println(number + " is odd.");  
        }  
  
        scanner.close();  
    }  
}
```

Here is a breakdown explanation of the above code:

Input: The program prompts the user to enter a number.

Condition: The if statement checks if the number is divisible by 2 ($\text{number} \% 2 == 0$).

Output: If the condition is true, it prints that the number is even; otherwise, it prints that the number is odd.

9.3 Using the ELSE IF Statement:

The else if is used to check multiple conditions in a program:

Here is the syntax of it:

```
if (condition1) {  
    // Code for condition1  
} else if (condition2) {  
    // Code for condition2  
} else {  
    // Code if neither condition1 nor condition2 is true  
}
```

Here's a modified example that categorizes a number as positive, negative, or zero using the Else If statement:

```
import java.util.Scanner;

public class NumberSignChecker {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter a number: ");
        int number = scanner.nextInt();

        if (number > 0) {
            System.out.println(number + " is positive.");
        } else if (number < 0) {
            System.out.println(number + " is negative.");
        } else {
            System.out.println("The number is zero.");
        }

        scanner.close();
    }
}
```

9.4 Switch Statements in Java:

Here is an example code to demonstrate the use of switch statement:

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

    int user = 18;

    switch ( user ) {
        case 18:
            System.out.println("You're 18");
            break;
        case 19:
            System.out.println("You're 19");
            break;
        case 20:
            System.out.println("You're 20");
            break;
        default:
            System.out.println("You're not 18, 19 or 20");
    }
}
```

must be “spelt out”9.5 Loop in Java

Java loops are fundamental constructs that allow you to execute a block of code repeatedly based on certain conditions. There are three primary types of loops in Java: `for`, `while`, and `do-while`. Each serves different

use cases and has unique syntax and behavior. Let's explore these loops in detail, along with examples and common use cases.

9.6 For Loop in Java:

The *for* loop is typically used when the number of iterations is known beforehand. It consists of three parts: initialization, condition, and iteration.

Syntax:

```
for (initialization; condition; iteration) {  
    // Code to be executed  
}
```

Example:

```
for (int i = 0; i < 5; i++) {  
    System.out.println("Iteration: " + i);  
}
```

Explanation:

- **Initialization:** `int i = 0` initializes the loop counter.
- **Condition:** `i < 5` checks if the loop should continue.
- **Iteration:** `i++` increments the counter after each iteration.

Common Use Cases:

- Iterating over arrays or collections.
- Executing a block of code a specific number of times.

9.7 While Loop:

The *while* loop is used when the number of iterations is not known in advance and depends on a condition. The loop continues as long as the specified condition evaluates to true.

Syntax:

```
while (condition) {  
    // Code to be executed  
}
```

Example:

```
int i = 0;  
while (i < 5) {  
    System.out.println("Iteration: " + i);  
}
```

```
    i++;  
}
```

Explanation:

- The loop continues as long as $i < 5$ is true.
- The counter i is incremented within the loop.

Here's a Java program that demonstrates the use of a while loop. This program will prompt the user to enter a number and will keep asking for numbers until the user enters a negative number. The program will also calculate and display the sum of all positive numbers entered.

```
import java.util.Scanner;  
  
public class WhileLoopDemo {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
        int number;  
        int sum = 0;  
  
        System.out.println("Enter numbers to sum them up (enter a negative  
number to stop):");  
  
        // Using a while loop to continuously ask for input  
        while (true) {  
            number = scanner.nextInt(); // Read user input  
  
            if (number < 0) {  
                break; // Exit the loop if the number is negative  
            }  
  
            sum += number; // Add the number to the sum  
        }  
  
        System.out.println("The sum of all positive numbers entered is: " +  
sum);  
        scanner.close();  
    }  
}
```

Common Use Cases:

- Reading data until a certain condition is met (e.g., user input).
- Continuously checking for an event or a state.

9.8 Do While Loop:

In Java, a *do-while* loop is used to execute a block of code at least once and then repeat the loop as long as a specified condition is true. The syntax for a *do-while* loop is as follows:

```
do {  
    // Code to be executed  
} while (condition);
```

Key Points to Note When Using Do while statement

- The code inside the `do` block is executed first.
- After executing the block, the condition is checked.
- If the condition is true, the loop repeats; if false, the loop ends.

Here's a simple example that demonstrates the use of a do-while loop to print numbers from 1 to 5:

```
public class DoWhileExample {  
    public static void main(String[] args) {  
        int number = 1;  
  
        do {  
            System.out.println(number);  
            number++;  
        } while (number <= 5);  
    }  
}
```

Explanation of the above code:

1. The variable `number` is initialized to 1.
2. The `do` block prints the value of `number` and increments it.
3. The condition `number <= 5` is checked after the block executes.
4. The loop continues until `number` is greater than 5.

When to use the Do While statement:

- Use a do-while loop when you need to ensure that the loop body is executed at least once, such as when prompting user input.

10.0 OBJECT ORIENTED PROGRAMMING IN JAVA

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data and code: data in the form of fields (often known as attributes or properties) and code in the form of procedures (often known as methods). Java, being a fully object-oriented programming language, supports the four main principles of OOP: Object, Class, Method, encapsulation, inheritance, polymorphism, and abstraction.

10.1 Class in Java

A **class** is a blueprint for creating objects. It defines a datatype by bundling data (attributes) and methods (functions) that operate on that data. A class encapsulates the properties and behaviors of an object.

Syntax of Class

```
class ClassName {  
    // Attributes (fields)  
    type fieldName;  
  
    // Constructor  
    ClassName(parameters) {  
        // Initialization code  
    }  
  
    // Methods  
    returnType methodName(parameters) {  
        // Method body  
    }  
}
```

Program example on class

```
class Dog {  
    // Attributes  
    String name;  
    String breed;  
    int age;  
  
    // Constructor  
    public Dog(String name, String breed, int age) {  
        this.name = name;  
        this.breed = breed;  
        this.age = age;  
    }  
  
    // Method to display dog details  
    public void displayInfo() {  
        System.out.println("Dog Name: " + name);  
        System.out.println("Breed: " + breed);  
        System.out.println("Age: " + age);  
    }  
}
```

10.2 Object in Java

An **object** is an instance of a class. When a class is defined, no memory is allocated until an object of that class is created. Objects can represent real-world entities with states and behaviors.

Creating Object

```
public class Main {  
    public static void main(String[] args) {  
        // Creating an object of the Dog class  
        Dog myDog = new Dog("Buddy", "Golden Retriever", 3);  
  
        // Calling method on the object  
        myDog.displayInfo();  
    }  
}
```

```
}  
}
```

Output:

```
Dog Name: Buddy  
Breed: Golden Retriever  
Age: 3
```

10.3 Method in Java

A **method** is a block of code that performs a specific task. Methods are defined inside a class and can manipulate the object's attributes. Methods can be of different types, including instance methods, class methods (static methods), and more.

Defining Methods

```
class Calculator {  
    // Method to add two integers  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    // Method to multiply two integers  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
}
```

Using Methods

```
public class Main {  
    public static void main(String[] args) {  
        Calculator calculator = new Calculator();  
  
        // Using methods of the Calculator class  
        int sum = calculator.add(5, 3);  
        int product = calculator.multiply(4, 6);  
  
        System.out.println("Sum: " + sum);           // Output: Sum: 8  
        System.out.println("Product: " + product);  // Output: Product: 24  
    }  
}
```

10.4 Method Overloading

In Java, you can have multiple methods in the same class with the same name but different parameter lists. This is known as method overloading.

```
class MathUtils {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }
}
```

Let's combine everything into a single example that demonstrates classes, objects, and methods in action.

```
class Car {
    // Attributes
    String model;
    String color;
    int year;

    // Constructor
    public Car(String model, String color, int year) {
        this.model = model;
        this.color = color;
        this.year = year;
    }

    // Method to display car details
    public void displayInfo() {
        System.out.println("Model: " + model);
        System.out.println("Color: " + color);
        System.out.println("Year: " + year);
    }

    // Method to start the car
    public void start() {
        System.out.println(model + " is starting.");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating an object of the Car class
        Car myCar = new Car("Toyota Camry", "Blue", 2022);

        // Calling methods on the Car object
        myCar.displayInfo();
        myCar.start();
    }
}
```

Output:

Model: Toyota Camry
Color: Blue
Year: 2022
Toyota Camry is starting.

10.5 Encapsulation

Encapsulation is the bundling of data and methods that operate on that data within a single unit, or class. It restricts access to certain components, which helps to prevent accidental interference and misuse of the methods and data.

Example:

```
class BankAccount {
    private String accountNumber;
    private double balance;

    public BankAccount(String accountNumber) {
        this.accountNumber = accountNumber;
        this.balance = 0.0;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println("Deposited: " + amount);
        } else {
            System.out.println("Deposit amount must be positive.");
        }
    }

    public void withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            System.out.println("Withdrew: " + amount);
        } else {
            System.out.println("Insufficient funds or invalid amount.");
        }
    }

    public double getBalance() {
        return balance;
    }
}
```

In this example, the *BankAccount* class encapsulates the properties *accountNumber* and *balance*. These properties are private, meaning they cannot be accessed directly from outside the class. Instead, public methods like *deposit*, *withdraw*, and *getBalance* are provided for interacting with these properties.

10.6 Inheritance

Inheritance is a mechanism in which one class can inherit the fields and methods of another class. This promotes code reusability.

Example:

```
class SavingsAccount extends BankAccount {
    private double interestRate;

    public SavingsAccount(String accountNumber, double interestRate) {
        super(accountNumber); // Call the constructor of the superclass
        this.interestRate = interestRate;
    }

    public void applyInterest() {
        double interest = getBalance() * interestRate / 100;
        deposit(interest); // Use the deposit method from the parent class
        System.out.println("Applied interest: " + interest);
    }
}
```

In this example, *SavingsAccount* extends *BankAccount*, inheriting its methods and properties while also introducing a new method *applyInterest* specific to savings accounts.

10.7 Polymorphism

Polymorphism allows methods to do different things based on the object that it is acting upon, which can be achieved through method overloading and method overriding.

```
class CheckingAccount extends BankAccount {
    public CheckingAccount(String accountNumber) {
        super(accountNumber);
    }

    @Override
    public void withdraw(double amount) {
        // Additional check for overdraft
        if (amount > getBalance()) {
            System.out.println("Overdraft allowed, with withdrawal: " + amount);
        } else {
            super.withdraw(amount);
        }
    }
}
```

Here, *CheckingAccount* overrides the *withdraw* method of the *BankAccount* class. If the amount to be withdrawn exceeds the balance, it allows the overdraft.

Example: Method Overloading

```
class Calculator {
```



```
public int add(int a, int b) {  
    return a + b;  
}  
  
public double add(double a, double b) {  
    return a + b;  
}  
  
public int add(int a, int b, int c) {  
    return a + b + c;  
}  
}
```

In this case, the add method is overloaded to handle different parameter types and numbers.

10.8 Abstraction

Abstraction is the concept of hiding complex implementation details and showing only the essential features of the object. In Java, abstraction can be achieved using abstract classes and interfaces.

Example: Abstract Class

```
abstract class Shape {  
    abstract void draw();  
}  
  
class Circle extends Shape {  
    void draw() {  
        System.out.println("Drawing a circle");  
    }  
}  
  
class Rectangle extends Shape {  
    void draw() {  
        System.out.println("Drawing a rectangle");  
    }  
}
```

The *Shape* class is an abstract class that cannot be instantiated. It defines a method *draw*, which is implemented by its subclasses *Circle* and *Rectangle*.

Example: Interface

```
interface Drawable {  
    void draw();  
}  
  
class Triangle implements Drawable {  
    public void draw() {  
        System.out.println("Drawing a triangle");  
    }  
}  
  
class Square implements Drawable {
```

```
public void draw() {  
    System.out.println("Drawing a square");  
}  
}
```

In this example, *Drawable* is an interface that declares a *draw* method. Both *Triangle* and *Square* classes implement this interface.

12.0 CONCLUSION

In conclusion, Java remains a powerful and versatile programming language that continues to play a significant role in software development. Its platform independence, robust community support, and extensive libraries make it an ideal choice for a wide range of applications, from web development to mobile apps and enterprise solutions. Java's emphasis on object-oriented principles promotes code reusability and maintainability, allowing developers to create scalable applications efficiently. As technology evolves, Java's adaptability ensures that it remains relevant, empowering developers to tackle modern challenges with confidence. Whether you're a beginner or an experienced programmer, mastering Java opens up numerous opportunities in the ever-changing landscape of technology.

BIBLIOGRAPHY

1. Herbert Schildt (2019). Java: A Beginner's Guide. McGraw-Hill Education Publishers
2. E. Balagurusamy (2015). Programming in Java. McGraw-Hill Education Publishers.
3. JAVA WITH NETBEANS. Source: <http://www.homeandlearn.co.uk/java/java.html>
4. JAVA INTRODUCTION. Source: https://www.w3schools.com/java/java_intro.asp
5. COURSE MODULE OUTLINE: <https://www.chatgpt.com/>
6. JAVA PROGRAMMING. Source: <https://w3resource.com/java-tutorial/java-program-structure.php>