Java Programming Language:

7500 (2500+2500+2500). If you pay full fees on or before 24th Nov, 2022 then the fees will be 7000. If you clear your fees after 17-Feb-2023, then the total fees will be 8000.

I - Introduction:

Java is related to C++, which is a direct descendant of C Programming Language. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++.

History:

The C Programming Language was developed by Dennis Ritchie, Brian Kernighan and team at Bell Laboratories in 1972. C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted.
C++, which is the successor of C, was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language "C with Classes". However, in 1983, the name was changed to C++. C++ extends C by adding object-oriented features. Because C++ is built on the foundation of C, it includes all of C's features, attributes, and benefits. This is a crucial reason for the success of C++ as a language.

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems Inc, Oakland, USA, in 1991. It took 18 months to develop the first working version. This language was initially called "Oak", but was renamed to "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Java:

Java is a programming language and a platform. Java is a high level, robust, object-oriented and secure programming language.
Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995. With the advancement of Java and its widespread popularity, multiple configurations were built to suit various types of platforms. For example: J2SE for Standard Applications, J2EE for Enterprise Applications, J2ME for Mobile Applications, etc. The new J2 versions were renamed as Java SE, Java EE, and Java ME respectively.
Sun released the first public implementation as Java 1.0 in 1995. It promised "Write Once, Run Anywhere" (WORA), providing no-cost run-times on popular platforms.
On 13 November, 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).
On 8 May, 2007, Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Platform:

Any hardware or software environment in which a program runs, is known as a platform. Since Java has its own runtime environment (JRE) and API (Application Programming Interface), it itself is a platform. Thus, Java is platform independent.

Java Platform/Editions:

There are 4 platforms or editions of Java:

1. Java SE (Java Standard Edition):

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2. Java EE (Java Enterprise Edition):

It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.

3. Java ME (Java Micro Edition):

It is a micro platform that is dedicated to mobile applications.

4. Java FX:

It is used to develop rich internet applications. It uses a lightweight user interface API. The applications built in Java FX, can run on multiple platforms including Web, Mobile and Desktops. Most likely it stands for Java "special EFF-ECTS" as FX is normally the abbreviation given to special effects mostly sound or visual.

Features:

- **Object Oriented** − In Java, everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform Independent** − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machines, rather into platform independent bytecode. This bytecode is distributed over the web and interpreted by the Java Virtual Machine (JVM) on whichever platform it is being run on.
- **Simple** − Java is designed to be easy to learn. If you understand the basic concept of OOP Java, it would be easy to master.
- **Secure** − With Java's secure feature it enables the development of virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** − Java compiler generates an architecture-neutral object file format (bytecode), which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compilers in Java are written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust** − Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.
- **Interpreted** − Java bytecode is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance** − With the use of Just-In-Time (JIT) compilers, Java enables high performance.
- **Distributed** − Java is designed for the distributed environment of the internet.
- **Dynamic** − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry an extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

First Java Program:

To design/write a Java program you can use any text editor tool software, such as notepad.

01. Program:

```
class MyFirstProgram
{
	public static void main(String[]args)
	{
		System.out.print("Hello All! Welcome to Java Programming.");
	}
}
```

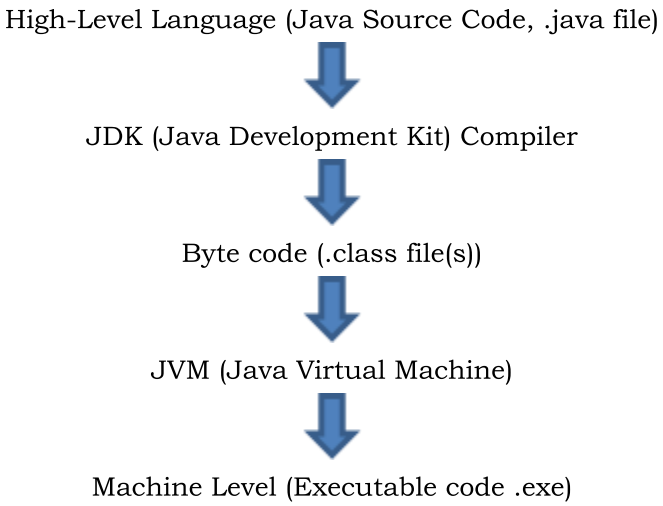
Program Explanation:

- The first statement is the declaration and definition of the **class** and we have given our own name to the **class**. As Java is a high level and object oriented programming language, here each and every thing is declared and defined within classes.
- The second statement is the declaration and definition of the **main( )** method. The **main( )** method is the entry point of our program, that is the execution of the program always starts with the **main( )** method. The **main( )** method is also the exit point of our program, that is execution of the program terminates with the end of **main( )** method.
  Observe that the **main( )** method has a "**public**" access modifier, that is because the **main( )** method is invoked (called) by the compiler which is not the part of our class/program. So the **main( )** method should always be declared as **public** so that it should be accessible to the compiler.
  The **main( )** method is decorated as "**static**", because the method should be accessible to the compiler without creating/initializing/using any object.
  The return type of the **main( )** method is "**void**", this is because the **main( )** method does not return any value.
  The argument to the **main( )** method is the **String[ ]** array. While invoking and executing the Java program, we can pass 0 or more arguments to the **main( )** method (command line arguments). These arguments are accepted in this **String[ ]** array. Thus we need to define the **String[ ]** array argument to the **main( )** method.
- The next statement is to print a String literal (text) on the output screen (console/command prompt). To print the output we have used the **print( )** method that is a built-in method defined inside the "**System**" class. We invoke this method using the "**out**" object. This "**out**" object is of output stream class, which is predefined inside the "**System**" class.

Once you complete writing the program you need to save it. You can save Java programs anywhere with any name. Generally a meaningful name is expected. Mostly it is recommended to give the **main( )** method's class name to the program file, with the "**.java**" extension. The "**.java**" is the mandatory extension using which the Java program file has to be saved.
Once the program has been saved, compile and execute it. To compile and execute the Java program you need to have **Java JDK**. JDK stands for **Java Development Kit**. This is the compiler provided by Java for compilation and execution of Java programs.

Compiler:

A compiler is a computer program (software) that translates a source program written in some high-level programming language (such as C, C++, Java, C#, etc.) into machine code (binary). The generated machine code can be then later executed many times against different data (input) each time.



Java provides us with their own open source, free, architecture neutral, platform independent compiler, that is the JDK (Java Development Kit) compiler.

JDK Installation:

To install Java JDK visit Java's official website and download the desired version of JDK.
https://www.oracle.com/in/java/technologies/javase-downloads.html
Once you have downloaded the Java JDK, install it.

After installation of Java JDK you can verify whether it has successfully installed or not. To verify open Command Prompt and run the "**java -version**" command as follows:

```
C:\Users\sourabha>java -version
java version "19.0.1" 2022-10-18
Java(TM) SE Runtime Environment (build 19.0.1+10-21)
Java HotSpot(TM) 64-Bit Server VM (build 19.0.1+10-21, mixed mode, sharing)

C:\Users\sourabha>
```

As shown above you will be able to see the installed version of Java JDK which means it has successfully installed on your machine.

Now you can compile and execute your Java program.

Open the Command Prompt. Browse to the location where you have saved the Java program and use the "**javac**" (Java Compile) command to compile the Java program.

Example:
```
C:\Users\sourabha>E:
E:\>cd "Java Programs"
E:\Java Programs>javac MyFirstProgram.java
E:\Java Programs>
```

Once the program is successfully compiled, you can run it. After the successful compilation of the program a **class** file is created at the location where the source code program (MyFirstProgram.java) file is saved. This file is the **bytecode** of a Java program which is only understood and targeted to the **JVM**. This **class** file has the name that matches with the class name(s) present in your program.
JVM stands for Java Virtual Machine. JVM is the part of the Java compiler (JDK) which is responsible for the execution of Java programs.

Now to execute Java program use the "**java MainMethod'sClassName**" command as shown below:

Example:

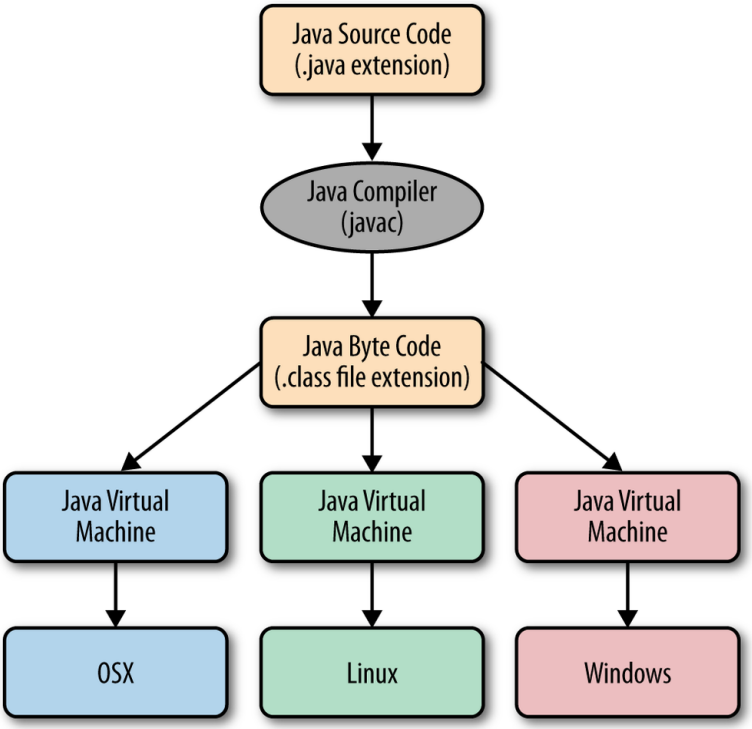
```
E:\Java Programs>java MyFirstProgram
Hello All! Welcome to Java Programming.
E:\Java Programs>
```

The command has a **java** application followed by the class name in which the **main( )** method resides.

It is recommended that Java programs should be saved/stored with the name of the **class** in which the **main( )** method resides, followed by the "**.java**" extension. That is the above program should be saved with the name as "**MyFirstProgram.java**".

This is just a recommendation, and not any mandatory rule. If you follow this recommendation it becomes easy for you to compile and execute your program, with one single same name.

Compilation process of Java Program:



In the new version of Java, that is in Java SE 12 (JDK12) onwards you can directly execute Java programs without compiling it. No doubt Java JDK will implicitly compile the program and run it.
Note that on directly executing Java program class files will not be created.

All Java programs can directly be compiled and executed using the "**java**" command as shown below:

```
java programFileName.java
```

Example:

```
E:\Java Programs>java MyFirstProgram.java
Hello All! Welcome to Java Programming.
E:\Java Programs>
```

This method of execution of Java programs is mainly used in the design, development and testing phase only. Once your Java Application is ready, to distribute it to the client, you should never share your source code to the client. There you will require the bytecode (.class file) to be installed on the client machine. In this case, you should follow the old traditional method of compilation of Java programs, which creates the bytecode (.class) file, which can be distributed to the client.

As you know that the byte code is the compiled code. It can only be read by the JVM (Java Virtual Machine). No user, or machine, or any kind of application/code can open, read, or tamper it. If tampered, JVM will never execute the tampered bytecode. This provides security to the Java Application.

Applications of Java:

According to Sun Microsystems, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are:

1. Desktop applications such as Acrobat Reader, Media Player, Antivirus, etc.
2. Web applications such as www.irctc.co.in, etc.
3. Enterprise applications such as banking applications.
4. Mobile.
5. Embedded system.
6. Smart card.
7. Robotics.
8. Games, etc.

Types of Java Applications:

There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application:

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that you need to install on every machine. Examples of standalone applications are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone GUI applications.

2) Web Application:

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struds, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application:

An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application:

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

II - Java Data Types:

Java is a Strongly typed language. Every variable has a type, every expression has a type, and every type is strictly defined. All assignments, whether explicit via parameter passing in method calls or implicit, are checked for type compatibility.
There are no automatic conversions of incompatible types as in some languages (C, C++, etc).
The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class.

Primitive Types:

Java defines eight primitive types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**.
These primitive types can be categorized as follows:

Integer Types:

The integer category includes **byte**, **short**, **int** and **long**, which are for integer-valued signed numbers.

Sr.	Name	Size	Range
1.	`byte`	1 byte (8-bits)	-128 to 127
2.	`short`	2 bytes (16-bits)	-32768 to 32767
3.	`int (default)`	4 bytes (32-bits)	-2147483648 to 2147483647
4.	`long`	8 bytes (64-bits)	-9223372036854775808 to 9223372036854775807

Declaration of Variable:

The variable declaration methodology in Java is the same as that in the C, C++ programming Languages. That is, you just have to specify the data type of the variable and then the identifier (name) of the variable.

Syntax:
```
dataType identifier;
```

Example:
```
short s;
s = 5;
```

As you can see, variable "s" has been declared, of type "short", and value has been assigned to it. The declaration and assignment of value to the variable, both can be done at the same time, which is known as initialization of the variable.

Syntax:
```
dataType identifier = value;
```

Example:
```
short s = 5;
```

Note that, in Java local variables have to be initialized before you use them. Without initialization or without assigning values to the declared variables, you cannot use them.

Storage of Values in Memory:

When a variable is declared, memory is allocated to it and the size of memory that is allocated to the declared variable is dependent on the data type of the variable. Next when you assign value to the variable, the binary equivalent of the assigned value will be stored into the memory.

Example:

```
short s = 6, r = -6;
```

Memory storage:

```
s = 0000 0000 0000 0110
r = 1000 0000 0000 0110
```

When a negative value is assigned to a variable, the negative sign is stored on the first, most significant, leftmost binary bit of the variable's value. That is binary bit 1 is stored, which represents negative value, and binary bit 0 is stored, which represents positive value. The rest of the 15 bits, in case of short data type, will be actually used to store the data value.

02. Program:

```
class IntegerDataTypes1 {
    public static void main(String[]args) {
        byte b = 0;
        short s = 0;
        int i = 0;
        long l = 0;
        System.out.println("Initial values of integer type variables :");
        System.out.println("b : " + b);
        System.out.println("s : " + s);
        System.out.println("i : " + i);
        System.out.println("l : " + l);
        b = 60;
        s = 15000;
        i = 6543210;
        l = 665544332;
        System.out.println("After assigning new values of integer type variables :");
        System.out.println("b : " + b);
        System.out.println("s : " + s);
        System.out.println("i : " + i);
        System.out.println("l : " + l);
    }
}
```

Note that, if you assign a value to a variable which is out of the range of the given variable's data type, then the compiler generates an error.

03. Program:

```
class IntegerDataTypes2 {
    public static void main(String[]args) {
        byte b = 24;
        Object obj = (Object)b;
        System.out.println("Type of variable b : " + obj.getClass().getName());
```

```
        System.out.println("Value of b : " + b);
        short s = 32000;
        obj = (Object)s;
        System.out.println("Type of variable s : " + obj.getClass().getName());
        System.out.println("Value of s : " + s);
        int i = 543210;
        obj = (Object)i;
        System.out.println("Type of variable i : " + obj.getClass().getName());
        System.out.println("Value of i : " + i);
        long l = 55443322;
        obj = (Object)l;
        System.out.println("Type of variable l : " + obj.getClass().getName());
        System.out.println("Value of l : " + l);
    }
}
```

Floating-Point Types:

The floating-point number category includes **float** and **double**, which represents numbers with fractional precision.

Sr.	Name	Size	Range
1.	float	4 bytes (32-bits)	1.4e-045 to 3.4e+038
2.	double (default)	8 bytes (64-bits)	4.9e-324 to 1.8e+308

Syntax:

```
float var;
var = valuef;
double var2;
var2 = value;
```

Example:

```
float fl = 3.14f;
double du = 12.2022;
```

04. Program:

```
class FloatDataType1 {
    public static void main(String[]str) {
        float fl = 3.14f;
        System.out.println("Float variable fl : " + fl);
        Object o = (Object)fl;
        System.out.println("Data type of variable fl : " + o.getClass().getName());
        double du = 12.2022;
        System.out.println("Double type variable du : " + du);
        o = (Object)du;
        System.out.println("Data type of variable du : " + o.getClass().getName());
    }
}
```

Character Type:

The character type is used to represent symbols in a character set, like letters, alphabets, digits, symbols, etc. In Java, the data type used to store characters is **char**. Character data type is the subtype of integer. Character type in Java is not the same as **char** in C or C++. In C and C++, char is 8 bits (1 byte) wide. This is not the case in Java.

C and C++ uses ASCII character code for interpretation and storage of character data. ASCII stands for American Standard Code for Information Interchange.
Under the ASCII code all digital electronic devices use an integer value to represent (and store into binary) the character values.

Decimal ASCII Chart

0	NUL	16	DLE	32	SP	48	0	64	@	80	P	96	`	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	'	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Java uses the Unicode character set to represent characters. Unicode defines a fully international character set that can represent all of the characters found in almost all human languages. The Unicode character set is the advanced version of ASCII, thus all the English Language alphabets (A-Z, a-z, 0-9) use the same ASCII code numbers in the Unicode Character set.
At the time of Java's creation, **Unicode** required 16 bits (**2 bytes**). Thus, in Java **char** is a 16-bit type. The range of a char data type in Java is from 0 to 65535.

Syntax:

```
char variableName;
variableName = 'value';
```

Note that, the character value that you, the developer, assigns to a character data type variable should always be enclosed within single quotes.

Example:

```
char ch;
ch = 'A';
ch = 's';
ch = '6';
ch = '+';
ch = ' ';
```

05. Program:

```
class CharDataType1 {
    static public void main(String[]args) {
        char ch = 'A';
        System.out.println("Char variable ch : " + ch);
        System.out.println("Type : " + ((Object)ch).getClass().getName());
        System.out.println("Unicode of " + ch + " is " + (int)ch);
        ch = 's';
        System.out.println("Value : " + ch + ", it's Unicode : " + (int)ch);
        ch = '6';
        System.out.println("Value : " + ch + ", it's Unicode : " + (int)ch);
        ch = 100;
        System.out.println("Value : " + ch + ", it's Unicode : " + (int)ch);
    }
}
```

Boolean Type:

In computer programming languages, **0 (zero)** and **NULL** value is used to represent "**false**" boolean value, and on the other hand, all **non-zero** and **non-NULL** values can be used to represent "**true**" boolean value.
Java has a primitive type, called **boolean**, for logical values. It can have one of two possible values, "**true**" or "**false**". 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** and **for**.

Syntax:

```
boolean variableName;
variableName = value;
```

Example:

```
boolean b;
b = true;
b = false;
```

06. Program:

```
class BooleanType1 {
    public static void main(String[]args) {
        boolean b = true;
        System.out.println("Value of boolean variable b : " + b);
        b = false;
        System.out.println("After assigning new value to the boolean variable b : " + b);
        System.out.println("Type : " + ((Object)b).getClass().getName());
        // b = 0;  // ERROR : incompatible types: int cannot be converted to boolean
        // b = (boolean)1; // ERROR : incompatible types: int cannot be converted to boolean
    }
}
```

III - Operators in Java

Operator in Java is a symbol that is used to perform operations. Java provides a rich set of operators to manipulate variables/values.

Types of operators:

- Arithmetic operators
- Increment and Decrement operators
- Relational/Comparison operators
- Bitwise operators
- Logical operators
- Assignment operators
- Miscellaneous operators

Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. Assume the following declaration of variables:

```
int a = 17, b = 7, c;
```

Sr.	Operator	Operation	Example
1.	+	Addition	c = a + b; // Output: a = 17, b = 7, c = 24
2.	-	Subtraction	c = a - b; // Output: a = 17, b = 7, c = 10
3.	*	Multiplication	c = b * a; // Output: a = 17, b = 7, c = 119
4.	/	Division	c = a / b; // Output: a = 17, b = 7, c = 2
5.	%	Modulus (Reminder)	c = a % b; // Output: a = 17, b = 7, c = 3

07. Program:

```
class ArithmeticOperators1 {
    public static void main(String [] args) {
        int a = 17, b = 7, c;
        System.out.println("Arithmetic Operators:");
        c = a + b;
        System.out.println("Addition of " + a + " and " + b + " is " + c);
        c = a - b;
        System.out.println("Subtraction of " + a + " and " + b + " is " + c);
        c = a * b;
        System.out.println("Multiplication of " + a + " and " + b + " is " + c);
        c = a / b;
        System.out.println("Division of " + a + " and " + b + " is " + c);
        c = a % b;
        System.out.println("Reminder of division of " + a + " and " + b + " is " + c);
    }
}
```

08. Program:

```
class ArithmeticOperators2 {
    public static void main(String [] args) {
        int a = 17, b = 7;
        System.out.println("Arithmetic Operators:");
        System.out.println("Concatenation of " + a + " and " + b + " is " + a + b);
        System.out.println("Addition of " + a + " and " + b + " is " + (a + b));
        System.out.println("Subtraction of " + a + " and " + b + " is " + (a - b));
        System.out.println("Multiplication of " + a + " and " + b + " is " + a * b);
        System.out.println("Division of " + a + " and " + b + " is " + a / b);
        System.out.println("Reminder of division of " + a + " and " + b + " is " + a % b);
    }
}
```

Increment and Decrement Operators:

The increment /decrement operator increases/decreases the value of a given integer type variable by 1. These are the unary operators.

The Increment Operator (++):

Increment operator increments (increases) the value of given variable by 1.

Syntax:

```
var++;       // var = var + 1;
```

Example:

```
int i = 13;
i++;         // Memory: i = 14
```

Types of Increment Operator:

1. Prefix-increment (++var):

When the increment operator is placed before the variable (operand), then the value of the variable is incremented first and then is assigned to another variable (if any).

Syntax:

```
++var;
var2 = ++var1;
```

Example:

```
int i = 13, j;
j = ++i;     // i = 14, j = 14
```

2. Postfix-increment (var++):

In incrementation if the increment operator is placed after the variable (operand), then the value of the variable is assigned as it is to another variable (if any), and then its own value is incremented.

Syntax:

```
var++;
var2 = var1++;
```

Example:
```
int i = 13, j;
j = i++;     // i = 14, j = 13
```

09. Program:
```
class IncrementOperatorDemo1 {
    static public void main(String[]args) {
        int i = 13;
        System.out.println("Value of i = " + i);     // 13
        i++;
        System.out.println("Value of i after i++ = " + i);  // 14
        ++i;
        System.out.println("Value of i after ++i = " + i);  // 15
        int j;
        j = ++i;
        System.out.println("Prefix-increment : values after j = ++i");
        System.out.println("i = " + i + ", j = " + j);  // i = 16, j = 16
        j = i++;
        System.out.println("Postfix-increment : values after j = i++");
        System.out.println("i = " + i + ", j = " + j);  // i = 17, j = 16
        System.out.println("++i = " + (++i));    // Output = 18, i = 18
        System.out.println("i++ = " + (i++));    // Output = 18, i = 19
        System.out.println("After i++, i = " + i);  // 19
    }
}
```

The Decrement Operator (--):

Decrement operator decrements (decreases) the value of given variable by 1. This is an unary operator.

Syntax:
```
var--;       // var = var - 1;
```

Example:
```
int i = 13;
i--;  // i = 12
```

Types of Decrement Operator:

1. Prefix-decrement (--var):

When the decrement operator is placed before the variable (operand), then the value of the variable is decremented first and then is assigned to another variable (if any).

Syntax:
```
--var;
var2 = --var1;
```

Example:
```
int i = 13, j;
j = --i;     // i = 12, j = 12
```

2. Postfix-decrement (var--):

In decrementation if the decrement operator is placed after the variable (operand), then the value of the variable is assigned as it is to another variable (if any), and then its own value is decremented.

Syntax:

```
var--;
var2 = var1--;
```

Example:

```
int i = 13, j;
j = i--;     // i = 12, j = 13
```

10. Program:

```
class DecrementOperatorDemo1 {
    public static void main(String[] args) {
        int i = 13;
        System.out.println("Value of i = " + i); // 13
        i--;
        System.out.println("Value of i after i-- = " + i); // 12
        --i;
        System.out.println("Value of i after --i = " + i); // 11
        int j;
        j = --i;
        System.out.println("Pre-decrement: values after j = --i");
        System.out.println("i = " + i + ", j = " + j); // i = 10, j = 10
        j = i--;
        System.out.println("Post-decrement: values after j = i--");
        System.out.println("i = " + i + ", j = " + j); // i = 9, j = 10
        System.out.println("--i = " + (--i)); // 8
        System.out.println("i-- = " + (i--));   // 8
        System.out.println("After i--, i = " + i); // 7
    }
}
```

Relational/Comparison Operators:

In Java relational operators are also known as comparison operators and they compare 2 operands and give us the output as either **true** (1) or **false** (0).
In programming languages all non-zero and non-null values are considered as TRUE.
On the other hand, zero (0) or NULL is considered as FALSE.

Consider the variables as:

```
int a = 15, b = 10;
```

Sr.	Operator	Description	Example
1.	Equality (==)	Checks if the values of 2 operands are equal or not. If yes, then the condition becomes **true**, otherwise **false**.	a == b Output : false

2.	Not-equal (!=)	Checks if the values of 2 operands are equal or not. If the values are not equal, then the condition becomes **true**, otherwise **false**.	a != b Output : true
3.	Greater than (>)	Checks if the left hand side value of 2 operands is greater than the right hand side value or not. If yes, then the condition becomes **true** otherwise **false**.	a > b Output : true
4.	Smaller than (<)	Checks if the left hand side value of 2 operands is smaller than the right hand side value or not. If yes, then the condition becomes **true** otherwise **false**.	a < b Output : false
5.	Greater than or equal to (>=)	Checks if the left hand side value of 2 operands is greater than or is equal to the right hand side value or not. If yes, then the condition becomes **true** otherwise **false**.	a >= b Output : true
6.	Smaller than or equal to (<=)	Checks if the left hand side value of 2 operands is smaller than or is equal to the right hand side value or not. If yes, then the condition becomes **true** otherwise **false**.	a <= b Output : false

11. Program:

```
class RelationalOperatorsDemo1 {
    public static void main(String[]args) {
        System.out.println("Relational Operators:");
        int no1 = 15, no2 = 10, no3 = 15;
        System.out.println("no1 = " + no1 + ", no2 = " + no2 + ", and no3 = " + no3);
        System.out.println("no1 equal to no2 = " + (no1 == no2));
        boolean b = no1 == no3;
        System.out.println("no1 equal to no3 = " + b);
        System.out.println("no1 not equal to no2 = " + (no1 != no2));
        System.out.println("no1 not equal to no3 = " + (no1 != no3));
        System.out.println("no1 greater than no2 = " + (no1 > no2));
        System.out.println("no1 greater than no3 = " + (no1 > no3));
        System.out.println("no1 smaller than no2 = " + (no1 < no2));
        System.out.println("no1 smaller than no3 = " + (no1 < no3));
        System.out.println("no1 greater than or equal to no2 = " + (no1 >= no2));
        System.out.println("no1 greater than or equal to no3 = " + (no1 >= no3));
        System.out.println("no1 smaller than or equal to no2 = " + (no1 <= no2));
        System.out.println("no1 smaller than or equal to no3 = " + (no1 <= no3));
    }
}
```

Logical Operators:

Logical operators perform logical operations on the given operands. There are 3 types of logical operators: Logical AND, Logical OR, Logical NOT.

A	B	A AND B	A OR B	A ExOR B	NOT A
false	false	false	false	false	true

false	true	false	true	true	true
true	false	false	true	true	false
true	true	true	true	false	false

Assume boolean variables as:

```
boolean a = true, b = false, c;
```

Sr.	Operator	Description	Example
1.	&& (Logical AND)	Logical AND operator performs AND operation among the given 2 operands. If both the operands are "true" (non-zero), then the operator returns "true" boolean value, otherwise "false".	c = a && b; Output: a = true, b = false, c = false
2.	|| (Logical OR)	Logical OR operator performs OR operation among the given 2 operands. If any of the two operands are "true" (non-zero), then the operator returns "true" boolean value, otherwise if both the operands are false then it returns "false".	c = a || b; Output : a = true, b = false, c = true
3.	! Logical NOT	Logical NOT gives the inverted value of the operand. This is an unary operator. For "true" input, the logical NOT operator gives the value as "false" and vise-versa.	c = ! a; Output : a = true, c = false

Homework:
WAP to perform all the operations using logical operators:

Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte. Bitwise operators perform operations on the binary bits of the given operands. That is, they work on bits and perform bit-by-bit operations.

Assume that,

```
short a = 60, b = 13, c;
Binary of a (60) = 0000 0000 0011 1100
Binary of a (13) = 0000 0000 0000 1101
```

Sr.	Operator	Description	Example
1.	Bitwise AND (&)	Bitwise AND operator performs AND operation bit-by-bit on the input operands.	a(60) = 0000 0000 0011 1100 b(13) = 0000 0000 0000 1101 a & b = 0000 0000 0000 1100 = (12)
2.	Bitwise OR (|)	Bitwise OR operator performs OR operation bit-by-bit on the input operands.	a(60) = 0000 0000 0011 1100 b(13) = 0000 0000 0000 1101 a | b = 0000 0000 0011 1101 = (61)
3.	Bitwise ExOR (^)	Bitwise ExOR operator performs ExOR operation bit-by-bit on the input operands.	a(60) = 0000 0000 0011 1100 b(13) = 0000 0000 0000 1101 a ^ b = 0000 0000 0011 0001 = (49)
4.	Bitwise NOT (~)	Bitwise NOT operator performs the operation on a single operand, that is it is an unary	a(60) = 0000 0000 0011 1100 +1

		operator. Here binary bit 1 is added in the given operand's binary bits and the sign bit is inverted. ~a = -(a+1) ~49 = -(49+1) = -(50) = -50 ~-60 = -(-60 + 1) = -(-59) = 59	~a = 1000 0000 0011 1101 = (-61) b(13) = 0000 0000 0000 1101 +1 ~b = 1000 0000 0000 1110 = (-14)
5.	Bitwise Left shift (<<)	The left shift operator moves the binary bits towards the left of the left operand by the number of bits specified by the right operand.	a(60) = 0000 0000 0011 1100 a << 3= 0000 0001 1110 0000 = (480) b(13) = 0000 0000 0000 1101 b << 2= 0000 0000 0011 0100 = (52)
6.	Bitwise Right shift (>>)	The right shift operator moves the binary bits towards the right of the left operand by the number of bits specified by the right operand.	a(60) = 0000 0000 0011 1100 a >> 3= 0000 0000 0000 0111 = (7) b(13) = 0000 0000 0000 1101 b >> 2= 0000 0000 0000 0011 = (3)

12. Program:

```
class BitwiseOperatorsDemo1 {
    public static void main(String[]args) {
        System.out.println("Relational Operators:");
        int a = 60, b = 13;
        System.out.println("a = " + a + ", and b = " + b);
        System.out.println("Bitwise AND : a & b = " + (a & b));
        System.out.println("Bitwise OR : a | b = " + (a | b));
        System.out.println("Bitwise ExOR : a ^ b = " + (a ^ b));
        System.out.println("Bitwise NOT : ~a = " + (~a));
        System.out.println("Bitwise NOT : ~b = " + (~b));
        System.out.println("Bitwise Left Shift : a << 3 = " + (a << 3));
        System.out.println("Bitwise Left Shift : b << 2 = " + (b << 2));
        System.out.println("Bitwise Right Shift : a >> 3 = " + (a >> 3));
        System.out.println("Bitwise Right Shift : b >> 2 = " + (b >> 2));
    }
}
```

Assignment Operators:

Assignment Operator (=):

The assignment operator assigns the right hand side value to the left hand side variable. You can also define an expression at the right hand side and then the assignment operator evaluates the right hand side expression and assigns the outcome to the left hand side variable.
Note that, the left hand side of the assignment operator should always be a variable.

Syntax:

```
var = value;
```

OR

```
var = expression;
```

Example:

```
int a = 51;
```

```
int b = a * 5;
int c = a && b;
boolean d = a > b;
```

Compound Assignment Operators:

Compound Assignment Operators perform 2 operations. First it performs operation on left side and right side operands and then assigns the outcome of the operation to the left side operand.
Note that the value of the left hand side operand will be overwritten by the new value which is the outcome of the operation.

I. Arithmetic Assignment Operators:

a. Addition Assignment Operator (+=):

The addition assignment performs addition of left and right hand side operands and the sum value is assigned to the left side operand.

Syntax:

```
op1 += op2;        // op1 = op1 + op2;
```

b. Subtraction Assignment Operator (-=):

It performs subtraction of left and right side operands and the difference value is assigned to the left side operand.

Syntax:

```
op1 -= op2;     // op1 = op1 - op2;
```

c. Multiplication Assignment Operator: (*=):

It performs multiplication of left and right side operands and the resultant value is assigned to the left side operand.

Syntax:

```
op1 *= op2;     // op1 = op1 * op2;
```

d. Division Assignment Operator: (/=):

It performs division of left and right side operands and the quotient value is assigned to the left side operand.

Syntax:

```
op1 /= op2;     // op1 = op1 / op2;
```

e. Modulus Assignment Operator: (%=):

It performs division of left and right side operands and the remainder value is assigned to the left side operand.

Syntax:

```
op1 %= op2;     // op1 = op1 % op2;
```

Example:

```
int a = 15, b = 9;
a += b;      // a = 24, b = 9
a -= b;      // a = 15, b = 9
b *= a;      // a = 15, b = 135
b /= a;      // a = 15, b = 9
a %= a;      // a = 0, b = 9
```

Homework:

1. WAP to perform all the Arithmetic Assignment Operations.

II. Bitwise Compound Assignment Operators:

a. Bitwise AND Assignment operator: (&=):

It performs Bitwise AND operation of left and right side operands and the result is assigned to the left side operand.

Syntax:

```
op1 &= op2;      // op1 = op1 & op2;
```

b. Bitwise OR Assignment operator: (|=):

It performs Bitwise OR operation of left and right side operands and the result is assigned to the left side operand.

Syntax:

```
op1 |= op2;      // op1 = op1 | op2;
```

c. Bitwise ExOR Assignment operator: (^=):

It performs Bitwise ExOR operation of left and right side operands and the result is assigned to the left side operand.

Syntax:

```
op1 ^= op2;      // op1 = op1 ^ op2;
```

d. Left shift Assignment operator: (<<=):

It performs Bitwise Left shift operation of left and right side operands and the result is assigned to the left side operand.

Syntax:

```
op1 <<= op2;      // op1 = op1 << op2;
```

e. Right shift Assignment operator: (>>=):

It performs Bitwise Right shift operation of left and right side operands and the result is assigned to the left side operand.

Syntax:

```
op1 >>= op2;      // op1 = op1 >> op2;
```

Homework:

1. WAP to perform all the Bitwise Assignment Operations.

Miscellaneous Operators:

1. Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The condition operator is also called the if-then-else operator.

Syntax:

```
var = (expression) ? valueIfTrue : valueIfFalse;
```

13. Program:

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

```
        int no1 = 60, no2 = 150;
        int max = no1 > no2 ? no1 : no2;
        System.out.println("Max value among " + no1 + " and " + no2 + " is " + max);
        System.out.println("Max value among 250 and 240 is " + (250 > 240 ? 250 : 240));
    }
}
```

IV - Control Statements:

Control statements are the statements in the Java Programming Language that control the flow of a program.

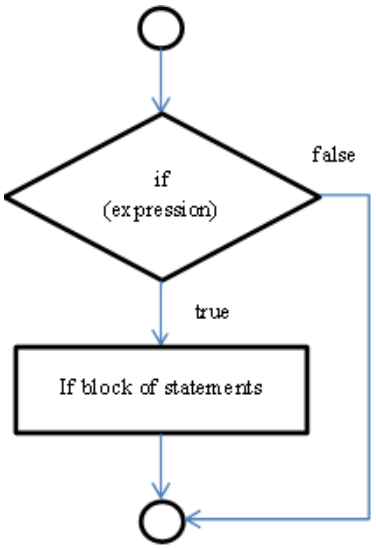
Decision Making:

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or group of statements to be executed if the condition is determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**.
Java Programming Language assumes all **NON-ZERO** and **NON-NULL** values as **TRUE**, and if the value is **ZERO** or **NULL** then it is assumed as **FALSE**. Apart from this Java has its own **boolean** data type, which is used to evaluate the condition/expression which results in either **true** or **false**.

The if Statement:

An **if** statement consists of a boolean expression followed by one or more statements. If the boolean expression evaluates to **TRUE**, then the block of code inside the **if** statement will be executed. If the boolean expression evaluates to **FALSE**, then the first set of code after the end of the **if** statement will be executed, escaping the statement(s) present inside the **if** statement block.

Flowchart:



Syntax:

```
if(booleanExpression) {
      // block of statement(s) to be executed if the booleanExpression evaluates to TRUE
}
...
```

14. Program:

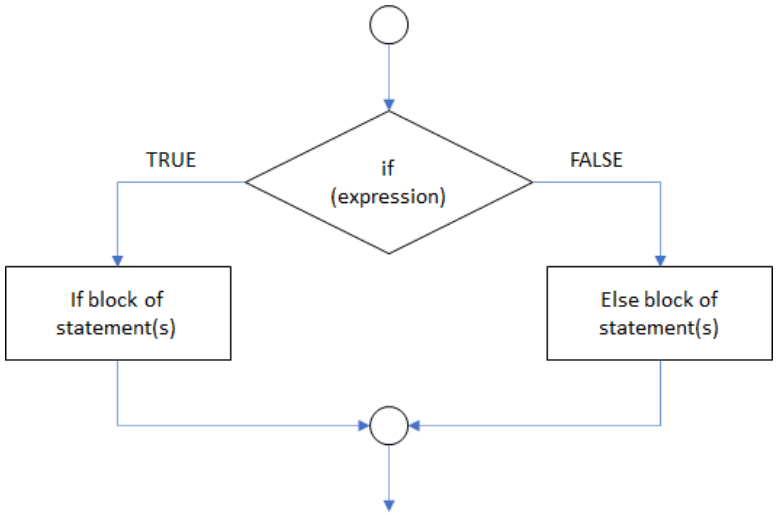
```
class IfStatementDemo1 {
    public static void main(String[]args) {
        int age = 20;
        if(age >= 18) {
            System.out.println("Congratulations! You are eligible for driving a vehicle.");
            System.out.println("Please get a driving license and drive carefully.");
        }
        System.out.println("Thank you.");
```

```
    }
}
```

The if … else Statement:

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression evaluates to **FALSE**. If the boolean expression evaluates to **TRUE**, then the block of code inside the **if** statement will be executed. If the boolean expression evaluates to **FALSE**, then the block of code inside the **else** statement will be executed.

Flowchart:



Syntax:

```
if(booleanExpression) {
      // block of statement(s) to be executed if the booleanExpression evaluates to TRUE
}
else {
      // block of statement(s) to be executed if the booleanExpression evaluates to FALSE
}
...
```

Note that in any case only one single block of code, that is either TRUE or FALSE block of code will be executed.
In any case, Java compiler will execute only one block, that is either **if** block or **else** block.

15. Program:

```
class IfElseStatementDemo1 {
    public static void main(String[]args) {
        int age = 15;
        if(age >= 18) {
            System.out.println("Congratulations! You are eligible for driving a vehicle.");
            System.out.println("Please get a driving license and drive carefully.");
        }
        else {
            System.out.println("Please wait...");
            System.out.println("You are not yet eligible to drive any vehicle.");
        }
        System.out.println("Thank you.");
```

```
    }
}
```

Homework:

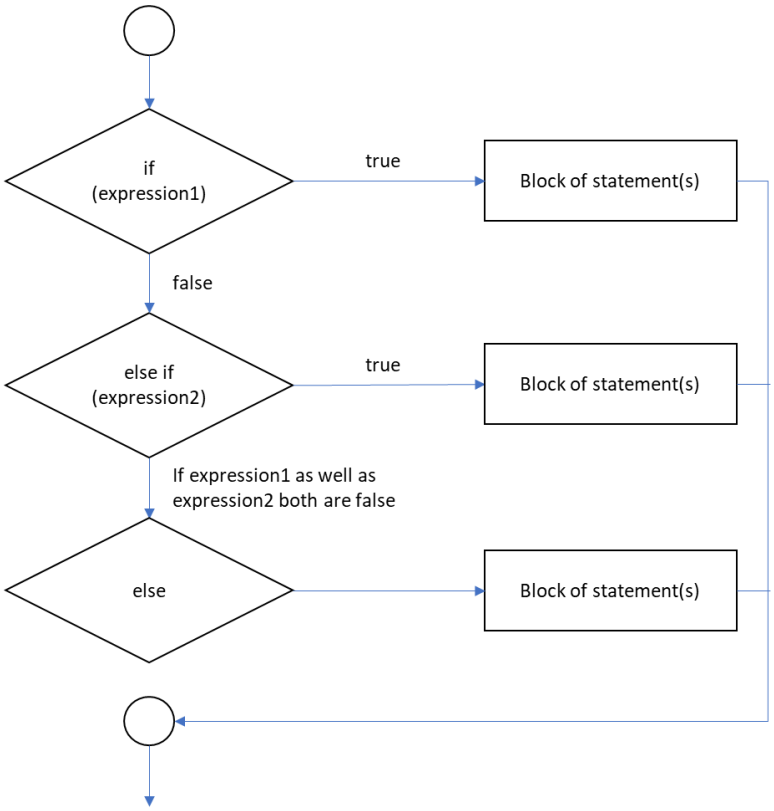
1. WAP to check whether a number is an EVEN or ODD number.
2. WAP to check and print whether a number is POSITIVE or NEGATIVE value.
3. WAP to check the percentage of a student and print if he/she has passed or not.

The if … else if … Statement:

An **if** statement can be followed by an optional **else if** statement, which is very useful to test various conditions using single **if … else if …** statement. When using **if … else if …** statements, there are few points to keep in mind:

- An **if** can have zero or one **else** statement and it must come after any **else if**.
- An **if** can have zero to many **else if** statement(s) and they must come after the **if** statement and before the **else** statement, if any.
- Once an **if** or **else if** statement succeeds, none of the remaining **else if** or **else** will be tested.

Flowchart:



Syntax:

```
if(condition1/expression1) {
      // this block executes if the condition1/expression1 evaluates to TRUE
}
else if(condition2/expression2) {
      // this block executes if the condition1/expression1 evaluates to FALSE and
      condition2/expression2 evaluates to TRUE
}
else {
      // this block executes if the condition1/expression1 as well as condition2/expression2 both
      evaluates to FALSE
}
...
```

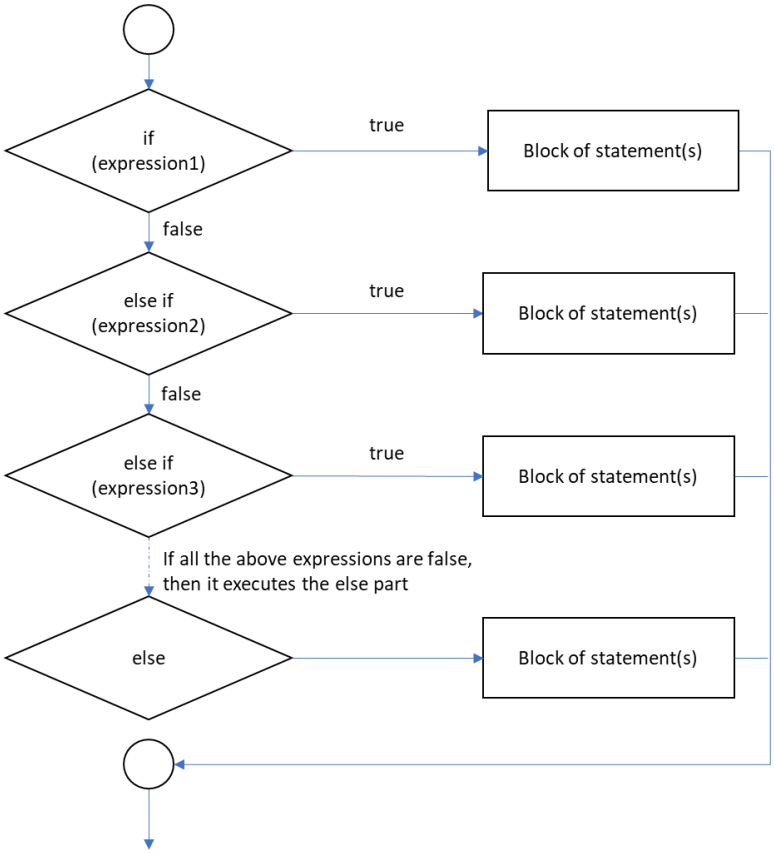
16. Program:

```
class IfElseIfDemo1 {
    public static void main(String[]args) {
        int no = 6;
        if(no == 0)
            System.out.println("Number is NEUTRAL.");
        else if(no > 0)
            System.out.println("Number is POSITIVE.");
        else
            System.out.println("Number is NEGATIVE.");
        System.out.println("Exiting from the main() method.");
    }
}
```

The if … else if … Ladder Statement:

The **if... else if...** ladder statement is an extension to the **if... else if...** statement. It is used in the scenario where one case among multiple cases are to be performed for different conditions.

Flowchart:



Syntax:

```
if(condition1/expression1) {
      // this block executes if the condition1/Expression1 evaluates to TRUE
}
else if(condition2/expression2) {
```

```
      // this block executes if the condition1/expression1 evaluates to FALSE and
      condition2/expression2 evaluates to TRUE
}
else if(condition3/expression3) {
      // this block executes if the condition1/expression1 and condition2/expression2 both
      evaluates to FALSE and condition3/expression3 evaluates to TRUE
}
else if(condition4/expression4) {
      // this block executes if the condition1/expression1, condition2/expression2 & condition3
      /expression3 both evaluates to FALSE and condition4/expression4 evaluates to TRUE
}
...
else {
      // this block executes if the all the above conditions/expressions evaluates to FALSE
}
...
```

17 Program:

```
class IfElseIfLadderDemo1 {
    public static void main(String[]args) {
        int age = 24;
        if(age <= 3)
            System.out.println("The person is an infant.");
        else if(age < 12)
            System.out.println("The person is a child.");
        else if(age < 18)
            System.out.println("The person is a teen ager.");
        else if(age < 60)
            System.out.println("The person is an Adult.");
        else
            System.out.println("The person is Sr. Citizen.");
        System.out.println("Thank you.");
    }
}
```

Homework:

1. WAP to check whether the number is even or odd, without using the mod operator.
2. WAP to calculate the percentage obtained by a student and print the grade.

Nested if … else Statement:

Java Program allows us to place an **if … else** statement inside another **if, else if,** or **else** block. This is **nesting of if … else** statements.

Syntax:

```
if(condition1/expression1) {
      ...
      if(condition2/expression2) {   // Nested if
            // executes when condition1/expression1 is TRUE and condition2/expression2 is also
            TRUE
            ...
```

```
	}
	else ...
...
}
else if(condition3/expression3) {
	...
	if(condition4/expression4) {
		// executes when condition1/expression1 is FALSE and condition3/expression3 as well as
		condition4/expression4 is TRUE
	}
	...
}
else ...
```

18. Program:

```
class NestedIfElseDemo1 {
    public static void main(String[]args) {
        int age = 60;
        if(age >= 0 && age <= 130) {
            if(age <= 3)    // Nested if...else...
                System.out.println("The person is an infant.");
            else if(age < 12)
                System.out.println("The person is a child.");
            else if(age < 18)
                System.out.println("The person is a teen ager.");
            else if(age < 60)
                System.out.println("The person is an Adult.");
            else
                System.out.println("The person is Sr. Citizen.");
        }
        else
            System.out.println("Invalid age.");
        System.out.println("Thank you.");
    }
}
```

The switch-case Statement:

The **switch** statement in Java is an alternative to the **if... else-if** ladder statement which allows you to execute multiple operations for the different possible values of a single variable/expression.

Syntax:

```
switch(expression) {
	case value1:
		// code to be executed if the expression evaluates to value1
		break;
	case value2:
		// code to be executed if the expression evaluates to value2
		break;
	case value3:
		// code to be executed if the expression evaluates to value3
```

```
        break;
    ...
    default:
        // code to be executed if the expression evaluation value does not match with any
        case's value
}
...
```

19. Program:

```
class SwitchCaseDemo1 {
    public static void main(String[]args) {
        int weekNo = 3;
        switch(weekNo) {
            case 6:
                System.out.println("Its Saturday.");
                break;
            case 1:
                System.out.println("Its Monday.");
                break;
            case 4:
                System.out.println("Its Thursday.");
                break;
            case 3:
                System.out.println("Its Wednesday.");
                break;
            case 5:
                System.out.println("Its Friday.");
                break;
            case 7:
                System.out.println("Its Sunday.");
                break;
            case 2:
                System.out.println("Its Tuesday.");
                break;
            default:
                System.out.println("Invalid week day number");
        }
        System.out.println("Thank you.");
    }
}
```

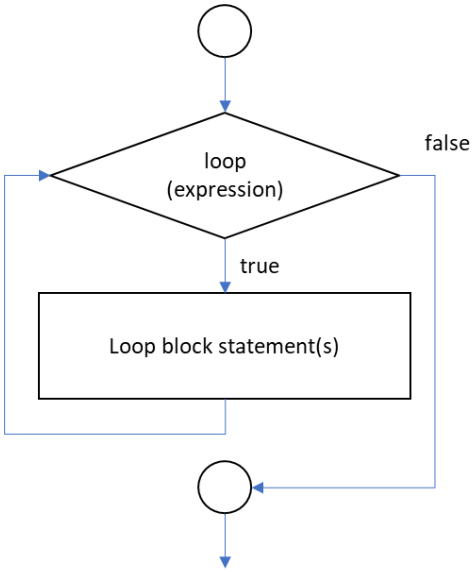
Homework:

1. WAP to check and print the month name from the month number.

Looping Statements:

You may encounter situations when a block of code needs to be executed several times.
In general statements are executed sequentially. The first statement in a method or block of code is executed first, followed by the second, then the third one, and so on.
Programming languages provide various control structures that allow for more complicated execution paths.
A looping statement allows you to execute a statement or group of statements multiple times.

Flowchart:



Types of Loops:

The while Loop:

A while loop in Java programming language repeatedly executes a target statement(s) as long as the given condition is TRUE.

Syntax:

```
while(condition/expression) {
      statement(s);
      ...
}
...
```

Here, statement(s) may be a single statement or block of statements. The condition may be any expression, and TRUE is any non-zero, non-NULL value. The loop iterates while the given condition is TRUE.
When the condition becomes FALSE, the program control passes to the statement immediately following the loop.
The while loop is also known as a pre-tested loop/entry controlled loop.

20. Program:

```
class WhileDemo1 {
    public static void main(String[]args) {
        System.out.println("Numbers from 1 to 10 are :");
        int cnt = 1;
        while(cnt <= 10) {
            System.out.print(" " + cnt);
```

```
            ++cnt;
        }
        System.out.println("\nAfter the while loop, cnt = " + cnt);
        System.out.println("Thank you.");
    }
}
```

Homework:

1. WAP to print the table of a given number.
2. WAP to print all ODD numbers from 1 to 100.
3. WAP to print all EVEN numbers from 1 to 100.
4. WAP to print numbers divisible by 7 from 1 to 350.

The for Loop:

The **for** loop in Java Programming Language is used to iterate a block of statement(s) several times. It is frequently used to traverse the data structures like the array and linked list.

Syntax:

```
for(initialization; condition; iteration) {
      statement(s);
}
```

Execution/Working:

```
for(initialization [Step-0]; condition [Step-1]; incr/decr [Step-3]) {
      statement(s); [Step-2]
}
For condition false [Step-4]
```

21. Program:

```
class ForDemo1 {
    public static void main(String[]args) {
        System.out.println("Numbers from 1 to 10 are :");
        /* int cnt = 1;
        while(cnt <= 10) {
            System.out.print(" " + cnt);
            ++cnt;
        }
        System.out.println("\nAfter the while loop, cnt = " + cnt); */
        for(int cnt = 1; cnt <= 10; ++cnt) {
            System.out.print(" " + cnt);
        }
        System.out.println("\nThank you.");
    }
}
```

Homework:

1. WAP to print the table of the given number.
2. WAP to print all ODD numbers from 100 to 1.
3. WAP to print all EVEN numbers from 100 to 1.
4. WAP to print numbers divisible by 7 from 350 to 1.

The do…while Loop:

The do-while loop is a post tested loop. Using the do-while loop, you can repeat the execution of a block of statements. The do-while loop is mainly used in the case where you need to execute the loop body at least once. The do…while loop is mainly used to design/develop menu driven programs.

Syntax:

```
do {
      statement(s);
      ...
}while(condition);
```

22. Program:

```
class DoWhileDemo1 {
    public static void main(String[]args) {
        System.out.println("Numbers from 1 to 10 are :");
        int cnt = 1;
        /* while(cnt <= 10) {
            System.out.print(" " + cnt);
            ++cnt;
        } */
        do {
            System.out.print(" " + cnt);
            ++cnt;
        } while(cnt <= 10);
        System.out.println("\nAfter the do-while loop, cnt = " + cnt);
        System.out.println("Thank you.");
    }
}
```

The **do … while** loop is mainly used when you want your block of statement(s) to be executed at least once. It is mainly used to implement menu driven programs. Menu driven programs are nothing but the programs where the user decides what to do and when to stop.

23. Program:

```
class DoWhileDemo2 {
    public static void main(String[]args) {
        int no, option;
        no = 6;
        for(int i = 1; i < 11; ++i) {
            int ans = no * i;
            System.out.println(no + " x " + i + " = " + ans);
        }
    }
}
```

Homework:

1. WAP to print even numbers from 100 to 1.
2. WAP to print odd numbers from 100 to 1.
3. WAP to print fibonacci series from 0 to 500

The for-each Loop/Enhanced for Loop:

The Java **for-each** loop or enhanced for loop is introduced since J2SE 5.0. It provides an alternative approach to traverse the array or collection in Java. It is mainly used to traverse the array or collection elements. The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as the **for-each** loop because it traverses each element one by one from the given array or collection of elements.

The drawback of the enhanced for loop is that it cannot traverse the elements in reverse order. Here, you do not have the option to skip any element because it does not work on an index basis. Moreover, you cannot traverse the odd or even elements only.

But, it is recommended to use the Java for-each loop for traversing the elements of array and collection because it makes the code readable.

Advantages:

- It makes the code more readable.
- It eliminates the possibility of programming errors.

Syntax:

```
for(dataType iteratingVariable : array/collection) {
      // body of for each loop
}
```

24. Program:

```
class ForEachDemo1 {
    public static void main(String[]args) {
        int arr[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
        System.out.println("The array elements are :");
        /* for(int i = 0; i < arr.length; ++i)
            System.out.println("arr[" + i + "] = " + arr[i]); */
        for(int ele : arr)
            System.out.println(ele);
        System.out.println("Thank you.");
    }
}
```

Variations in Loops:

Variations in while, do…while loop:

You can set a while/do-while loop to always TRUE. Doing so will infinitely execute the loop body.

Example:

```
while(true) {
      Loop body...
}

do {
      Loop body...
}while(true);
```

25. Program:

```
class VariationsInLoops1 {
    public static void main(String[] args) {
        System.out.println("while(true)");
        int no = 1;
        while(true)
        {
            System.out.print(no + " ");
            no++;
        }
        // System.out.println("\nThank you."); // ERROR : unreachable statement
    }
}
```

26. Program:

```
class VariationsInLoops2 {
    public static void main(String[] args) {
        System.out.println("do-while(true)");
        int no = 1;
        do {
            System.out.print(no + " ");
            no++;
        } while(true);
        // System.out.println("\nThank you."); // ERROR : unreachable statement
    }
}
```

Variations in for Loop:

You can specify 0 or more number of assignment/input statement(s), separated by commas, in the initialization part of the for loop.

Example:

```
for( ... ; ... ; ...)
{
      Loop body
}
```

27. Program:

```
class VariationsInLoops3 {
    public static void main(String[]args) {
        System.out.println("Table of number 6 is :");
        for(int no = 6, cnt = 1, ans; cnt <= 10; ++cnt) {
            ans = no * cnt;
            System.out.println(no + " X " + cnt + " = " + ans);
        }
        System.out.println("Thank you.");
    }
}
```

You can eliminate any/all expressions from the **for** loop.

28. Program:

```
class VariationsInLoops4 {
    public static void main(String[]args) {
        System.out.println("Table of number 6 is :");
        int no = 6, cnt = 1, ans;
        for( ; cnt <= 10; ++cnt) {
            ans = no * cnt;
            System.out.println(no + " X " + cnt + " = " + ans);
        }
        System.out.println("Thank you.");
    }
}
```

29. Program:

```
class VariationsInLoops5 {
    public static void main(String[]args) {
        System.out.println("Table of number 6 is :");
        int no = 6, cnt = 1, ans;
        for( ; ; ++cnt) {
            ans = no * cnt;
            System.out.println(no + " X " + cnt + " = " + ans);
        }
        // System.out.println("Thank you."); // ERROR : unreachable statement
    }
}
```

30. Program:

```
class VariationsInLoops6 {
    public static void main(String[]args) {
        System.out.println("Table of number 6 is :");
        int no = 6, cnt = 1, ans;
        for( ; ; ) {
            ans = no * cnt;
            System.out.println(no + " X " + cnt + " = " + ans);
            ++cnt;
        }
        // System.out.println("Thank you."); // ERROR : unreachable statement
    }
}
```

Loop Control Statements:

The iteration of loops can be controlled, stopped, or skipped using the loop control statements. There are basically two loop control statements, the break, and the continue statement.

The break Statement:

The **break** statement in Java Programming has 2 usage:

- The **break** statement is used to terminate a **case** in the **switch** statement.
- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

Syntax:

```
break;
```

31. Program:

```
class BreakStatementDemo1 {
    public static void main(String[]args) {
        System.out.println("The while(true) loop, with the break statement:");
        System.out.println("Numbers from 1 to 100 are :");
        int no = 1;
        while(true) {
            System.out.print(no + " ");
            ++no;
            if(no > 100)
                break;
        }
        System.out.println("\nThank you.");
    }
}
```

The continue Statement:

The **continue** statement in Java Programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.
In the **for** loop, the **continue** statement causes the increment/decrement portions and conditional test of the loop to execute.
In the **while** and **do-while** loops, the **continue** statement causes the program control to pass to the conditional test.

Syntax:

```
continue;
```

32. Program:

```
class ContinueStatementDemo1 {
    static public void main(String[]args) {
        System.out.println("The continue statement in the for loop :");
        System.out.println("Even numbers from 1 to 100 are :");
        for(int no = 1; no <= 100; ++no) {
```

```
            if(no % 2 == 1)
                continue;
            System.out.print(" " + no);
        }
        System.out.println("\nThank you.");
    }
}
```

Homework:

1. WAP to check whether the input number is PRIME or NOT.
2. WAP to find the factorial of the input number.
3. WAP to print fibonacci series from 0 to 200

```
0 1 1 2 3 5 8 13 21 34 55 89 ...
```

4. WAP to print tribonacci series from 0 to 500

```
0 1 1 2 4 7 13 24 44 81 ...
```

5. WAP to check whether the number is an armstrong number or not.

```
153 = 1*1*1 + 5*5*5 + 3*3*3
= 1 + 125 + 27
= 153
As LHS and RHS are equal to each other, 153 is an armstrong number.
Number 407,
```

6. WAP to check whether the integer number is palindrome or not.

```
0-9, 11, 22, 33, 44, 55, 66, 77, 88, 99, 101, 111, 121, 131, 141, 151, 161, 171, 181, 191, 202,
212, 12321, 4545454, 5005, etc.
```

Nested Loops:

Java Programming language allows one loop inside another loop. This is known as nesting of loops.

Syntax:

```
for(initialization; condition; iteration)  OR while(expression) {
      statements;
      ...
      for(initialization; condition; iteration)  OR while(expression) {
            statements;
            ...
      }
      ...
}
```

A note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a "**for**" loop can be inside a "**while**" loop or vice-versa.

Note: While execution of nested loops, whenever the outer loop is iterated once, the inner loop iterates to the fullest, that is till it exhausts.

33. Program:

```
class NestedLoop1 {
    public static void main(String[]args) {
        int out, in;
        System.out.println("Nested Loop :");
        for(out = 1; out <= 5; ++out) {
```

```
            for(in = 1; in <= 5; ++in) {
                System.out.println(out + " : " + in);
            }
            System.out.println();
        }
        System.out.println("Thank you.");
    }
}
```

34. Program:

```
class NestedLoops2 {
    public static void main(String[]args) {
        int out, in;
        System.out.println("Nested Loop:");
        out = 1;
        while(out <= 5) {
            in = 1;
            while(in <= 5) {
                System.out.print((out + in) + " ");
                ++in;
            }
            System.out.println();
            ++out;
        }
        System.out.println("Thank you.");
    }
}
```

35. Program:

```
class NestedLoops3 {
    public static void main(String[]args) {
        int out, in;
        System.out.println("Nested Loop:");
        for(out = 1; out <= 5; ++out) {
            for(in = 1; in <= out; ++in) {
                System.out.print(" * ");
            }
            System.out.println();
        }
        System.out.println("Thank you.");
    }
}
```

Homework:

1. WAP to print squares of numbers from 1 to 10, using nested loops.
2. WAP to print the tables from 1 to 10 numbers using a nested loop.
3. WAP to print the following patterns:

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

```
1
1   2
1   2   3
1   2   3   4
1   2   3   4   5
```

```
a
a   b
a   b   c
a   b   c   d
a   b   c   d   e
```

```
A
B B
C C C
D D D D
E E E E E
```

```
9
8   7
6   5   4
3   2   1   0
```

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

```
        A
      A   A
    A   A   A
  A   A A   A
A   A   A   A   A
```

V - Variables in Java:

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type. Variable is a name of memory location.
There are three types of variables in Java:

- Local variables
- Instance variables, and
- Static (class) variables

Local Variables:

A variable declared inside the body of a method, or a block of code which is inside a method (if conditional statement, looping statements, etc), is known as a local variable. You can use this variable only within that method or block and the other methods/blocks in the class/program aren't even aware that the variable exists.

36. Program:

```
class LocalVariablesDemo1 {
    public static void main(String[]args) {
        int a;
        /* ERROR : variable a might not have been initialized
        System.out.println("Inside main() method, variable a : " + a); */
        a = 15;
        System.out.println("Inside main() method, variable a : " + a);
        if(true) {
            System.out.println("Inside the if block, variable a : " + a);
            ++a;
            int b = 6;
            System.out.println("Inside the if block, variable b : " + b);
        }
        System.out.println("Inside main() method, after the if block, variable a : " + a);
        /* ERROR : : cannot find symbol
        System.out.println("Inside main() method, after the if block, variable b : " + b); */
    }
}
```

Note that, in Java local variables have to be initialized before using them. Without initialization or without assigning values to declared local variables, we cannot use them.
Also observe that a local variable, declared inside a local block, cannot be accessed outside of its declared local block, just link C and C++ programming Languages.

37. Program:

```
class LocalVariablesDemo2 {
    public static void main(String[]args) {
        int no1 = 24;
        System.out.println("Inside the main() method, variable no1 : " + no1);
        if(no1 != 0) {
            System.out.println("Inside the if block, variable no1 : " + no1);
            /* ERROR : variable no1 is already defined in method main(String[])
            int no1 = 150; */
```

```
            int no2 = 150;
            System.out.println("Inside the if block, variable no2 : " + no2);
        }
        System.out.println("Inside main() method, after if block, variable no1 : " + no1);
        /* ERROR : cannot find symbol
        System.out.println("Inside main() method, after if block, variable no2 : " + no2); */
        int no2 = 51;
        System.out.println("Inside main() method, after if block, variable no2 : " + no2);

    }
}
```

Note that, a local variable cannot be defined as **static**.

Instance Variables:

A variable declared inside the class but outside the body of any method, is called an instance variable. It is not declared as **static**.
It is called an instance variable because its value is instance-specific and is not shared among instances. Instance variable does not belong to any specific method or class, it belongs to the instance/object/entity of the class in which it is declared.
Instance variables can be used directly in any non-static method of the same class. These variables cannot be directly used inside any static methods or any other classes. To use an instance variable, inside a static method, or inside any other class, we first need to declare and define the instance/object of the class in which the instance variable is declared.

Declaring and Initializing Instance/Object of Class:

The "**new**" operator, that is the **dynamic memory allocation** operator, is used to initialize the instance/object of a class.

Syntax

```
ClassName objectName;           // This is the declaration of an object of ClassName class
objectName = new ClassName();  // This is initialization of the object
```

OR

```
// Declaration as well as initialization of an object
ClassName objectName = new ClassName();
```

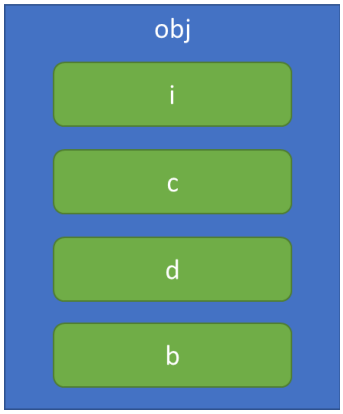
As soon as we initialize the object of our class, instance variables are initialized. A new memory is allocated to the object and also to the instance variables and default initial values are allocated to the instance variables.

38. Program:

```
class InstanceVariables1 {
    int i;
    char c;
    double d;
    boolean b;
    public static void main(String[]args) {
        System.out.println("Instance variables :");
        /* ERROR : non-static variable i cannot be referenced from a static context
```

```
        System.out.println("i : " + i);
        System.out.println("c : " + c);
        System.out.println("d : " + d);
        System.out.println("b : " + b); */
        InstanceVariables1 obj;
        obj = new InstanceVariables1();
        System.out.println("Instance/object of the class initialized.");
        System.out.println("obj.i : " + obj.i);      // 0
        System.out.println("obj.c : " + obj.c);      // NULL '\0'
        System.out.println("obj.d : " + obj.d);      // 0.0
        System.out.println("obj.b : " + obj.b);      // false
        obj.i = 60;
        obj.c = 'J';
        obj.d = 12.2022;
        obj.b = true;
        System.out.println("After assigning new values to the instance variables :");
        System.out.println("obj.i : " + obj.i);
        System.out.println("obj.c : " + obj.c);
        System.out.println("obj.d : " + obj.d);
        System.out.println("obj.b : " + obj.b);
        System.out.println("Thank you.\n");
    }
}
```

Memory Representation:



Multiple Objects:

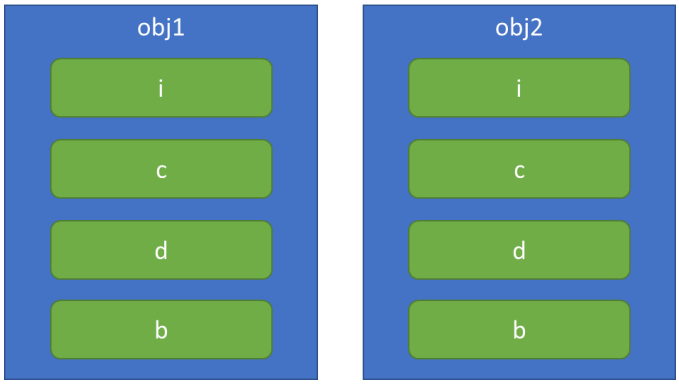
As stated, we can create/initialize multiple instances/objects of the class, which will create multiple copies of the instance variables.
We can also assign default initial values to instance variables at the time of their declaration.

39. Program:

```
class InstanceVariables2 {
    int i = 6;
    char c = 'S';
    double d = 12.2022;
    boolean b = true;
```

```
    public static void main(String[]args) {
        System.out.println("Instance variables : Multiple Objects/Instances :");
        InstanceVariables2 obj1;
        obj1 = new InstanceVariables2();
        System.out.println("\nObject 1 of class InstanceVariables2 initialized.");
        System.out.println("Default initial values of data members of Object 1 :");
        System.out.println("obj1.i : " + obj1.i + "\nobj1.c : " + obj1.c);
        System.out.println("obj1.d : " + obj1.d + "\nobj1.b : " + obj1.b);
        InstanceVariables2 obj2 = new InstanceVariables2();
        System.out.println("\nObject 2 of class InstanceVariables2 initialized.");
        System.out.println("Default initial values of data members of Object 2 :");
        System.out.println("obj2.i : " + obj2.i + "\nobj2.c : " + obj2.c);
        System.out.println("obj2.d : " + obj2.d + "\nobj2.b : " + obj2.b);
        obj1.i = 15;
        obj1.c = 'J';
        obj1.d = 3.14;
        obj1.b = false;
        obj2.i = 24;
        obj2.c = 'V';
        obj2.d = 2022.12;
        obj2.b = true;
        System.out.println("\nAfter assigning new values to data member of both objects :");
        System.out.println("\nNew values of data members of Object 1 :");
        System.out.println("obj1.i : " + obj1.i + "\nobj1.c : " + obj1.c);
        System.out.println("obj1.d : " + obj1.d + "\nobj1.b : " + obj1.b);
        System.out.println("\nNew values of data members of Object 2 :");
        System.out.println("obj2.i : " + obj2.i + "\nobj2.c : " + obj2.c);
        System.out.println("obj2.d : " + obj2.d + "\nobj2.b : " + obj2.b);
        System.out.println("Thank you.\n");
    }
}
```

Memory Representation:



Instance Methods:

In general to access/assign/show values of the instance variable (data members), instance methods (member methods) are used. Instance methods are non-static methods that are invoked/called using the class instance/object.
The declaration/definition of instance methods is exactly the same as of any method in Java.

40. Program:

```
class InstanceVariables3 {
    int i = 6;
    char c = 'S';
    double d = 12.2022;
    boolean b = true;
    public static void main(String[]args) {
        System.out.println("Instance variables and methods :");
        InstanceVariables3 obj1;
        obj1 = new InstanceVariables3();
        System.out.println("\nObject 1 of class InstanceVariables3 initialized.");
        System.out.println("Default initial values of data members of Object 1 :");
        obj1.showData();
        InstanceVariables3 obj2 = new InstanceVariables3();
        System.out.println("\nObject 2 of class InstanceVariables3 initialized.");
        System.out.println("Default initial values of data members of Object 2 :");
        obj2.showData();
        obj1.setData(15, 'J', 3.14, false);
        obj2.setData(24, 'V', 2022.12, true);
        System.out.println("\nAfter assigning new values to data member of both objects :");
        System.out.println("\nNew values of data members of Object 1 :");
        obj1.showData();
        System.out.println("\nNew values of data members of Object 2 :");
        obj2.showData();
        System.out.println("Thank you.\n");
    }
    void setData(int in, char ch, double du, boolean bo) {
        i = in;
        c = ch;
        d = du;
        b = bo;
    }
    void showData() {
        System.out.println("i : " + i + "\nc : " + c + "\nd : " + d + "\nb : " + b);
    }
}
```

Static Variables (Class Variables):

A variable that is declared as **static** is called a static variable. It cannot be local. A single copy of the static variable is created and is shared among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.
Static variables can be directly used inside static methods, both belonging to 1 single same class.
If a static variable is in a class and the method using which you access the static variable is in another class then we can access the static variable using its class name and the period/dot (.) operator.

41. Program:

```
class StaticVariables1 {
    static int i;
    static char c;
    static double d;
    static boolean b;
    public static void main(String[]args) {
        System.out.println("Static variables :");
        System.out.println("Default initial values :");
        System.out.println("i : " + i + "\nc : " + c + "\nd : " + d + "\nb : " + b);
        i = 60;
        c = 'J';
        d = 12.2022;
        b = true;
        System.out.println("\nAfter assigning new values to the static variables :");
        System.out.println("i : " + i + "\nc : " + c + "\nd : " + d + "\nb : " + b);
        System.out.println("Thank you.\n");
    }
}
```

Static Methods:

The **static** variables are generally accessed/used inside the class through the **static** method(s). Static methods are like static data members, which can be invoked/called directly into the other static methods, all belonging to the same class. Other classes can invoke/call static methods using the class name and the period/dot (.) operator.
The definition of a static method is similar to the definition of any class instance method, with a change that prefixes the definition with the **static** keyword.

42. Program:

```
class StaticMembers1 {
    static int i = 24;
    static char c = 'P';
    static double d = 12.2022;
    static boolean b = true;
    public static void main(String[]args) {
        System.out.println("Static variables :");
        System.out.println("Default initial values :");
        showData();
        StaticMembers1 obj = new StaticMembers1();
```

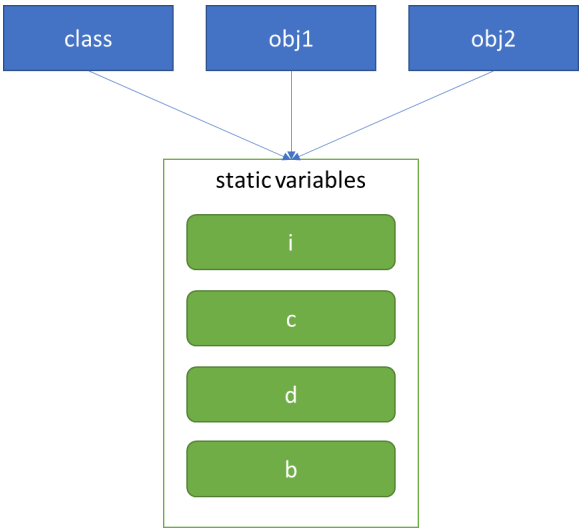
```
        System.out.println("\nDefault initial values of static variables, using object :");
        System.out.println("obj.i : " + obj.i + "\nobj.c : " + obj.c);
        System.out.println("obj.d : " + obj.d + "\nobj.b : " + obj.b);
        i = 60;
        c = 'J';
        obj.d = 12.2022;
        obj.b = true;
        System.out.println("\nAfter assigning new values to the static variables :");
        showData();
        System.out.println("\nValues of static variables, using object :");
        obj.showData();
        System.out.println("Thank you.\n");
    }
    static void showData() {
        System.out.println("i : " + i + "\nc : " + c + "\nd : " + d + "\nb : " + b);
    }
}
```

43. Program:

```
class StaticMembers2 {
    public static void main(String[]args) {
        System.out.println("Static variables :");
        System.out.println("Default initial values :");
        StaticMembers.showData();
        StaticMembers.i = 15;
        StaticMembers.c = 'G';
        StaticMembers.d = 3.14;
        StaticMembers.b = false;
        System.out.println("\nAfter setting new values :");
        StaticMembers.showData();
        StaticMembers obj1 = new StaticMembers();
        System.out.println("\nValues of static variables, through object 1 :");
        obj1.showData();
        obj1.setData(123, 'Q', 2.5, true);
        System.out.println("\nAfter setting new values, through object 1 :");
        StaticMembers.showData();
        StaticMembers obj2 = new StaticMembers();
        System.out.println("\nValues of static variables, through object 2 :");
        obj2.showData();
        obj2.i = 1122;
        obj2.c = 'Y';
        StaticMembers.d = 3.6;
        StaticMembers.b = false;
        System.out.println("\nAfter setting new values :");
        obj1.showData();
        System.out.println("Thank you.");
    }
}
```

```
class StaticMembers {
    static int i = 24;
    static char c = 'P';
    static double d = 12.2022;
    static boolean b = true;
    static void showData() {
        System.out.println("i : " + i + "\nc : " + c + "\nd : " + d + "\nb : " + b);
    }
    void setData(int in, char ch, double du, boolean bo) {
        i = in;
        c = ch;
        d = du;
        b = bo;
    }
}
```

Memory Representation:



VI - Type Conversion and Casting:

It is common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion implicitly (automatically). For example, it is always possible to assign a **short** value/variable to an **int** variable, or a **float** value/variable to a **double** variable.
However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no automatic conversion defined from **double** to **byte**.
Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a *cast*, which performs an explicit conversion between incompatible types.

Java's Implicit Conversions:

When one type of data is assigned to another type of variable, an *implicit* (*automatic) type conversion* will take place if the following two conditions are met:

- The two types are compatible
- The destination type is larger than the source type

44. Program:

```
class TypeConversion1 {
    static public void main(String[]args) {
        byte b = 15;
        int i;
        float f = 3.14f;
        double d;
        System.out.println("Byte variable b : " + b);
        System.out.println("Float variable f : " + f);
        i = b;
        d = f;
        System.out.println("\nImplicit type conversion of compatible types :");
        System.out.println("Byte variable b assigned to int variable i : " + i);
        System.out.println("Float variable f assigned to double type variable d : " + d);
        short s = 60;
        System.out.println("\nShort integer variable s : " + s);
        // b = s; // ERROR : incompatible types: possible lossy conversion from short to byte
        f = s;
        System.out.println("\nImplicit type conversion of incompatible types :");
        System.out.println("Short variable s assigned to float variable f : " + f);
        long l = 123;
        System.out.println("\nLong integer variable l : " + l);
        // l = f; // ERROR : incompatible types: possible lossy conversion from float to long
    }
}
```

Casting incompatible types:

Although the implicit type conversions are helpful, they will not fulfill all needs. To create a conversion between two incompatible types, you must use a **cast**.

A cast is simply an explicit type conversion. It has a general form as:

General form:

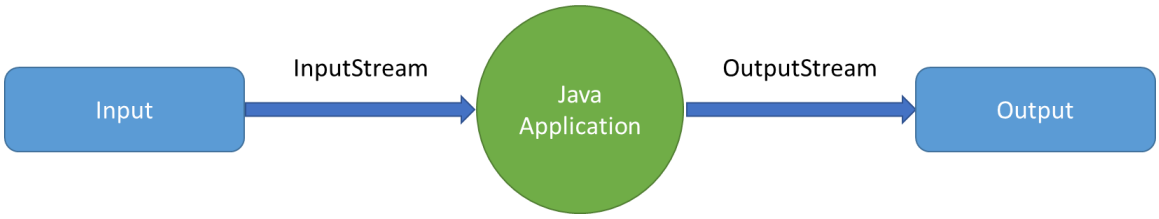
```
(targetType)value/variable
```

45. Program:

```
class TypeConversion2 {
    static public void main(String[]args) {
        byte b = 15;
        float f;
        short s = 60;
        System.out.println("Short integer variable s : " + s);
        System.out.println("Conversion of incompatible types :");
        b = (byte)s;
        f = (float)s;
        System.out.println("Short variable s assigned to byte variable b : " + b);
        System.out.println("Short variable s assigned to float variable f : " + f);
        long l;
        f = (double)12.34;  // OR   f = 12.34f;
        System.out.println("\nFloat variable f : " + f);
        l = (long)f;
        System.out.println("Conversion of incompatible types :");
        System.out.println("Float variable f assigned to long variable l : " + l);
        int i;
        double d = 1234.5678;
        System.out.println("\nDouble floating-point variable d : " + d);
        i = (int)d;
        System.out.println("Conversion of incompatible types :");
        System.out.println("Double floating-point variable d assigned to int i : " + i);
    }
}
```

VII - Input and Output Operations in Java:

Java provides various Streams with its I/O package that helps the user to perform all the input-output operations. These streams support all the types of objects, data-types, characters, files etc to fully execute the I/O operations.



Before exploring various input and output streams let us look at 3 standard or default streams that Java provides us which are most commonly used. 2 stream objects come under output and 1 stream object is for input.

Output Stream Objects:

- System.out
- System.err

System.out:

This is the standard output stream object that is used to produce the result of a program on an output device like the computer screen (console).
Here is a list of the various print methods that we use to output result:

1. The print( ) method:

The **print( )** method in Java is used to display a text on the console. It accepts just a single string parameter. This method prints the given string on the console and the cursor remains at the end of the string on the console. The next printing takes place from just after the previous string.
When you have to print the value of a variable, you have to concatenate (+) the variable/value/expression with the string and then the **print( )** method will show it as a single string on the output screen.

General form:

```
System.out.print("StringLiteral ... \n");
System.out.print("StringLiteral " + variable + "..." + (expression) + "... \n");
```

2. The println( ) method:

The **println( )** method in Java is also used to display a text on the console. It too accepts just a single string parameter. It prints the given string on the console and then the cursor moves to the start of the next line at the console after printing the given string. The next printing takes place from the next line.
Here too we will have to concatenate the value with the string in case of printing of any value/variable/expression.

General form:

```
System.out.println("StringLiteral ... ");
System.out.println("StringLiteral " + variable + " ..." + (expression) + "...");
```

3. The printf( ) method:

Sometimes in competitive programming, it is essential to print the output in a given specified format. Most users are familiar with the **printf( )** function in C. Let us discuss how we can format the output in Java, using the **printf( )** method.

This is the easiest of all methods as this is similar to the **printf( )** function in C. Note that **System.out.print( )** and **System.out.println( )** methods take a single string argument only, but the **System.out.printf( )** function may take multiple arguments.

General form:

```
System.out.printf("StringLiteral ... %formatCharacter ...\n", value/variable, ...);
```

46. Program:

```
class OutputMethods1 {
    public static void main(String[]args) {
        System.out.print("Output methods in Java :\n");
        System.out.print("1. System.out.print() method.\n");
        System.out.println("2. System.out.println() method.");
        System.out.printf("3. System.out.printf() method.\n");
        int i = 15;
        float f = 3.14f;
        String s = "Java Output Stream";
        boolean b = true;
        System.out.println("Printing values of variables using various print methods:");
        System.out.print("print() : i : "+i+", f : "+f+", s : " + s + " and b : " + b + "\n");
        System.out.println("println() : i : "+i+", f : " + f + ", s : "+s+" and b : " + b);
        System.out.printf("printf() : i : %d, f : %g, s : %s and b : %b\n", i, f, s, b);
        System.out.println("Thank you.");
    }
}
```

System.err:

This is the standard error stream object that is used to output all the error data that a program might generate, on a computer screen or any standard output device. Generally this standard error stream object is used in exception handling mechanisms, inside the **catch** block.

This stream object uses all the above-mentioned 3 methods to output the error data:

- print( )
- println( )
- printf( )

```
class OutputMethods2 {
    public static void main(String[]args) {
        System.err.println("Output Stream object : System.err.");
        int n = 10, d = 0, div;
        try {
            div = n / d;
            System.out.print("Division of " + n + " and " + d + " is " + div + "\n");
            System.out.println("Division of " + n + " and " + d + " is " + div);
            System.out.printf("Division of %i and %i is %i\n", n, d, div);
        }
        catch(Exception ex) {
            System.err.print("ERROR : ");
```

```
            System.err.printf("%s\n", ex);
        }
        System.out.println("Thank you.");
    }
}
```

Input Stream Object:

The input stream classes are used to read data that must be taken as an input from a source array or file or any peripheral device. For eg., FileInputStream, BufferedInputStream, ByteArrayInputStream, etc.
Although Java provides us with various input stream classes, it also provides a standard input stream object, that is the **System.in** object.

In Java, there are various different ways for reading input from the user in the command line environment (console). Among which some of the classes and their use is defined here.

The BufferedReader Class:

This is Java's classical method to accept input, introduced in JDK1.0. This method is used by wrapping the **System.in** (standard input stream) object in an **InputStreamReader** class instance which is wrapped in a **BufferedReader** class instance; we can read input from the user in the command line using this class' object.
The input is buffered for efficient reading. The wrapping code is hard to remember.

To use the **BufferedReader** class and **InputStreamReader** class you need to **import** the **java.io** package in which these classes have been defined.
These classes may throw run-time exceptions so we need to implement the code in the exception handling mechanism.

General form:

```
import java.io.*;
...
      BufferedReader obj = new BufferedReader(new InputStreamReader(System.in));
      try {
            var = obj.readLine();
      }
      catch(Exception object) {
            object...
      }
```

48. Program:

```
import java.io.*;
class BufferedReader1 {
    public static void main(String[]str) {
        BufferedReader objBR = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter your name :");
        try {
            String name = objBR.readLine();
            System.out.println("Hello : " + name);
        }
        catch(Exception ex) {
```

```
            System.err.println("ERROR : " + ex);
        }
        System.out.println("Thank you.");
    }
}
```

As we can see, the **readLine( )** method of **BufferedReader** class accepts input in **String** type only. To read other types, we can use type casting/conversion methods.

To read other types, we use type casting/conversion methods like **Integer.parseInt( )**, **Double.parseDouble( )**, etc. To read multiple values, we need to use the string's **split( )** method.

49. Program:

```
import java.io.*;
class BufferedReader2 {
    static public void main(String[]str) {
        BufferedReader objBR = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Enter 2 numbers :");
        try {
            int no1 = Integer.parseInt(objBR.readLine());
            int no2 = Integer.parseInt(objBR.readLine());
            System.out.println("Addition : " + (no1 + no2));
        }
        catch(Exception ex) {
            System.err.println("ERROR : " + ex);
        }
        System.out.println("Thank you.");
    }
}
```

Multiple inputs to multiple variables of multiple data types:

50. Program:

```
import java.io.*;
class BufferedReader3 {
    public static void main(String[]args) {
        try {
            BufferedReader objBR = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Enter student details:");
            System.out.print("Name:");
            String name = objBR.readLine();
            System.out.print("Roll No:");
            int rn = Integer.parseInt(objBR.readLine());
            System.out.print("Percentage:");
            float per = Float.parseFloat(objBR.readLine());
            System.out.println("\nStudent details are:");
            System.out.println("Name : "+name+", Roll No : "+rn+", and Percentage : "+per+"%");
        }
        catch(Exception ex) {
```

```
            System.err.println("ERROR : " + ex);
        }
        System.out.println("Thank you.");
    }
}
```

The Scanner Class:

Another way to accept input from the user is through the methods of the **Scanner** class. The **Scanner** class is present within the **java.util** package, so to use it you need to first **import** the **java.util** package.
This is probably the most preferred way to take input from the user. The main purpose of the **Scanner** class is to parse primitive types and strings using regular expressions, however, it can also be used to read input from the user in the command line.
The object of the **Scanner** class is initialized by wrapping the **System.in** (standard input stream) object in its constructor. We can read input from the user in the command line for which various methods are provided to us.
The **Scanner** class provides us with the convenient methods to accept primitive type inputs.

51. Program:

```
import java.util.*;
class ScannerDemo1 {
    public static void main(String[]args) {
        Scanner objScan = new Scanner(System.in);
        System.out.println("Enter student details :");
        System.out.print("Name :");
        String name = objScan.nextLine();
        System.out.print("Roll No :");
        int rn = objScan.nextInt();
        System.out.print("Percentage:");
        float per = objScan.nextFloat();
        System.out.println("\nStudent details are:");
        System.out.println("Name : "+name+", Roll No : "+rn+", and Percentage : "+per+"%");
    }
}
```

Difference Between BufferedReader and Scanner Class:

a. BufferedReader is a very basic way to read the input generally used to read the stream of characters. It gives an edge over Scanner as it is faster than Scanner because Scanner does lots of post-processing for parsing the input; as seen in **nextInt( )**, **nextFloat( )**, **nextDouble( )**, etc methods.
b. BufferedReader is more flexible as we can specify the size of stream input to be read. (In general, BufferedReader reads larger input than Scanner).
c. These two factors come into play when we are reading larger inputs. In general, the Scanner Class serves the input that belongs to various different types.
d. BufferedReader is preferred as it is synchronized. While dealing with multiple threads it is preferred.
e. For decent input, easy readability, the Scanner is preferred over BufferedReader.
f. The code to accept input using Scanner is easy, short and simple as compared to BufferedReader.

The Console Class:

Use of the **Console** class has been becoming a preferred way for reading user's input from the command line. In addition, it can be used for reading password-like input without echoing (*) the characters entered by the user; the format string syntax can also be used (like **System.out.printf( )**). The object of **Console** class is returned by the **System.console( )** method using which we accept input from the user.

52. Program:

```
class ConsoleDemo1 {
    public static void main(String[]args) {
        System.out.println("Enter student details :");
        System.out.print("Name :");
        String name = System.console().readLine();
        System.out.print("Roll No :");
        int rn = Integer.parseInt(System.console().readLine());
        System.out.print("Percentage:");
        float per = Float.parseFloat(System.console().readLine());
        System.out.println("\nStudent details are:");
        System.out.println("Name : "+name+", Roll No : "+rn+", and Percentage : "+per+"%");
    }
}
```

Command Line Arguments:

Sometimes we will want to pass information into a program when we run it. This is accomplished by passing *command-line arguments* to the **main( )** method.
It is the most used user input for competitive coding. The command-line arguments are stored in the String format.
The accepted input can be type casted into any desired type. Such as, the **parseInt( )** method of the **Integer** class converts string arguments into **Integer**. Similarly, the **parseFloat( )** method of the **Float** class can be used to type cast the command line string input value into float, as so on, during execution. The usage of **args[ ]** comes into existence in this input form. The passing of information takes place during the execution of a program. The command line is given to **args[ ]**. These programs have to be run on **cmd**. The arguments passed are accepted by the **String [ ]** parameter defined in the **main( )** method.

53. Program:

```
class CmdLineArgs1 {
    public static void main(String[]args) {
        System.out.println("Command Line Arguments passed are : " + args.length);
        if(args.length > 0) {
            System.out.println("Command Line Arguments are :");
            for(int i = 0; i < args.length; ++i)
                System.out.printf("args[%d] : %s\n", i, args[i]);
        }
        else
            System.out.println("No arguments passed.");
        System.out.println("Thank you.");
    }
}
```

VIII - Arrays:

An array is a group of like-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. Arrays offer a convenient means of grouping related information.
Array is a variable consisting of a sequence of elements belonging to the same data type.

One-Dimensional Array:

A one-dimensional array is, essentially, a list of like-typed variables. To use an array, we first must declare/create an array variable of the desired type.

Declaration of Array:
Syntax:
```
dataType arrayName[];
```
OR
```
dataType []arrayName;
```

Example:
```
int a[];
```
OR
```
int []a;
```

Once the array name has been declared, next we need to use the "**new**" (dynamic memory allocation) operator to allocate memory for the declared array.

Initialization of Array:
Syntax:
```
arrayName = new dataType[SIZE];
```

Example:
```
a = new int[10];
```

The "**new**" operator is the dynamic memory allocation operator. It is used to allocate memory dynamically, that is at run time. In Java arrays are declared at compile time, but the memory allocation is done at run-time.

The declaration of array name and allocation of memory, both can be done at the same time, that is in 1 single statement as follows:

Declaration and Initialization of Array:
Syntax:
```
dataType arrayName[] = new dataType[SIZE];
```
OR
```
dataType []arrayName = new dataType[size];
```

Example:
```
int a[] = new int[10];
```
OR
```
int []a = new int[10];
```

As arrays in Java are defined using the "**new**" operator, the arrays are initialized and by default all the elements are set to value **0**.

54. Program:

```
class ArrayDemo1 {
    public static void main(String args[]) {
        int a[];
        a = new int[5];
        System.out.println("Default initial values in the array \"a\" are :");
        System.out.println("a[0] : " + a[0]);
        System.out.println("a[1] : " + a[1]);
        System.out.println("a[2] : " + a[2]);
        System.out.println("a[3] : " + a[3]);
        System.out.println("a[4] : " + a[4]);
        System.out.println("Thank you.");
    }
}
```

Assigning Elements to Array:

Like C and C++, Java uses index numbers enclosed within subscripts (square brackets), that start from "**0**" and goes up to "**SIZE-1**" to access, or store elements from, or in the array.

Syntax:

```
arrayName[index] = value;
```

Example:

```
a[0] = 10;
a[1] = 20;
a[2] = 30;
a[3] = 40;
a[4] = 50;
```

55. Program:

```
class ArrayDemo2 {
    public static void main(String args[]) {
        int a[];
        a = new int[5];
        System.out.println("Array \"a\" defined with size 5.");
        a[0] = 10;
        a[1] = 20;
        a[2] = 30;
        a[3] = 40;
        a[4] = 50;
        /* a[5] = 60;
        Runtime Error : Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
        Index 5 out of bounds for length 5 */
        System.out.println("Values in the array \"a\" are :");
        System.out.println("a[0] : " + a[0]);
        System.out.println("a[1] : " + a[1]);
        System.out.println("a[2] : " + a[2]);
        System.out.println("a[3] : " + a[3]);
```

```
        System.out.println("a[4] : " + a[4]);
        /* System.out.println("a[5] : " + a[5]);
        Runtime Error : Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
        Index 5 out of bounds for length 5 */
        System.out.println("Thank you.");
    }
}
```

Array Length Property:

In Java arrays have an in-built property known as **length**. This property returns an integer value specifying the array's length/size.

Syntax:

```
arrayName.length
```

56. Program:

```
import java.io.*;
class ArrayDemo3 {
    static public void main(String[]args) {
        BufferedReader objBR = new BufferedReader(new InputStreamReader(System.in));
        try {
            System.out.print("Enter the number of elements for an array :");
            int s = Integer.parseInt(objBR.readLine());
            int arr[] = new int[s];
            for(int i = 0; i < s; ++i)
                arr[i] = (i + 1) * 10;
            System.out.println("Array's " + arr.length + " values are :");
            for(int i = 0; i < s; ++i)
                System.out.println("arr[" + i + "] : " + arr[i]);
        }
        catch(Exception ex) {
            System.err.println("ERROR : " + ex);
        }
        System.out.println("Thank you.");
    }
}
```

Array Initialization:

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An array initializer is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements we specify in the array initializer. There is no need to use the **new** operator.

Syntax:

```
dataType arrayName[] = {element1, element2, element3, element4, ...};
```

57. Program:

```
class ArrayInitialization1 {
    static public void main(String[]args) {
        int a[] = {2, 4, 6, -10, 24, 36, -5, 0, 51, 10};
        System.out.println("Array's " + a.length + " values are :");
        for(int i = 0; i < a.length; ++i)
            System.out.println("a[" + i + "] : " + a[i]);
        System.out.println("Thank you.");
    }
}
```

Homework:

1. WAP to accept 10 numbers from the user and print all even numbers among them.
2. WAP to accept 10 numbers from the user and print all negative numbers among them.
3. WAP to accept 10 numbers in an array and search for a value in it.
4. WAP to accept 10 values from the user and print the smallest value.
5. WAP to accept 10 values from the user and print the largest value.
6. WAP to accept 10 values from the user and print the sum of all values.
7. WAP to accept 10 values from the user and print the average value of all 10 values.
8. WAP to find the binary of the given number.
9. WAP to accept binary number from the user and find out its equivalent decimal number.
10. WAP to accept 10 values from the user and copy it into another array.
11. WAP to accept 10 values from the user and sort them in ascending order.

Multidimensional Array:

In Java, multidimensional arrays are actually arrays of arrays. These, as we might expect, look and act like regular multidimensional arrays. However, as we 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.

Two-Dimensional Array:

A two-dimensional array is an array of arrays. It can be thought of as a matrix, that is a table (row and column) like structure. Here we declare 2 dimensions using 2 sets of square brackets.

Declaration of a Two-dimensional Array:

As stated, while declaring a two-dimensional array specify 2 subscripts (2 sets of square brackets) with the array name.

Syntax:

```
dataType arrayName[][];
```

OR

```
dataType [][]arrayName;
```

Example:

```
int a2d[][];
```

OR

```
int [][]a2d;
```

Defining Two-Dimensional Array:

We can define a two-dimensional array using the same **new** operator and the size of both the dimensions.

Syntax:
```
arrayName = new dataType[rowSize][columnSize];
```

Example:
```
a2d = new int[3][4];
```

The above array can be represented as:

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]

Declaring and defining 2D array:

Declaration and definition of two-dimensional arrays can be done at the same time as:

Syntax:
```
dataType arrayName[][] = new dataType[rowSize][columnSize];
```
OR
```
dataType [][]arrayName = new dataType[rowSize][columnSize];
```

Example:
```
int a2d[][] = new int[3][4];
```
OR
```
int [][]a2d = new int[3][4];
```

Assigning elements to Two-Dimensional Array:

While assigning an element(s) to a two-dimensional array, we need to specify the name of the array, and both the index numbers, first of the row and second of the column, in 2 different sets of subscript operators (square brackets).

Syntax:
```
arrayName[rowIndex][columnIndex]  = element;
```

Example:
```
a2d[0][0] = 10; // Element value 10 set at location first row first cell
a2d[0][1] = 20; // Element value 20 set at location first row second cell
...
a2d[1][0] = 50; // Element value 50 set at location second row first cell
...
a2d[2][2] = 10; // Element value 10 set at location third row third cell
...
```

58. Program:
```
class TwoDimensionalArray1 {
    static public void main(String[]args) {
```

```
        int a2d[][];
        a2d = new int[3][4];
        a2d[0][0] = 10;
        a2d[0][1] = 20;
        a2d[0][2] = 30;
        a2d[0][3] = 40;

        a2d[1][0] = 50;
        a2d[1][1] = 60;
        a2d[1][2] = 70;
        a2d[1][3] = 80;

        a2d[2][0] = 90;
        a2d[2][1] = 100;
        a2d[2][2] = 110;
        a2d[2][3] = 120;

        System.out.println("Array with size 3x4 have total " + (a2d.length * a2d[0].length) + "
elements as :");
        for(int r = 0; r < a2d.length; ++r) {
            for(int c = 0; c < a2d[r].length; ++c) {
                System.out.print("\t" + a2d[r][c]);
            }
            System.out.println("");
        }
        System.out.println("Thank you.");
    }
}
```

Initialization of Two-Dimensional Array:

We can initialize a two-dimensional array using elements. The elements should be enclosed within curly braces, according to our desired size.

59. Program:

```
class TwoDimensionalArray2 {
    static public void main(String[]args) {
        /* int a2d[][];
        a2d = new int[3][4];
        a2d[0][0] = 10;
        a2d[0][1] = 20;
        a2d[0][2] = 30;
        a2d[0][3] = 40;

        a2d[1][0] = 50;
        a2d[1][1] = 60;
        a2d[1][2] = 70;
```

```
        a2d[1][3] = 80;

        a2d[2][0] = 90;
        a2d[2][1] = 100;
        a2d[2][2] = 110;
        a2d[2][3] = 120; */

        // OR

        int twoDArray[][] = {{10, 20, 30, 40}, {50, 60, 70, 80}, {90, 100, 110, 120}};
        System.out.println("Array with size 3x4 have total " + (twoDArray.length *
twoDArray[0].length) + " elements as :");
        for(int r = 0; r < twoDArray.length; ++r) {
            for(int c = 0; c < twoDArray[r].length; ++c) {
                System.out.print("\t" + twoDArray[r][c]);
            }
            System.out.println("");
        }
        System.out.println("Thank you.");
    }
}
```

The initialization of multi-dimensional (2-dimensional) arrays can be done by just specifying 1 dimension (row). Means, when we allocate memory for a multidimensional array, we need to only specify the memory for the first (leftmost) dimension. We can allocate the remaining dimensions separately.

Syntax:

```
dataType arrayName[][] = new dataType[rowSize][];
```

Example:

```
int a[][] = new int[3][];
```

Next, we have to separately allocate the second dimension.

Syntax:

```
arrayName[rowIndex] = new dataType[columnSize];
```

Example:

```
a[0] = new int[4];
a[1] = new int[4];
a[2] = new int[4];
```

Homework:

1. WAP to perform addition and subtraction of 2 matrices.
2. WAP to perform multiplication of 2 matrices.
3. WAP to find the transpose of the given matrix.

Sparse Matrix:

A sparse matrix is a matrix that is composed of mostly zero values. Sparse matrices are distinct from matrices with mostly non-zero values, which are referred to as dense matrices.

A matrix is sparse if many of its coefficients are zero. The interest in sparsity arises because its exploitation can lead to enormous computational savings and because many large matrix problems that occur in practice are sparse.

10	0	0	20
30	0	0	0
0	40	50	0

Jagged Array:

While there is no advantage to individually allocating the second dimension in arrays, as the above shown example/program, there may be in others.
For example, when we allocate dimensions manually/individually, we do not need to allocate the same number of elements for each dimension.
As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under our control.
For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal.

Example:

```
int sparseM[][] = new int[4][];
sparseM[0] = new int[1];
sparseM[1] = new int[2];
sparseM[2] = new int[1];
sparseM[3] = new int[3];
```

The above declare and defined jagged array can be represented in mathematics as:

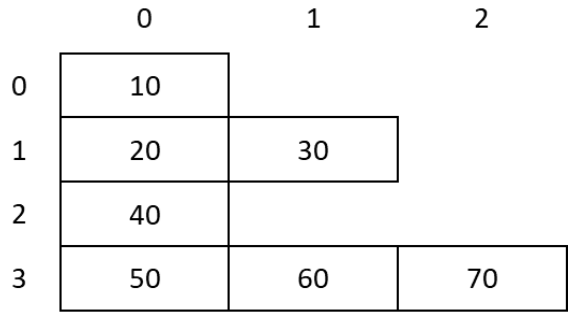
10	0	0
20	30	0
40	0	0
50	60	70

60. Program:

```
class JaggedArray1 {
    public static void main(String[]args) {
        int sparseM[][] = new int[4][];
        sparseM[0] = new int[1];
        sparseM[1] = new int[2];
        sparseM[2] = new int[1];
        sparseM[3] = new int[3];
        sparseM[0][0] = 10;
        sparseM[1][0] = 20;
        sparseM[1][1] = 30;
        sparseM[2][0] = 40;
        sparseM[3][0] = 50;
        sparseM[3][1] = 60;
```

```
        sparseM[3][2] = 70;
        System.out.println("Jagged array have elements as :");
        System.out.println("sparseM[0][0] : " + sparseM[0][0]);
        System.out.println("sparseM[1][0] : " + sparseM[1][0] + "\tsparseM[1][1] : " +
sparseM[1][1]);
        System.out.println("sparseM[2][0] : " + sparseM[2][0]);
        System.out.println("sparseM[3][0] : " + sparseM[3][0] + "\tsparseM[3][1] : " +
sparseM[3][1] + "\tspareseM[3][2] : " + sparseM[3][2]);
        System.out.println("Thank you.");
    }
}
```

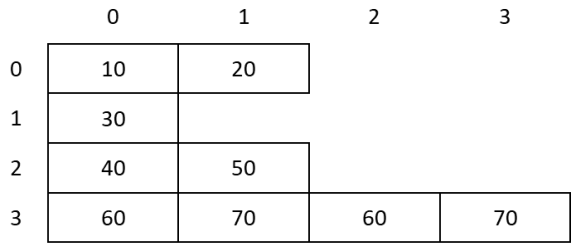
Memory representation of the above declared jagged array:



61. Program:

```
class JaggedArray2 {
    public static void main(String[]args) {
        int a[][] = {{10, 20}, {30}, {40, 50}, {60, 70, 80, 90}};
        System.out.println("Jagged array have elements as :");
        for(int i = 0; i < a.length; ++i) {
            for(int j = 0; j < a[i].length; ++j)
                System.out.print("\t" + a[i][j]);
            System.out.println();
        }
        System.out.println("Thank you.");
    }
}
```

Memory allocation:



IX - Strings:

As is the case in most other programming languages, in Java a string is a sequence of characters. But, unlike some other languages that implement strings as character arrays, Java implements strings as objects of type **String** class.
Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string, and so on. Also, **String** objects can be constructed in a number of ways, making it easy to obtain a string when needed.
Somewhat unexpectedly, when we create a **String** object, we are creating a string that cannot be changed. That is, once a **String** object has been created, we cannot change the characters that comprise that string.
At first, this may seem to be a serious restriction. However, such is not the case. We can still perform all types of string operations.
The difference is that each time we need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones.
For those cases in which a modifiable string is desired, Java provides two options/classes: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in the **java.lang** package.

The String Constructor:

The **String** class supports several constructors. To create an empty string, call the default constructor.

Example:
```
String objectName = new String();
```

This will create an instance of **String** with no characters in it.

Frequently, we will want to create strings that have initial values. The **String** class provides a variety of constructors to handle this. To create a **String** initialized by an array of characters, use the constructor shown here:

Syntax:
```
String(char chars[])
String(char chars[], int startIndex, int numChars)
```

Example:
```
char chars[] = {'a', 'b', 'c'};
String s = new String(chars);
```

This constructor initializes an object with name "**s**" and with the string value as "**abc**".

We can construct a **String** object that contains the same character sequence as another **String** object using the following constructor.

Syntax:
```
String(String strObj)
```
OR
```
String("stringLiteral")
```

Example:
```
String s = new String("Java");
```

This initializes the object "**s**" with the string literal "**Java**". The same can also be done using the following syntax.

Syntax:
```
String s = "stringLiteral";
```

Example:
```
String s1 = "Java";
String s2 = new String(s1);
```
OR
```
String s2 = s1;
```

This initializes the object "**s1**" and also the object "**s2**" with the string literal "**Java**".

Even though Java's char type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the **String** class provides constructors that initialize a string when given a byte array.

Syntax:
```
String(byte b[])
String(byte b[], int startIndex, int numChars)
```

Here, *b* specifies the array of bytes. The second form allows us to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform.

Example:
```
byte ascii[] = {65, 66, 67, 68, 69, 70};
String s = new String(ascii); // s = "ABCDEF"
```

Example 2:
```
byte ascii[] = {97, 98, 99, 100, 101, 102};
String s = new String(ascii, 2, 3);        // s = "cde"
```

62. Program:
```
class StringDemo1 {
    public static void main(String[]args) {
        char chars[] = {'a', 'b', 'c'};
        String s1 = new String(chars);
        System.out.println("String object using char array = " + s1);
        char chars2[] = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
        s1 = new String(chars2, 1, 5);
        System.out.println("String object having substring, using char array = " + s1);
        String s2 = "Java";
        System.out.println("String object using literal = " + s2);
        // String s3 = new String(s2);
        // OR
        String s3 = s2;
        System.out.println("String object using another string object = " + s3);
        byte bytes[] = {97, 98, 99, 100, 101, 102, 103};
        String s4 = new String(bytes);
        System.out.println("String object using byte array = " + s4);
        String s5 = new String(bytes, 2, 3);
        System.out.println("String object having substring, using byte array = " + s5);
```

```
        System.out.println("Thank you.");
    }
}
```

String length( ) method:

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown below:

Syntax:

```
int length()
```

Example:

```
String s = new String("Java String Data Type.");
int l = s.length();      // l = 22
```

String Concatenation:

In general, Java does not allow operators to be applied to String objects. The one exception to this rule is the **+** operator, which concatenates two strings, producing a new **String** object as the result. This allows us to chain together a series of + operations concatenating n-number of strings (or any other valid Java data type).

Example:

```
String s1 = "Shruti";
String name = s1 + "ka";
String s2 = "Hello! This is " + name;
int age = 18;
String college = "DBF, BSc.(ECS)";
float per = 98.56f;
s2 = "Hello! This is " + name + ". I am " + age + " years old. I study in " + college + ". I have
secured " + per + "%.";
```

63. Program:

```
class StringDemo2 {
    public static void main(String[]args) {
        String s = new String("Java String Data Type.");
        int l = s.length(); // l = 22
        System.out.println("String s = " + s);
        System.out.println("Length of string s = " + l);
        String s1 = "Shruti";
        String name = s1 + "ka";
        System.out.println("Name = " + name);
        String s2 = "Hello! This is " + name;
        System.out.println("s2 = " + s2);
        int age = 18;
        String college = "DBF, BSc.(ECS)";
        float per = 98.56f;
        s2 = "Hello! This is " + name + ". I am " + age + " years old. I study in " + college + ".
I have secured " + per + "%.";
        System.out.println("Concatenated String s2 = " + s2);
```

```
    }
}
```

String Conversion:

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by the **String** class. For objects, the **valueOf()** method calls the **toString( )** method on the objects that belongs to any data type, converting it into the string type.
Every class implements the **toString( )** method because it is defined by the **Object** class. However, the default implementation of the **toString( )** method is seldom sufficient. For most important classes that we create, we will want to override the **toString( )** method and provide our own string representations. Fortunately, this is easy to do.

Syntax:

```
String toString()
```

To implement the **toString( )** method, simply return a **String** object that contains the human-readable string that appropriately describes an object of your class. By overriding the **toString( )** method for classes that we create, we allow them to be fully integrated into Java's programming environment.

64. Program:

```
class StringDemo3 {
    int no1, no2;
    static public void main(String[]args) {
        StringDemo3 obj1 = new StringDemo3();
        StringDemo3 obj2 = new StringDemo3();
        obj1.no1 = 10;
        obj1.no2 = 20;
        obj2.no1 = 4;
        obj2.no2 = 6;
        System.out.println("obj1 : " + obj1.toString());
        System.out.println("obj2 : " + obj2);
    }
    public String toString() {
        return "no1 : " + no1 + " , no2 : " + no2;
    }
}
```

As we can see, the class **StringDemo3**'s **toString( )** method is implicitly (automatically) invoked when a **StringDemo3** object is used in a concatenation expression or in a call to the **println( )** method.

StringBuffer:

The **StringBuffer** class object supports a modifiable string. As we know, the object of class **String** represents fixed-length, immutable character sequences. In contrast, the object of class **StringBuffer** represents growable and writable character sequences. The **StringBuffer** class object may have characters and substrings inserted in the middle or appended to the end. The object of **StringBuffer** class will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

The StringBuffer Constructors:

The **StringBuffer** class defines these four constructors:

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
StringBuffer(CharSequence chars)
```

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation.
The second version accepts an integer argument that explicitly sets the size of the buffer.
The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation.
The fourth constructor creates an object that contains the character sequence contained in *chars* and reserves room for 16 more characters.

The **StringBuffer( )** constructor allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, the **StringBuffer** class reduces the number of reallocations that take place.

The length( ) and capacity( ) Methods:

The current length of a **StringBuffer** can be found via the **length( )** method, while the total allocated capacity (memory) can be found through the **capacity( )** method.

General form:

```
int length()
int capacity()
```

Example:

```
StringBuffer sb1 = new StringBuffer("Hello");
System.out.println("sb1 = " + sb1); // Hello
System.out.println("Length of sb1 = " + sb1.length()); // 5
System.out.println("Capacity of sb1 = " + sb1.capacity()); // 21
```

Since object *sb1* is initialized with the string "*Hello*" when it is created, its length is of *5* characters. Its capacity is *21* characters (or more) because room for *16* additional characters is automatically added.

The ensureCapacity( ) Method:

If we want to preallocate room for a certain number of characters after an object of **StringBuffer** class has been constructed, we can use the **ensureCapacity( )** method to set the size of the buffer. This is useful if we know in advance that we will be appending a large number of small strings to the **StringBuffer** object.

General form:

```
void ensureCapacity(int minCapacity)
```

Here, *minCapacity* specifies the minimum size of the buffer. (A buffer larger than *minCapacity* may be allocated for reasons of efficiency.)

Example:

```
StringBuffer sb1 = new StringBuffer("Hello");
System.out.println("sb1 = " + sb1);  // Hello
```

```
System.out.println("Length of sb1 = " + sb1.length()); // 5
System.out.println("Capacity of sb1 = " + sb1.capacity());    // 21
sb1.ensureCapacity(24);
System.out.println("Capacity of sb1 = " + sb1.capacity());    // 25 (or more)
```

65. Program:

```
class StringBuffer1 {
    static public void main(String[]args) {
        StringBuffer sb1 = new StringBuffer("Hello All.");
        System.out.println("sb1 : " + sb1);
        System.out.println("Length of sb1 : " + sb1.length());
        System.out.println("Capacity (memory) of sb1 : " + sb1.capacity());
        sb1.ensureCapacity(30);
        System.out.println("After ensureCapacity(30), capacity (memory) of sb1 : " +
sb1.capacity());
        System.out.println("Thank you.");
    }
}
```

The append( ) Method:

The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions.

General forms:

```
StringBuffer append(String str)
StringBuffer append(int num)
StringBuffer append(Object obj)
```

The string representation of each parameter is obtained, often by calling the **String.valueOf( )** method. The result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of the **append( )** method. This allows subsequent calls to be chained together.

Example:

```
String s1 = "Good Evening.";
StringBuffer sb1 = new StringBuffer("Hello All!");
sb1.append(s1);    // sb1 = "Hello All!Good Evening."
```

The insert( ) Method:

The **insert( )** method inserts one string into another. It is overloaded to accept values of all the primitive types, plus Strings, Objects, and CharSequences. Like the **append( )** method, it obtains the string representation of the value it is called with. This string is then inserted into the invoking **StringBuffer** object.

General forms:

```
StringBuffer insert(int index, String str)
StringBuffer insert(int index, char ch)
StringBuffer insert(int index, Object obj)
```

Here, *index* specifies the index at which point the string will be inserted into the invoking *StringBuffer* object.

Example:

```
StringBuffer sb1 = new StringBuffer("I Java!");
sb1.insert(2, "like ");  // sb1 = "I like Java!"
```

66. Program:

```
class StringBuffer2 {
    public static void main(String[]args) {
        String s1 = "Good Morning.";
        StringBuffer sb1 = new StringBuffer("I Java!");
        System.out.println("sb1 : " + sb1);
        sb1.insert(2, "like ");
        System.out.println("After sb1.insert(2, \"like\") : " + sb1);
        System.out.println("Thank you.");
    }
}
```

StringBuilder:

Introduced by JDK 5, **StringBuilder** is a relatively recent addition to Java's string handling capabilities. **StringBuilder** is similar to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of **StringBuilder** is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use **StringBuffer** rather than **StringBuilder**.

All the operations, methods, etc are the same for **StringBuilder** as they were for the **StringBuffer**.

X- Object Oriented Programming:

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept we wish to implement in a Java program must be encapsulated within a class.
The main aim of object-oriented programming is to implement real-world entities, for example, objects, classes, abstraction, inheritance, polymorphism, etc.

Object Oriented Programming System:

Object means a real-world entity such as a pen, chair, table, laptop, human being, etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing concepts such as:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Advantages of OOP over Procedure-Oriented Programming (POP) Languages:

1. OOPs makes development and maintenance easier, whereas, in a POP language, it is not easy to manage the application (software/website) if code grows as project size increases.
2. OOPs provide data hiding, whereas, in a POP language, global data can be accessed from anywhere.
3. OOPs provide the ability to simulate real-world events much more effectively. We can provide the solution of real world problems if we are using the Object-Oriented Programming Language. Whereas, this is not possible using POP language.

Objects and Classes in Java:

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.
A class is a template/blueprint/plan for an object, and an object is an instance/working/implemented model of a class.

Class:

A class is nothing but the encapsulation of data members and member methods into a single unit. A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It cannot be physical.
A class in Java can contain fields, methods, constructors, blocks, nested class and interface, etc.

Declaration of Class:

A class in Java is declared using the class keyword and then defined the members, fields, methods, etc within the class.

General form:

```
class ClassName {
	Fields...
	Methods...
	...
}
```

Example:

```
class Student {
    int rollNo;
    String name;
    double totalMarks;
}
```

Object:

Object is nothing but the instance of the class, that is the entity, or working model of the class.
An entity that has state and behavior is known as an object.
For example: chair, pen, bike, laptop, car, college, exam, result, etc are classes.
The object can be a physical or logical (tangible and intangible) entity, such as a brown chair, a blue Parker Pen, etc.

The object of the class has characteristics:

- State : represents the data (value) of an object (instance variables)
- Behavior : represents the behavior (functionality) of the object (member methods)
- Identity : An object identity is typically implemented via a unique ID. The value of the ID is not visible to external users.

Example:
Pen is a class. Parker pen is an object. The name of the object is Parker, color is Golden, written in blue color, known as its state. It is used to write, so writing is its behavior.
An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance (result) of a class.

Creating Object:

Before creating an object of a class, we must first declare/define the class. Once a class has been defined, we can declare and initialize any number of objects of that class. Declaration of an object can be done using the following syntax:

```
ClassName objectName;
```

Example:

```
Student s;
```

Declaration of an object just declares the object but no memory is allocated to it. So we cannot use such an object. Thus we need to initialize the object, which allocates memory to the object, that is done using the "**new**" operator.

Syntax:

```
objectName = new ClassName();
```

Example:

```
s = new Student();
```

Declaration and initialization of object can be done by one single statement, which is as follows:

Syntax:

```
ClassName objectName = new ClassName();
```

Example:

```
Student s = new Student();
```

Constructing the class and object program:

```
class ClassObjectDemo1 {
    public static void main(String[]args) {
        Student s;
        s = new Student();
        s.rollNo = 101;
        s.name = "Shrutika";
        s.totalMarks = 492.8;
        System.out.println("Student details are :");
        System.out.println("Roll No : " + s.rollNo);
        System.out.println("Name : " + s.name);
        System.out.println("Total Marks : " + s.totalMarks);
        double per = (s.totalMarks / 500.0) * 100.0;
        System.out.println("Percentage : " + per + "%");
    }
}
class Student {
    int rollNo;
    String name;
    double totalMarks;
}
```

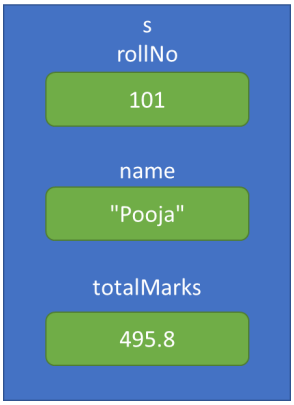
Defining multiple classes:

In real time development, we create multiple classes and use them from another class, as done in the above example. It is a better approach than using a single class. That is many times we create a class to declare and define the data members and member methods that belong to an entity. And then, another class is defined for the main( ) method, in which the objects of the other classes are created, initialized and used.

The new Keyword:

The **new** keyword is used to allocate memory to the object at runtime. Thus this **new** keyword is also called as "dynamic memory allocation operator".
All objects get memory in the Heap memory area that is allocated by this **new** keyword.

Representation of memory allocated to the object "**s**" of class "**Student**".



Creating Multiple Objects:

We can declare and initialize multiple objects of a class. This will allocate us a new set of memory for each new object that we initialize. This approach is commonly used when we have to work on multiple sets of data.

Syntax:

```
ClassName objectName1, objectName2, objectName3, ...;
objectName1 = new ClassName();
objectName2 = new ClassName();
objectName3 = new ClassName();
...
```

OR

```
ClassName objectName1 = new ClassName();
ClassName objectName2 = new ClassName();
ClassName objectName3 = new ClassName();
...
```

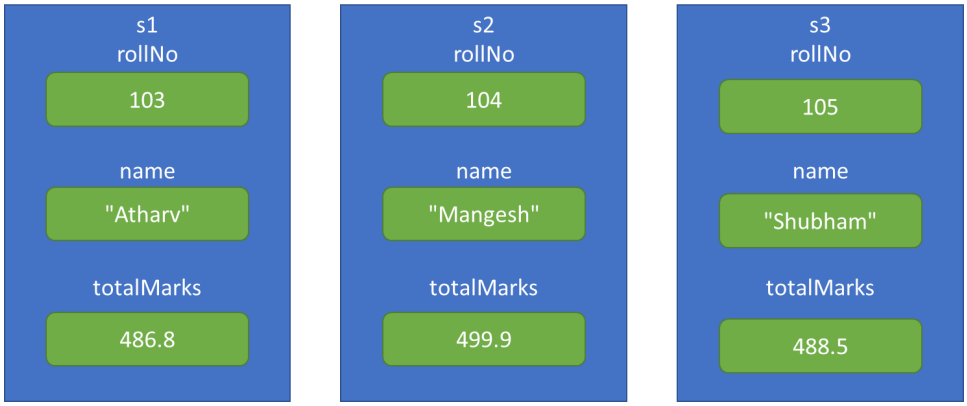
68. Program:

```
class ClassObjectDemo2 {
    public static void main(String[]args) {
        Student s1 = new Student();
        s1.rollNo = 103;
        s1.name = "Atharv";
        s1.totalMarks = 486.8;

        Student s2 = new Student();
        s2.rollNo = 104;
        s2.name = "Mangesh";
        s2.totalMarks = 499.9;

        Student s3 = new Student();
        System.out.println("Enter student details :");
        System.out.print("Roll No : ");
        s3.rollNo = Integer.parseInt(System.console().readLine());
        System.out.print("Name : ");
```

```
        s3.name = System.console().readLine();
        System.out.print("Total marks (out of 500) : ");
        s3.totalMarks = Double.parseDouble(System.console().readLine());

        System.out.println("\nStudent 1 details are :");
        System.out.println("Roll No : " + s1.rollNo);
        System.out.println("Name : " + s1.name);
        System.out.println("Total Marks : " + s1.totalMarks);
        double per = (s1.totalMarks / 500.0) * 100.0;
        System.out.println("Percentage : " + per + "%");

        System.out.println("\nStudent 2 details are :");
        System.out.println("Roll No : " + s2.rollNo);
        System.out.println("Name : " + s2.name);
        System.out.println("Total Marks : " + s2.totalMarks);
        per = (s2.totalMarks / 500.0) * 100.0;
        System.out.println("Percentage : " + per + "%");

        System.out.println("\nStudent 3 details are :");
        System.out.println("Roll No : " + s3.rollNo);
        System.out.println("Name : " + s3.name);
        System.out.println("Total Marks : " + s3.totalMarks);
        per = (s3.totalMarks / 500.0) * 100.0;
        System.out.println("Percentage : " + per + "%");
    }
}
class Student {
    int rollNo;
    String name;
    double totalMarks;
}
```

Memory Allocation:

As we see have declared, defined and initialized 3 objects of the class Student, 3 different set of memory is allocated to these 3 different objects, as shown below:



Defining Class Member Methods:

In general, a method is a way to perform some task. Similarly, the method in Java is a collection of instructions that perform a specific task.
A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times generally to perform operations related to, or on the instance variables. It is not required to write code again and again. It also provides the ease of modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

Types of methods:

- Predefined methods
- User-defined methods

Predefined Methods:

In Java, predefined methods are the methods that are already defined in the Java class libraries. They are also called standard library methods or built-in methods. These predefined methods are declared and defined inside the classes/interfaces which are present in Java packages.
As the predefined methods are predefined we just need to call/invoke them by first importing the package within which they are present, and then create, declare, and define the object of the class within which the method is defined.
While calling/invoking the method, passing the required parameters (if any) to the calling method.

General form:

```
[import packagePathAndName;]
...
ClassName object = new ClassName();
object.method([arguments]);
...
```

69. Program:

```
import java.util.Date;
import java.text.DateFormat;
class BuiltInMethod1 {
    static public void main(String[]para) {
        Date objD = new Date();
        System.out.println("Date : " + objD);
        System.out.println("Formatted date : " + DateFormat.getInstance().format(objD));
        System.out.println("Thank you.");
    }
}
```

In the above example, we have used some built-in methods, such as **getInstance( )**, **format( )**, and some special methods, that is a constructor, which is **Date( )**, and apart from these the **println( )** method. To use these built-in methods we need to import the packages within which these methods reside, or just use their fully qualified names, as used in case of **println( )** method.

User-defined Methods:

The methods that we declare and define inside our classes/interfaces in our program are known as user-defined methods.
In general, a method is a way to perform some task. And these tasks are defined inside classes/interfaces using methods.

Method Declaration and Definition:

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments.
Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.
In Java we provide method definition which is the declaration as well as definition of method and its body (block of code).

General form:

```
returnType methodName([parameterList]) {
      methodBody...
      ...
      return [exp/value];
}
```

Once a method is defined, to execute it we must call/invoke the method. A program with method definition, but with no call to the method is meaningless, as the method that we have defined will never execute unless we call/invoke it.
The method can be called/invoked by just specifying its name, and passing the parameters (if any).

General form:

```
[variableName = ] methodName([arguments]);
```

70. Program:

```
class UserDefinedMethods1 {
    public static void main(String[]args) {
        Employee e1 = new Employee();
        e1.id = 101;
        e1.name = "Harshal";
        e1.post = "Sr. Developer";
        e1.ctc = 15;
        Employee e2 = new Employee();
        e2.id = 102;
        e2.name = "Santosh";
        e2.post = "Jr. Developer";
        e2.ctc = 9;
        System.out.println("Employee 1 details are :");
        e1.showDetails();
        System.out.println("\nEmployee 2 details are :");
        e2.showDetails();
    }
}
class Employee {
    int id;
```

```
    String name;
    String post;
    double ctc;
    void showDetails() {
        System.out.println("ID : " + id);
        System.out.println("Name : " + name);
        System.out.println("Post : " + post);
        System.out.println("Package (CTC) (in Lakh) Rs : " + ctc);
    }
}
```

Methods with Arguments:

Methods can have arguments. A method can have minimum 0 and maximum N number of arguments with any data type. The argument list is defined inside the parenthesis in method definition along with the data type of each argument.

General form:

```
returnType methodName(argType argName, argType argName2, argType argName3, ...) {
      methodBody...
      ...
      return [exp/value];
}
```

Note that, a method can have any number of arguments, and each argument should be defined with its data type separately.

71. Program:

```
class UserDefinedMethods2 {
    public static void main(String[]para) {
        Circle c1 = new Circle();
        c1.getRadius(5.0);
        Circle c2 = new Circle();
        c2.getRadius(7.5);
        Rectangle r1 = new Rectangle();
        r1.getDimensions(60, 70);
        Rectangle r2 = new Rectangle();
        r2.getDimensions(4, 6);
        System.out.println("Circle 1 details are :");
        c1.showArea();
        System.out.println("Rectangle 1 details are :");
        r1.showArea();
        System.out.println("Circle 2 details are :");
        c2.showArea();
        System.out.println("Rectangle 2 details are :");
        r2.showArea();
    }
}
class Circle {
```

```
    double pi = 3.14;
    double rad;
    void getRadius(double r) {
        rad = r;
    }
    void showArea() {
        System.out.println("Radius : " + rad + ", Area : " + (pi * rad * rad));
    }
}
class Rectangle {
    double len, bre;
    void getDimensions(double l, double b) {
        len = l;
        bre = b;
    }
    void showArea() {
        System.out.println("Length : " +len+ ", Breadth : " +bre+ ", Area : " + (len * bre));
    }
}
```

Variable-Length Arguments:

Beginning with JDK-5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called **varargs** and it is short for **variable-length arguments**. A method that takes a variable length of arguments is called a **variable-arity method**, or **varargs method**.

Before JDK-5, we had to use arrays for passing a variable number of arguments. But now the **varargs** methodology has been introduced using which we can define a method in Java to accept a variable number of arguments.

Old Approach : Using Array:
72. Program:

```
class VarArgsUsingArray {
    static public void main(String[]args) {
        VarArgsArray obj = new VarArgsArray();
        int a[] = {10, 20};
        obj.numbers(a);
        int b[] = {2, 4, 6, 8, 10};
        obj.numbers(b);
        int c[] = {};
        obj.numbers(c);
        System.out.print("Enter number of values you want to input :");
        int size = Integer.parseInt(System.console().readLine());
        int d[] = new int[size];
        System.out.print("Enter " + size + " values :");
        for(int i = 0; i < size; ++i)
            d[i] = Integer.parseInt(System.console().readLine());
        obj.numbers(d);
        System.out.println("Thank you.");
```

```
    }
}
class VarArgsArray {
    void numbers(int arr[]) {
        System.out.println("Number of arguments passed (size of array) : " + arr.length);
        if(arr.length > 0)
            for(int i = 0; i < arr.length; ++i)
                System.out.println("arr[" + i + "] = " + arr[i]);
        System.out.println();
    }
}
```

New Approach : Using the varargs(...):
As stated, from JDK-5 the methodology of varargs came into existence, to define and pass any number arguments to Java's method.

Syntax:

```
class ClassName {
      ....
      returnType methodName([requiredArgument, ] dataType ... varArgsName) {
            ....
            Method body
      }
      ....
}
```

73. Program:

```
class VarArgsDemo {
    static public void main(String[]args) {
        VarArgs obj = new VarArgs();
        obj.numbers(10, 20);
        obj.numbers(2, 4, 6, 8, 10);
        obj.numbers();
        System.out.print("Enter number of values you want to input :");
        int size = Integer.parseInt(System.console().readLine());
        int d[] = new int[size];
        System.out.print("Enter " + size + " values :");
        for(int i = 0; i < size; ++i)
            d[i] = Integer.parseInt(System.console().readLine());
        obj.numbers(d);
        System.out.println("Thank you.");
    }
}
class VarArgs {
    void numbers(int ... nos) {
        System.out.println("Number of arguments passed : " + nos.length);
        if(nos.length > 0)
            for(int i = 0; i < nos.length; ++i)
                System.out.println("nos[" + i + "] = " + nos[i]);
        System.out.println();
```

```
    }
}
```

Returning Value:

A method can return either 0 or only one value upon calling it. The return type has to be defined in method definition. If any method does not return any value then its return type must be **void**. Else if a method returns any value then the return type of that method should be properly defined.
When a method returns a value, we need to state the **return** statement to return the value.

74. Program:

```
class ReturningValue1 {
    public static void main(String[]para) {
        Shapes c = new Shapes();
        c.setDimensions(3.14, 5.0);
        Shapes t = new Shapes();
        t.setDimensions(24, 51);
        Shapes r = new Shapes();
        r.setDimensions(40, 60);
        System.out.println("Shape Circle details are : ");
        System.out.println("Radius : " + c.dim2 + ", Area : " + c.areaOfCircle());
        System.out.println("\nShape Rectangle details are : ");
        double a = r.areaOfRectangle();
        System.out.println("Length : " + r.dim1 + ", Breadth : " + r.dim2 + ", Area : " + a);
        System.out.println("\nShape Triangle details are : ");
        a = t.areaOfTriangle();
        System.out.println("Base : " + t.dim1 + ", Height : " + t.dim2 + ", Area : " + a);
        System.out.println("\nThank you.");
    }
}
class Shapes {
    double dim1, dim2;
    double area;
    void setDimensions(double d1, double d2) {
        dim1 = d1;
        dim2 = d2;
    }
    double areaOfCircle() {
        area = dim1 * dim2 * dim2;
        return area;
    }
    double areaOfRectangle() {
        return (dim1 * dim2);
    }
    double areaOfTriangle() {
        return (0.5 * dim1 * dim2);
    }
```

```
}
```

Overloading of Methods:

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case the methods are said to be overloaded, and the process is referred to as method overloading.
Method overloading is one of the ways that Java supports polymorphism.
When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of parameters, that is, they must differ with different sets of signatures (list of parameters).

Note that, the return type of the method is never used to differentiate methods in method overloading.

General form:

```
returnType methodName([parameterList1]) {
	...
	Method body
}
returnType methodName([parameterList2]) {
	...
	Method body
}
returnType methodName([parameterList3]) {
	...
	Method body
}
...
```

75. Program:

```
class OverloadingOfMethods1 {
    static public void main(String[]args) {
        Addition obj = new Addition();
        obj.add(25);                // 4th method overloads the others
        obj.add(30, 40.5);          // 2nd method overloads the others
        obj.add(25, 35);            // 5th method overloads the others
        obj.add("Hello", "All");    // 7th method overloads the others
        System.out.println("Addition of 2, 3, 4 and 5 is " + obj.add(2, 3, 4, 5)); // 8th method
overloads the others
        obj.add();                  // 1st method overloads the others
        obj.add(2, 3, 4);           // 6th method overloads the others
        obj.add(2.5, 3.5);          // 3rd method overloads the others
    }
}
class Addition {
    void add() {
        System.out.println("\nFirst add() method, without any parameters.");
        int n1, n2;
        System.out.print("Enter 2 numbers :");
        n1 = Integer.parseInt(System.console().readLine());
```

```
        n2 = Integer.parseInt(System.console().readLine());
        System.out.printf("Addition of %d and %d is %d.\n", n1, n2, (n1 + n2));
    }
    void add(int a, double b) {
        System.out.println("\nSecond add() method, with 1 int and 1 double parameter.");
        System.out.printf("Addition of %d and %g is %g.\n", a, b, (a + b));
    }
    void add(double a, double b) {
        System.out.println("\nThird add() method, with 2 double type parameters.");
        System.out.printf("Addition of %g and %g is %g.\n", a, b, (a + b));
    }
    void add(int no) {
        System.out.println("\nFourth add() method, with 1 int parameter.");
        System.out.printf("Addition of %d with itself is %d.\n", no, (no + no));
    }
    void add(int a, int b) {
        System.out.println("\nFifth add() method, with 2 int parameters.");
        System.out.printf("Addition of %d and %d is %d.\n", a, b, (a + b));
    }
    void add(int x, int y, int z) {
        System.out.println("\nSixth add() method, with 3 int parameters.");
        System.out.printf("Addition of %d, %d and %d is %d.\n", x, y, z, (x + y + z));
    }
    void add(String s1, String s2) {
        System.out.println("\nSeventh add() method, with 2 String parameters.");
        System.out.printf("Addition (concatenation) of %s and %s is %s.\n", s1, s2, (s1 + s2));
    }
    int add(int a, int b, int c, int d) {
        System.out.println("\nEighth add() method, with 4 integer parameters and a return
value.");
        return (a + b + c + d);
    }
}
```

76. Program:

```
class OverloadingOfMethods2 {
    public static void main(String[]args) {
        Shapes obj = new Shapes();
        System.out.println("\nDetails of Circle :");
        System.out.println("Radius : 5.5. Area : " + obj.area(5.5));
        System.out.println("\nDetails of Triangle :");
        System.out.println("Base : 25.5, height : 40.2. Area : " + obj.area(0.5, 25.5, 40.2));
        System.out.println("\nDetails of Rectangle :");
        System.out.println("Length : 50.5, breadth : 40.2. Area : " + obj.area(50.5, 40.2));
        System.out.println("\nThank you.");
    }
}
class Shapes {
```

```
    double area(double radius) {
        double a = 3.14 * radius * radius;
        return a;
    }
    double area(double l, double b) {
        double a = l * b;
        return a;
    }
    double area(double half, double base, double height) {
        return (half * base * height);
    }
}
```

Assigning one object to another:

When a class is defined, to use its members we need to declare and initialize/define the object of that class. Upon initialization of an object, Java assigns memory to the object dynamically using the dynamic memory allocation operator, i.e., "**new**" operator, at runtime.
When we create and initialize an object and the same object is assigned to another object of the same class then the reference of the first object is assigned to the other(s). No new memory is allocated to the other object.
This means when one object is assigned to another, Java does not create 2 different objects. Instead 2 identifiers (names) are declared and both refer to one single same object (memory). That is why when we make changes (assign values) to the instance members using one identifier, it will reflect on the other one.

77. Program:

```
class AssigningOneObjectToAnother1 {
    public static void main(String[]args) {
        Student s1, s2, s3;
        s1 = new Student();
        System.out.println("Enter student details for object s1 :");
        s1.setData();
        s2 = s1;
        s3 = new Student();
        System.out.println("Student details using object s1 :");
        s1.showData();
        System.out.println("Student details using object s2 :");
        s2.showData();
        System.out.println("Student details using object s3 :");
        s3.showData();
        System.out.println("\nEnter student details for object s2 :");
        s2.setData();
        System.out.println("Enter student details for object s3 :");
        s3.setData();
        System.out.println("\nAfter accepting new data in objects s2 and s3 :");
        System.out.println("Student details using object s1 :");
        s1.showData();
        System.out.println("Student details using object s2 :");
```

```
        s2.showData();
        System.out.println("Student details using object s3 :");
        s3.showData();
        System.out.println("Thank you.");
    }
}
class Student {
    int rollNo;
    String name;
    double totalMarks;
    void setData() {
        System.out.print("Roll No :");
        rollNo = Integer.parseInt(System.console().readLine());
        System.out.print("Name :");
        name = System.console().readLine();
        System.out.print("Total Marks :");
        totalMarks = Double.parseDouble(System.console().readLine());
    }
    void showData() {
        System.out.printf("Roll No : %d, Name : %s, Total Marks : %g.\n", rollNo, name,
totalMarks);
    }
}
```

XI - Constructors:

Java allows objects to initialize themselves when they are created. This implicit (automatic) initialization is performed through the use of a **constructor**.
A **constructor** is a special type of method of the class that initializes an object immediately upon creation. It is implicitly (automatically) invoked (called) when the object is created (initialized), before the **new** operator completes.

Types of Constructors:

1. Implicit constructor
2. Default constructor
3. Parameterized constructor

Implicit Constructor:

When we, the developer, have not defined any constructor within our class, then Java implicitly uses its own default implicit constructor. Using this implicit constructor Java initializes the object of the class. When we do not explicitly define a constructor for a class, then Java creates and uses this implicit constructor for the class to initialize the objects and all the instance members of the objects. This is why in all the preceding examples/programs, the objects of the defined classes were constructed and initialized implicitly.
The implicit constructor implicitly (automatically) initializes all instance variables to their default values, which are **zero**, **null**, and **false**, for numeric types, reference types, and boolean, respectively.

78. Program:

```
class ImplicitConstructor1 {
    public static void main(String[]args) {
        System.out.println("Implicit Constructor :");
        ImplicitConstructor obj = new ImplicitConstructor();
        System.out.println("Object of class initialized via implicit constructor :");
        System.out.println("Default initial values of instance variables :");
        obj.showData();
        obj.i = 60;
        obj.c = 'J';
        obj.d = 1.2023;
        obj.b = true;
        obj.s = "Java String";
        System.out.println("\nAfter assigning new values to the instance variables :");
        obj.showData();
        System.out.println("Thank you.");
    }
}
class ImplicitConstructor {
    int i;
    char c;
    double d;
    boolean b;
    String s;
    void showData() {
        System.out.println("int i : " + i);
```

```
        System.out.println("char c : " + c);
        System.out.println("double d : " + d);
        System.out.println("boolean b : " + b);
        System.out.println("String s : " + s);
    }
}
```

Default Constructor (non-parameterized constructor):

The implicit constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once we define our own constructor, the implicit constructor is no longer used.
The constructor has the same name as the class in which it resides and is syntactically similar to a method. Once defined, it is implicitly (automatically) invoked (called) when the object is created (initialized), before the **new** operator completes.
Java uses the constructor that we define to construct/initialize the object of the class that we create.
A constructor is called "Default constructor" when it is defined within a class with no any parameter(s). Generally, here the developer defines the default initial values to the instance variables of the class.

Rules for Creating Constructor:

- Constructor name must be exactly the same as its class name.
- A Constructor must have no explicit return type, not even void.
- A Java constructor cannot be abstract, static, final, and synchronized.

General form for the declaration and definition of Constructor:

```
class ClassName {
      //data members
      ...
      ClassName([parameterList]) {
            //Constructor body
            ...
      }
      //member methods
      ...
}
```

Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fly initialized, usable object immediately.

79. Program:

```
class DefaultConstructor1 {
    public static void main(String[]args) {
        System.out.println("Default Constructor :");
        DefaultConstructor obj = new DefaultConstructor();
        System.out.println("Object of class initialized via Default constructor :");
        System.out.println("Default initial values of instance variables :");
        obj.showData();
        obj.i = 60;
        obj.c = 'J';
        obj.d = 3.14;
```

```
        obj.b = true;
        obj.s = "Java String";
        System.out.println("\nAfter assigning new values to the instance variables :");
        obj.showData();
        System.out.println("Thank you.");
    }
}
class DefaultConstructor {
    int i;
    char c;
    double d;
    boolean b;
    String s;
    DefaultConstructor() {
        System.out.println("Default Constructor invoked.");
        i = 24;
        c = 'A';
        d = 1.2023;
        b = true;
        s = "Default String";
    }
    void showData() {
        System.out.println("int i : " + i);
        System.out.println("char c : " + c);
        System.out.println("double d : " + d);
        System.out.println("boolean b : " + b);
        System.out.println("String s : " + s);
    }
}
```

80. Program:

```
class DefaultConstructor2 {
    public static void main(String[]a) {
        Shape objC = new Shape();
        objC.setData(5.0, "Circle");
        Shape objR = new Shape();
        objR.setData(45.3, 65.6, "Rectangle");
        Shape objS = new Shape();
        objS.setData(6.0, "Square");
        Shape objT = new Shape();
        objT.setData(5.3, 6.6, "Triangle");
        System.out.println("Area of " + objC.name + " : " + objC.area());
        System.out.println("Area of " + objS.name + " : " + objS.area());
        System.out.println("Area of " + objT.name + " : " + objT.area());
        System.out.println("Area of " + objR.name + " : " + objR.area());
    }
}
class Shape {
```

```
    double dim1, dim2, dim3;
    String name;
    Shape() {
        dim1 = 1.0;
        dim2 = 3.14;
        dim3 = 0.5;
        name = "Not defined";
        System.out.println("Default constructor for class Shape invoked.");
    }
    void setData(double d1, String shapeName) {
        dim1 = d1;
        name = shapeName;
    }
    void setData(double d1, double d2, String shapeName) {
        dim1 = d1;
        dim2 = d2;
        name = shapeName;
    }
    double area() {
        if(name.equals("Circle"))
            return (dim2 * dim1 * dim1);
        else if(name.equals("Square"))
            return (dim1 * dim1);
        else if(name.equals("Rectangle"))
            return (dim1 * dim2);
        else if(name.equals("Triangle"))
            return (dim3 * dim1 * dim2);
        else
            return 0.0;
    }
}
```

Parameterized Constructor:

A constructor which has a specific number of parameters (1 or more) is called a parameterized constructor. The parameterized constructor is used to provide different values to distinct objects. However, we can provide the same values also.

81. Program:

```
class ParameterizedConstructor1 {
    public static void main(String[]args) {
        System.out.println("The main() method started.");
        ParamConstructor obj1 = new ParamConstructor(1, 2.3, 'a', "Object 1", true);
        System.out.println("Object 1 of class initialized via Parameterized constructor :");
        ParamConstructor obj2 = new ParamConstructor(13, 1.2023, 'b', "Object 2", false);
        System.out.println("Object 2 of class initialized via Parameterized constructor :");
        ParamConstructor obj3 = new ParamConstructor(456, 3.14, 'z', "Object 3", true);
```

```
        System.out.println("Object 3 of class initialized via Parameterized constructor :");
        System.out.println("\nDefault initial values of obj1 :");
        obj1.showData();
        System.out.println("Default initial values of obj2 :");
        obj2.showData();
        System.out.println("Default initial values of obj3 :");
        obj3.showData();
        System.out.println("\nThank you.");
    }
}
class ParamConstructor {
    int i;
    char c;
    double d;
    boolean b;
    String s;
    ParamConstructor(int in, double du, char ch, String st, boolean bo) {
        System.out.println("Parameterized Constructor invoked.");
        i = in;
        c = ch;
        d = du;
        b = bo;
        s = st;
    }
    void showData() {
        System.out.println("int i : " + i + "\nchar c : " + c);
        System.out.println("double d : " + d + "\nboolean b : " + b + "\nString s : " + s);
    }
}
```

82. Program:

```
class ParameterizedConstructor2 {
    public static void main(String[]a) {
        Shape objC = new Shape(5.0, 3.14, "Circle");
        Shape objR = new Shape(45.3, 65.6, "Rectangle");
        Shape objS = new Shape(6.0, 0.0, "Square");
        Shape objT = new Shape(5.3, 6.6, "Triangle");
        /* ERROR : constructor Shape in class Shape cannot be applied to given types;
        Shape obj = new Shape(); */
        System.out.println("Area of " + objC.name + " : " + objC.area());
        System.out.println("Area of " + objS.name + " : " + objS.area());
        System.out.println("Area of " + objT.name + " : " + objT.area());
        System.out.println("Area of " + objR.name + " : " + objR.area());
        System.out.println("Thank you.");
    }
}
class Shape {
    double dim1, dim2;
```

```
    String name;
    Shape(double d1, double d2, String shapeName) {
        dim1 = d1;
        dim2 = d2;
        name = shapeName;
        System.out.println("Parameterized constructor for Shape " + name + "  invoked.");
    }
    double area() {
        if(name.equals("Circle"))
            return (dim2 * dim1 * dim1);
        else if(name.equals("Square"))
            return (dim1 * dim1);
        else if(name.equals("Rectangle"))
            return (dim1 * dim2);
        else if(name.equals("Triangle"))
            return (0.5 * dim1 * dim2);
        else
            return 0.0;
    }
}
```

Overloading of Constructors:

In Java, a constructor is just like a method but without return type. It can also be overloaded like the other Java methods.
Constructor overloading in Java is a technique of having more than one constructor with different parameter lists (signatures). They are arranged in a way that each constructor performs a different task. The compiler differentiate them with the help of the number of parameters in the list or/and their types.

83. Program:

```
class OverloadingConstructors1 {
    public static void main(String[]args) {
        System.out.println("The main() method started.");
        ConstructorDemo obj1 = new ConstructorDemo(1, 2.3, 'a', "Object 1", true);
        System.out.println("Object 1 of class initialized via Parameterized constructor :");
        ConstructorDemo obj2 = new ConstructorDemo();
        System.out.println("Object 2 of class initialized via Default constructor :");
        ConstructorDemo obj3 = new ConstructorDemo(456, 3.14, 'z', "Object 3", false);
        System.out.println("Object 3 of class initialized via Parameterized constructor :");
        System.out.println("\nDefault initial values of obj1 :");
        obj1.showData();
        System.out.println("Default initial values of obj2 :");
        obj2.showData();
        System.out.println("Default initial values of obj3 :");
        obj3.showData();
        System.out.println("\nThank you.");
    }
```

```
}
class ConstructorDemo {
    int i;
    char c;
    double d;
    boolean b;
    String s;
    ConstructorDemo() {
        System.out.println("Default Constructor invoked.");
        i = 24;
        c = 'A';
        d = 1.2023;
        b = true;
        s = "Default String";
    }
    ConstructorDemo(int in, double du, char ch, String st, boolean bo) {
        System.out.println("Parameterized Constructor invoked.");
        i = in;
        c = ch;
        d = du;
        b = bo;
        s = st;
    }
    void showData() {
        System.out.println("int i : " + i + "\nchar c : " + c);
        System.out.println("double d : " + d + "\nboolean b : " + b + "\nString s : " + s);
    }
}
```

84. Program:

```
class OverloadingConstructors2 {
    public static void main(String[]a) {
        Shape objC = new Shape(5.0, "Circle");
        Shape objR = new Shape(45.3, 65.6, "Rectangle");
        Shape objS = new Shape(6.0, "Square");
        Shape objT = new Shape(5.3, 6.6, "Triangle");
        Shape obj = new Shape();
        System.out.println("Area of " + objC.name + " : " + objC.area());
        System.out.println("Area of " + objS.name + " : " + objS.area());
        System.out.println("Area of " + objT.name + " : " + objT.area());
        System.out.println("Area of " + objR.name + " : " + objR.area());
        System.out.println("Area of " + obj.name + " : " + obj.area());
        System.out.println("Thank you.");
    }
}
class Shape {
    double dim1, dim2;
    String name;
```

```
    Shape() {
        dim1 = 1.0;
        dim2 = 1.0;
        name = "Shape not defined";
        System.out.println("Default constructor for class Shape invoked.");
    }
    Shape(double d1, String shapeName) {
        dim1 = d1;
        dim2 = 3.14;
        name = shapeName;
        System.out.println("2 Parameters constructor for Shape " + name + " invoked.");
    }
    Shape(double d1, double d2, String shapeName) {
        dim1 = d1;
        dim2 = d2;
        name = shapeName;
        System.out.println("3 Parameters constructor for Shape " + name + "  invoked.");
    }
    double area() {
        if(name.equals("Circle"))
            return (dim2 * dim1 * dim1);
        else if(name.equals("Square"))
            return (dim1 * dim1);
        else if(name.equals("Rectangle"))
            return (dim1 * dim2);
        else if(name.equals("Triangle"))
            return (0.5 * dim1 * dim2);
        else
            return 0.0;
    }
}
```

The this Keyword:

Sometimes a method needs to refer to the object that invoked it. To allow this, Java defines the "**this**" keyword. The "**this**" keyword can be used inside any (non-static) method to refer to the current object. That is, "**this**" is always a reference to the current object of the class type.

85. Program:

```
class ThisKeyword1 {
    public static void main(String[]args) {
        ThisKeyword obj1 = new ThisKeyword();
        ThisKeyword obj2 = new ThisKeyword();
        System.out.println("obj1 ID : " + obj1);
        obj1.showID();
        System.out.println("obj2 ID : " + obj2);
        obj2.showID();
```

```
    }
}
class ThisKeyword {
    void showID() {
        System.out.println("Object ID that invoked this method : " + this);
    }
}
```

86. Program:

```
class ThisKeyword2 {
    public static void main(String[]str) {
        Person stud = new Person(101, "Harshal", 98.56f);
        Person staff = new Person("Santosh", 75.99f);
        System.out.println("Details of stud object : " + stud + " are : ");
        stud.showDetails();
        System.out.println("Details of staff object : " + staff + " are : ");
        staff.showDetails();
        System.out.println("Thank you.");
    }
}
class Person {
    String name, role;
    float d;
    int i;
    Person(String name, float d) {
        this.name = name;
        this.d = d;
        role = "Staff";
        System.out.println(role + " object initialized.");
    }
    Person(int i, String name, float d) {
        this.i = i;
        this.name = name;
        this.d = d;
        role = "Student";
        System.out.println(role + " object initialized.");
    }
    void showDetails() {
        System.out.println(role + " details, using the object " + this + " are :");
        if(this.role == "Staff") {
            System.out.println("Name : " + this.name);
            System.out.println("Salary Rs : " + this.d);
        }
        else if(this.role == "Student") {
            System.out.println("Name : " + this.name);
            System.out.println("Roll No : " + this.i);
            System.out.println("Percentage : " + this.d);
        }
```

```
    }
}
```

Garbage Collections:

Since object initialization and memory allocation is done dynamically using the **new** operator, in Java, we might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for us implicitly (automatically).

The technique that accomplishes this is called **garbage collection**. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects in Java. Garbage collection only occurs sporadically (if at all) during the execution of our program.

Objects as Parameters:

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods.
In Java the objects are passed by reference when an object is passed to any method as a parameter. Thus, if we perform any modifications/changes on the data members of the object that is passed as argument, the changes will be performed/ reflected into the original object, that is in the caller method.

87. Program:

```
class ObjectAsParameter1 {
    public static void main(String[]args) {
        ObjectAsParameter obj1 = new ObjectAsParameter();
        obj1.setData("One", 15, 2.4);
        ObjectAsParameter obj2 = new ObjectAsParameter();
        obj2.setData("Two", 24, 3.6);
        System.out.println("Object 1 details are :");
        obj1.showData();
        System.out.println("Object 2 details are :");
        obj2.showData();
        ObjectAsParameter obj3 = new ObjectAsParameter();
        obj3.incrementData(obj1);
        System.out.println("\nAfter incrementing values of obj1 and then assigned to obj3 :");
        System.out.println("Object 1 details are :");
        obj1.showData();
        System.out.println("Object 2 details are :");
        obj2.showData();
        System.out.println("Object 3 details are :");
        obj3.showData();
        System.out.println("Thank you.");
    }
}
class ObjectAsParameter {
    String objName;
```

```
    int i;
    double d;
    void setData(String name, int i, double d) {
        objName = name;
        this.i = i;
        this.d = d;
    }
    void showData() {
        System.out.printf("Object name : %s, i : %d, and d : %g.\n", objName, i, d);
    }
    void incrementData(ObjectAsParameter obj) {
        i = ++obj.i;
        d = ++obj.d;
    }
}
```

88. Program:

```
class ObjectAsParameter2 {
    public static void main(String[]strs) {
        Distance d1 = new Distance(12, 5.7f);
        Distance d2 = new Distance(12, 2.5f);
        System.out.print("Distance 1 : ");
        d1.showDistance();
        System.out.print("Distance 2 : ");
        d2.showDistance();
        if(d1.greaterDistance(d2))
            System.out.println("Distance 1 is greater than distance 2.");
        else
            System.out.println("Distance 2 is greater than distance 1.");
        System.out.println("Thank you.");
    }
}
class Distance {
    int feet;
    float inches;
    Distance(int f, float i) {
        feet = f;
        inches = i;
    }
    boolean greaterDistance(Distance obj) {
        if(feet > obj.feet)
            return true;
        else if(feet == obj.feet)
            if(inches > obj.inches)
                return true;
            else
                return false;
        else
```

```
        return false;
    }
    void showDistance() {
        System.out.println("Feet : " + feet + ", inches : " + inches);
    }
}
```

Returning Objects:

A method can return any type of data, including class types that we create. That is, a method can return an object of the class.
As we know that when an object is passed as a parameter, Java passes the object's reference to the calling method (Call-by-reference). The same is the case while returning an object from a method. Java compiler returns the reference of the object while returning an object from any method.

89. Program:

```
class ReturningObject1 {
    public static void main(String[]args) {
        ReturningObject no1 = new ReturningObject(24);
        ReturningObject no2;
        no2 = no1.incrementData();
        System.out.print("Number 1 : ");
        no1.showData();
        System.out.print("Number 2 : ");
        no2.showData();
    }
}
class ReturningObject {
    int no;
    ReturningObject(int n) {
        no = n;
    }
    ReturningObject incrementData() {
        ReturningObject obj = new ReturningObject(no + 1);
        return obj;
    }
    void showData() {
        System.out.println(no);
    }
}
```

90. Program:

```
class ReturningObject2 {
    public static void main(String[]args) {
        Distance d1 = new Distance("Length", 12, 5.7f);
        Distance d2 = new Distance("Width", 12, 2.5f);
        System.out.print("Distance 1 : ");
        d1.showDistance();
```

```
        System.out.print("Distance 2 : ");
        d2.showDistance();
        Distance d3;
        d3 = d1.greaterDistance(d2);
        System.out.println("Greater distance : ");
        d3.showDistance();
        System.out.println("Thank you.");
    }
}
class Distance {
    String name;
    int feet;
    float inches;
    Distance(String name, int f, float i) {
        this.name = name;
        feet = f;
        inches = i;
    }
    Distance greaterDistance(Distance obj) {
        if(feet > obj.feet)
            return this;
        else if(feet == obj.feet)
            if(inches > obj.inches)
                return this;
            else
                return obj;
        else
            return obj;
    }
    void showDistance() {
        System.out.println(name + ", Dimensions, Feet : " + feet + ", inches : " + inches);
    }
}
```

XII - Access Control Mechanism:

Introduction to Access Control:

As we know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: **access control**. Through encapsulation, we can control what parts of a program can access the members of a class. By controlling access, we can prevent misuse of data and/or methods.

For example, by allowing access to data only through a well defined set of methods, we can prevent the misuse of that data. Thus, when correctly implemented, a class creates a "black box" which may be used, but the inner workings of which are not open to tampering. However, the classes that were presented earlier do not completely meet this goal.
That is, it is possible for another part of the program to bypass these methods and access the stack of data directly. Of course, in the wrong hands, this could lead to trouble.

In this section, we will be introduced to the mechanism by which we can precisely control access to the various members of a class.
How a member can be accessed is determined by the **access modifier** attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance and/or packages. (A package is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later. Here, let's begin by examining access control as it applies to a single class, or a single package. Once we understand the fundamentals of access control, the rest will be easy.

Java's access modifiers are **public**, **private**, and **protected**. Java also defines a **default** access level. The **protected** access modifier applies only when inheritance is involved. The other access modifiers are described here.
Let's begin by defining **public** and **private** access modifiers.
When a member of a class is defined by the **public** access modifier, then that member can be accessed by any other code.
When a member of a class is specified as **private**, then that member can only be accessed by other members of its own class, and not by the other code.

Now we can understand why the **main( )** method has always been preceded by the **public** modifier. It is called by code that is outside the program—that is, by the Java run-time system.

When no access modifier is used, then by default the member of a class is like public within its own package, but cannot be accessed outside of its package. (Packages are discussed in the next chapters.)
In the classes developed so far, all members of a class have used the **default** access mode. However, this is not what we will typically want to be the case. Usually, we will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when we will want to define some methods that are **private** to a class.

An access modifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement.

General form:

```
accessModifier dataType variableName;
...
accessModifier returnType methodName([parameterList]) {
      // method body
      ...
}
...
```

91. Program:

```
class AccessModifiers1 {
    public static void main(String[]args) {
        AccessModifiersDemo obj = new AccessModifiersDemo();
        obj.defNo = 10;
        System.out.println("Default variable : " + obj.defNo);
        obj.pubNo = 20;
        System.out.println("Public variable : " + obj.pubNo);
        /* ERROR : priNo has private access in AccessModifiersDemo
        obj.priNo = 30;
        System.out.println("Private variable : " + obj.priNo); */
    }
}
class AccessModifiersDemo {
    int defNo;
    public int pubNo;
    private int priNo;
}
```

To get or set the value of a **private** member variable we can define a member method using which we will be able to get/set the value of private member variables.

92. Program:

```
class AccessModifiers2 {
    public static void main(String[]args) {
        AccessModifiersDemo obj = new AccessModifiersDemo();
        obj.defNo = 10;
        System.out.println("Default variable : " + obj.defNo);
        obj.pubNo = 20;
        System.out.println("Public variable : " + obj.pubNo);
        obj.setData(30);
        System.out.println("Private variable : " + obj.getData());
    }
}
class AccessModifiersDemo {
    int defNo;
    public int pubNo;
    private int priNo;
    void setData(int no) {
        priNo = no;
    }
    int getData() {
        return priNo;
    }
}
```

XIII - The static Keyword:

There will be times when we want to define a class member that will be used independently of any object of that class. However, it is possible to create a member that can be used by itself, without reference to any specific instance/object.
To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any instance/object.
We can declare both methods and variables to be **static**. The most common example of a **static** member is the **main()** method. The **main( )** method is declared as **static** because it must be invoked/called before any objects exist.

The static Variables:

Variables declared as **static** are, essentially, global variables, global to its class only. That is, a **static** variable should not be declared inside any method, or block. When objects of its class are defined, no instance (copy) of a **static** variable is made. Insead, all instances of the class share the same **static** variable.

93. Program:

```
class StaticVariables1 {
    static int a = 60;
    static int b;
    int c;
    public static void main(String[]str) {
        System.out.println("Default initial value of variables :");
        System.out.println("Static variable a : " + a);
        System.out.println("Static variable b : " + b);
        /* ERROR : non-static variable c cannot be referenced from a static context
        System.out.println("Non-static variable c : " + c); */
        StaticVariables1 obj = new StaticVariables1();
        System.out.println("Non-static variable obj.c : " + obj.c);
        obj.a = 150;
        b = 240;
        obj.c = 320;
        System.out.println("\nAfter assigning new values to variables :");
        System.out.println("Static variable a : " + a);
        System.out.println("Static variable obj.b : " + obj.b);
        System.out.println("Non-static variable obj.c : " + obj.c);
    }
}
```

The static Methods:

Java has a methodology to define methods as **static**. The static method is mainly defined to access other static members.
However, when a method is defined as **static**, it has some restrictions:

- They can only directly call other **static** methods.
- They can only directly access **static** data.
- They cannot refer to **this**, or **super** in any way. (The keyword **super** relates to inheritance).

94. Program:

```
class StaticMethod1 {
    static int a = 60;
    int b = 150;
    public static void main(String[]args) {
        System.out.println("The main() method invoked.");
        System.out.println("Static variable a : " + a);
        StaticMethod1 obj = new StaticMethod1();
        System.out.println("Non-static variable obj.b : " + obj.b);
        staticMethod();
        System.out.println("\nReturned from the staticMethod() :");
        System.out.println("Static variable a : " + a);
        System.out.println("Non-static variable obj.b : " + obj.b);
        System.out.println("Thank you.");
    }
    static void staticMethod() {
        System.out.println("\nThe staticMethod() invoked.");
        System.out.println("Static variable a : " + a);
        StaticMethod1 obj = new StaticMethod1();
        System.out.println("Non-static variable obj.b : " + obj.b);
        a = 240;
        obj.b = 320;
        System.out.println("After assigning new values to variables :");
        System.out.println("Static variable a : " + a);
        System.out.println("Non-static variable obj.b : " + obj.b);
    }
}
```

Static Members from Another Class:

Static members from another class can be referred to using the class name and the period/dot (.) operator.

Syntax:

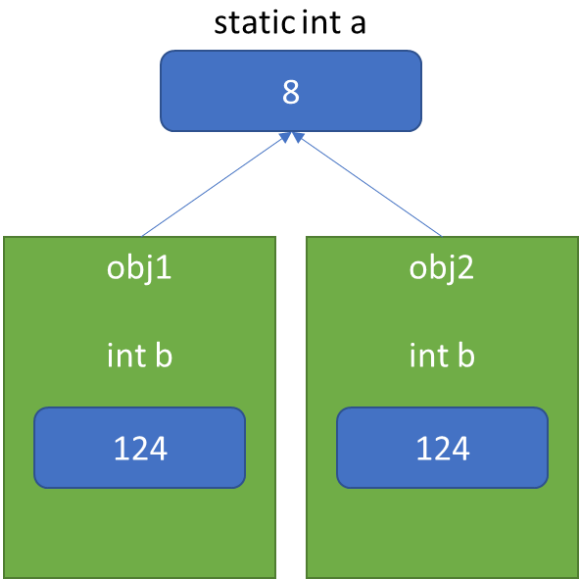
```
ClassName.staticDataMember = value;
ClassName.staticMethod([arguments]);
```

95. Program:

```
class StaticMembers1 {
    public static void main(String[]args) {
        System.out.println("The main() invoked.");
        System.out.println("StaticAndNonStaticMembers.a : " + StaticAndNonStaticMembers.a);
        StaticAndNonStaticMembers obj1 = new StaticAndNonStaticMembers();
        System.out.println("Non-static variable obj1.b : " + obj1.b);
        System.out.println("Calling the staticMethod() :");
        StaticAndNonStaticMembers.staticMethod();
        System.out.println("\nReturned from the staticMethod() :");
        System.out.println("StaticAndNonStaticMembers.a : " + StaticAndNonStaticMembers.a);
```

```
        System.out.println("Non-static variable obj1.b : " + obj1.b);
        System.out.println("Calling the obj1.nonStaticMethod() :");
        obj1.nonStaticMethod();
        System.out.println("\nReturned from the obj1.nonStaticMethod() :");
        System.out.println("StaticAndNonStaticMembers.a : " + StaticAndNonStaticMembers.a);
        System.out.println("Non-static variable obj1.b : " + obj1.b);
        System.out.println("Thank you.");
    }
}
class StaticAndNonStaticMembers {
    static int a = 6;
    int b = 123;
    static void staticMethod() {
        System.out.println("The staticMethod() invoked.");
        ++a;
        System.out.println("a : " + a);
        StaticAndNonStaticMembers obj2 = new StaticAndNonStaticMembers();
        ++obj2.b;
        System.out.println("obj2.b : " + obj2.b);
    }
    void nonStaticMethod() {
        System.out.println("The nonStaticMethod() invoked.");
        ++a;
        System.out.println("a : " + a);
        ++b;
        System.out.println("b : " + b);
    }
}
```

Memory representation of static and instance (non-static) variables of class:



The static Block:

If we need to do computation in order to initialize our **static** variables, we can declare a **static** block that gets executed exactly once, when the class is first loaded.

General form:

```
static {
      // block of code
}
```

96. Program:

```
class StaticBlock1 {
    static double pi;
    double radius;
    public static void main(String[]para) {
        System.out.print("Enter radius of a Circle :");
        StaticBlock1 obj = new StaticBlock1();
        obj.radius = Double.parseDouble(System.console().readLine());
        System.out.println("Area of Circle : " + obj.area());
        System.out.println("Thank you.");
    }
    static {
        pi = 3.14;
        System.out.println("Value of PI initialized.");
    }
    double area() {
        return pi * radius * radius;
    }
}
```

The final Keyword:

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant field. This means that we must initialize a **final** field when it is declared.
We can do this in one of two ways:

- First, we can give it a value when it is declared, or
- Second, we can assign it a value within a constructor.

97. Program:

```
class FinalDemo1 {
    int a;
    int b = 20;
    final int c;
    final int d = 40;
    public static void main(String[]arg) {
        FinalDemo1 obj = new FinalDemo1();
        System.out.println("Default initial values :");
        System.out.println("obj.a : " + obj.a);
```

```
        System.out.println("obj.b : " + obj.b);
        System.out.println("obj.c : " + obj.c);
        System.out.println("obj.d : " + obj.d);
        obj.a = 100;
        obj.b = 200;
        // obj.c = 300; // ERROR : cannot assign a value to final variable c
        // obj.d = 400; // ERROR : cannot assign a value to final variable d
        System.out.println("After assigning new values :");
        System.out.println("obj.a : " + obj.a);
        System.out.println("obj.b : " + obj.b);
        System.out.println("obj.c : " + obj.c);
        System.out.println("obj.d : " + obj.d);
        System.out.println("Thank you.");
    }
    FinalDemo1() {
        c = 30;
    }
}
```

98. Program:

```
class FinalDemo2 {
    public static void main(String[]args) {
        Shapes objC = new Shapes("Circle", 5.0f, 0.0f);
        Shapes objT = new Shapes("Triangle", 25.0f, 30.0f);
        Shapes objR = new Shapes("Rectangle", 45.0f, 60.0f);
        Shapes objS = new Shapes("Square", 6.0f, 0.0f);
        System.out.println("\nShape " + objC.name + ". Area : " + objC.getArea());
        System.out.println("Shape " + objR.name + ". Area : " + objR.getArea());
        System.out.println("Shape " + objS.name + ". Area : " + objS.getArea());
        System.out.println("Shape " + objT.name + ". Area : " + objT.getArea());
        objC.dim1 = 7.5f;
        // objC.constant = 3.1415f; // ERROR : cannot assign a value to final variable constant
        objT.dim1 = 75.5f;
        objT.dim2 = 90.0f;
        // objT.constant = 1/2; // ERROR : cannot assign a value to final variable constant
        objR.dim1 = 80.0f;
        objR.dim2 = 100.0f;
        objS.dim1 = 15.0f;
        System.out.println("\nAfter assigning new dimensions :");
        System.out.println("Shape " + objC.name + ". Area : " + objC.getArea());
        System.out.println("Shape " + objR.name + ". Area : " + objR.getArea());
        System.out.println("Shape " + objS.name + ". Area : " + objS.getArea());
        System.out.println("Shape " + objT.name + ". Area : " + objT.getArea());
        System.out.println("Thank you.");
    }
}
class Shapes {
    String name;
```

```
    float dim1, dim2;
    final float constant;
    Shapes(String name, float dim1, float dim2) {
        this.name = name;
        this.dim1 = dim1;
        this.dim2 = dim2;
        if(this.name.equals("Circle"))
            constant = 3.14f;
        else if(this.name.equals("Triangle"))
            constant = 0.5f;
        else
            constant = 0.0f;
        System.out.println("Shape " + this.name + " created.");
    }
    float getArea() {
        if(name.equals("Circle"))
            return constant * dim1 * dim1;
        else if(name.equals("Triangle"))
            return constant * dim1 * dim2;
        else if(name.equals("Rectangle"))
            return dim1 * dim2;
        else if(name.equals("Square"))
            return dim1 * dim1;
        else
            return 0.0f;
    }
}
```

Nested and Inner Classes:

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A.
A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
A nested class that is declared directly within its enclosing class scope is a member of its enclosing class. It is also possible to declare a nested class that is local to a block.
The most important type of nested class is the **inner** class. An **inner** class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

99. Program:

```
class InnerClass1 {
    public static void main(String[]args) {
        OuterClass obj = new OuterClass();
        obj.outerMethod();
        // obj.innerMethod();
        // InnerClass objIn = new InnerClass();
```

```
        System.out.println("Thank you.");
    }
}
class OuterClass {
    int out = 150;
    void outerMethod() {
        System.out.println("Inside the outerMethod().");
        System.out.println("out : " + out);
        // System.out.println("in : " + in);
        // innerMethod();
        InnerClass obj = new InnerClass();
        System.out.println("obj.in : " + obj.in);
        obj.in = 330;
        obj.innerMethod();
    }
    class InnerClass {
        int in = 240;
        void innerMethod() {
            System.out.println("Inside the innerMethod().");
            System.out.println("in : " + in);
            System.out.println("out : " + out);
        }
    }
}
```

100. Program:

```
class InnerClass2 {
    public static void main(String [] arg) {
        Circle objC = new Circle(2.5, 6.0);
        System.out.println("Circle details are :");
        System.out.println("Radius : " + objC.radius);
        System.out.println("Area : " + objC.area());
        System.out.println("\nCylinder details are :");
        System.out.println("Radius : " + objC.radius);
        System.out.println("Height : " + objC.objCy.height);
        System.out.println("Volume : " + objC.objCy.volume());
        objC.radius = 5.5;
        System.out.println("\nNew Circle details are :");
        System.out.println("Radius : " + objC.radius);
        System.out.println("Area : " + objC.area());
        objC.objCy.height = 15.5;
        System.out.println("\nNew Cylinder details are :");
        System.out.println("Radius : " + objC.radius);
        System.out.println("Height : " + objC.objCy.height);
        System.out.println("Volume : " + objC.objCy.volume());
        System.out.println("Thank you.");
    }
}
```

```
class Circle {
    double radius;
    final double pi = 3.14;
    Cylinder objCy;
    Circle(double radius, double height) {
        this.radius = radius;
        objCy = new Cylinder(height);
        System.out.println("Circle created");
    }
    double area() {
        return pi * radius * radius;
    }
    class Cylinder {
        double height;
        Cylinder(double height) {
            this.height = height;
            System.out.println("Cylinder created.");
        }
        double volume() {
            return area() * height;
        }
    }
}
```

XIV - Inheritance:

Introduction:

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, we can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that can be unique to them.
In the terminology of Java, a class that is inherited is called a SuperClass. SuperClass is also called the Base or the Parent class.
The class that does the inheriting is called a SubClass. SubClass is also called the Derived or the Child class.
Therefore, a Subclass is a specialized version of a Superclass. It inherits all of the members defined by the Superclass and adds its own, unique elements.
That is, a Subclass extends the traits of the Superclass.

Inheritance Basics:

To inherit a class, we simply incorporate the definition of one class into another by using the **extends** keyword.

General form:

```
class SuperClass {
      Data SuperMembers...
      ...
      Member SuperMethods([arguments])...
      ...
}
class SubClass extends SuperClass {
      Data SubMembers...
      ...
      Member SubMethods([arguments])...
      ...
}
...
SubClass objectSub = new SubClass([args]);
objectSub.SubMembers = value;
objectSub.SubMethods([args]);
objectSub.SuperMembers = value;
objectSub.SuperMethods([args]);
...
```

101. Program:

```
class Inheritance1 {
    static public void main(String[]args) {
        System.out.println("Inheritance Demo.");
        ChildClass objCh = new ChildClass();
        objCh.a = 10;
        objCh.b = 20;
        objCh.c = 30;
        objCh.d = 40;
        objCh.showAB();
```

```
        objCh.showCD();
        objCh.sum();
        System.out.println("Thank you.");
    }
}
class ParentClass {
    int a, b;
    void showAB() {
        System.out.println("Class Parent. a : " + a + ", b : " + b);
    }
} // Total members : 3, Data members : 2, Methods : 1
class ChildClass extends ParentClass {
    int c, d;
    void showCD() {
        System.out.println("Class Child. c : " + c + ", d : " + d);
    }
    void sum() {
        System.out.println("Sum of a, b, c and d : " + (a + b + c + + d));
    }
} // Total members : 3 + 4 = 7, Data members : 2 + 2 = 4, Methods : 1 + 2 = 3
```

102. Program:

```
class Inheritance2 {
    static public void main(String[]args) {
        System.out.println("Inheritance Demo.");
        DerivedClass objD = new DerivedClass();
        objD.a = 10;
        objD.b = 20;
        objD.c = 30;
        objD.d = 40;
        objD.showAB();
        objD.showCD();
        objD.sum();

        BaseClass objB = new BaseClass();
        objB.a = 100;
        objB.b = 200;
        /* objB.c = 300;
        objB.d = 400; */
        objB.showAB();
        /* objB.showCD();
        objB.sum(); */
        System.out.println("Thank you.");
    }
}
class BaseClass {
    int a, b;
    void showAB() {
```

```
        System.out.println("Class Base. a : " + a + ", b : " + b);
    }
} // Total members : 3, Data members : 2, Methods : 1
class DerivedClass extends BaseClass {
    int c, d;
    void showCD() {
        System.out.println("Class Derived. c : " + c + ", d : " + d);
    }
    void sum() {
        System.out.println("Sum of a, b, c and d : " + (a + b + c + + d));
    }
} // Total members : 3 + 4 = 7, Data members : 2 + 2 = 4, Methods : 1 + 2 = 3
```

Accessing Members in Inheritance:

Although a subclass derives all the members of its superclass, it cannot access those members of the superclass that have been declared as **private**.
That means, when an object of derived class is initialized, the memory to all the members (private, public, protected, default) of base as well as derived class is assigned and initialized, but the **private** members of base class are inaccessible, due to its access privileges, to the derived class. Thus to access the **private** members of base class into derived class you have to use the method of the base class that gives the access to the **private** members.

103. Program:

```
class Inheritance3 {
    public static void main(String[]args) {
        Sub obj = new Sub();
        /* obj.str = "Java Inheritance";
        obj.d = 3.14; */
        obj.setSuperData("Java Inheritance", 3.14);
        obj.i = 15;
        // obj.dd = 2.2023;
        obj.setSubData(2.2023);
        obj.id = 8;
        obj.showSuperData();
        obj.showSubData();
        System.out.println("Thank you.");
    }
}
class Super {
    private String str;
    private double d;
    int i;
    void setSuperData(String s, double d) {
        str = s;
        this.d = d;
    }
    void showSuperData() {
        System.out.println("Super class data :");
```

```
        System.out.printf("str : %s, d : %g and i : %d\n", str, d, i);
    }
} // Total members : 5, Data members : 3, Methods : 2
class Sub extends Super {
    private double dd;
    int id;
    void setSubData(double dd) {
        this.dd = dd;
    }
    void showSubData() {
        System.out.println("Subclass data :");
        // System.out.printf("str : %s and d : %g\n", str, d);
        System.out.printf("i : %d\n", i);
        System.out.printf("dd : %g and id : %d\n", dd, id);
    }
} // Total members : 5 + 4 = 9, Data members : 3 + 2 = 5, Methods : 2 + 2 = 4
```

Superclass Object References Subclass:

A reference variable, that is the object of a superclass can be assigned a reference (object) to any subclass derived from that superclass. We will find this aspect of inheritance quite useful in a variety of situations.

104. Program:

```
class Inheritance4 {
    public static void main(String[]args) {
        BoxWeight b1 = new BoxWeight();
        b1.setBox(10.5, 20.2, 15.6);
        b1.setWeight(12.5);
        Box b2 = new BoxWeight();
        b2.setBox(5.2, 6.3, 7.4);
        // b2.setWeight(6.5); // ERROR : cannot find symbol
        System.out.println("Box 1 (Derived) details are :");
        b1.showBox();
        b1.showWeight();
        System.out.println("Volume of Box 1 : " + b1.getVolume());
        System.out.println("Box 2 (Base) details are :");
        b2.showBox();
        // b2.showWeight(); // ERROR : cannot find symbol
        System.out.println("Volume of Box 2 : " + b2.getVolume());
    }
}
class Box {
    private double w, d, h;
    void setBox(double width, double depth, double height) {
        w = width;
        d = depth;
        h = height;
```

```
    }
    void showBox() {
        System.out.printf("Width : %g, depth : %g and height : %g.\n", w, d, h);
    }
    double getVolume() {
        return w * d * h;
    }
} // Total members : 6, Data members : 3, Methods : 3
class BoxWeight extends Box {
    private double weight;
    void setWeight(double w) {
        weight = w;
    }
    void showWeight() {
        System.out.println("Weight : " + weight);
    }
} // Total members : 6 + 3 = 9, Data members : 3 + 1 = 4, Methods : 3 + 2 = 5
```

Assigning/Modifying Data:

When the reference (object) of a subclass is assigned to the reference (object) of superclass, we actually use the memory and members of the subclass reference (object) via superclass reference (object). Thus if we assign/modify values of the data members using the reference (object) of superclass, the changes will be reflected into the reference (object) of subclass.
However, even if we assign the reference (object) of subclass to the reference of superclass, we are not able to access the members of subclass using the reference (object) of superclass.

105. Program:

```
class Inheritance5 {
    public static void main(String[]args) {
        Cylinder cy1 = new Cylinder();
        cy1.setRadius(5.0);
        cy1.setHeight(10.0);
        Circle ci1 = new Circle();
        ci1.setRadius(6.0);
        System.out.println("Cylinder details are :");
        cy1.showRadius();
        cy1.showHeight();
        System.out.println("Volume : " + cy1.getVolume());
        System.out.println("\nCircle details are :");
        ci1.showRadius();
        System.out.println("Area : " + ci1.getArea());
        ci1 = cy1;
        System.out.println("\nAssigning derived class (Cylinder) object to base class (Circle)
object :");
        ci1.setRadius(7.0);
        // ci1.setHeight(15.0); // ERROR : cannot find symbol
        System.out.println("Cylinder details, using derived class (Cylinder) object, are :");
```

```
        cy1.showRadius();
        System.out.println("Area : " + cy1.getArea());
        cy1.showHeight();
        System.out.println("Volume : " + cy1.getVolume());
        System.out.println("\nCylinder details, using base class (Circle) object, are :");
        ci1.showRadius();
        System.out.println("Area : " + ci1.getArea());
        // ci1.showHeight(); // ERROR : cannot find symbol
        // System.out.println("Volume : " + ci1.getVolume()); // ERROR : cannot find symbol
        System.out.println("\nThank you.");
    }
}
class Circle {
    private double r;
    void setRadius(double radius) {
        r = radius;
    }
    void showRadius() {
        System.out.println("Radius : " + r);
    }
    double getArea() {
        return 3.14 * r * r;
    }
} // Total members : 4, Data members : 1, Methods : 3
class Cylinder extends Circle {
    private double h;
    void setHeight(double height) {
        h = height;
    }
    void showHeight() {
        System.out.println("Height : " + h);
    }
    double getVolume() {
        return getArea() * h;
    }
} // Total members : 4 + 4 = 8, Data members : 1 + 1 = 2, Methods : 3 + 3 = 6
```

Constructors in Inheritance:

A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass is invoked during the initialization of the subclass.
First the constructor of the superclass is invoked, and then the subclass constructor is invoked. This is implicitly present in the Java language, and also in all OOP languages, so that the members (data and methods) of the superclass should be initialized first and then those will be inherited by the subclass.

Default Constructor in SubClass:

When we define a constructor of the subclass, upon initializing its object the constructor will be implicitly invoked, just like for any other class. When we have not defined the constructor of the superclass, Java uses the implicit constructor to construct the instance members of the superclass. This implicit constructor of the superclass is invoked first, and then the default constructor of the subclass will be invoked.

106. Program:

```
class DefaultConstructorInSub1 {
    static public void main(String[]args) {
        System.out.println("Default constructor in SubClass : ");
        Cylinder cy = new Cylinder();
        cy.showRadius();
        cy.showArea();
        cy.showHeight();
        cy.showVolume();
        System.out.println("Thank you.");
    }
}
class Circle {
    double r;
    void showRadius() {
        System.out.println("Radius : " + r);
    }
    void showArea() {
        System.out.println("Area : " + (3.14 * r * r));
    }
} // Total members : 3, Data members : 1, Methods : 2
class Cylinder extends Circle {
    double h;
    Cylinder() {
        r = 1.0;
        h = 1.0;
        System.out.println("Cylinder details set.");
    }
    void showHeight() {
        System.out.println("Height : " + h);
    }
    void showVolume() {
        System.out.println("Volume : " + (3.14 * r * r * h));
```

```
    }
} // Total members : 3 + 3 = 6, Data members : 1 + 1 = 2, Methods : 2 + 2 = 4
```

Default Constructor in SuperClass:

When we define a constructor in the superclass, upon initializing the object of the subclass the constructor of the superclass will be implicitly invoked. This happens in inheritance and due to this invocation of superclass constructor on initializing the object of subclass all the members of superclass are given to the object of subclass.
Here as we have not defined the constructor of subclass, Java uses the implicit constructor of subclass. No doubt the default constructor of superclass, that is define, will be invoked first, and then the implicit constructor of subclass will be invoked during the initialization of the subclass object.

107. Program:

```
class DefaultConstructorInSuper1 {
    public static void main(String[]args) {
        System.out.println("Default constructor in SuperClass : ");
        Cylinder cy = new Cylinder();
        cy.showRadius();
        cy.showArea();
        cy.showHeight();
        cy.showVolume();
        System.out.println("Thank you.");
    }
}
class Circle {
    double r;
    Circle() {
        r = 1.0;
        System.out.println("Circle details set.");
    }
    void showRadius() {
        System.out.println("Radius : " + r);
    }
    void showArea() {
        System.out.println("Area : " + (3.14 * r * r));
    }
} // Total members : 3, Data members : 1, Methods : 2
class Cylinder extends Circle {
    double h;
    void showHeight() {
        System.out.println("Height : " + h);
    }
    void showVolume() {
        System.out.println("Volume : " + (3.14 * r * r * h));
    }
} // Total members : 3 + 3 = 6, Data members : 1 + 1 = 2, Methods : 2 + 2 = 4
```

Default Constructor in Super as well as SubClass:

Java allows us to define constructors in both, superclass as well as subclass. Both the constructors will be implicitly invoked upon initializing the object of subclass. The constructor of superclass is invoked first and then the constructor of subclass is invoked, this is because the members of superclass need to be initialized first and then will be inherited by the subclass.

108. Program:

```
class DefaultConstructorInSuperSub1 {
    public static void main(String[]args) {
        System.out.println("Default constructor in SuperClass as well as SubClass : ");
        Cylinder cy = new Cylinder();
        cy.showRadius();
        cy.showArea();
        cy.showHeight();
        cy.showVolume();
        System.out.println("Thank you.");
    }
}
class Circle {
    double r;
    Circle() {
        r = 1.0;
        System.out.println("Circle details set.");
    }
    void showRadius() {
        System.out.println("Radius : " + r);
    }
    void showArea() {
        System.out.println("Area : " + (3.14 * r * r));
    }
} // Total members : 3, Data members : 1, Methods : 2
class Cylinder extends Circle {
    double h;
    Cylinder() {
        h = 1.0;
        System.out.println("Cylinder details set.");
    }
    void showHeight() {
        System.out.println("Height : " + h);
    }
    void showVolume() {
        System.out.println("Volume : " + (3.14 * r * r * h));
    }
} // Total members : 3 + 3 = 6, Data members : 1 + 1 = 2, Methods : 2 + 2 = 4
```

Parameterized Constructors in Inheritance:

Java language permits us to define parameterized constructors in inheritance. Parameterized constructor require parameters/values to be passed while the object of its class is being initialized. The parameters/values that we pass are mostly assigned to the instance variable/members of the class which further give us results as per the operations defined in the class.

Parameterized Constructor in SubClass:

When we define a parameterized constructor in the subclass, while initializing the object we need to pass the arguments and then the parameterized constructor will be invoked.
Here we are not going to define any constructor in the superclass, so the implicit default constructor will be invoked for it.

109. Program:

```
class ParameterizedConstructorInSub {
    public static void main(String[]args) {
        System.out.println("Parameterized constructor in SubClass : ");
        Circle c = new Circle(5.0f);
        c.showCircle();
        System.out.println("Thank you.");
    }
}
class Shape {
    String name;
    float d1, d2;
    void showShape() {
        System.out.println("Shape name : " + name);
        System.out.println("Dimension 1 : " + d1 + ", dimension 2 : " + d2);
    }
} // Total members : 4, Data members : 3, Methods : 1
class Circle extends Shape {
    float a;
    Circle(float radius) {
        name = "Circle";
        d1 = 3.14f;
        d2 = radius;
        System.out.println(name + " shape defined and radius set.");
    }
    void showCircle() {
        a = d1 * d2 * d2;
        showShape();
        System.out.println("Area : " + a);
    }
} // Total members : 4 + 2 = 6, Data members : 3 + 1 = 4, Methods : 1 + 1 = 2
```

Default in Super and Parameterized constructor in SubClass:

When we define a parameterized constructor in the subclass, while initializing the object we need to pass the arguments and then the parameterized constructor will be invoked.
Here we are going to define a default constructor in the superclass, so this default constructor will be invoked first, of the superclass and then the parameterized constructor of the subclass will be invoked.

110. Program:

```
class ParameterizedConstructorInSub2 {
    static public void main(String[]args) {
        System.out.println("Parameterized constructor in SubClass & Default in SuperClass :");
        Circle c = new Circle("Circle");
        System.out.println("\nDefault initial values :");
        c.showCircle();
        c.setDimensions(3.14f, 5.0f);
        System.out.println("\nAfter assigning  values :");
        c.showCircle();
        System.out.println("Thank you.");
    }
}
class Shape {
    String name;
    private float d1, d2;
    Shape() {
        name = "Not defined.";
        d1 = d2 = 1.0f;
        System.out.println("Shape details set.");
    }
    void setDimensions(float dim1, float dim2) {
        d1 = dim1;
        d2 = dim2;
    }
    float getDimension1() {
        return d1;
    }
    float getDimension2() {
        return d2;
    }
    void showShape() {
        System.out.println("Shape name : " + name);
        System.out.println("Dimension 1 : " + d1 + ", dimension 2 : " + d2);
    }
} // Total members : 7, Data members : 3, Methods : 4
class Circle extends Shape {
    float a;
    Circle(String n) {
        name = n;
        System.out.println(name + " shape defined.");
    }
```

```
    void showCircle() {
        float radius = getDimension2();
        a = getDimension1() * radius * radius;
        showShape();
        System.out.println("Area : " + a);
    }
} // Total members : 7 + 2 = 9, Data members : 3 + 1 = 4, Methods : 4 + 1 = 5
```

Parameterized Constructor in Superclass:

In the case of a parameterized constructor in a superclass, we need to explicitly call this constructor from the constructor of the subclass. To invoke the parameterized constructor of the superclass in the subclass you have to use the **super** keyword.

The super Keyword:

There will be times when we want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem.

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.
The **super** keyword has two general forms.

- Invokes/calls the superclass' constructor.
- Refers/accesses member(s) of the superclass that have been hidden by member(s) of a subclass.

Invoking the SuperClass Constructor:

A subclass can call a constructor defined by its superclass by use of the **super** keyword.

General form:

```
class SuperClass {
      ...
      SuperClass([paramList]) {
            ...
      }
      ...
}
class SubClass extends SuperClass{
      ...
      SubClass([paramList]) {
            super([argList]);
            ...
      }
      ...
}
```

111. Program:

```
class ParameterizedConstructorInSup1 {
    static public void main(String[]args) {
        System.out.println("Parameterized constructor in SuperClass and in SubClass too :");
```

```
        Circle c = new Circle("Circle", 5.0f);
        System.out.println("\nDefault initial values :");
        c.showCircle();
        System.out.println("Thank you.");
    }
}
class Shape {
    String name;
    private float d1, d2;
    Shape(float dim1, float dim2) {
        d1 = dim1;
        d2 = dim2;
        System.out.println("Shape details set.");
    }
    float getDimension1() {
        return d1;
    }
    float getDimension2() {
        return d2;
    }
    void showShape() {
        System.out.println("Shape name : " + name);
        System.out.println("Dimension 1 : " + d1 + ", dimension 2 : " + d2);
    }
} // Total members : 6, Data members : 3, Methods : 3
class Circle extends Shape {
    float a;
    Circle(String n, float radius) {
        super(3.14f, radius);
        name = n;
        System.out.println(name + " shape defined.");
    }
    void showCircle() {
        float radius = getDimension2();
        a = getDimension1() * radius * radius;
        showShape();
        System.out.println("Area : " + a);
    }
} // Total members : 6 + 2 = 8, Data members : 3 + 1 = 4, Methods : 3 + 1 = 4
```

Note that, **super([argList])** which is used to invoke/call the superclass constructor must always be in the subclass' constructor and should be the first statement executed inside that subclass' constructor.

Referring SuperClass Members:

When a data member of subclass as well as superclass have the same name, the data member of subclass overrides (hides) the data member of superclass. So, whenever we use the data member which is present in both superclass as

well as subclass, with the same name, the Java compiler uses the data member of subclass, overriding/hiding the data member of superclass.
Thus, when we need to use the data member of a superclass we need to refer to it using the **super** keyword, in the subclass.
This **super** keyword can only be used inside the subclass, which refers to the members of its superclass.

Syntax:
```
super.dataMember = value;
```

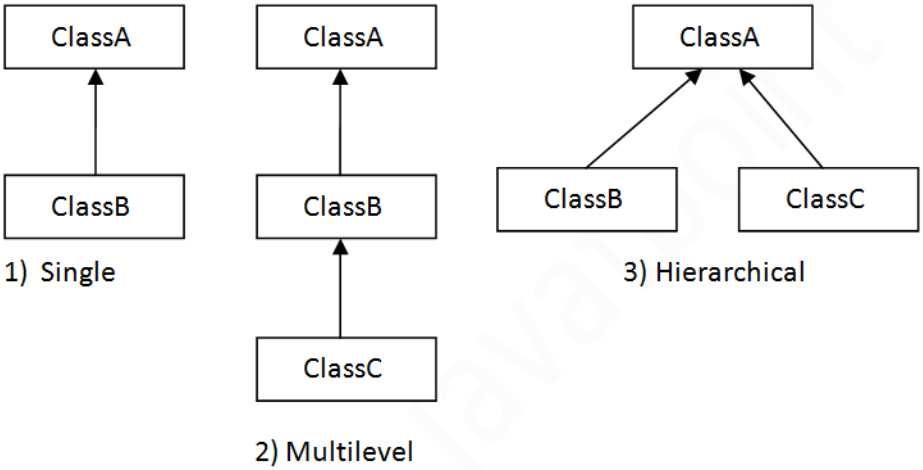
112. Program:

```
class ParameterizedConstructorInSup2 {
    public static void main(String[]args) {
        System.out.println("Parameterized constructor in SuperClass and in SubClass too :");
        Bike b1 = new Bike("Bajaj", "Pulsar", 150f, 1.1f, 1.45f);
        Bike b2 = new Bike("BMW", "G310RR", 310f, 3.15f, 3.99f);
        System.out.println("\nBike 1 details are :");
        b1.showBike();
        System.out.println("\nBike 2 details are :");
        b2.showBike();
        System.out.println("\nThank you.");
    }
}
class Vehicle {
    private String co, mo;
    float pr;
    Vehicle(String co, String mo) {
        this.co = co;
        this.mo = mo;
        System.out.println("Company name and model defined for your vehicle.");
    }
    void showVehicle() {
        System.out.println("Company name : " + co + ", model : " + mo);
    }
} // Total members : 4, Data members : 3, Methods : 1
class Bike extends Vehicle {
    private float cc;
    float pr;
    Bike(String c, String m, float cc, float ex, float or) {
        super(c, m);
        this.cc = cc;
        super.pr = ex;  // SuperClass member
        pr = or;        // Own member
        System.out.println("Specification details defined for your bike.");
    }
    void showBike() {
        showVehicle();
        System.out.println("Engine CC : " + cc);
        System.out.println("Ex-showroom Price (in Lakh) Rs : " + super.pr + "/-");
        System.out.println("On-road Price (in Lakh) Rs : " + pr + "/-");
```

```
    }
} // Total members : 4 + 3 = 7, Data members : 3 + 2 = 5, Methods : 1 + 1 = 2
```

Types of Inheritance:

On the basis of class, there can be three types of inheritance in Java: Single, Multilevel and Hierarchical.
In Java programming, multiple and hybrid inheritance is supported through "interface" only. We will learn about interfaces later.



Single Inheritance:

Java supports single inheritance in which one subclass can have only one superclass. That is a child class can have only one single parent.
The examples that we have seen up till now in this Inheritance chapter, all belong to the type Single Inheritance.

Multilevel Inheritance:

The multilevel inheritance is actually a single inheritance followed by another single inheritance. Here the members of 1 class are inherited (acquired) by the another class from which one more class inherits (acquires).
The multilevel inheritance can be thought of as, a Child class inherits the traits of a Parent class which has already inherited the traits of a GrandParent class. Here the Child class acquires the members of Parent as well as GrandParent class.

113. Program:

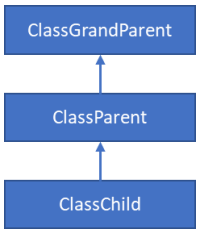
```
class MultilevelInheritance1 {
    static public void main(String[]args) {
        System.out.println("Multilevel Inheritance :\n");
        Cylinder cy = new Cylinder();
        cy.setCylinder(5.0f, 10.0f);
        cy.showCylinder();
        System.out.println("\nThank you.");
    }
}
class Shape {
```

```
    String name;
    float d1, d2;
    void setShape(String n) {
        name = n;
        System.out.println(name + " shape name defined.");
    }
    void showShape() {
        System.out.println("Shape name : " + name);
        System.out.println("Dimension 1 : " + d1 + ", dimension 2 : " + d2);
    }
} // Total members : 5, Data members : 3, Methods : 2
class Circle extends Shape {
    float a;
    void setCircle(String name, float radius) {
        setShape(name);
        d1 = 3.14f;
        d2 = radius;
        System.out.println(name + " shape radius set.");
    }
    void showCircle() {
        a = d1 * d2 * d2;
        showShape();
        System.out.println("Area : " + a);
    }
} // Total members : 5 + 3 = 8, Data members : 3 + 1 = 4, Methods : 2 + 2 = 4
class Cylinder extends Circle {
    float h, v;
    void setCylinder(float radius, float height) {
        setCircle("Cylinder", radius);
        h = height;
        System.out.println(name + " shape height set.");
    }
    void showCylinder() {
        showCircle();
        v = a * h;
        System.out.println("Height : " + h);
        System.out.println("Volume : " + v);
    }
} // Total members : 5 + 3 + 4 = 12, Data members : 3 + 1 + 2 = 6, Methods : 2 + 2 + 2 = 6
```

Constructors in Multilevel Inheritance:

As we already know that the constructors of the class are invoked implicitly while initializing the objects. In inheritance the constructor of superclass is invoked first and then the constructor of subclass.
The same methodology is followed in multilevel inheritance. The constructor of the superclass that is on the very top of the multilevel hierarchy is invoked first, then the constructor of subclass is invoked, and then the constructor of subclass of the subclass is invoked, and so on.

That is, consider classes ClassGradParent, ClassParent and ClassChild. Here ClassGrandParent is at the top of multilevel inheritance, ClassParent is the subclass of ClassGrandParent, and ClassChild is the subclass of ClassParent.



Default Constructor:

While initializing the object of ClassChild, the constructor of ClassGrandParent will be invoked first, then the constructor of ClassParent, and lastly the constructor of ClassChild will be invoked.

114. Program:

```
class MultilevelInheritance2 {
    static public void main(String[]args) {
        System.out.println("Default constructors in Multilevel Inheritance :\n");
        Cylinder cy = new Cylinder();
        System.out.println("\nDefault initial values :");
        cy.showCylinder();
        cy.setCylinder(5.0f, 10.0f);
        System.out.println("\nAfter assigning values :");
        cy.showCylinder();
        System.out.println("\nThank you.");
    }
}
class Shape {
    String name;
    float d1, d2;
    Shape() {
        name = "Not defined" ;
        d1 = d2 = 1.0f;
        System.out.println("Shape defined.");
    }
    void setShape(String n) {
        name = n;
    }
    void showShape() {
        System.out.println("Shape name : " + name);
        System.out.println("Dimension 1 : " + d1 + ", dimension 2 : " + d2);
    }
} // Total members : 5, Data members : 3, Methods : 2
class Circle extends Shape {
    float a;
    Circle() {
        a = 1.0f;
        System.out.println("Circle defined.");
```

```
    }
    void setCircle(String name, float radius) {
        setShape(name);
        d1 = 3.14f;
        d2 = radius;
    }
    void showCircle() {
        a = d1 * d2 * d2;
        showShape();
        System.out.println("Area : " + a);
    }
} // Total members : 5 + 3 = 8, Data members : 3 + 1 = 4, Methods : 2 + 2 = 4
class Cylinder extends Circle {
    float h, v;
    Cylinder() {
        h = v = 1.0f;
        System.out.println("Cylinder defined.");
    }
    void setCylinder(float radius, float height) {
        setCircle("Cylinder", radius);
        h = height;
    }
    void showCylinder() {
        showCircle();
        v = a * h;
        System.out.println("Height : " + h);
        System.out.println("Volume : " + v);
    }
} // Total members : 5 + 3 + 4 = 12, Data members : 3 + 1 + 2 = 6, Methods : 2 + 2 + 2 = 6
```

Parameterized Constructors in Multilevel Inheritance:

As we know, the Java language permits us to define parameterized constructors in inheritance. The same can be used in multilevel inheritance.
In case of implicit or default constructors the Java compiler implicitly invokes the constructors of superclass and then of the subclass.
In the case of a parameterized constructor we have to explicitly invoke and pass parameters to the constructors. To explicitly invoke parameterized constructor of superclass we have to use the **super** keyword, which should be the first statement in the constructor of the subclass.

115. Program:

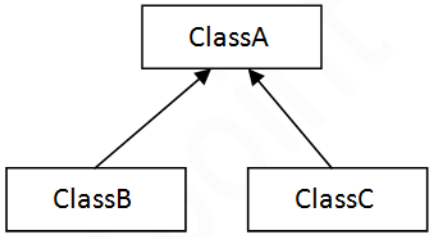
```
class MultilevelInheritance3 {
    static public void main(String[]args) {
        System.out.println("Parameterized constructors in Multilevel Inheritance :\n");
        BoxShipment obj = new BoxShipment(20.5f, 25.4f, 30.2f, 10.6f, 20.0f);
        System.out.println("Box details are :");
        obj.printBox();
        obj.printWeight();
        obj.printShipment();
```

```
        obj.printVolume();
        System.out.println("Thank you.");
    }
}
class Box {
    float w, h, d;
    Box(float width, float height, float depth) {
        w = width;
        h = height;
        d = depth;
        System.out.println("Dimensions to the Box set.");
    }
    void printBox() {
        System.out.printf("Width : %g, Height : %g, and Depth : %g.\n", w, h, d);
    }
} // Total members : 4, Data members : 3, Methods : 1
class BoxWeight extends Box {
    float we;
    BoxWeight(float wi, float he, float de, float wei) {
        super(wi, he, de);
        we = wei;
        System.out.println("Weight of the Box set.");
    }
    void printWeight() {
        System.out.println("Weight : " + we);
    }
} // Total members : 5 + 2 = 7, Data members : 3 + 1 = 4, Methods : 2 + 1 = 3
class BoxShipment extends BoxWeight {
    float c;
    BoxShipment(float wi, float he, float de, float we, float co) {
        super(wi, he, de, we);
        c = co;
        System.out.println("Shipment cost of the Box set.");
    }
    void printVolume() {
        System.out.println("Volume of the Box : " + (w * h * d));
    }
    void printShipment() {
        System.out.println("Shipment cost per kg, Rs : " + c);
        System.out.println("Total Shipment cost, Rs : " + (we * c));
    }
} // Total members : 5 + 2 + 3 = 10, Data members : 3 + 1 + 1 = 5, Methods : 2 + 1 + 2 = 5
```

Hierarchical Inheritance:

Hierarchical inheritance is the third type of inheritance where a superclass has 2 or more subclasses. That is, one parent having more than one child is called hierarchical inheritance.

Hierarchical inheritance can be thought of as 2 single inheritance from 1 same superclass.



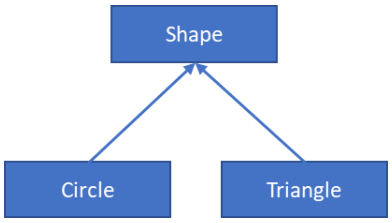
3) Hierarchical

116. Program:

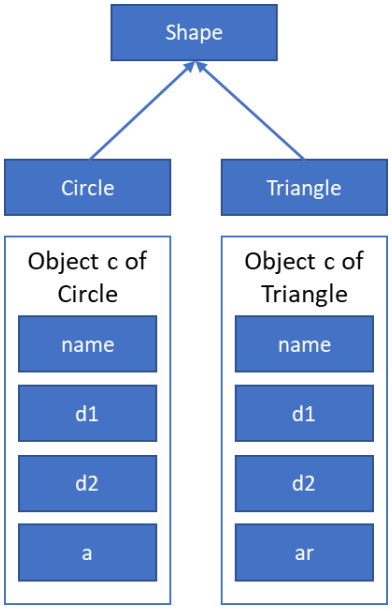
```
class HierarchicalInheritance1 {
    public static void main(String[]args) {
        System.out.println("Hierarchical Inheritance :\n");
        Circle c = new Circle();
        Triangle t = new Triangle();
        System.out.println("Assigning values to both the shapes :\n");
        c.setShape("Circle");
        c.setCircle(5.0f);
        t.setShape("Triangle");
        t.setTriangle(15.0f, 24.5f);
        System.out.println("\nDetails of shapes are :");
        c.showCircle();
        t.showTriangle();
        System.out.println("Thank you");
    }
}
class Shape {
    String name;
    float d1, d2;
    void setShape(String n) {
        name = n;
        System.out.println(name + " shape name defined.");
    }
    void showShape() {
        System.out.println("Shape name : " + name);
        System.out.println("Dimension 1 : " + d1 + ", dimension 2 : " + d2);
    }
} // Total members : 5, Data members : 3, Methods : 2
class Circle extends Shape {
    float a;
    void setCircle(float radius) {
        d1 = 3.14f;
        d2 = radius;
        System.out.println(name + " shape radius set.");
    }
    void showCircle() {
```

```
        a = d1 * d2 * d2;
        showShape();
        System.out.println("Area : " + a);
    }
} // Total members : 5 + 3 = 8, Data members : 3 + 1 = 4, Methods : 2 + 2 = 4
class Triangle extends Shape {
    float a;
    void setTriangle(float base, float height) {
        d1 = base;
        d2 = height;
        System.out.println(name + " shape base and height set.");
    }
    void showTriangle() {
        a = 0.5f * d1 * d2;
        showShape();
        System.out.println("Area : " + a);
    }
} // Total members : 5 + 3 = 8, Data members : 3 + 1 = 4, Methods : 2 + 2 = 4
```

Representation of above Hierarchical Inheritance:



Memory Allocation:



Constructors in Hierarchical Inheritance:

Constructors, as we already know, work the same way in hierarchical inheritance, as they work in any other type of inheritance. That is, the constructor of the base class will be invoked first and then the constructor of the derived class will be invoked, upon initialization of the derived class object.

Default Constructor:

Hierarchical inheritance is just like 2 single inheritance from 1 same superclass. So when we define a default constructor in a subclass the implicit constructor of superclass will be invoked and then the default constructor of subclass.
When we define a default constructor in superclass as well as in subclass, on initialization of an object of the subclass the constructor of superclass is invoked first and then the constructor of subclass.

117. Program:

```
class HierarchicalInheritance2 {
    public static void main(String[]args) {
        System.out.println("Default constructors in Hierarchical Inheritance :\n");
        Circle c = new Circle();
        Triangle t = new Triangle();
        System.out.println("\nDefault initial values of shape circle are :");
        c.showCircle();
        System.out.println("\nDefault initial values of shape triangle are :");
        t.showTriangle();
        System.out.println("Thank you");
    }
}
class Shape {
    String name;
    float d1, d2;
    Shape() {
        name = "Not defined";
        System.out.println("Shape defined.");
    }
    void showShape() {
        System.out.println("Shape name : " + name);
        System.out.println("Dimension 1 : " + d1 + ", dimension 2 : " + d2);
    }
} // Total members : 4, Data members : 3, Methods : 1
class Circle extends Shape {
    float a;
    Circle() {
        d1 = 3.14f;
        d2 = 0.0f;
        System.out.println("Circle defined.");
    }
    void showCircle() {
        a = d1 * d2 * d2;
        showShape();
        System.out.println("Area : " + a);
```

```
    }
} // Total members : 4 + 2 = 6, Data members : 3 + 1 = 4, Methods : 1 + 1 = 2
class Triangle extends Shape {
    float ar;
    Triangle() {
        d1 = 0.0f;
        d2 = 0.0f;
        System.out.println("Triangle defined.");
    }
    void showTriangle() {
        ar = 0.5f * d1 * d2;
        showShape();
        System.out.println("Area : " + ar);
    }
} // Total members : 4 + 2 = 6, Data members : 3 + 1 = 4, Methods : 1 + 1 = 2
```

Parameterized Constructors in Superclass and subclass:

When we define a parameterized constructor in a subclass, we need to explicitly invoke and pass the parameters while initializing the object of the subclass.
To invoke the parameterized constructor of superclass from subclass, we need to use the **super** keyword. A subclass calls the constructor defined by its superclass using the **super** keyword and by passing the parameters.

118. Program:

```
class HierarchicalInheritance3 {
    public static void main(String[]args) {
        System.out.println("Parameterized constructors in Hierarchical Inheritance :\n");
        Circle c = new Circle("Circle", 5.0f);
        Triangle t = new Triangle("Triangle", 24.5f, 33.3f);
        System.out.println("\nDetails of shapes are :");
        c.showCircle();
        t.showTriangle();
        System.out.println("Thank you");
    }
}
class Shape {
    String name;
    float d1, d2;
    Shape(String name) {
        this.name = name;
        System.out.println("Shape defined.");
    }
    void showShape() {
        System.out.println("Shape name : " + name);
        System.out.println("Dimension 1 : " + d1 + ", dimension 2 : " + d2);
    }
} // Total members : 4, Data members : 3, Methods : 1
class Circle extends Shape {
```

```
    float a;
    Circle(String name, float r) {
        super(name);
        d1 = 3.14f;
        d2 = r;
        System.out.println("Circle defined.");
    }
    void showCircle() {
        a = d1 * d2 * d2;
        showShape();
        System.out.println("Area : " + a);
    }
} // Total members : 4 + 2 = 6, Data members : 3 + 1 = 4, Methods : 1 + 1 = 2
class Triangle extends Shape {
    float ar;
    Triangle(String name, float base, float height) {
        super(name);
        d1 = base;
        d2 = height;
        System.out.println("Triangle defined.");
    }
    void showTriangle() {
        ar = 0.5f * d1 * d2;
        showShape();
        System.out.println("Area : " + ar);
    }
} // Total members : 4 + 2 = 6, Data members : 3 + 1 = 4, Methods : 1 + 1 = 2
```

Runtime Polymorphism:

Overriding of Methods:

In a class hierarchy (inheritance), when a method in a subclass has the same name and type signature (list of parameters) as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden.

Accessing superclass methods using super keyword:

The hidden method of superclass can be called from within the subclass using the **super** keyword.

Syntax:

```
super.superClassMethod([args]);
```

119. Program:

```
class MethodOverriding1 {
    public static void main(String[]args) {
```

```
        System.out.println("Overriding of method.\n");
        Product obj1=new Product("Asus", "ROG 17", "Laptop", "AMD Ryzen 7", 8, 512f, 115000f);
        Product obj2=new Product("Dell", "Vostro 35", "Desktop", "AMD A6", 8, 1024f, 35000f);
        System.out.println("\nProduct 1 details :");
        obj1.showData();    // Subclass method overrides superclass method
        System.out.println("\nProduct 2 details :");
        obj2.showData();    // Subclass method overrides superclass method
        System.out.println("\nThank you.");
    }
}
class Company {
    private String name, model;
    String type;
    Company(String name, String model) {
        this.name = name;
        this.model = model;
        System.out.println("Company name and model defined.");
    }
    void showData() {
        System.out.println("Company name : " + name);
        System.out.println("Model : " + model);
        System.out.println("Product type : " + type);
    }
} // Total members : 4, Data members : 3, Methods : 1
class Product extends Company {
    private String processor;
    private int ram;
    private float rom, mrp;
    Product(String n, String m, String type, String pro, int ram, float rom, float mrp) {
        super(n, m);
        this.type = type;
        processor = pro;
        this.ram = ram;
        this.rom = rom;
        this.mrp = mrp;
        System.out.println("Product specification defined.");
    }
    void showData() {
        super.showData();   // Invoking the overridden method of the superclass
        System.out.println("Processor : " + processor);
        System.out.println("RAM (in GB) : " + ram);
        System.out.println("ROM (in GB) : " + rom);
        System.out.println("MRP Rs : " + mrp + "/-");
    }
} // Total members : 4 + 5 = 9, Data members : 3 + 4 = 7, Methods : 1 + 1 = 2
```

Dynamic Method Dispatch:

Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

As we know, a superclass reference variable (object) can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how.
When an overridden method is called through a superclass reference (object), Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.
In other words, it is the type of the object being referred to (not the type of the reference variable/object) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

120. Program:

```
class RuntimePolymorphism1 {
    public static void main(String[]args) {
        System.out.println("Runtime Polymorphism. Dynamic Method Dispatch.\n");
        Company objCo=new Company("Not defined", "NA");
        Product objPr=new Product("Asus", "ROG 17", "Laptop", "AMD Ryzen 7", 8, 512f, 115000f);
        System.out.println("\nCompany details, using object of its own (Company) class :");
        objCo.showData();
        System.out.println("\nProduct details, using object of its own (Product) class :");
        objPr.showData();   // Subclass method overrides superclass method

        Company obj;
        obj = objCo;
        System.out.println("\nSuperclass object referring to its own initialized object:");
        obj.showData();    // Dispatches superclass method. Dynamic Method Dispatch
        obj = objPr;
        System.out.println("\nSuperclass object referring to its subclass object:");
        obj.showData();    // Dispatches subclass method. Dynamic Method Dispatch
        System.out.println("\nThank you.");
    }
}
class Company {
    private String name, model;
    String type;
    Company(String name, String model) {
        this.name = name;
        this.model = model;
        System.out.println("Company name and model defined.");
    }
    void showData() {
        System.out.println("Company name : " + name);
        System.out.println("Model : " + model);
        System.out.println("Product type : " + type);
```

```
    }
} // Total members : 4, Data members : 3, Methods : 1
class Product extends Company {
    private String processor;
    private int ram;
    private float rom, mrp;
    Product(String name, String model, String type, String pro, int ram, float rom, float mrp) {
        super(name, model);
        this.type = type;
        processor = pro;
        this.ram = ram;
        this.rom = rom;
        this.mrp = mrp;
        System.out.println("Product specification defined.");
    }
    void showData() {
        System.out.println("Processor : " + processor);
        System.out.println("RAM (in GB) : " + ram);
        System.out.println("ROM (in GB) : " + rom);
        System.out.println("MRP Rs : " + mrp + "/-");
    }
} // Total members : 4 + 5 = 9, Data members : 3 + 4 = 7, Methods : 1 + 1 = 2
```

Runtime Polymorphism of Data Members:

When data members override, the reference of the subclass object to the superclass object will not be used to refer to the overridden data members. Thus, even if we assign the object of subclass to superclass, the superclass' object will always refer to its own data member, irrespective of the reference.

The abstract Class:

There are situations in which we will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes we will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
In this case, we want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the **abstract** method.
We require that certain methods be overridden by subclasses by specifying the **abstract** type modifier.
These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

General form:

```
abstract class SuperClass {
	...
	abstract returnType methodName([listOfParameters]);
	...
}
```

As we can see, the **abstract** method, "**methodName**", has no body defined.

Any class that contains one or more **abstract** methods must also be declared **abstract**. To declare a class **abstract**, we simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration, as shown above. There can be no objects of an **abstract** class. That is, an **abstract** class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an **abstract** class is not fully defined. Also, we cannot declare **abstract** constructors, or **abstract static** methods. Any subclass of an **abstract** class must either implement all of the **abstract** methods from the superclass, or be declared **abstract** itself.

121. Program:

```
class AbstractMethod1 {
    static public void main(String[]args) {
        System.out.println("Abstract method :");
        // Shape obj = new Shape(); // ERROR : Shape is abstract; cannot be instantiated
        Rectangle obj = new Rectangle();
        obj.setData("Rectangle", 10.5f, 12.0f);
        obj.showRectangle();
        obj.area();
        System.out.println("Thank you.");
    }
}
abstract class Shape {
    float d1, d2;
    String name;
    void setData(String name, float dim1, float dim2) {
        this.name = name;
        d1 = dim1;
        d2 = dim2;
    }
    abstract void area();
} // Total members : 4, Data members : 3, Methods : 1
```

```
class Rectangle extends Shape {
    void area() {
        System.out.println("Area : " + (d1 * d2));
    }
    void showRectangle() {
        System.out.println(name + " has length " + d1 + ", and width " + d2);
    }
} // Total members : 4 + 1 = 5, Data members : 3 + 0 = 3, Methods : 1 + 1 = 2
```

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

122. Program:

```
class AbstractMethod2 {
    static public void main(String[]args) {
        System.out.println("Abstract Methods :");
        Vehicle objB = new Bike();
        Vehicle objC = new Car();
        objC.setDetails();
        // objC.setCar();
        objB.setDetails();
        // objB.setBike();
        objC.showDetails();
        objB.showDetails();
        System.out.println("\nThank you.");
    }
}
abstract class Vehicle {
    String co, mo, type;
    void setDetails() {
        System.out.println("\nEnter " + type + " details :");
        System.out.print("Company name : ");
        co = System.console().readLine();
        System.out.print("Model : ");
        mo = System.console().readLine();
    }
    abstract void showDetails();
} // Total members : 5, Data members : 3, Methods : 2
class Bike extends Vehicle {
    int ecc;
    float price;
    Bike() {
        type = "Bike";
    }
    void setBike() {
        System.out.print("Engine CC : ");
        ecc = Integer.parseInt(System.console().readLine());
```

```
        System.out.print("Price, Rs : ");
        price = Float.parseFloat(System.console().readLine());
    }
    void showDetails() {
        System.out.println("\n" + type + " details are :");
        System.out.println("Company name : " + co + ", Model : " + mo);
        System.out.println("Engine CC : " + ecc + ", Price Rs : " + price);
    }
} // Total members : 5 + 3 = 8, Data members : 3 + 2 = 5, Methods : 2 + 1 = 3
class Car extends Vehicle {
    int ecc;
    float price;
    int seats;
    Car() {
        type = "Car";
    }
    void setCar() {
        System.out.print("Engine CC : ");
        ecc = Integer.parseInt(System.console().readLine());
        System.out.print("Seating Capacity : ");
        seats = Integer.parseInt(System.console().readLine());
        System.out.print("Price, Rs : ");
        price = Float.parseFloat(System.console().readLine());
    }
    void showDetails() {
        System.out.println("\n" + type + " details are :");
        System.out.println("Company name : " + co + ", Model : " + mo);
        System.out.println("Engine CC : " + ecc + ", Seating Capacity : " + seats);
        System.out.println("Price Rs : " + price);
    }
} // Total members : 5 + 4 = 9, Data members : 3 + 3 = 6, Methods : 2 + 1 = 3
```

The final Keyword in Inheritance:

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the previous chapters.
The other two uses of **final** apply to inheritance. Both are examined here.

Second use of the **final** keyword is to prevent overriding of methods.

Using final to Prevent Overriding:

While method overriding is one of Java's most powerful features, there will be times when we want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration.
Thus, then the methods that are declared as **final** cannot be overridden.

123. Program:

```
class FinalMethod1 {
```

```
    static public void main(String[]args) {
        System.out.println("The final Method :");
        Bike objB = new Bike();
        objB.setDetails();
        objB.setBike();
        objB.showDetails();
        System.out.println("\nThank you.");
    }
}
class Vehicle {
    private String co, mo;
    String type;
    void setDetails() {
        System.out.println("\nEnter " + type + " details :");
        System.out.print("Company name : ");
        co = System.console().readLine();
        System.out.print("Model : ");
        mo = System.console().readLine();
    }
    final void showDetails() {
        System.out.println("Company name : " + co + ", Model : " + mo);
    }
} // Total members : 5, Data members : 3, Methods : 2
class Bike extends Vehicle {
    int ecc;
    float price;
    Bike() {
        type = "Bike";
    }
    void setBike() {
        System.out.print("Engine CC : ");
        ecc = Integer.parseInt(System.console().readLine());
        System.out.print("Price, Rs : ");
        price = Float.parseFloat(System.console().readLine());
    }
    void showDetails() {
        System.out.println("\n" + type + " details are :");
        System.out.println("Engine CC : " + ecc + ", Price Rs : " + price);
    }
} // Total members : 5 + 3 = 9, Data members : 3 + 2 = 5, Methods : 2 + 1 = 3
```

On compiling the above program, the Java JDK throws the following ERROR:

```
E:\Java> java FinalMethod1.java
FinalMethod1.java:37: error: showDetails() in Bike cannot override showDetails() in Vehicle
    void showDetails() {
         ^
  overridden method is final
1 error
```

```
error: compilation failed
E:\Java>
```

Methods declared as **final** can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass. When a small **final** method is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is an option only with **final** methods. Normally, Java resolves calls to methods dynamically, at run time, which is called late binding. However, since **final** methods cannot be overridden, a call to one can be resolved at compile-time. This is called early binding.

The above program can be re-written, without overriding the **final** method, as shown below, so that it will compile and run without any error.

124. Program:

```
class FinalMethod2 {
    static public void main(String[]args) {
        System.out.println("The final Method :");
        Bike objB = new Bike();
        objB.setDetails();
        objB.setBike();
        objB.showBike();
        System.out.println("\nThank you.");
    }
}
class Vehicle {
    private String co, mo;
    String type;
    void setDetails() {
        System.out.println("\nEnter " + type + " details :");
        System.out.print("Company name : ");
        co = System.console().readLine();
        System.out.print("Model : ");
        mo = System.console().readLine();
    }
    final void showDetails() {
        System.out.println("Company name : " + co + ", Model : " + mo);
    }
} // Total members : 5, Data members : 3, Methods : 2
class Bike extends Vehicle {
    int ecc;
    float price;
    Bike() {
        type = "Bike";
    }
    void setBike() {
        System.out.print("Engine CC : ");
        ecc = Integer.parseInt(System.console().readLine());
        System.out.print("Price, Rs : ");
        price = Float.parseFloat(System.console().readLine());
```

```
    }
    // ERROR : showDetails() in Bike cannot override showDetails() in Vehicle
    // void showDetails() {
    void showBike() {
        System.out.println("\n" + type + " details are :");
        showDetails();
        System.out.println("Engine CC : " + ecc + ", Price Rs : " + price);
    }
} // Total members : 5 + 4 = 9, Data members : 3 + 2 = 5, Methods : 2 + 2 = 4
```

Note that a method can never be **final** as well as **abstract**. This is because an **abstract** method completely relies on overriding, and the **final** is completely against the concept of overriding.

Third use of the **final** keyword is to prevent inheritance.

Using final to Prevent Inheritance:

Sometimes we want to prevent a class from being inherited. To do this, precede the class declaration with the **final** keyword. Declaring a class as **final** implicitly declares all of its methods as **final**, too.

125. Program:

```
class FinalClass1 {
    public static void main(String[]args) {
        System.out.println("The final class.");
        Square obj = new Square();
        obj.setSquare(5.6f);
        obj.showSquare();
        obj.setRectangle(5.6f, 8.1f);
        obj.showRectangle();
        System.out.println("\nThank you.\n");
    }
}
final class Rectangle {
    float l, b;
    void setRectangle(float len, float bre) {
        l = len;
        b = bre;
    }
    void showRectangle() {
        System.out.println("Rectangle details are :");
        System.out.println("Length : " + l + ", breadth : " + b + ", area : " + (l * b));
    }
} // Total members : 4, Data members : 2, Methods : 2
class Square extends Rectangle {    // ERROR : cannot inherit from final Rectangle
    float area;
    void setSquare(float side) {
        l = side;
    }
    void showSquare() {
```

```
        area = l * l;
        System.out.println("Square details are :");
        System.out.println("Side : " + l + ", area : " + area);
    }
} // Total members : 3, Data members : 1, Methods : 2
```

CMD Output:

```
E:\Java> java FinalClass1.java
FinalClass1.java:23: error: cannot inherit from final Rectangle
class Square extends Rectangle {
                     ^
1 error
error: compilation failed
E:\Java>
```

The above program can be re-written, without extending (inheriting) the **final** class, as shown below, so that it will compile and run without any error.

126. Program:

```
class FinalClass1 {
    public static void main(String[]args) {
        System.out.println("The final class.");
        Square objS = new Square();
        objS.setSquare(5.6f);
        objS.showSquare();
        Rectangle objR = new Rectangle();
        objR.setRectangle(5.6f, 8.1f);
        objR.showRectangle();
        System.out.println("\nThank you.\n");
    }
}
final class Rectangle {
    float l, b;
    void setRectangle(float len, float bre) {
        l = len;
        b = bre;
    }
    void showRectangle() {
        System.out.println("Rectangle details are :");
        System.out.println("Length : " + l + ", breadth : " + b + ", area : " + (l * b));
    }
} // Total members : 4, Data members : 2, Methods : 2
class Square {
    float side, area;
    void setSquare(float side) {
        this.side = side;
    }
    void showSquare() {
```

```
        area = side * side;
        System.out.println("Square details are :");
        System.out.println("Side : " + side + ", area : " + area);
    }
} // Total members : 4, Data members : 2, Methods : 2
```

As we might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations, and the **final** prevents inheritance.

XV - Packages:

Java provides a mechanism for partitioning the class namespace into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. We can define classes inside a package that are not accessible by code outside that package. We can also define class members that are exposed only to other members of the same package. This allows our classes to have internal knowledge of each other, but not expose that knowledge to the rest of the world.
Packages are containers for classes. They are used to keep the class namespace compartmentalized.

Defining a Package:

Creating a package is quite easy: simply include a **package** command (with the **package** keyword and package name) as the "first statement in a Java source file". Any classes declared within that file will belong to the specified **package**. The **package** statement defines a namespace in which classes are stored.

If we omit the **package** statement, the class names are put into the **default package**, which has no name. This is why we haven't had to worry about packages before now. While the **default package** is fine for short, simple programs, it is inadequate for real applications. Most of the time, we will define a **package** for our code.

Syntax:

```
package packageName;
...
```

127. Program:

```
package myPackage;
class PackageDemo1 {
    public static void main(String[]args) {
        System.out.println("The package class.");
        Rectangle obj = new Rectangle();
        obj.setRectangle(5.6f, 8.1f);
        obj.showRectangle();
        System.out.println("\nThank you.\n");
    }
}
class Rectangle {
    float l, b;
    void setRectangle(float len, float bre) {
        l = len;
        b = bre;
    }
    void showRectangle() {
        System.out.println("Rectangle details are :");
        System.out.println("Length : " + l + ", breadth : " + b + ", area : " + (l * b));
    }
} // Total members : 4, Data members : 2, Methods : 2
```

Java uses file system directories to store packages. For example, the **.class** (byte code) files for the classes that we declare to be part of the **package** must be stored in a directory named as the "**packageName**". Remember that the classes are significant, and the directory name must match the **package** name exactly.

Save the above Java program anywhere in your computer/laptop. Suppose that we have stored this program in "**E:\Java**", with the name "**PackageDemo1.java**".

To compile the program, we first have to browse to the path where our Java source code file is stored, and then compile it, using Command Prompt (CMD).

Command to compile the Java's Package Program:

```
E:\Java> javac PackageDemo1.java
E:\Java>
```

Once the program is successfully compiled we have to place all the **class files** of the program inside the package name's directory, that is, for the above program save (copy) all the **.class** files in the **myPackage** directory.

To execute the program, we need to be in the **directory above** the **myPackage** directory, and then execute the program. The command for the execution of Java Program now must include the **package name** and the **class name**, in which the **main( )** method resides.

Command to execute the PackageDemo1.java program:

```
E:\Java> java myPackage.PackageDemo1
The package class.
Rectangle details are :
Length : 5.6, breadth : 8.1, area : 45.36

Thank you.

E:\Java>
```

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file(s) belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

Multiple Java Files but Same Package:

Using packages we can declare and define multiple classes, placed within multiple, physically different Java source files into a single package. When we place such multiple classes into a single package we are then able to use all these classes, and their members, all together as if they all belong to a single program file.

128. Program:
Here we are going to define 5 different classes belonging to 1 same package, but defined in 5 different Java files. Even if there are 5 different physical Java files, they all belong to 1 single same program, just because of the same package name.
All the 5 Java files will be stored on the "**E:\Java**" locations, inside the "**ArithmeticOperations**" directory, that is at location : "**E:\Java\ArithmeticOperations**".

File 1: PackageDemoAddition.java:

```
package ArithmeticOperations;
class Addition {
    int add(int a, int b) {
        return (a + b);
    }
    double add(double a, double b){
```

```
        return (a + b);
    }
    String add(String a, String b) {
        return (a + b);
    }
}
```

File 2: PackageDemoSubtraction.java:

```
package ArithmeticOperations;
class Subtraction {
    int sub(int a, int b) {
        return (a - b);
    }
    double sub(double a, double b){
        return (a - b);
    }
}
```

File 3: PackageDemoMultiplication.java:

```
package ArithmeticOperations;
class Multiplication {
    double mul(double var1, double var2) {
        return var1 * var2;
    }
}
```

File 4: PackageDemoDivision.java:

```
package ArithmeticOperations;
class Division {
    int div(int no1, int no2) {
        return (no1 / no2);
    }
    double div(double no1, double no2) {
        return (no1 / no2);
    }
    int mod(int a, int b) {
        return (a % b);
    }
}
```

File 5: PackageDemo.java:

```
package ArithmeticOperations;
class PackageDemo {
    public static void main(String[]args) {
        Addition objA = new Addition();
        Subtraction objS = new Subtraction();
        Multiplication objM = new Multiplication();
        Division objD = new Division();
        System.out.println("Arithmetic operations on integer values:");
```

```
        System.out.println("Addition of 5, 6 : " + objA.add(5, 6));
        System.out.println("Subtraction of 5, 6 : " + objS.sub(5, 6));
        System.out.println("Multiplication of 5, 6 : " + objM.mul(5, 6));
        System.out.println("Division of 5, 6 : " + objD.div(5, 6));
        System.out.println("Reminder of division of 5, 6 : " + objD.mod(5, 6));
        System.out.println("\nArithmetic operations on floating-point values:");
        System.out.println("Addition of 25.5, 5.5 : " + objA.add(25.5, 5.5));
        System.out.println("Subtraction of 50.4, 6.8 : " + objS.sub(50.4, 6.8));
        System.out.println("Multiplication of 5.2, 2.5 : " + objM.mul(5.2, 2.5));
        System.out.println("Division of 2.5, 1.5 : " + objD.div(2.5, 1.5));
        System.out.println("\nAddition (Concatenation) operations on String values:");
        System.out.println("Addition of \"Java\" & \"Packages\" : " + objA.add("Java",
"Packages"));
    }
}
```

Compilation of package program:

To compile the above written program (5 files), we need to compile all the files at once, to avoid getting errors. Use a single **javac** command to compile all the files belonging to 1 package.

Syntax:

```
javac file1.java file2.java file3.java ...
```

Example:

```
C:\Users\Sourabha>E:
E:\>cd "Java"
E:\Java>cd ArithmeticOperations
E:\Java\ArithmeticOperations> javac PackageDemo.java PackageDemoAddition.java
PackageDemoDivision.java PackageDemoMultiplication.java PackageDemoSubtraction.java
E:\Java\ArithmeticOperations>
```

Now to execute this program use the **java** command and the package name followed by the dot operator and main( ) function's class name.
Remember to put all the class files that got created at the time of compilation of the program into a directory named as package name.
That is, for the above program, create a new directory with name "**ArithmeticOperations**", and put all the class (byte code) files into the newly created "**ArithmeticOperations**" directory.
As we have already created the "**ArithmeticOperations**" directory and placed our source code files (all 5 Java files) in that "**ArithmeticOperations**" directory, we need not have to place the class files in this "**ArithmeticOperations**" directory, because those are already present in that "**ArithmeticOperations**" directory. And that is why we first navigated (through the **cd ArithmeticOperations** command on CMD) to that "**ArithmeticOperations**" directory and then compiled the program (all 5 Java files).

Now using the command prompt and from outside the package named directory run the program.

Syntax:

```
java packageName.MainFunctionClassName
```

Example:

```
E:\Java\ArithmeticOperations> cd..
E:\Java> java ArithmeticOperations.PackageDemo
Arithmetic operations on integer values:
Addition of 5, 6 : 11
Subtraction of 5, 6 : -1
Multiplication of 5, 6 : 30.0
Division of 5, 6 : 0
Reminder of division of 5, 6 : 5

Arithmetic operations on floating-point values:
Addition of 25.5, 5.5 : 31.0
Subtraction of 50.4, 6.8 : 43.6
Multiplication of 5.2, 2.5 : 13.0
Division of 2.5, 1.5 : 1.6666666666666667

Addition (Concatenation) operations on String values:
Addition of "Java" & "Packages" : JavaPackages
E:\Java>
```

Hierarchy of Packages:

We can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period/dot (.) operator.

Syntax:

```
package packageName1.packageName2. ...;
```

A package hierarchy must be reflected in the file system of our Java development system.

For example:
A package declared as:

```
package operations.arithmeticOperations.additive;
```

The above declared package needs to be stored in the **operations\arithmeticOperations\additive** directory in a Windows environment. Be sure to choose our package names carefully. We cannot rename a package without renaming the directory in which the classes (class files) are stored.

129. Program:

Consider creating 3 program files, in the **E:\Java\operations\arithmeticOperations\additive** directory.

File 1 : Addition.java:

```
package operations.arithmeticOperations.additive;
class Addition {
    int add(int a, int b) {
        return (a + b);
    }
    double add(double a, double b) {
        return (a + b);
```

```
    }
    String add(String a, String b) {
        return (a + b);
    }
}
```

File 2: Subtraction.java

```
package operations.arithmeticOperations.additive;
class Subtraction {
    int sub(int a, int b) {
        return (a - b);
    }
    double sub(double no1, double no2) {
        return (no1 - no2);
    }
}
```

File 3: PackageHierarchy1.java

```
package operations.arithmeticOperations.additive;
class PackageHierarchy1 {
    public static void main(String[]args) {
        System.out.println("The package hierarchy.");
        Addition objA = new Addition();
        Subtraction objS = new Subtraction();
        System.out.println("Arithmetic operations on integer values:");
        System.out.println("Addition of 5, 6 = " + objA.add(5, 6));
        System.out.println("Subtraction of 5, 6 = " + objS.sub(5, 6));
        System.out.println("\nArithmetic operations on floating-point values:");
        System.out.println("Addition of 25.5, 5.5 = " + objA.add(25.5, 5.5));
        System.out.println("Subtraction of 50.4, 6.8 = " + objS.sub(50.4, 6.8));
        System.out.println("\nAddition (Concatenation) operations on String values:");
        System.out.println("Addition of \"Java\" & \"Package Hierarchy\" = " + objA.add("Java",
"Package Hierarchy"));
        System.out.println("Thank you.");
    }
}
```

Save these program files (java source code files) anywhere, with any name.
For instance we store these files in **E:\Java\operations\arithmeticOperations\additive** with the names **Addition.java, Subtraction.java and PackageHierarchy1.java**, respectively. Now compile this program using **CMD**.

```
C:\Users\Sourabha> E:
E:\> cd "Java"
E:\Java> cd .\operations\
E:\Java\operations> cd .\arithmeticOperations\
E:\Java\operations\arithmeticOperations> cd .\additive\
E:\Java\operations\arithmeticOperations\additive> javac .\Addition.java .\Subtraction.java
.\PackageHierarchy1.java
E:\Java\operations\arithmeticOperations\additive>
```

Once the program is successfully compiled, class (object code/byte code) files will be created for each and every class that we declare and define in our program. As our program has 3 classes, first **Addition**, second **Subtraction**, and the third one **PackageHierarchy1**, the compiler will create 3 class files, with name **Addition.class**, **Subtraction.class**, and **PackageHierarchy1.class** on successful compilation.

Now place these newly created class files in the directory hierarchy that should be created by following the package hierarchy that we have declared in this program. And then copy/cut paste the class files in the package hierarchy directory.
That is, the package hierarchy is **operations.arithmeticOperations.additive**, thus create these directory and place the class files in them, as:

```
E:\Java\operations\arithmeticOperations\additive\
```

As we have already created these directories hierarchy and placed the Java (source code) files in them, no need to create these directories once again. And also as the program has been compiled from within these directories, no need to again copy/cut paste the class files in the directories hierarchy.

Now using CMD browse to the directory above the package hierarchy directories where the class files have been kept and run the program.

```
E:\Java\operations\arithmeticOperations\additive>
E:\Java\operations\arithmeticOperations\additive> cd..
E:\Java\operations\arithmeticOperations> cd..
E:\Java\operations> cd..
E:\Java> java operations.arithmeticOperations.additive.PackageHierarchy1
Arithmetic operations on integer values:
Addition of 5, 6 = 11
Subtraction of 5, 6 = -1

Arithmetic operations on floating-point values:
Addition of 25.5, 5.5 = 31.0
Subtraction of 50.4, 6.8 = 43.6

Addition (Concatenation) operations on String values:
Addition of "Java" & "Package Hierarchy" = JavaPackage Hierarchy
E:\Java>
```

Compilation and Execution at Once:

The new versions of Java jdk directly compile as well as execute the Java programs using the "**java**" command. In case of Package programming, we need not have to worry about the hierarchy of directories to be created to place the class file(s) of the package program.

In the below given program we shall create only a single physical file, named **CompilationAndExecution1.java** and write the following Java code in it.

130. Program:

```
package operations.arithmeticOperations;
class CompilationAndExecution1 {
```

```
    public static void main(String[]args) {
        System.out.println("Compilation and Execution at Once.");
        Addition objA = new Addition();
        Subtraction objS = new Subtraction();
        Multiplication objM = new Multiplication();
        Division objD = new Division();
        System.out.println("Arithmetic operations on integer values:");
        System.out.println("Addition of 5, 6 = " + objA.add(5, 6));
        System.out.println("Subtraction of 5, 6 = " + objS.sub(5, 6));
        System.out.println("Multiplication of 5, 6 = " + objM.mul(5, 6));
        System.out.println("Division of 5, 6 = " + objD.div(5, 6));
        System.out.println("Reminder of division of 5, 6 = " + objD.mod(5, 6));
        System.out.println("\nArithmetic operations on floating-point values:");
        System.out.println("Addition of 25.5, 5.5 = " + objA.add(25.5, 5.5));
        System.out.println("Subtraction of 50.4, 6.8 = " + objS.sub(50.4, 6.8));
        System.out.println("Multiplication of 5.2, 2.5 = " + objM.mul(5.2, 2.5));
        System.out.println("Division of 2.5, 1.5 = " + objD.div(2.5, 1.5));
        System.out.println("\nAddition (Concatenation) operations on String values:");
        System.out.println("Addition of \"Java\" & \"Packages\" = " + objA.add("Java",
"Packages"));
        System.out.println("Thank you.");
    }
}
class Addition {
    int add(int a, int b) {
        return (a + b);
    }
    double add(double a, double b) {
        return (a + b);
    }
    String add(String a, String b) {
        return (a + b);
    }
}
class Subtraction {
    int sub(int a, int b) {
        return (a - b);
    }
    double sub(double no1, double no2) {
        return (no1 - no2);
    }
}
class Multiplication {
    double mul(double var1, double var2) {
        return var1 * var2;
    }
}
class Division {
```

```
    int div(int no1, int no2) {
        return (no1 / no2);
    }
    double div(double no1, double no2) {
        return (no1 / no2);
    }
    int mod(int a, int b) {
        return (a % b);
    }
}
```

Save this file anywhere in our computer/laptop and compile as well as run it using the single **java** command.

```
E:\Java> java CompilationAndExecution1.java
Compilation and Execution at Once.
Arithmetic operations on integer values:
Addition of 5, 6 = 11
Subtraction of 5, 6 = -1
Multiplication of 5, 6 = 30.0
Division of 5, 6 = 0
Reminder of division of 5, 6 = 5

Arithmetic operations on floating-point values:
Addition of 25.5, 5.5 = 31.0
Subtraction of 50.4, 6.8 = 43.6
Multiplication of 5.2, 2.5 = 13.0
Division of 2.5, 1.5 = 1.6666666666666667

Addition (Concatenation) operations on String values:
Addition of "Java" & "Packages" = JavaPackages
Thank you.
E:\Java>
```

As we can see, here we have not created any of the directory for the packages of this program. This is possible only because we have designed and defined the entire program in a single physical file and compiled and run it using the new single **java** command.

Note that, this new single **java** command to compile as well as run the program works without the directory structure of the packages, only in case of a single physical Java source code file.

Although the use of the "**java**" command makes it easy to compile and execute Java programs, it is rarely used. As we have to distribute (share) only the byte code to the clients we have to follow the traditional (previous) method to compile and then execute the program separately.

Importing Packages:

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must

be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class we want to use.

For example, the following statement represents how a class can be used which is present within a package.

Example:

```
java.util.Scanner objScan = new java.util.Scanner(System.in);
```

As shown above, to use any class that belongs to a package, we have to use the fully qualified name, which is a tedious job.

For this reason, Java provides the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If we are going to refer to a few dozen classes in our application, however, the **package** and **import** statement will save a lot of work/typing.
In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.

Syntax:

```
package packageName...
import package1[.package2. ...].(ClassName|*);
import package3[.package4. ...].(ClassName|*);
class ClassName {
...
}
```

Here, *package1* is the name of the top-level package, and *package2* is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, we specify either an explicit *ClassName* or a star (*), which indicates that the Java compiler should import the entire package.

131. Program:

```
import java.util.Scanner;
class ImportingPackage1 {
    public static void main(String[]args) {
        System.out.println("Importing package.");
        String name;
        System.out.print("Enter your name :");
        // java.util.Scanner objScan = new java.util.Scanner(System.in);
        Scanner objScan = new Scanner(System.in);
        name = objScan.nextLine();
        System.out.println("Hello " + name);
        System.out.println("Thank you.");
    }
}
```

Defining Multiple Packages:

Java allows us to define any number of packages in a single Java program. By defining multiple packages, we can compartmentalize our classes in different compartments (packages).

To define different packages, we need to define different Java source code files, each having the first statement as the **package** statement along with the package name.
Once multiple packages are defined, we can import them in each other with the help of the **import** statement.

General form:
File 1: (for package 1):

```
package packageName1[.name2. ...];
[import packageName2. ...;]
class ClassName {
	...
}
```

File 2: (for package 2):

```
package packageName2[.name3. ...];
[import packageName1. ...;]
class ClassName {
	...
}
```

132. Program:
File 1: E:\Java\Circle.java

```
package figures.shapes;
public class Circle {
    float r;
    final float pi = 3.14f;
    public Circle() {
        r = 1.0f;
        System.out.println("Default Circle defined.");
    }
    public void setRadius(float radius) {
        r = radius;
    }
    public void showArea() {
        float a = pi * r * r;
        System.out.println("Area of Circle with radius " + r + " is " + a);
    }
}
```

File 2: E:\Java\MultiplePackages1.java

```
import figures.shapes.Circle;
class MultiplePackages1 {
    public static void main(String[]args) {
        System.out.println("Defining multiple packages.\n");
        Circle objC = new Circle();
        System.out.println("Default values of Circle :");
        objC.showArea();
        // objC.r; // ERROR : r is not public in Circle; cannot be accessed from outside package
        objC.setRadius(5.0f);
        System.out.println("\nAfter setting new radius value to Circle :");
        objC.showArea();
        System.out.println("\nThank you.");
```

```
    }
}
```

Save these program files (Java source code files) anywhere, with any names (public class name). For instance we store these files on the location **E:\Java** with the name **Circle.java**, and **MultiplePackages1.java**. Observe that we have given the name of the class to the file. This is because the class has been defined as **public** in the Java source code. Java makes it mandatory to save the source code into the file with the class name which is defined as **public**.

Now compile these files using CMD:

```
E:\Java> javac Circle.java MultiplePackages1.java
E:\Java>
```

Once the program (both the files) is successfully compiled, class (object code/byte code) files will be created for each and every class that we declare and define in our program. As our program has 2 classes, belonging to 2 different packages, first **MultiplePackages1**, that belongs to unnamed default package, and the second one the **Circle**, that belongs to **figures.shapes** package, the compiler will create 2 class files, with name **MultiplePackages1.class** and **Circle.class**, on successful compilation.

Now place the **Circle.class** file in the directory hierarchy that should be created by the package hierarchy that we have declared in the **Circle.java** program file. That is, the package hierarchy is **figures.shapes**, thus create directory as:

```
E:\Java\figures\shapes\
```

Now copy/cut paste the **Circle.class** file in the directory **figures.shapes**.

The other class file, the **MultiplePackages1.class** file, belongs to the unnamed default package. Thus no need to create any directory for it. The class file can remain as it is in the **E:\Java** directory.

Now using CMD browse to the directory which is above the directory of the package hierarchy directory, that is in the **E:\Java**, and run the program.

```
E:\Java> java MultiplePackages1
Defining multiple packages.

Default Circle defined.
Default values of Circle :
Area of Circle with radius 1.0 is 3.14

After setting new radius value to Circle :
Area of Circle with radius 5.0 is 78.5

Thank you.
E:\Java>
```

133. Program:
File 1: E:\Java\figures\shapes\circle\Circle.java

```
package figures.shapes.circle;
public class Circle {
```

```
    float r;
    final float pi = 3.14f;
    public Circle(float radius) {
        r = radius;
        System.out.println("Circle defined.");
    }
    public void showArea() {
        float a = pi * r * r;
        System.out.println("Area of Circle with radius " + r + " is " + a);
    }
}
```

File 2: E:\Java\figures\shapes\rectangle\Rectangle.java

```
package figures.shapes.rectangle;
public class Rectangle {
    float l, b;
    public Rectangle(float len, float bre) {
        l = len;
        b = bre;
        System.out.println("Rectangle defined.");
    }
    public void showArea() {
        System.out.printf("Area of Rectangle with length %g, breadth %g is %g\n", l, b, (l*b));
    }
}
```

File 3: E:\Java\figures\shapes\triangle\Triangle.java

```
package figures.shapes.triangle;
public class Triangle {
    float b, h;
    final float half = 0.5f;
    public Triangle(float base, float height) {
        b = base;
        h = height;
        System.out.println("Triangle defined.");
    }
    public void showArea() {
        float a = half * b * h;
        System.out.printf("Area of Triangle with base %g, height %g is %g\n", b, h, a);
    }
}
```

File 4: E:\Java\MultiplePackages2.java

```
import figures.shapes.circle.Circle;
import figures.shapes.rectangle.Rectangle;
import figures.shapes.triangle.Triangle;
class MultiplePackages2 {
    public static void main(String[]args) {
```

```
        System.out.println("Defining multiple packages.\n");
        Circle objC = new Circle(5.0f);
        Rectangle objR = new Rectangle(40.2f, 60.0f);
        Triangle objT = new Triangle(40.2f, 60.0f);
        System.out.println("\nCircle details :");
        objC.showArea();
        System.out.println("\nTriangle details :");
        objT.showArea();
        System.out.println("\nRectangle details :");
        objR.showArea();
        System.out.println("\nThank you.");
    }
}
```

Save these program files (Java source code files) anywhere, with the **public** class names of the respective files. For instance we store the first three files in the "**E:\Java\figures\shapes**" directory.
The first file is stored inside the **circle** directory with the name **Circle.java**, second file inside the **rectangle** directory as **Rectangle.java**, third file in the **triangle** directory as **Triangle.java**, and the last fifth file with is stored outside all these directories that is at location "**E:\Java**" with name **MultiplePackages2.java**. Now compile these programs using **CMD**.

```
E:\Java> javac MultiplePackages2.java figures\shapes\circle\Circle.java
figures\shapes\rectangle\Rectangle.java figures\shapes\triangle\Triangle.java
E:\Java>
```

Once the program is successfully compiled, class (object code/byte code) files will be created for each and every class that we declare and define in our program. As our program has 4 packages and their 4 classes, first **Circle** class of the **figures.shapes.circle** package, second **Rectangle** class of **figures.shapes.rectangle** package, third **Triangle** class of **figures.shapes.triangle** package, and forth, the last one, the **MultiplePackages2** class that belongs to the unnamed **default** package, the compiler will create 4 class files, all with the class name and **.class** extensions, on successful compilation.

Now place these newly created **class files** in the directory hierarchy that should be created by following the package hierarchy that we have declared in each of these program files. That is, the package hierarchy is **figures.shapes.circle** for the **Circle** class, **figures.shapes.rectangle** for the **Rectangle** class and **figures.shapes.triangle** for the **Triangle** class, thus create directories as:

```
E:\Java\figures\shapes\circle
E:\Java\figures\shapes\rectangle
E:\Java\figures\shapes\triangle
```

As we have already created the above show directories and placed their respective Java source code files into those directory hierarchy, no need to again create them.

Note that there is no any requirement of the directory structure for the fourth class, **MultiplePackages2.class** file, because it belongs to the unnamed **default** package.

Now copy/cut paste the class files in their respective package named directories. This is also not needed as the source code files were already present in those directories and the class files have also been created there itself.

Next using CMD browse to the directory where the **main( )** method's class file resides and run the program, that is in the "**E:\Java**" directory.

```
E:\Java> java MultiplePackages2
Defining multiple packages.

Circle defined.
Rectangle defined.
Triangle defined.

Circle details :
Area of Circle with radius 5.0 is 78.5

Triangle details :
Area of Triangle with base 40.2000, height 60.0000 is 1206.00

Rectangle details :
Area of Rectangle with length 40.2000, breadth 60.0000 is 2412.00

Thank you.
E:\Java>
```

Access Protection:

Classes and packages are both means of encapsulating and containing the namespace and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access modifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

Sr.	Classes\Access Modifier	Private	No Modifier	Protected	Public
1.	Same class	Yes	Yes	Yes	Yes
2.	Same package subclass	No	Yes	Yes	Yes
3.	Same package non-subclass	No	Yes	Yes	Yes
4.	Different package subclass	No	No	Yes	Yes
5.	Different package non-subclass	No	No	No	Yes

134. Program: Accessing members of a class within itself, within subclass and then within non-subclass, all belonging to the same package.

File 1: "E:\Java\accessProtection\packageClasses\SuperClass.java":

```
package accessProtection.packageClasses;
```

```
public class SuperClass {
    private int pri = 10;
    int def = 20;
    protected int pro = 30;
    public int pub = 40;
    public SuperClass() {
        System.out.println("SuperClass object initialized:");
        System.out.println("pri = " + pri);
        System.out.println("def = " + def);
        System.out.println("pro = " + pro);
        System.out.println("pub = " + pub);
    }
}
```

File 2: "E:\Java\accessProtection\packageClasses\SubClass.java":

```
package accessProtection.packageClasses;
public class SubClass extends SuperClass {
    public SubClass() {
        System.out.println("SubClass object initialized:");
        // ERROR : pri has private access in SuperClass
        // System.out.println("pri : " + pri);
        System.out.println("def : " + def);
        System.out.println("pro : " + pro);
        System.out.println("pub : " + pub);
    }
}
```

File 3: "E:\Java\accessProtection\packageClasses\NonSubClass.java":

```
package accessProtection.packageClasses;
public class NonSubClass {
    public NonSubClass() {
        SuperClass obj = new SuperClass();
        System.out.println("NonSubClass object initialized:");
        // ERROR : pri has private access in SuperClass
        // System.out.println("obj.pri : " + obj.pri);
        System.out.println("obj.def : " + obj.def);
        System.out.println("obj.pro : " + obj.pro);
        System.out.println("obj.pub : " + obj.pub);
    }
}
```

File 4: "E:\Java\MainClass1.java":

```
import accessProtection.packageClasses.*;
class MainClass1 {
    public static void main(String[]args) {
        System.out.println("Access Protection Mechanism.");
        System.out.println("Super, Sub and Non-sub classes of the same package.");
        System.out.println("\nInitializing SuperClass object.");
```

```
        SuperClass obj1 = new SuperClass();
        System.out.println("\nInitializing SubClass object.");
        SubClass obj2 = new SubClass();
        System.out.println("\nInitializing NonSubClass object.");
        NonSubClass obj3 = new NonSubClass();
        System.out.println("\nThank you.");
    }
}
```

Save all these files and compile them. While saving, save the first three files in their package hierarchy directory structure and save the forth file, the **main( )** method's file which belongs to the default (unnamed) package, in the directory above the package hierarchy directory structure.

That is, the files **SuperClass.java**, **SubClass.java** and **NonSubClass.java** are to be saved into the directory **\accessProtection\packageClasses**. Save the **MainClass1.java** file in the directory above the **\accessProtection\packageClasses**.

Now Compile the program, using CMD with the following command, as:

```
E:\Java> javac MainClass1.java .\accessProtection\packageClasses\SuperClass.java
.\accessProtection\packageClasses\SubClass.java .\accessProtection\packageClasses\NonSubClass.java
E:\Java>
```

On successful compilation we will get the class files already placed into their respective package directory. This is because we have already placed the source code files in their respective package named directory structure.

Now run the program, as:

```
E:\Java> java MainClass1
Access Protection Mechanism.
Super, Sub and Non-sub classes of the same package.

Initializing SuperClass object.
SuperClass object initialized:
pri = 10
def = 20
pro = 30
pub = 40

Initializing SubClass object.
SuperClass object initialized:
pri = 10
def = 20
pro = 30
pub = 40
SubClass object initialized:
def : 20
pro : 30
pub : 40

Initializing NonSubClass object.
```

```
SuperClass object initialized:
pri = 10
def = 20
pro = 30
pub = 40
NonSubClass object initialized:
obj.def : 20
obj.pro : 30
obj.pub : 40

Thank you.
E:\Java>
```

135. Program: Accessing members of a class within itself, within subclass and then within non subclass, all belonging to the same package. At the same time accessing members of a class within a subclass and a non subclass belonging to another package.

File 1, Package 1: "E:\Java\accessProtection\packageClasses\SuperClass.java":

```
package accessProtection.packageClasses;
public class SuperClass {
    private int pri = 10;
    int def = 20;
    protected int pro = 30;
    public int pub = 40;
    public SuperClass() {
        System.out.println("SuperClass object initialized:");
        System.out.println("pri = " + pri);
        System.out.println("def = " + def);
        System.out.println("pro = " + pro);
        System.out.println("pub = " + pub);
    }
}
```

File 2, Package 1: "E:\Java\accessProtection\packageClasses\SubClass.java":

```
package accessProtection.packageClasses;
public class SubClass extends SuperClass {
    public SubClass() {
        System.out.println("SubClass object initialized:");
        // ERROR : pri has private access in SuperClass
        // System.out.println("pri : " + pri);
        System.out.println("def : " + def);
        System.out.println("pro : " + pro);
        System.out.println("pub : " + pub);
    }
}
```

File 3, Package 1: "E:\Java\accessProtection\packageClasses\NonSubClass.java":

```
package accessProtection.packageClasses;
public class NonSubClass {
```

```
    public NonSubClass() {
        SuperClass obj = new SuperClass();
        System.out.println("NonSubClass object initialized:");
        // ERROR : pri has private access in SuperClass
        // System.out.println("obj.pri : " + obj.pri);
        System.out.println("obj.def : " + obj.def);
        System.out.println("obj.pro : " + obj.pro);
        System.out.println("obj.pub : " + obj.pub);
    }
}
```

File 4, Package 2: "E:\Java\accessProtection\packageClasses2\SubClass2.java":

```
package accessProtection.packageClasses2;
import accessProtection.packageClasses.SuperClass;
public class SubClass2 extends SuperClass {
    public SubClass2() {
        System.out.println("Object of SubClass2 from packageClasses2 initialized:");
        // ERROR : pri has private access in SuperClass
        // System.out.println("pri : " + pri);
        // ERROR : def is not public in SuperClass; cannot be accessed from outside package
        // System.out.println("def : " + def);
        System.out.println("pro : " + pro);
        System.out.println("pub : " + pub);
    }
}
```

File 5, Package 2: "E:\Java\accessProtection\packageClasses2\NonSubClass2.java":

```
package accessProtection.packageClasses2;
import accessProtection.packageClasses.SuperClass;
public class NonSubClass2 {
    public NonSubClass2() {
        SuperClass obj = new SuperClass();
        System.out.println("Object of NonSubClass2 from packageClasses2 initialized:");
        // ERROR : pri has private access in SuperClass
        // System.out.println("obj.pri : " + obj.pri);
        // ERROR : def is not public in SuperClass; cannot be accessed from outside package
        // System.out.println("obj.def : " + obj.def);
        // ERROR : pro has protected access in SuperClass
        // System.out.println("obj.pro : " + obj.pro);
        System.out.println("obj.pub : " + obj.pub);
    }
}
```

File 6, Default (unnamed package): "E:\Java\MainClass2.java":

```
class MainClass2 {
    public static void main(String[]args) {
        System.out.println("Access Protection Mechanism.");
        System.out.println("Super, Sub & Non-sub classes in same and in different packages.");
```

```
        System.out.println("\nInitializing SuperClass object of package 1.");
        new accessProtection.packageClasses.SuperClass();
        System.out.println("\nInitializing SubClass object of package 1.");
        new accessProtection.packageClasses.SubClass();
        System.out.println("\nInitializing NonSubClass object of package 1.");
        new accessProtection.packageClasses.NonSubClass();
        System.out.println("\nInitializing SubClass2 object of package 2.");
        new accessProtection.packageClasses2.SubClass2();
        System.out.println("\nInitializing NonSubClass2 object of package 2.");
        new accessProtection.packageClasses2.NonSubClass2();
        System.out.println("\nThank you.");
    }
}
```

Save all these files and compile them. While saving, save the first three files in their package hierarchy directory structure, next 2 files in their different package's hierarchy directory structure and save the sixth file, the **main( )** method's file which belongs to the default (unnamed) package, in the directory above the package hierarchy directory structures.
That is, the first 3 files, **SuperClass.java**, **SubClass.java** and **NonSubClass.java** are to be saved into the directory **/accessProtection/packageClasses/**.
The next 2 files, **SubClass2.java** and **NonSubClass2.java** are to be saved into the directory **/accessProtection/packageClasses2/**.
Lastly save the **MainClass2.java** in the directory above the **/accessProtection/packageClasses/** and **/accessProtection/packageClasses2/**.

Now Compile the program, using CMD with the following command, as:

```
C:\Users\Sourabha> E:
E:\> cd "Java"
E:\Java> javac MainClass2.java accessProtection\packageClasses\SuperClass.java
accessProtection\packageClasses\SubClass.java .\accessProtection\packageClasses\NonSubClass.java
accessProtection\packageClasses2\SubClass2.java accessProtection\packageClasses2\NonSubClass2.java
E:\Java>
```

On successful compilation we will get the class files already placed into their respective package directories. This is because we have already placed the source code files in their respective package named directory structures.

Now run the program, as:

```
E:\Java> java MainClass2
Access Protection Mechanism.
Super, Sub and Non-sub classes of the same as well as different packages.

Initializing SuperClass object of package 1.
SuperClass object initialized:
pri = 10
def = 20
pro = 30
pub = 40

Initializing SubClass object of package 1.
SuperClass object initialized:
```

```
pri = 10
def = 20
pro = 30
pub = 40
SubClass object initialized:
def : 20
pro : 30
pub : 40

Initializing NonSubClass object of package 1.
SuperClass object initialized:
pri = 10
def = 20
pro = 30
pub = 40
NonSubClass object initialized:
obj.def : 20
obj.pro : 30
obj.pub : 40

Initializing SubClass2 object of package 2.
SuperClass object initialized:
pri = 10
def = 20
pro = 30
pub = 40
Object of SubClass2 from packageClasses2 initialized:
pro : 30
pub : 40

Initializing NonSubClass2 object of package 2.
SuperClass object initialized:
pri = 10
def = 20
pro = 30
pub = 40
Object of NonSubClass2 from packageClasses2 initialized:
obj.pub : 40

Thank you.
E:\Java>
```

As we can see we have used various members from various classes belonging to various packages, having various different access modifiers.

XVI - Interfaces:

Introduction:

Using the keyword **interface** we can fully **abstract** a class' interface from its implementation. That is, Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule, their methods are declared without any body. Once it is defined, any number of classes can **implement** an **interface**. Also, one class can implement any number of interfaces.
To implement an interface, a class must provide the complete set of methods required by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows us to fully utilize the "one interface, multiple methods" aspect of polymorphism.
Interfaces are designed to support dynamic method resolution at run time.

Defining an interface:

An **interface** is defined much like a class. Instead of using the **class** keyword, use the **interface** keyword, and apart from this, here in interfaces, we need not have to define any method.

General form of an interface:

```
accessModifier interface InterfaceName {
	returnType methodName1([parameterList]);
	returnType methodName2([parameterList]);
	...
	returnType methodNameN([parameterList]);
	type finalFieldName1 = value1;
	type finalFieldName2 = value2;
	...
	type finalFieldNameN = valueN;
}
```

Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, **abstract** methods. Each class that includes such an **interface** must **implement** all of the methods.

Fields (variables) can be declared inside of **interface** declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must be initialized in the interface itself.
All the methods and fields of an interface are implicitly **public**. That is why while implementing the methods of an interface in any class, we need to define them as **public** only.

Example:

```
interface Shape {
    void area();
}
```

Implementing interface:

Once an **interface** has been defined, one or more classes can **implement** that **interface**. To implement an interface, include the **implements** clause in a **class** definition, and then define the methods declared by the **interface**.

General form:

```
class ClassName [extends SuperClass] implements InterfaceName1 [, InterfaceName2 ...] {
	// class body
}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interfaces. The methods of the **interface** that we implement must be implemented as **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

136. Program:

```
class InterfaceDemo1 implements Shape {
    float l, b;
    public static void main(String[]args) {
        System.out.println("Interface Demo.");
        InterfaceDemo1 obj = new InterfaceDemo1();
        obj.setRectangle();
        obj.area();
        System.out.println("Thank you.");
    }
    void setRectangle() {
        System.out.print("Enter length and breadth of a Rectangle : ");
        l = Float.parseFloat(System.console().readLine());
        b = Float.parseFloat(System.console().readLine());
    }
    public void area() {
        float a = l * b;
        System.out.printf("Rectangle length : %g, breadth : %g, and area : %g.\n", l, b, a);
    }
}
interface Shape {
    void area();
}
```

Default interface Method:

Interfaces may provide **default** implementation for its method. This new capability to **interface** was added with the release of JDK8, and is called the Default Method. JDK8 added this feature to the **interface** to make a significant change to its capabilities. Prior to JDK8, an **interface** could not define any implementation whatsoever. Beginning with JDK8, it is possible to add a **default** implementation to an **interface** method. Thus, it is now possible for the **interface** to specify some behavior.
A Default Method lets us define a default implementation for an interface method. In other words, by use of a default method, it is now possible for an interface method to provide a body, rather than being abstract.
While defining the body for a method inside the interface, we need to decorate the method with the **default** keyword.

General form:

```
accessModifier interface InterfaceName {
	returnType methodName1([parameterList]);
```

```
    default returnType methodName2([parameterList]) {
            Method body...
            ...
    }
    ...
    returnType methodNameN([parameterList]);
    type finalVariableName1 = value1;
    type finalVariableName2 = value2;
    ...
}
```

137. Program:

```
class InterfaceDemo2 {
    public static void main(String[]args) {
        System.out.println("Default method in interface.");
        Cylinder obj = new Cylinder();
        obj.setDetails();
        obj.showDetails();
        obj.area();
        obj.volume();
        System.out.println("Thank you.");
    }
}
interface Circle {
    float pi = 3.14f;
    void area();
    default void showDetails() {
        System.out.println("Circle details :");
        System.out.println("PI : " + pi);
    }
}
class Cylinder implements Circle {
    float r, h;
    void setDetails() {
        System.out.print("Enter radius and height of a Cylinder : ");
        r = Float.parseFloat(System.console().readLine());
        h = Float.parseFloat(System.console().readLine());
    }
    public void area() {
        float a = pi * r * r;
        System.out.println("Radius : " + r);
        System.out.println("Area : " + a);
    }
    void volume() {
        float v = pi * r * r * h;
        System.out.println("Height : " + h);
        System.out.println("Volume : " + v);
    }
}
```

Interface Reference:

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable/object. When we call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces.
The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the “callee”. This process is similar to using a superclass reference to access a subclass object.

138. Program:

```
class InterfaceDemo3 {
    public static void main(String[]args) {
        System.out.println("Interface Demo.");
        Shape obj;
        Circle c = new Circle();
        obj = c;
        obj.setDimensions(5.0f);
        obj.area();
        obj = new Square();
        obj.setDimensions(5.0f);
        obj.area();
        System.out.println("Thank you.");
    }
}
interface Shape {
    float PI = 3.14f;
    void setDimensions(float dim1);
    void area();
}
class Circle implements Shape {
    float r;
    public void setDimensions(float dim1) {
        r = dim1;
    }
    public void area() {
        float a = PI * r;
        System.out.printf("Circle radius : %g, and area : %g.\n", r, a);
    }
}
class Square implements Shape {
    float s;
    public void setDimensions(float dim1) {
        s = dim1;
    }
    public void area() {
        float a = s * s;
        System.out.printf("Square side : %g, and area : %g.\n", s, a);
    }
```

```
}
```

Partial Implementation:

If a class includes/implements an interface but does not fully implement the methods required by that interface, then the class must be declared as **abstract**.

139. Program:

```
class PartialImplementation1 {
    public static void main(String[]args) {
        System.out.println("Partial implementation of Interface.");
        Circle c = new Circle();
        c.setRadius(5.0f);
        c.area();
        System.out.println("Thank you.");
    }
}
interface Shape {
    float PI = 3.14f;
    void setDimensions(float dim1);
    void area();
}
class Circle implements Shape {
    float r;
    void setRadius(float radius) {
        r = radius;
    }
    public void area() {
        float a = PI * r * r;
        System.out.printf("Circle radius : %g, and area : %g.\n", r, a);
    }
}
```

The above program will produce a compile time error, as :

```
E:\Java> java PartialImplementation1.java
PartialImplementation1.java:15: error: Circle is not abstract and does not override abstract
method setDimensions(float) in Shape
class Circle implements Shape {
^
1 error
error: compilation failed
E:\Java>
```

To overcome this error, either override the abstract method from the interface in the implementing class, or else declare the partial implementing class as **abstract**. Next this **abstract** class needs to be inherited and then implement the unimplemented **abstract** methods of the **interface** in the child of the partially implemented **abstract** class.

140. Program:

```
class PartialImplementation2 {
    public static void main(String[]args) {
        System.out.println("Partial implementation of Interface.");
        Cylinder cy = new Cylinder();
        cy.setRadius(5.0f);
        cy.area();
        cy.setDimensions(10.0f);
        cy.volume();
        System.out.println("Thank you.");
    }
}
interface Shape {
    float PI = 3.14f;
    void setDimensions(float dim1);
    void area();
}
abstract class Circle implements Shape {
    float r;
    void setRadius(float radius) {
        r = radius;
    }
    public void area() {
        float a = PI * r * r;
        System.out.printf("Circle radius : %g, and area : %g.\n", r, a);
    }
}
class Cylinder extends Circle {
    float h;
    public void setDimensions(float height) {
        h = height;
    }
    void volume() {
        float v = PI * r * r * h;
        System.out.printf("Cylinder height : %g, and volume : %g.\n", h, v);
    }
}
```

Nested Interface:

An interface can be declared as a member of a class or another interface. Such an interface is called a member interface or a nested interface. A nested interface can be declared as public, private, or protected. This differs from a top-level interface, which must either be declared as public or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

141. Program:

```
class NestedInterface1 {
    public static void main(String[]args) {
        System.out.println("Nested Interface.");
        Shape objS = new Shape();
        Cylinder objCy = new Cylinder(objS);
        objCy.obj.setDimensions(5.0f, 10.0f);
        objCy.volume();
        System.out.println("Thank you.");
    }
}
class Shape {
    float d1, d2;
    public interface Circle {
        float PI = 3.14f;
        float area();
    }
    void setDimensions(float dim1, float dim2) {
        d1 = dim1;
        d2 = dim2;
    }
}
class Cylinder implements Shape.Circle {
    float v;
    Shape obj;
    Cylinder(Shape obj) {
        this.obj = obj;
    }
    public float area() {
        return PI * obj.d1 * obj.d1;
    }
    void volume() {
        System.out.println("Cylinder details :");
        System.out.println("Radius : " + obj.d1 + ", Height : " + obj.d2);
        System.out.println("Area : " + area() + ", Volume : " + (area() * obj.d2));
    }
}
```

Extending Interface:

One interface can inherit another by use of the **extends** keyword. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

142. Program:

```
class ExtendingInterface1 {
    public static void main(String[]args) {
```

```
        System.out.println("Extending Interface.");
        TriangleShape obj = new TriangleShape();
        obj.setDimensions(20.0f, 15.0f);
        obj.area();
        obj.showTriangle();
        System.out.println("Thank you.");
    }
}
interface Shape {
    void setDimensions(float d1, float d2);
}
interface Triangle extends Shape {
    float half = 0.5f;
    void area();
}
class TriangleShape implements Triangle {
    float b, h, a;
    public void setDimensions(float base, float height) {
        b = base;
        h = height;
    }
    public void area() {
        a = half * b * h;
    }
    void showTriangle() {
        System.out.println("Triangle details are :");
        System.out.println("Base : " + b + ", height : " + h);
        System.out.println("Area : " + a);
    }
}
```

Implementing Multiple Interfaces:

Java does not support the multiple inheritance of classes. But it is allowed using interfaces.

Multiple Inheritance:

A class can implement multiple interfaces, which can be indirectly referred to as multiple inheritance. To implement multiple interfaces the class must implement all the methods of all the implementing interfaces.
The syntax to implement multiple interfaces is similar to that of implementing a single interface with one change that is just specify the comma separated list of interfaces after the **implements** keyword.

143. Program:

```
class MultipleInheirtance1 {
    public static void main(String[]args) {
        System.out.println("Multiple inheritance using interfaces.");
        int flag = 0;
```

```
        MultipleInterfaces obj = new MultipleInterfaces();
        do {
            System.out.println("\nArithmetic Operations.");
            System.out.print("Press 1 for integer, 2 for floating-point operations :");
            flag = Integer.parseInt(System.console().readLine());
            if(flag == 1) {
                System.out.print("Enter 2 integer values :");
                int no1 = Integer.parseInt(System.console().readLine());
                int no2 = Integer.parseInt(System.console().readLine());
                obj.integerOps(no1, no2);
            }
            else if(flag == 2) {
                System.out.print("Enter 2 floating-point values :");
                float no1 = Float.parseFloat(System.console().readLine());
                float no2 = Float.parseFloat(System.console().readLine());
                obj.floatOps(no1, no2);
            }
            System.out.print("Press 1 to Continue, 0 to Exit :");
            flag = Integer.parseInt(System.console().readLine());
        }while(flag == 1);
        System.out.println("Thank you.");
    }
}
interface AdditiveOps {
    int add(int n1, int n2);
    float add(float n1, float n2);
    int sub(int n1, int n2);
    float sub(float n1, float n2);
}
interface MultiplicativeOps {
    int mul(int n1, int n2);
    float mul(float a, float b);
    int div(int no1, int no2);
    float div(float x, float y);
    int mod(int n, int d);
}
class MultipleInterfaces implements AdditiveOps, MultiplicativeOps {
    public int add(int a, int b) {
        return (a + b);
    }
    public float add(float n1, float n2) {
        return (n1 + n2);
    }
    public int sub(int r, int s) {
        return (r - s);
    }
    public float sub(float t, float u) {
        return (t - u);
```

```
    }
    public int mul(int a, int b) {
        return (a * b);
    }
    public float mul(float n1, float n2) {
        return (n1 * n2);
    }
    public int div(int r, int s) {
        return (r / s);
    }
    public float div(float t, float u) {
        return (t / u);
    }
    public int mod(int n, int d) {
        return (n % d);
    }
    void integerOps(int a, int b) {
        System.out.println("Arithmetic Operations of integer values " + a + ", and " + b);
        System.out.println("Addition : " + add(a, b));
        System.out.println("Subtraction : " + sub(a, b));
        System.out.println("Multiplication : " + mul(a, b));
        System.out.println("Division : " + div(a, b));
        System.out.println("Remainder of division : " + mod(a, b));
    }
    void floatOps(float x, float y) {
        System.out.println("Arithmetic Operations of floating-point values " + x + " & " + y);
        System.out.println("Addition : " + add(x, y));
        System.out.println("Subtraction : " + sub(x, y));
        System.out.println("Multiplication : " + mul(x, y));
        System.out.println("Division : " + div(x, y));
    }
}
```

Static Interface Method:

JDK8 added another new capability to the interface: the ability to define one or more **static** methods. Like static methods in a class, a static method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a static method.

144. Program:

```
class StaticInterfaceMethod1 {
    static public void main(String[]args) {
        System.out.println("Static interface methods.");;
        NumberOps.square(5);
        NumberOps.cube(5);
        // ERROR : non-static method squareRoot(int) cannot be referenced from a static context
```

```
        // NumberOps.squareRoot(5);
        System.out.println("Thank you.");
    }
}
interface NumberOps {
    static void square(int n) {
        System.out.println("Square of " + n + " is " + (n * n));
    }
    static void cube(int n) {
        System.out.println("Cube of " + n + " is " + (n * n * n));
    }
    void squareRoot(int n);
}
```

XVII - Exception Handling:

Exception:

Exception is nothing but runtime error, or also known as runtime abnormality.
Exception is an abnormal condition. In Java, an exception is an event that disrupts the normal flow of the program.
Exception is an object which is thrown at runtime.

A Java Program with an exception:

145. Program:

```
class ExceptionDemo1 {
    public static void main(String[]args) {
        System.out.println("Exception Demo.");
        System.out.println("Arithmetic Operations:");
        System.out.print("Enter 2 numbers :");
        int no1 = Integer.parseInt(System.console().readLine());
        int no2 = Integer.parseInt(System.console().readLine());
        int result = no1 / no2;
        System.out.printf("Division of %d and %d is %d.\n", no1, no2, result);
        result = no1 * no2;
        System.out.printf("Multiplication of %d and %d is %d.\n", no1, no2, result);
        result = no1 - no2;
        System.out.printf("Subtraction of %d and %d is %d.\n", no1, no2, result);
        result = no1 + no2;
        System.out.printf("Addition of %d and %d is %d.\n", no1, no2, result);
        System.out.println("Thank you.");
    }
}
```

Another Java Program with an exception:

146. Program:

```
class ExceptionDemo2 {
    public static void main(String[]args) {
        System.out.println("Exception Demo 2.");
        arithmeticOperations();
        System.out.println("Thank you.");
    }
    static void arithmeticOperations() {
        System.out.println("Arithmetic Operations :");
        int[]a = new int[5];
        System.out.print("Enter 2 numbers :");
        a[0] = Integer.parseInt(System.console().readLine());
        a[1] = Integer.parseInt(System.console().readLine());
        a[2] = a[0] / a[1];
        System.out.println("Division : " + a[2]);
```

```
        a[3] = a[0] * a[1];
        System.out.println("Multiplication : " + a[3]);
        a[4] = a[0] - a[1];
        System.out.println("Subtraction : " + a[4]);
        a[5] = a[0] + a[1];
        System.out.println("Addition : " + a[5]);
        System.out.println("Returning from the arithmeticOperations().");
    }
}
```

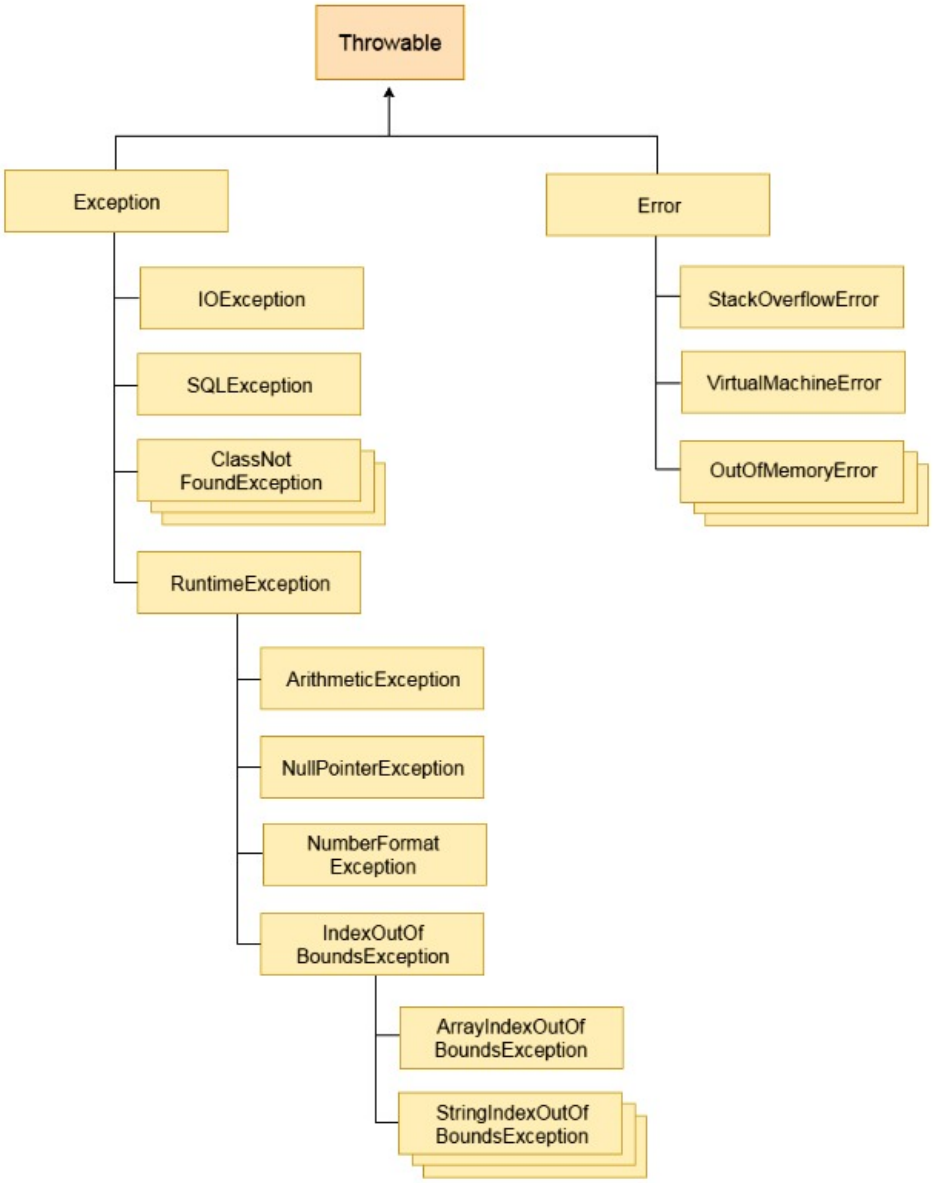
Exception Handling:

Exception Handling in Java is one of the powerful mechanisms to handle the runtime errors so that flow of the program can be maintained.
Exception Handling is a mechanism to handle runtime errors through exception classes, such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Exception Types:

All exception types are objects of classes which are the subclasses of the built-in class **Throwable**. Thus, the **Throwable** class is at the top of the exception class hierarchy. Immediately below the **Throwable** class are two subclasses that partition exceptions into two distinct branches.
One branch is headed by the **Exception** class. This class is used for exceptional conditions that user programs should catch (handle).
The other branch is topped by the **Error** class, which defines exceptions that are not expected to be caught under normal circumstances by our program.

Hierarchy of Java Exception Classes:



The **java.lang.Throwable** class is the root class of Java Exception hierarchy which is inherited by two subclasses: **Exception** and **Error**.
All other exception classes come under these two classes, that is under the **Exception**, or **Error** class.

Exception-Handling Fundamentals:

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
Java exception handling is managed via five keywords:

- **try**: Program statements that we want to monitor for exceptions are contained within the **try** block. If an exception occurs within the **try** block, it is thrown.
- **catch**: Our code can **catch** the exception thrown from its **try** block and handle it in some rational manner, avoiding the abnormal termination of the program using the **catch** block.
- **throw**: To explicitly (manually) **throw** an exception, use the keyword **throw**.
- **throws**: Any exception that is thrown out of a method must be specified as such by a **throws** clause.

- **finally**: Any code that absolutely must be executed, any exception occurs or not, after a **try** block completes is put in a **finally** block.

General form:

```
try {
      // block of code to monitor for errors
      ...
}
catch(ExceptionType1 obj1) {
      // exception handling for ExceptionType1
      ...
}
catch(ExceptionType2 obj2) {
      // exception handling for ExceptionType1
      ...
}
...
finally {
      // block of code to be executed after try block ends
      ...
}
...
```

147. Program:

```
class ExceptionHandling1 {
    public static void main(String[]args) {
        System.out.println("Exception Handling Demo.");
        System.out.println("Arithmetic Operations:");
        System.out.print("Enter 2 numbers :");
        int no1 = Integer.parseInt(System.console().readLine());
        int no2 = Integer.parseInt(System.console().readLine());
        int result = 0;
        try {
            result = no1 / no2;
            System.out.printf("Division of %d and %d is %d.\n", no1, no2, result);
        }
        catch(ArithmeticException obj) {
            System.out.println("ERROR : Denominator cannot be 0 for division.");
        }
        result = no1 * no2;
        System.out.printf("Multiplication of %d and %d is %d.\n", no1, no2, result);
        result = no1 - no2;
        System.out.printf("Subtraction of %d and %d is %d.\n", no1, no2, result);
        result = no1 + no2;
        System.out.printf("Addition of %d and %d is %d.\n", no1, no2, result);
        System.out.println("Thank you.");
    }
}
```

148. Program:

```
class ExceptionDemo2 {
```

```
    public static void main(String[]args) {
        System.out.println("Exception Demo 2.");
        arithmeticOperations();
        System.out.println("Thank you.");
    }
    static void arithmeticOperations() {
        System.out.println("Arithmetic Operations :");
        System.out.print("Enter 2 numbers :");
        int no1 = 0, no2 = 0;
        try {
            no1 = Integer.parseInt(System.console().readLine());
            no2 = Integer.parseInt(System.console().readLine());
        }
        catch(NumberFormatException obj) {
            System.out.println("ERROR : Invalid integer input value.");
        }
        int result = 0;
        try {
            result = no1 / no2;
            System.out.println("Division : " + result);
        }
        catch(ArithmeticException obj) {
            System.out.println("ERROR : Denominator cannot be 0 for division.");
        }
        result = no1 * no2;
        System.out.println("Multiplication : " + result);
        result = no1 - no2;
        System.out.println("Subtraction : " + result);
        result = no1 + no2;
        System.out.println("Addition : " + result);
        System.out.println("Returning from the arithmeticOperations().");
    }
}
```

Multiple catch Clauses:

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, we can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block.

149. Program:

```
class MultipleCatch1 {
    public static void main(String[]args) {
        System.out.println("Multiple catch clauses.");
        System.out.println("Arithmetic Operations:");
        System.out.print("Enter 2 numbers :");
```

```
        int no1 = 1, no2 = 1, result = 0;
        try {
            no1 = Integer.parseInt(System.console().readLine());
            no2 = Integer.parseInt(System.console().readLine());
            result = no1 / no2;
            System.out.printf("Division of %d and %d is %d.\n", no1, no2, result);
        }
        catch(NumberFormatException obj) {
            System.out.println("ERROR : Invalid integer input value.");
        }
        catch(ArithmeticException obj) {
            System.out.println("ERROR : Denominator cannot be 0 for division.");
        }
        result = no1 * no2;
        System.out.printf("Multiplication of %d and %d is %d.\n", no1, no2, result);
        result = no1 - no2;
        System.out.printf("Subtraction of %d and %d is %d.\n", no1, no2, result);
        result = no1 + no2;
        System.out.printf("Addition of %d and %d is %d.\n", no1, no2, result);
        System.out.println("Thank you.");
    }
}
```

Exception handling of the called method by the calling method:

150. Program:

```
class MultipleCatch2 {
    public static void main(String[]args) {
        System.out.println("Exception Demo 2.");
        try {
            arithmeticOperations();
        }
        catch(NumberFormatException obj) {
            System.out.println("ERROR : Invalid integer input value.");
        }
        catch(ArithmeticException obj) {
            System.out.println("ERROR : Denominator cannot be 0 for division.");
        }
        catch(ArrayIndexOutOfBoundsException obj) {
            System.out.println("ERROR : Array index out of the size of array.");
        }
        System.out.println("Thank you.");
    }
    static void arithmeticOperations() {
        System.out.println("Arithmetic Operations :");
        int[]a = new int[5];
        System.out.print("Enter 2 numbers :");
        a[0] = Integer.parseInt(System.console().readLine());
```

```
        a[1] = Integer.parseInt(System.console().readLine());
        a[2] = a[0] / a[1];
        System.out.println("Division : " + a[2]);
        a[3] = a[0] * a[1];
        System.out.println("Multiplication : " + a[3]);
        a[4] = a[0] - a[1];
        System.out.println("Subtraction : " + a[4]);
        a[5] = a[0] + a[1];
        System.out.println("Addition : " + a[5]);
        System.out.println("Returning from the arithmeticOperations().");
    }
}
```

Displaying Description of an Exception:

The **Throwable** class overrides the **toString( )** method (defined by **Object** class) so that it returns a string containing a description of the exception. We can display this description using the **println( )** method by simply passing the exception object as an argument.

The err Object:

Java provides another output stream object, apart from **out**, the **err** object. It is recommended to use this **err** object to print the error messages in the **catch** block. The **err** object is just like the **out** object of the **System** class.

151. Program:

```
class ExceptionDescription1 {
    public static void main(String[]args) {
        System.out.println("Exception description.");
        try {
            arithmeticOperations();
        }
        catch(NumberFormatException obj) {
            System.err.println("ERROR : " + obj);
        }
        catch(ArithmeticException ex) {
            System.err.println("ERROR : " + ex);
        }
        catch(ArrayIndexOutOfBoundsException ai) {
            System.err.println("ERROR : " + ai);
        }
        System.out.println("Thank you.");
    }
    static void arithmeticOperations() {
        System.out.println("Arithmetic Operations :");
        int[]a = new int[5];
        System.out.print("Enter 2 numbers :");
        a[0] = Integer.parseInt(System.console().readLine());
```

```
        a[1] = Integer.parseInt(System.console().readLine());
        a[2] = a[0] / a[1];
        System.out.println("Division : " + a[2]);
        a[3] = a[0] * a[1];
        System.out.println("Multiplication : " + a[3]);
        a[4] = a[0] - a[1];
        System.out.println("Subtraction : " + a[4]);
        a[5] = a[0] + a[1];
        System.out.println("Addition : " + a[5]);
        System.out.println("Returning from the arithmeticOperations().");
    }
}
```

The printStackTrace( ) method:

The **printStackTrace( )** method of **java.lang.Throwable** class can be used to print the **Throwable** exception object along with other details like class name and line number where the exception occurred, which means it is a backtrace. This method prints a stack trace for the **Throwable** object on the standard error output stream.
The first line of output shows the same string which was returned by the **toString( )** method for the **Throwable** object means Exception class name and later lines represent data previously recorded by the method **fillInStackTrace( )**.

General form:

```
public void printStackTrace()
```

152. Program:

```
class ExceptionDescription2 {
    public static void main(String[]args) {
        System.out.println("Exception description using the printStackTrace() method.");
        arithmeticOperations();
        System.out.println("Thank you.");
    }
    static void arithmeticOperations() {
        System.out.println("Arithmetic Operations :");
        int[]a = new int[5];
        System.out.print("Enter 2 numbers :");
        try {
            a[0] = Integer.parseInt(System.console().readLine());
            a[1] = Integer.parseInt(System.console().readLine());
            a[2] = a[0] / a[1];
            System.out.println("Division : " + a[2]);
            a[3] = a[0] * a[1];
            System.out.println("Multiplication : " + a[3]);
            a[4] = a[0] - a[1];
            System.out.println("Subtraction : " + a[4]);
            a[5] = a[0] + a[1];
            System.out.println("Addition : " + a[5]);
        }
```

```
        catch(NumberFormatException obj) {
            obj.printStackTrace();
        }
        catch(ArithmeticException ex) {
            ex.printStackTrace();
        }
        catch(ArrayIndexOutOfBoundsException ai) {
            ai.printStackTrace();
        }
        System.out.println("Returning from the arithmeticOperations().");
    }
}
```

Handling Multiple Exceptions (Single catch Block):

In Java all exception classes that we can handle are the derived classes of the **Exception** class. This **Exception** class can be used to handle all types of exceptions at once, using a single **catch** block.

153. Program:

```
class MultipleExceptions1 {
    public static void main(String[]args) {
        System.out.println("Handling multiple exceptions using single catch clause.");
        arithmeticOperations();
        System.out.println("Thank you.");
    }
    static void arithmeticOperations() {
        System.out.println("Arithmetic Operations :");
        int[]a = new int[5];
        System.out.print("Enter 2 numbers :");
        try {
            a[0] = Integer.parseInt(System.console().readLine());
            a[1] = Integer.parseInt(System.console().readLine());
            a[2] = a[0] / a[1];
            System.out.println("Division : " + a[2]);
            a[3] = a[0] * a[1];
            System.out.println("Multiplication : " + a[3]);
            a[4] = a[0] - a[1];
            System.out.println("Subtraction : " + a[4]);
            a[5] = a[0] + a[1];
            System.out.println("Addition : " + a[5]);
        }
        catch(Exception ex) {
            ex.printStackTrace();
        }
        System.out.println("Returning from the arithmeticOperations().");
    }
}
```

Nested try statements:

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception.

154. Program:

```
class NestedTryBlocks1 {
    public static void main(String[]args) {
        System.out.println("Nesting of try blocks to handle multiple exceptions.");
        try {
            arithmeticOperations();
        }
        catch(Exception ex) {
            System.err.println("The main()'s try block, ERROR : " + ex);
        }
        System.out.println("Thank you.");
    }
    static void arithmeticOperations() {
        System.out.println("Arithmetic Operations :");
        int[]a = new int[5];
        System.out.print("Enter 2 numbers :");
        try {
            a[0] = Integer.parseInt(System.console().readLine());
            a[1] = Integer.parseInt(System.console().readLine());
            a[2] = a[0] + a[1];
            System.out.println("Addition : " + a[2]);
            a[3] = a[0] - a[1];
            System.out.println("Subtraction : " + a[3]);
            a[4] = a[0] * a[1];
            System.out.println("Multiplication : " + a[4]);
            try {
                a[5] = a[0] / a[1];
                System.out.println("Division : " + a[5]);
            }
            catch(ArithmeticException ae) {
                System.err.println("Inner try block, ERROR : " + ae);
            }
            System.out.println("Arithmetic operations finished.");
        }
        catch(NumberFormatException nfe) {
            System.err.println("Outer try block, ERROR : " + nfe);
        }
        System.out.println("Returning from the arithmeticOperations().");
```

```
    }
}
```

The throw clause:

System-generated exceptions are implicitly (automatically) thrown by the Java runtime system. To manually throw an exception, use the keyword **throw**. Using the **throw** statement we can throw an exception explicitly.

General form:

```
throw ThrowableInstance;
```

155. Program:

```
class ThrowingAnException1 {
    public static void main(String[]args) {
        System.out.println("Throwing an Exception.");
        try {
            arithmeticOperations();
        }
        catch(Exception obj) {
            System.err.println("ERROR : " + obj);
        }
        System.out.println("Thank you.");
    }
    static void arithmeticOperations() {
        System.out.println("Arithmetic Operations :");
        int[]a = new int[5];
        System.out.print("Enter 2 numbers :");
        a[0] = Integer.parseInt(System.console().readLine());
        a[1] = Integer.parseInt(System.console().readLine());
        a[2] = a[0] + a[1];
        System.out.println("Addition : " + a[2]);
        a[3] = a[0] - a[1];
        System.out.println("Subtraction : " + a[3]);
        a[4] = a[0] * a[1];
        System.out.println("Multiplication : " + a[4]);
        if(a[1] == 0) {
            ArithmeticException obj = new ArithmeticException("Division by 0 not possible.");
            throw obj;
        }
        a[5] = a[0] / a[1];
        System.out.println("Division : " + a[5]);
        System.out.println("Returning from the arithmeticOperations().");
    }
}
```

The throws clause:

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. We do this by including a **throws** clause in the method's declaration.
A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses.
All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compiler-time error will result.

General form:

```
returnType methodName([parameterList]) throws exceptionClassList {
      // method body...
      ...
}
```

156. Program:

```
class TheThrowsClause1 {
    public static void main(String[]args) {
        System.out.println("The throws clause.");
        DivisionOperation obj = new DivisionOperation();
        try {
            obj.division();
        }
        catch(Exception ex) {
            System.err.println("ERROR : " + ex);
        }
        System.out.println("Thank you.");
    }
}
class DivisionOperation {
    void division() throws ArithmeticException, NumberFormatException {
        int no1, no2;
        System.out.print("Enter 2 numbers :");
        no1 = Integer.parseInt(System.console().readLine());
        no2 = Integer.parseInt(System.console().readLine());
        int div = no1 / no2;
        System.out.println("Division : " + div);
        System.out.println("Returning from the division() method.");
    }
}
```

The finally Clause:

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
For example, if a method opens a file upon entry and closes it upon exit, then we will not want the code that closes the file to be bypassed by the exception handling mechanism.

The **finally** keyword is designed to address this contingency. The **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is abrupt to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit **return** statement, the **finally** clause is also executed just before the method returns.

The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

157. Program:

```
class FinallyClause1 {
    public static void main(String[]args) {
        System.out.println("The finally Clause.");
        try {
            arithmeticOperations();
        }
        catch(Exception obj) {
            System.err.println("ERROR : " + obj);
        }
        System.out.println("Thank you.");
    }
    static void arithmeticOperations() {
        System.out.println("Arithmetic Operations :");
        int[]a = new int[5];
        System.out.print("Enter 2 numbers :");
        try {
            a[0] = Integer.parseInt(System.console().readLine());
            a[1] = Integer.parseInt(System.console().readLine());
            a[2] = a[0] + a[1];
            System.out.println("Addition : " + a[2]);
            a[3] = a[0] - a[1];
            System.out.println("Subtraction : " + a[3]);
            a[4] = a[0] * a[1];
            System.out.println("Multiplication : " + a[4]);
            if(a[1] == 0)
                throw new ArithmeticException("Division by 0 not possible.");
            if(a.length < 6) {
                System.out.println("Returning from the method due to small array size.");
                return;
            }
            a[5] = a[0] / a[1];
            System.out.println("Division : " + a[5]);
        }
        catch(NumberFormatException ex) {
            System.err.println("ERROR : " + ex);
        }
        finally {
            System.out.println("This is the finally block.");
        }
        System.out.println("Returning from the arithmeticOperations().");
```

```
    }
}
```

Types of Exceptions:

There are mainly 2 types of exceptions: **checked** and **unchecked**. An error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception:

The classes which directly inherit the **Throwable** class except RuntimeException and Error are known as checked exceptions. Eg: IOException. SQLException, ClassNotFoundException, etc.
Checked exceptions are checked at compile-time. Thus when we compile the program that has the Java code that might throw an Exception of type Checked, the program will never compile, if we have not implemented the Exception Handling mechanism for our code.

158. Program:

```
class CheckedException1 {
    public static void main(String[]args) {
        System.out.println("The checked exception.");
        System.out.println("Throwing the \"ClassNotFoundException\" exception.");
        throw new ClassNotFoundException("My Exception");
        // System.out.println("Thank you."); // ERROR : unreachable statement
    }
}
```

On compilation of the above program the following error is shown:

```
E:\Java> javac CheckedException1.java
CheckedException1.java:5: error: unreported exception ClassNotFoundException; must be caught or
declared to be thrown
        throw new ClassNotFoundException("My Exception");
        ^
1 error
error: compilation failed
E:\Java>
```

To overcome the above compile time error, implement the Exception Handling mechanism for the code:

Solution 1:
159. Program:

```
class CheckedException2 {
    public static void main(String[]args) throws ClassNotFoundException {
        System.out.println("The checked exception.");
        System.out.println("Throwing the \"ClassNotFoundException\" exception.");
        throw new ClassNotFoundException("My Exception");
        // System.out.println("Thank you."); // ERROR : unreachable statement
```

```
    }
}
```

Solution 2:

160. Program:

```
class CheckedException3 {
    public static void main(String[]args) {
        System.out.println("The checked exception.");
        try {
            System.out.println("Throwing the \"ClassNotFoundException\" exception.");
            throw new ClassNotFoundException("My Exception");
        }
        catch(ClassNotFoundException ex) {
            System.err.println("ERROR : " + ex);
        }
        System.out.println("Thank you.");
    }
}
```

Note that, the above 2 programs will compile successfully and will also run.

Unchecked Exception:

The classes which inherit **RuntimeException** are known as unchecked exceptions. Eg: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.
Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

161. Program:

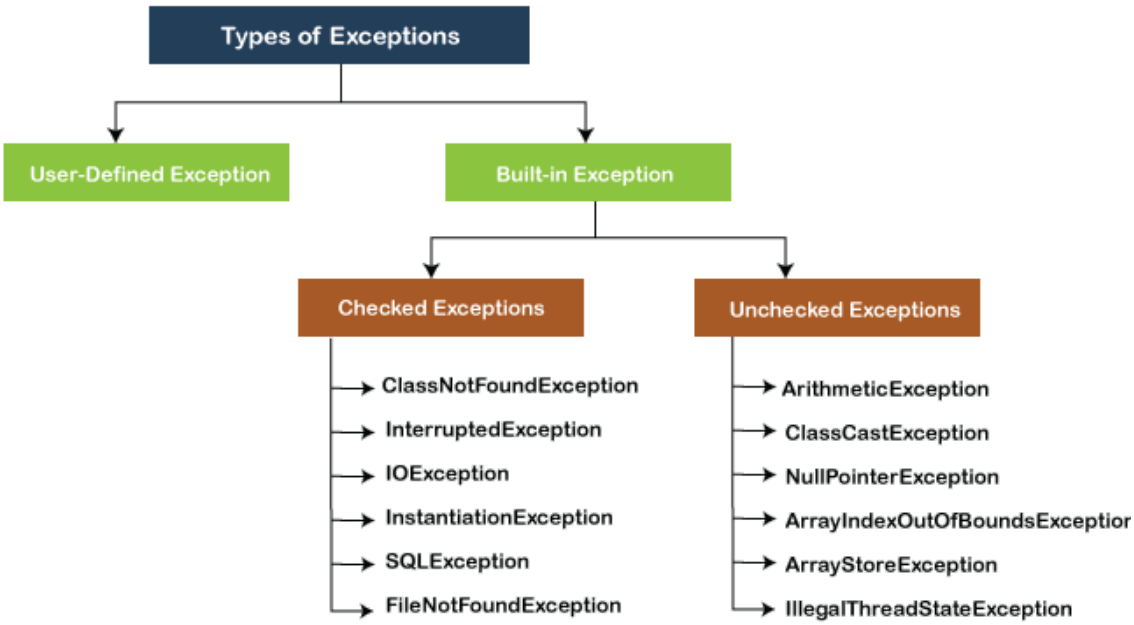
```
class UncheckedException1 {
    static public void main(String[]args) {
        System.out.println("The unchecked exception.");
        System.out.println("Throwing the \"ArithmeticException\" exception.");
        throw new ArithmeticException("My Exception.");
        // System.out.println("Thank you."); // ERROR : unreachable statement
    }
}
```

The above program will compile and run, and then at run-time it throws the ArithmeticException.

Error:

Error is irrecoverable. The errors cannot be handled. They are under the **Error** class which is the child of the **Throwable** class, in the **Exception** class hierarchy. Eg: OutOfMemoryError, VirtualMachineError, AssertionError, etc.

Hierarchy of Types of Exception:



User-Defined Exceptions:

Although Java's built-in exceptions handle most common errors, we will probably want to create our own exception types to handle situations specific to our applications. This is quite easy to do: just define a subclass of **Exception** class (which is, of course, a subclass of **Throwable** class).

162. Program:

```
class UserDefinedException1 {
    static public void main(String[]args) {
        System.out.println("User-defined exception.");
        validateAge();
        System.out.println("Thank you.");
    }
    static void validateAge() {
        System.out.print("Enter your age :");
        try {
            int age = Integer.parseInt(System.console().readLine());
            if(age < 0 || age > 130)
                throw new AgeValidation(age);
            else
                if(age >= 18)
                    System.out.println("Congratulations! You are eligible to drive vehicle.");
                else
                    System.out.println("Sorry. You are not yet eligible to drive any vehicle.");
        }
        catch(AgeValidation av) {
            System.err.println("ERROR : " + av);
        }
        catch(Exception ex) {
```

```
            System.err.println("ERROR : " + ex);
        }
    }
}
class AgeValidation extends Exception {
    private int age;
    AgeValidation(int a) {
        age = a;
    }
    public String toString() {
        return "AgeValidation : Invalid Age [" + age + "].";
    }
}
```

Chained Exception:

Beginning with JDK 1.4, a feature was incorporated into the exception subsystem: chained exception. The chained exception feature allows us to associate another exception with an exception. This second exception describes the cause of the first exception.
To do so the initial cause of the exception can be set using the **initCause( )** method and to get that initial cause use the **getCause( )** methods from the Exception class.

163. Program:

```
class ChainedException1 {
    public static void main(String[]args) {
        System.out.println("Chained Exception.");
        System.out.print("Enter 2 numbers:");
        try {
            int n1 = Integer.parseInt(System.console().readLine());
            int n2 = Integer.parseInt(System.console().readLine());
            if(n2 == 0) {
                NumberFormatException nfe = new NumberFormatException("Invalid input.");
                nfe.initCause(new ArithmeticException("Division by Zero."));
                throw nfe;
            }
            int div = n1 / n2;
            System.out.println("Division : " + div);
        }
        catch(NumberFormatException ex) {
            System.err.println("ERROR : "+ ex);
            System.err.println("Cause : " + ex.getCause());
        }
        System.out.println("Thank you.");
    }
}
```

XVIII - Multithreaded Programming:

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. Thus, multithreading is a specialized form of multitasking.

Thread:

A thread is a lightweight sub-process, the smallest unit of processing. A single process or software program consists of multiple sub-processes, each one can be referred to as a thread.

Thread-based Multitasking:

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.
For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
The thread-based multitasking handles the details of a single process. Multitasking threads require less overhead than multitasking processes.
Threads are light-weight. They share the same memory address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is lower in cost.

Process-based Multitasking:

In a process-based multitasking environment, separate processes are executing following their own separate execution path.
For instance, 2 separate processes, such as a text editor and a web browser, both at the same time are in execution performing their own separate operations.
Thus, process-based multitasking deals with the “big picture”.
Processes are heavyweight tasks that require their own separate memory address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.

While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java’s control. However, multithreaded multitasking is.

Multithreading:

Multithreading in Java is a process of executing multiple threads simultaneously. Multiprocessing and multithreading, both are used to achieve multitasking.
However, we use multithreading rather than multiprocessing because threads use a shared memory area.
They don't allocate a separate memory area so saves memory, and context-switching between the threads takes less time than process.
Java Multithreading is mostly used in games, animation, etc.

Advantages:

1. It doesn't block the user because threads are independent and we can perform multiple operations at the same time.

2. We can perform many operations together, so it saves time.
3. Threads are independent, so it does not affect other threads if an exception occurs in a single thread.

Multitasking:

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU more efficiently. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process gets a separate memory area allocated.
- The process is heavyweight.
- Cost of communication between the processes is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

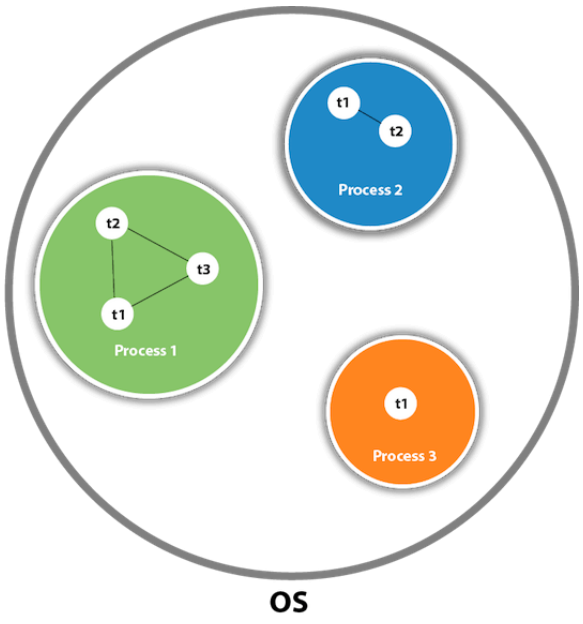
Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the threads is low.

Note: At least one process is required for each thread.

What is a Thread in Java:

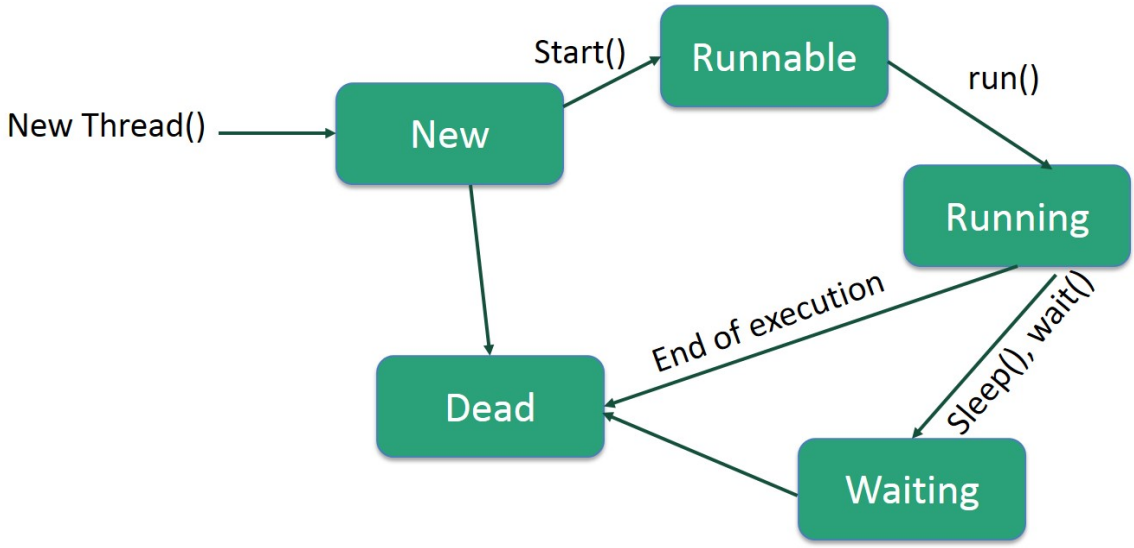
A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.
Threads are independent. If an exception occurs in one thread, it does not affect other threads. It uses a shared memory area.



Multi-threading enables us to write-in a procedure where multiple activities can proceed concurrently in the same program.

Thread Life Cycle:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies.



Following are the stages of the life cycle of a Thread −

- **New** − A new thread begins its life cycle in the **new** state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable** − After a newly born thread is started, the thread becomes **runnable**. A thread in this state is considered to be executing its task.
- **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Implementation of Thread:

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. The **Thread** class encapsulates a thread of execution. Since we can't directly refer to the core state of a running thread, we will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, our program will either **extend** the **Thread** class or **implement** the **Runnable** interface.

Thread class:

Java provides the **Thread** class to achieve thread programming. The **Thread** class provides constructors and methods to create and perform operations on a thread. The **Thread** class extends the **Object** class and implements the **Runnable** interface.

The **Thread** class defines several methods that help manage threads. Several of those used in are shown below:

1. `public void run():`
   It is used to perform action for a thread.
2. `public void start():`
   It starts the execution of the thread. JVM calls the **run( )** method on the thread.
3. `public void sleep(long milliseconds):`
   This method causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

4. `public void join():`
   It waits for a thread to die.
5. `public void join(long milliseconds):`
   It waits for a thread to die for the specified milliseconds.
6. `public int getPriority():`
   It returns the priority of the thread.
7. `public int setPriority(int priority):`
   It changes the priority of the thread.
8. `public String getName():`
   It returns the name of the thread.
9. `public void setName(String name):`
   It changes the name of the thread.
10. `public Thread currentThread():`
    It returns the reference of the currently executing thread.
11. `public int getId():`
    It returns the id of the thread.
12. `public Thread.State getState():`
    It returns the state of the thread.
13. `public boolean isAlive():`
    It tests if the thread is alive.
14. `public void yield():`
    It causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. `public boolean isDaemon():`
    It tests if the thread is a daemon thread.
16. `public void setDaemon(boolean b):`
    It marks the thread as a daemon or user thread.
17. `public void interrupt():`
    It interrupts the thread.
18. `public boolean isInterrupted():`
    It tests if the thread has been interrupted.
19. `public static boolean interrupted():`
    It tests if the current thread has been interrupted.

The main Thread:

When a Java program starts up, one thread begins running immediately. This is usually called the **main thread** of our program, because it is the one that is executed when our program begins. The **main thread** is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the **main thread** is created automatically when our program is started, it can be controlled through a **Thread** object. To do so, we must obtain a reference to it by calling the method **currentThread( )**, which is a public static member of the **Thread** class.

Its general form is shown here:

```
public static Thread currentThread()
```

This method returns a reference to the Thread in which it is called. Once we have a reference to the thread, we can control it just like any other thread.

164. Program:

```
class MainThread1 {
```

```
    public static void main(String[]args) {
        System.out.println("The main thread.");
        Thread objT = Thread.currentThread();
        System.out.println("Current thread : " + objT);
        System.out.println("Thank you.");
    }
}
```

Managing main Thread:

As we get the reference of the main thread, using the **currentThread( )** method of the **Thread** class, we can manage it too. That is, we shall now change the name of this current, main, thread, and shall also perform some operations, such as printing a count-down from 10 to 1.

165. Program:

```
class MainThread2 {
    static public void main(String[]args) {
        System.out.println("Managing the current, main, thread.");
        ManagingThread obj = new ManagingThread();
        obj.threadDemo();
        System.out.println("Thank you.");
    }
}
class ManagingThread {
    void threadDemo() {
        System.out.println("Inside the threadDemo() method.");
        Thread objT = Thread.currentThread();
        System.out.println("Current thread : " + objT);
        objT.setName("My Thread");
        System.out.println("Current thread after setting a new name : " + objT);
        System.out.println("Count-down from 10 to 1 :\n");
        try {
            for(int cnt = 10; cnt > 0; --cnt) {
                System.out.println(cnt);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException ie) {
            System.err.println(objT + " interrupted : " + ie);
        }
        System.out.println("Exiting from the threadDemo() method.");
    }
}
```

Creating New Thread:

In the most general sense, we create a thread by instantiating an object of type **Thread** class.
Java defines two ways in which this can be accomplished:

- We can implement the **Runnable** interface.
- We can extend the **Thread** class, itself.

Implementing Runnable Interface:

The easiest way to create a thread is to create a class that **implements** the **Runnable** interface. The **Runnable** interface abstracts a unit of executable code. We can construct a thread on any object that **implements** the **Runnable** interface. To implement the **Runnable** interface, a class needs to only implement a single method called **run( )**.

After we create a class that implements the **Runnable** interface, we instantiate an object of type **Thread** class from within that class. The **Thread** class defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadObj, String threadName)
```

In this constructor, *threadObj* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until we call its **start( )** method, which is declared within the **Thread** class. In essence, the **start( )** method executes a call to the **run( )** method. The **start( )** method is shown here:

```
void start()
```

This **start( )** method invokes the **run( )** method on the newly created thread object.

The general form of the **run( )** method from the Runnable interface is as follows:

```
public void run() {
      ...
}
```

The **run( )** method is used to perform action for a thread. Inside the **run( )** method, we can define the code that constitutes the new thread. That is, the **run( )** method begins execution of a new thread. The newly created thread ends when the **run( )** method returns.

166. Program:

```
class NewThreadDemo1 {
    public static void main(String[]args) {
        System.out.println("Creating a new thread.");
        new NewThread();
        System.out.println("Count-down of main thread from 100 to 91 :\n");
        try {
            for(int cnt = 100; cnt > 90; --cnt) {
                System.out.println("Main thread : " + cnt);
                Thread.sleep(500);
            }
            System.out.println("\nCount-down of main thread finished.");
        }
```

```
        catch(InterruptedException ie) {
            System.err.println("Main thread interrupted : " + ie);
        }
        System.out.println("Thank you.");
    }
}
class NewThread implements Runnable {
    Thread objT;
    NewThread() {
        objT = new Thread(this, "My Thread");
        System.out.println("New child thread created : " + objT);
        objT.start();
    }
    public void run() {
        System.out.println("Child thread started...");
        System.out.println("Count-down of child thread from 10 to 1 :\n");
        try {
            for(int cnt = 10; cnt > 0; --cnt) {
                System.out.println("Child thread : " + cnt);
                Thread.sleep(500);
            }
            System.out.println("\nCount-down of child thread finished.");
        }
        catch(InterruptedException ie) {
            System.err.println(objT + " interrupted : " + ie);
        }
        System.out.println("Exiting from the child thread.");
    }
}
```

Extending Thread Class:

The second way to create a thread is to create a new class that extends the **Thread** class, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call the **start( )** method, which invokes the **run( )** method, to begin execution of the new thread.

Following is the preceding program rewritten to extend the **Thread** class which creates a new thread:

167. Program:

```
class NewThreadDemo2 {
    public static void main(String[]args) {
        System.out.println("Creating a new thread.");
        new NewThread();
        System.out.println("Count-down of main thread from 100 to 91 :\n");
        try {
            for(int cnt = 100; cnt > 90; --cnt) {
                System.out.println("Main thread : " + cnt);
                Thread.sleep(500);
```

```
            }
            System.out.println("\nCount-down of main thread finished.");
        }
        catch(InterruptedException ie) {
            System.err.println("Main thread interrupted : " + ie);
        }
        System.out.println("Thank you.");
    }
}
class NewThread extends Thread {
    NewThread() {
        super("My Thread");
        System.out.println("New child thread created : " + this);
        start();
    }
    public void run() {
        System.out.println("Child thread started...");
        System.out.println("Count-down of child thread from 10 to 1 :\n");
        try {
            for(int cnt = 10; cnt > 0; --cnt) {
                System.out.println("Child thread : " + cnt);
                Thread.sleep(500);
            }
            System.out.println("\nCount-down of child thread finished.");
        }
        catch(InterruptedException ie) {
            System.err.println(this + " interrupted : " + ie);
        }
        System.out.println("Exiting from the child thread.");
    }
}
```

Creating Multiple Threads:

So far, we have been using only one or two threads: the **main** thread and one child thread. However, our program can spawn as many threads as it needs. For example, the following program creates three child threads:

168. Program:

```
class NewThreadDemo3 {
    public static void main(String[]args) {
        System.out.println("Creating multiple threads.");
        NewThread obj1 = new NewThread("First", 60, 50);
        NewThread obj2 = new NewThread("Second", 40, 50);
        NewThread obj3 = new NewThread("Third", 150, 140);
        System.out.println("Count-down of main thread from 100 to 91 :\n");
        try {
            for(int cnt = 100; cnt > 90; --cnt) {
```

```
                System.out.println("Main thread : " + cnt);
                Thread.sleep(500);
            }
            System.out.println("\nCount-down of main thread finished.");
        }
        catch(InterruptedException ie) {
            System.err.println("Main thread interrupted : " + ie);
        }
        System.out.println("Thank you.");
    }
}
class NewThread implements Runnable {
    Thread objT;
    String threadName;
    int iStart, iStop;
    NewThread(String threadName, int start, int stop) {
        this.threadName = threadName;
        iStart = start;
        iStop = stop;
        if(iStart < iStop) {
            System.out.println("Invalid count-down values. Cannot create the new thread.");
            return;
        }
        objT = new Thread(this, threadName);
        System.out.println("New child thread created : " + objT);
        objT.start();
    }
    public void run() {
        System.out.println(threadName + " thread started...");
        System.out.println("Count-down of "+threadName+" thread from "+iStart+" to "+iStop+"
:\n");
        try {
            for(int cnt = iStart; cnt > iStop; --cnt) {
                System.out.println(threadName + " thread : " + cnt);
                Thread.sleep(500);
            }
            System.out.println("\nCount-down of " + threadName + " thread finished.");
        }
        catch(InterruptedException ie) {
            System.err.println(objT + " interrupted : " + ie);
        }
        System.out.println("Exiting from the " + threadName + " thread.");
    }
}
```

The isAlive( ) and join( ) method:

As mentioned, often we will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep( )** method within **main( )** method, with a long enough delay to ensure that all child threads terminate prior to the **main** thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, the **Thread** class provides a means by which we can answer this question.

Two ways exist to determine whether a thread has finished. First, we can call the **isAlive( )** method on the thread object. This method is defined by the **Thread** class, and its general form is shown here:

```
final boolean isAlive()
```

The **isAlive( )** method returns **true** if the thread object upon which it is called is still running. It returns **false** otherwise.

While the **isAlive( )** method is occasionally useful, the method that we will more commonly use to wait for a thread to finish is the **join( )** method, shown here:

```
final void join() throws InterruptedException
```

The **join( )** method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of the **join( )** method allow us to specify a maximum amount of time that we want to wait for the specified thread to terminate.

Here is an improved version of the preceding example that uses **join( )** method to ensure that the main thread is the last to stop. It also demonstrates the use of the **isAlive( )** method.

169. Program:

```
class IsAliveAndJoin1 {
    public static void main(String[]args) {
        System.out.println("Creating multiple threads.");
        NewThread obj1 = new NewThread("First", 60, 50);
        NewThread obj2 = new NewThread("Second", 40, 30);
        NewThread obj3 = new NewThread("Third", 150, 140);
        System.out.println("Thread First is alive : " + obj1.objT.isAlive());
        System.out.println("Thread Second is alive : " + obj2.objT.isAlive());
        System.out.println("Thread Third is alive : " + obj3.objT.isAlive());
        try {
            obj1.objT.join();
            obj2.objT.join();
            obj3.objT.join();
            System.out.println("\nAll child threads finished.");
        }
        catch(InterruptedException ie) {
            System.err.println("Main thread interrupted : " + ie);
        }
        System.out.println("Thread First is alive : " + obj1.objT.isAlive());
        System.out.println("Thread Second is alive : " + obj2.objT.isAlive());
        System.out.println("Thread Third is alive : " + obj3.objT.isAlive());
        System.out.println("Thank you.");
```

```
    }
}
class NewThread implements Runnable {
    Thread objT;
    String threadName;
    int iStart, iStop;
    NewThread(String threadName, int start, int stop) {
        this.threadName = threadName;
        iStart = start;
        iStop = stop;
        if(iStart < iStop) {
            System.out.println("Invalid count-down values. Cannot create the new thread.");
            return;
        }
        objT = new Thread(this, threadName);
        System.out.println("New child thread created : " + objT);
        objT.start();
    }
    public void run() {
        System.out.println(threadName + " thread started...");
        System.out.println("Count-down of " + threadName + " thread from " + iStart + " to " +
iStop +" :\n");
        try {
            for(int cnt = iStart; cnt > iStop; --cnt) {
                System.out.println(threadName + " thread : " + cnt);
                Thread.sleep(500);
            }
            System.out.println("\nCount-down of " + threadName + " thread finished.");
        }
        catch(InterruptedException ie) {
            System.err.println(objT + " interrupted : " + ie);
        }
        System.out.println("Exiting from the " + threadName + " thread.");
    }
}
```

Thread Priority:

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, over a given period of time, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.)

A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread. In theory, threads of equal priority should get equal access to the CPU.

To set a thread's priority, use the **setPriority( )** method, which is a member of the **Thread** class. This is its general form:

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of level must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are **1** and **10**, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently **5**. These priorities are defined as **static final** data fields within the **Thread** class.

We can obtain the current priority setting by calling the **getPriority( )** method of the **Thread** class, shown here:

```
final int getPriority()
```

170. Program:

```
class ThreadPriority1 {
    public static void main(String[]args) {
        System.out.println("Creating multiple threads.");
        NewThread obj1 = new NewThread("Min", Thread.MIN_PRIORITY);
        NewThread obj2 = new NewThread("Max", Thread.MAX_PRIORITY);
        NewThread obj3 = new NewThread("Normal", Thread.NORM_PRIORITY);
        obj1.begin();
        obj2.begin();
        obj3.begin();
        try {
            Thread.sleep(5000);
        }
        catch(InterruptedException ie) {
            System.err.println("Main thread interrupted : " + ie);
        }
        obj1.end();
        obj2.end();
        obj3.end();
        System.out.println("Thread "+ obj1.threadName + "'s Priority : " +
obj1.objT.getPriority());
        System.out.println("Thread "+ obj2.threadName + "'s Priority : " +
obj2.objT.getPriority());
        System.out.println("Thread "+ obj3.threadName + "'s Priority : " +
obj3.objT.getPriority());
        System.out.println("Thank you.");
    }
}
class NewThread implements Runnable {
    int cnt = 1;
    boolean flag;
    Thread objT;
    String threadName;
    NewThread(String threadName, int prio) {
        this.threadName = threadName;
```

```
        objT = new Thread(this, threadName);
        System.out.println("New child thread created : " + objT);
        objT.setPriority(prio);
    }
    public void begin() {
        flag = true;
        objT.start();
    }
    public void run() {
        System.out.println(threadName + " thread started...");
        while(flag) {
            System.out.println(threadName + " thread : " + cnt);
            ++cnt;
        }
        System.out.println("Exiting from " + threadName + " thread with cnt : " + cnt);
    }
    public void end() {
        flag = false;
    }
}
```

Synchronization:

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. As we will see, Java provides unique, language-level support for it.
We can synchronize our code in either of two ways. Both involve the use of the **synchronized** keyword.

A Non-synchronized Method:

While sharing the same resource if threads are not in synchronization with each other, they might result in some unexpected output.

171. Program:

```
class NonSyncThreads1 {
    public static void main(String args[]) {
        System.out.println("Non-Synchronized Threads.");
        NonSyncDemo obj = new NonSyncDemo();
        NonSyncThread obj1 = new NonSyncThread(obj, "Hello");
        NonSyncThread obj2 = new NonSyncThread(obj, "Thread");
        NonSyncThread obj3 = new NonSyncThread(obj, "Synchronization");
        try {
            obj1.objT.join();
            obj2.objT.join();
            obj3.objT.join();
        }
        catch(InterruptedException ie) {
```

```
            ie.printStackTrace();
        }
        System.out.println("Exiting from the main thread.");
    }
}
class NonSyncDemo {
    void nonSyncMethod(String msg) {
        System.out.print("[");
        try {
            Thread.sleep(500);
            System.out.print(msg);
            Thread.sleep(500);
        }
        catch(InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("]");
    }
}
class NonSyncThread implements Runnable {
    String msg;
    Thread objT;
    NonSyncDemo obj;
    public NonSyncThread(NonSyncDemo obj, String msg) {
        this.obj = obj;
        this.msg = msg;
        objT = new Thread(this);
        objT.start();
    }
    public void run() {
        obj.nonSyncMethod(msg);
    }
}
```

Output:

```
E:\Java> java NonSyncThreads.java
Non-Synchronized Threads.
[[[HelloSynchronizationThread]
]
]
Exiting from the main thread.
E:\Java>
```

Using synchronized method:

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified/declared with the **synchronized** keyword. While a

thread is inside a **synchronized** method, all other threads that try to call it (or any other **synchronized** method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the **synchronized** method.

172. Program:

```
class SyncThreads1 {
    public static void main(String args[]) {
        System.out.println("Synchronized Threads, using synchronized method.");
        SyncDemo obj = new SyncDemo();
        SyncThread obj1 = new SyncThread(obj, "Hello");
        SyncThread obj2 = new SyncThread(obj, "Thread");
        SyncThread obj3 = new SyncThread(obj, "Synchronization");
        try {
            obj1.objT.join();
            obj2.objT.join();
            obj3.objT.join();
        }
        catch(InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("Exiting from the main thread.");
    }
}
class SyncDemo {
    synchronized void syncMethod(String msg) {
        System.out.print("[");
        try {
            Thread.sleep(500);
            System.out.print(msg);
            Thread.sleep(500);
        }
        catch(InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("]");
    }
}
class SyncThread implements Runnable {
    String msg;
    Thread objT;
    SyncDemo obj;
    public SyncThread(SyncDemo obj, String msg) {
        this.obj = obj;
        this.msg = msg;
        objT = new Thread(this);
        objT.start();
    }
    public void run() {
```

```
        obj.syncMethod(msg);
    }
}
```

Output:

```
E:\Java> java SyncThreads1.java
Synchronized Threads, using synchronized method.
[Hello]
[Synchronization]
[Thread]
Exiting from the main thread.
E:\Java>
```

Using synchronized statement:

While creating **synchronized** methods within classes that we create is an easy and effective means of achieving synchronization, however this will not work in all cases. To understand why, consider the following.
Imagine that we want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by us, but by a third party, and we do not have access to the source code. Thus, we can't add the **synchronized** keyword to the appropriate methods within the class. How can access to an object of this class be synchronized?
Fortunately, the solution to this problem is quite easy: we simply put calls to the methods defined by such a class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(objRef) {
      // statements to be synchronized
}
```

Here, *objRef* is a reference to the object being synchronized. A **synchronized** block ensures that a call to a **synchronized** method that is a member of objRef's class occurs only after the current thread has successfully entered objRef's monitor.

173. Program:

```
class SyncThreads2 {
    public static void main(String args[]) {
        System.out.println("Synchronized Threads, using synchronized block.");
        NonSyncDemo obj = new NonSyncDemo();
        SyncThread obj1 = new SyncThread(obj, "Hello");
        SyncThread obj2 = new SyncThread(obj, "Thread");
        SyncThread obj3 = new SyncThread(obj, "Synchronization");
        try {
            obj1.objT.join();
            obj2.objT.join();
            obj3.objT.join();
        }
        catch(InterruptedException ie) {
            ie.printStackTrace();
        }
```

```
        System.out.println("Exiting from the main thread.");
    }
}
class NonSyncDemo {
    void nonSyncMethod(String msg) {
        System.out.print("[");
        try {
            Thread.sleep(500);
            System.out.print(msg);
            Thread.sleep(500);
        }
        catch(InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("]");
    }
}
class SyncThread implements Runnable {
    String msg;
    Thread objT;
    NonSyncDemo obj;
    public SyncThread(NonSyncDemo obj, String msg) {
        this.obj = obj;
        this.msg = msg;
        objT = new Thread(this);
        objT.start();
    }
    public void run() {
        synchronized(obj) {
            obj.nonSyncMethod(msg);
        }
    }
}
```

Output:

```
E:\Java> java SyncThreads2.java
Synchronized Threads, using synchronized block.
[Hello]
[Synchronization]
[Thread]
Exiting from the main thread.
E:\Java>
```

Inter-Thread Communication:

Multithreading replaces event loop programming by dividing our tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time.
For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object** class, so all classes have them. All three methods can be called only from within a **synchronized** context.

Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- **wait( )** - The **wait( )** method tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )** or **notifyAll( )**.
- **notify( )** - The **notify( )** method wakes up a thread that calls **wait( )** on the same object.
- **notifyAll( )** - The **notifyAll( )** method wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

These methods are declared within **Object** class, as shown here:

```
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

174. Program:
Example without interthread communication:

```
class NonInterThreadComm1 {
    public static void main(String[]args) {
        System.out.println("Without Interthread Communication.");
        Number objN = new Number();
        new Producer(objN);
        new Consumer(objN);
        System.out.println("Press \"Ctrl + c\" to stop...");
    }
}
class Number {
    int no;
    synchronized void get() {
        System.out.println("Consumer : " + no);
    }
    synchronized void put(int no) {
        this.no = no;
        System.out.println("Producer : " + no);
    }
}
class Producer implements Runnable {
```

```
    Number objN1;
    Producer(Number objN) {
        this.objN1 = objN;
        new Thread(this, "Non-inter-thread communication Procduce.").start();
    }
    public void run() {
        int n = 0;
        while(true) {
            objN1.put(n++);
        }
    }
}
class Consumer implements Runnable {
    Number objN2;
    Consumer(Number objN) {
        this.objN2 = objN;
        new Thread(this, "Non-inter-thread communication Consumer.").start();
    }
    public void run() {
        while(true) {
            objN2.get();
        }
    }
}
```

Output:

```
E:\Java> java NonInterThreadComm.java
Without Interthread Communication.
Press "Ctrl + c" to stop...
Producer : 0
Producer : 1
Producer : 2
Producer : 3
Producer : 4
Producer : 5
Producer : 6
Consumer : 6
Consumer : 6
Consumer : 6
...
```

As we can see in the above program, 2 threads which are in synchronization with each other, but do not have any communication are unable to work one after another. That is, the expected output was, the Producer thread should produce a single value, and the Consumer should consume it. After the Consumer thread consumes the value, the Producer thread should produce the next new value. But this is not happening.
So the solution for this problem is inter thread communication.

175. Program:

```
class InterThreadComm1 {
```

```
    public static void main(String[]args) {
        System.out.println("Interthread Communication.");
        Number objN = new Number();
        new Producer(objN);
        new Consumer(objN);
        System.out.println("Press \"Ctrl + c\" to stop...");
    }
}
class Number {
    int no;
    boolean valueSet = false;
    synchronized void get() {
        while(!valueSet) {
            try {
                wait();
            }
            catch(InterruptedException ie) {
                ie.printStackTrace();
            }
        }
        System.out.println("Consumer : " + no);
        valueSet = false;
        notify();
    }
    synchronized void put(int no) {
        while(valueSet) {
            try {
                wait();
            }
            catch(InterruptedException ie) {
                ie.printStackTrace();
            }
        }
        this.no = no;
        valueSet = true;
        System.out.println("Producer : " + no);
        notify();
    }
}
class Producer implements Runnable {
    Number objN1;
    Producer(Number objN) {
        this.objN1 = objN;
        new Thread(this, "Non-inter-thread communication Procduce.").start();
    }
    public void run() {
        int n = 0;
        while(true) {
```

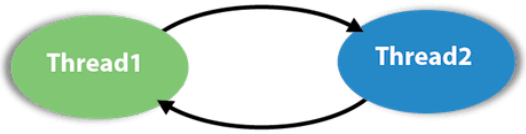
```
            objN1.put(n++);
        }
    }
}
class Consumer implements Runnable {
    Number objN2;
    Consumer(Number objN) {
        this.objN2 = objN;
        new Thread(this, "Non-inter-thread communication Consumer.").start();
    }
    public void run() {
        while(true) {
            objN2.get();
        }
    }
}
```

Output:

```
E:\Java> java InterThreadComm1.java
Interthread Communication.
Press "Ctrl + c" to stop...
Producer : 0
Consumer : 0
Producer : 1
Consumer : 1
Producer : 2
Consumer : 2
Producer : 3
Consumer : 3
Producer : 4
Consumer : 4
Producer : 5
Consumer : 5
Producer : 6
Consumer : 6
...
```

Deadlock:

Deadlock in Java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock that is acquired by another thread and a second thread is waiting for an object lock that is acquired by the first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



A special type of error that we need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.
For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Both objects will wait indefinitely for each other, and this is nothing but a deadlock condition.

Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

176. Program:

```
class DeadlockDemo1 implements Runnable {
    public static void main(String[]args) {
        System.out.println("Deadlock demo.");
        new DeadlockDemo1();
        System.out.println("Exiting from the main() method.");
    }
    Deadlock1 obj1 = new Deadlock1();
    Deadlock2 obj2 = new Deadlock2();
    DeadlockDemo1() {
        Thread.currentThread().setName("Main Thread");
        Thread objT = new Thread(this, "Child thread");
        objT.start();
        obj1.method1(obj2);
        System.out.println("Back in the main thread.");
    }
    public void run() {
        obj2.method2(obj1);
        System.out.println("Back in the child thread");
    }
}
class Deadlock1 {
    String name;
    synchronized void method1(Deadlock2 obj2) {
        name = Thread.currentThread().getName();
        System.out.println("Thread " + name + " entered Deadlock1's method1()");
        try {
            Thread.sleep(1000);
        }
        catch(InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("Thread " + name + " trying to call Deadlock2's lastMethod2().");
        obj2.lastMethod2();
    }
```

```
    synchronized void lastMethod1() {
        System.out.println("Thread " + name + " inside Deadlock1's lastMethod1()");
    }
}
class Deadlock2 {
    String name;
    synchronized void method2(Deadlock1 obj1) {
        name = Thread.currentThread().getName();
        System.out.println("Thread " + name + " entered Deadlock2's method2()");
        try {
            Thread.sleep(1000);
        }
        catch(InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("Thread " + name + " trying to call Deadlock1's lastMethod1().");
        obj1.lastMethod1();
    }
    synchronized void lastMethod2() {
        System.out.println("Thread " + name + " inside Deadlock2's lastMethod2()");
    }
}
```

Output:

```
E:\Java> java DeadlockDemo1.java
Deadlock demo.
Thread Main Thread entered Deadlock1's method1()
Thread Child thread entered Deadlock2's method2()
Thread Main Thread trying to call Deadlock2's lastMethod2().
Thread Child thread trying to call Deadlock1's lastMethod1().
```

To come out of this deadlock we will have to kill the process, that is by pressing the "Ctrl + c" keys to kill the process, or else this process/program will never end, and will remain in the deadlock condition forever.

Solution for this deadlock is avoidance. That is, the deadlock does not have any solution, and the only possible solution is the avoidance of the deadlock.

177. Program:

```
class ThreadSuspendResumeDemo1 {
    public static void main(String[]args) {
        System.out.println("Avoiding the deadlock situation.");
        NewThread obj1 = new NewThread("First");
        NewThread obj2 = new NewThread("Second");
        try {
            Thread.sleep(1000);
            System.out.println("Suspending First thread...");
            obj1.suspendThread();
            Thread.sleep(1000);
            System.out.println("Resuming First thread...");
```

```
            obj1.resumeThread();
            System.out.println("Suspending Second thread...");
            obj2.suspendThread();
            Thread.sleep(1000);
            System.out.println("Resuming Second thread...");
            obj2.resumeThread();
        }
        catch(InterruptedException ie) {
            System.err.println("ERROR : " + ie);
        }
        try {
            System.out.println("Main thread waiting for child threads to finish...");
            obj1.objT.join();
            obj2.objT.join();
        }
        catch(InterruptedException ie) {
            System.err.println("ERROR : " + ie);
        }
        System.out.println("Exiting from the main thread.");
    }
}
class NewThread implements Runnable {
    String name;
    Thread objT;
    boolean suspendFlag;
    NewThread(String name) {
        this.name = name;
        objT = new Thread(this, name);
        System.out.println("New Thread : " + objT + " created.");
        suspendFlag = false;
        objT.start();
    }
    public void run() {
        System.out.println("Count-down from 10 to 1 inside the thread " + name + ":");
        try {
            for(int i = 10; i > 0; --i) {
                System.out.println(name + " : " + i);
                Thread.sleep(500);
                synchronized(this) {
                    while(suspendFlag) {
                        wait();
                    }
                }
            }
        }
        catch(InterruptedException ie) {
            ie.printStackTrace();
        }
```

```
    }
    synchronized void suspendThread() {
        suspendFlag = true;
    }
    synchronized void resumeThread() {
        suspendFlag = false;
        notify();
    }
}
```

Output:

```
E:\Java> java ThreadSuspendResumeDemo1.java
Avoiding the deadlock situation.
New Thread : Thread[#30,First,5,main] created.
New Thread : Thread[#31,Second,5,main] created.
Count-down from 10 to 1 inside the thread First:
Count-down from 10 to 1 inside the thread Second:
First : 10
Second : 10
Second : 9
First : 9
Suspending First thread...
Second : 8
Second : 7
Resuming First thread...
Suspending Second thread...
First : 8
First : 7
Resuming Second thread...
Main thread waiting for child threads to finish...
Second : 6
First : 6
Second : 5
First : 5
Second : 4
First : 4
Second : 3
First : 3
Second : 2
First : 2
Second : 1
First : 1
Exiting from the main thread.
E:\Java>
```

XIX - GUI Programming:

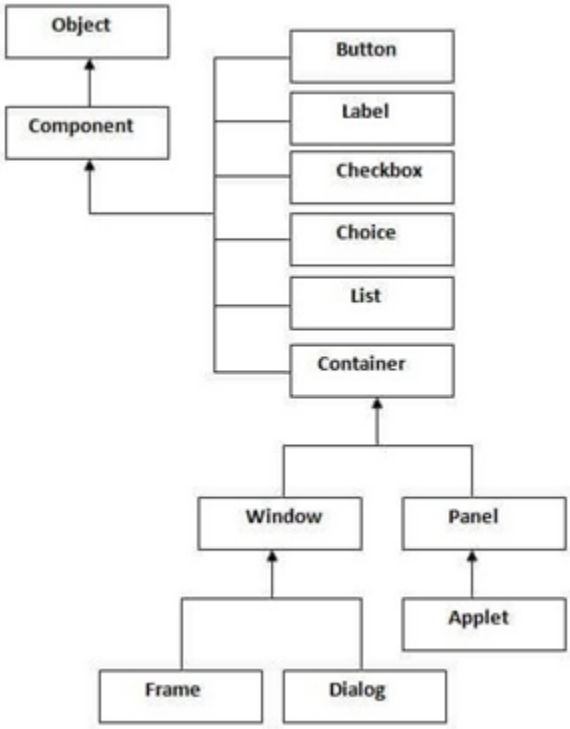
Introduction:

Graphical User Interface (GUI) offers user interaction via some graphical components. For example our underlying Operating System also offers GUI via window, frame, Panel, Button, Textfield, TextArea, Listbox, Combobox, Label, Checkbox, RadioButton, etc. These all are known as components. Using these components we can create an interactive user interface for an application.
GUI provides results to end users in response to raised events (actions performed by user). GUI is entirely based on events. For example clicking on a button, closing a window, opening a window, typing something in a textarea etc. These activities are known as events. GUI makes it easier for the end user to use an application. It also makes them interesting.
To design and define GUI applications Java provides us with the **AWT** API.

AWT:

AWT stands for Abstract Window Toolkit, which is an API that provides the graphical user interface classes, interfaces, objects, methods, etc to design and develop GUI applications.
Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in Java.
Java AWT components are platform-dependent i.e. components are displayed according to the view of the OS. AWT is heavyweight i.e. its components use resources of the OS.
The **java.awt** package provides classes for AWT API such as TextField, Label, TextArea, Button, RadioButton, etc.

Java AWT class Hierarchy:



Java AWT:

To create a simple AWT example, we need a frame. There are two ways to create a frame in AWT.

- By extending the **Frame** class (inheritance)
- By creating the object of the **Frame** class (association)

Extending (inheriting) the Frame class:

178. Program:

```
import java.awt.*;
class AWTDemo1 extends Frame {
    public static void main(String args[]) {
        System.out.println("AWT Frame Window.");
        new AWTDemo1();
        System.out.println("Thank you.");
    }
    AWTDemo1() {
        Button btn = new Button("Click Me");
        btn.setBounds(50, 100, 120, 40);
        add(btn);
        setSize(800, 600);
        setLayout(null);
        setVisible(true);
    }
}
```

Creating object (association) of the Frame class:

179. Program:

```
import java.awt.*;
class AWTDemo2 {
    public static void main(String args[]) {
        System.out.println("AWT Frame Window.");
        new AWTDemo2();
        System.out.println("Thank you.");
    }
    AWTDemo2() {
        Button btn = new Button("Click Me");
        btn.setBounds(50, 100, 120, 40);
        Frame objF = new Frame();
        objF.add(btn);
        objF.setSize(800, 600);
        objF.setLayout(null);
        objF.setVisible(true);
    }
}
```

Event Handling:

Action performed by the user is known as an event, or changing the state of an object is also known as an event. Eg: Click on a button, dragging mouse, entering text in an input (text) field, closing the frame, etc.

The **java.awt.event** package provides many event classes and event listener interfaces for event handling.

Java Event classes and Listener interfaces are as follows:

Sr. No.	Event class	Listener interfaces
1.	ActionEvent	ActionListener
2.	MouseEvent	MouseListener and MouseMotionListener
3.	MouseWheelEvent	MouseWheelListener
4.	KeyEvent	KeyListener
5.	ItemEvent	ItemListener
6.	TextEvent	TextListener
7.	AdjustmentEvent	AdjustmentListener
8.	WindowEvent	WindowListener
9.	ComponentEvent	ComponentListener
10.	ContainerEvent	ContainerListener
11.	FocusEvent	FocusListener

Implementation of Event Handling:

Following steps are required to perform event handling:

1. Register component with Listener, and
2. Implement event handling code

Registering the component with the Listener:
For registering the component with the Listener, many event classes provide the registration methods.

For example:

- Button
  public void addActionListener(ActionListener a) {}
- MenuItem
  public void addActionListener(ActionListener a) {}
- TextField
  public void addActionListener(ActionListener a) {}
  public void addTextListener(TextListener a) {}
- TextArea
  public void addTextListener(TextListener a) {}

- Checkbox
  public void addItemListener(ItemListener a) {}
- Choice
  public void addItemListener(ItemListener a) {}
- List
  public void addActionListener(ActionListener a) {}
  public void addItemListener(ItemListener a) {}

Implementation of Event Handling Code:
We can put the event handling code present in the event listener method into one of the following places:

a. Within class
b. Other class
c. Anonymous class

Within class:
180. Program:

```
import java.awt.*;
import java.awt.event.*;
class AWTEvent1 implements ActionListener {
    Label lbl;
    public static void main(String args[]) {
        System.out.println("AWT Frame Window.");
        new AWTEvent1();
        System.out.println("Thank you.");
    }
    AWTEvent1() {
        Button btn = new Button("Click Me");
        btn.setBounds(80, 100, 120, 40);
        btn.addActionListener(this);
        lbl = new Label();
        lbl.setBounds(80, 150, 200, 25);
        Frame objF = new Frame();
        objF.add(lbl);
        objF.add(btn);
        objF.setSize(800, 600);
        objF.setLayout(null);
        objF.setVisible(true);
    }
    public void actionPerformed(ActionEvent ae) {
        lbl.setText("Hello! You clicked the Button.");
    }
}
```

Other class:
181. Program:

```
import java.awt.*;
import java.awt.event.*;
class AWTEvent3 {
    Label lbl;
```

```
    public static void main(String args[]) {
        System.out.println("AWT Frame Window.");
        new AWTEvent3();
        System.out.println("Thank you.");
    }
    AWTEvent3() {
        Button btn = new Button("Click Me");
        btn.setBounds(80, 100, 120, 40);
        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                lbl.setText("Hello! You clicked the Button.");
            }
        });
        lbl = new Label();
        lbl.setBounds(80, 150, 200, 25);
        Frame objF = new Frame();
        objF.add(lbl);
        objF.add(btn);
        objF.setSize(800, 600);
        objF.setLayout(null);
        objF.setVisible(true);
    }
}
```

Anonymous class:

182. Program:

```
import java.awt.*;
import java.awt.event.*;
class AWTEvent3 {
    Label lbl;
    public static void main(String args[]) {
        System.out.println("AWT Frame Window.");
        new AWTEvent3();
        System.out.println("Thank you.");
    }
    AWTEvent3() {
        Button btn = new Button("Click Me");
        btn.setBounds(80, 100, 120, 40);
        btn.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                lbl.setText("Hello! You clicked the Button.");
            }
        });
        lbl = new Label();
        lbl.setBounds(80, 150, 200, 25);
        Frame objF = new Frame();
        objF.add(lbl);
        objF.add(btn);
```

```
        objF.setSize(800, 600);
        objF.setLayout(null);
        objF.setVisible(true);
    }
}
```

Swing:

Java Swing is a part of Java Foundation Classes (JFC) that is used to create window-based applications. It is built on the top of AWT (Abstract Window Toolkit) API and entirely written in Java.
Unlike AWT, Java Swing provides platform-independent and lightweight components.
The **javax.swing** package provides classes for Java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser, etc.

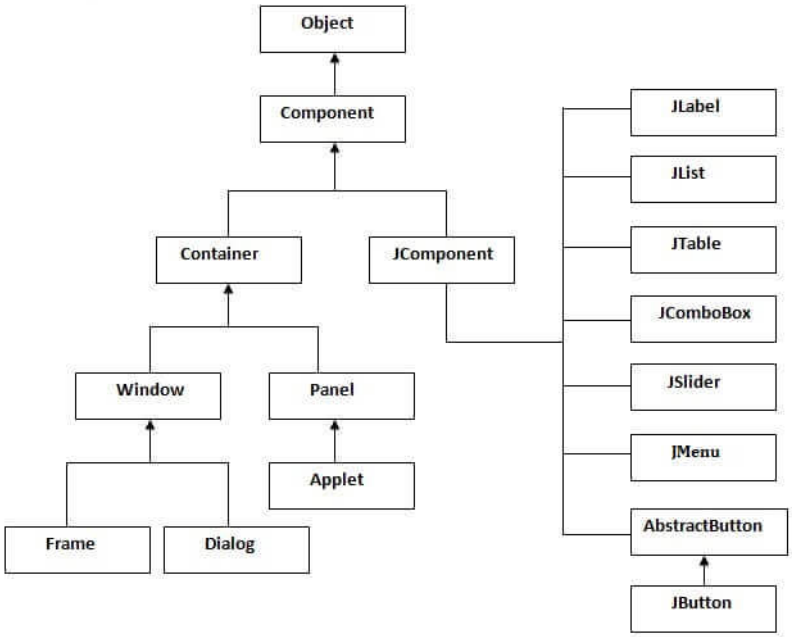
Difference between AWT and Swing:

Sr	Java AWT	Java Swing
1.	AWT components are platform-dependent.	Swing components are platform-independent.
2.	AWT components are heavyweight.	Swing components are lightweight.
3.	AWT does not support pluggable look and feel.	Swing supports pluggable look and feel.
4.	AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollPanes, colorChooser, tabbedPane, etc.
5.	AWT does not follow MVC (Model View Controller) architecture.	Swing follows MVC, where model represents data, view represents presentation and controller acts as an interface between model and view.

Java Foundation Class (JFC):

The Java foundation classes are a set of GUI components which simplify the development of desktop applications.

Hierarchy of class in Java Swing:



Java Swing:

To create a simple Swing example, we need a frame. There are two ways to create a frame in Swing, just like the AWT frame example.

- By extending the **JFrame** class (inheritance)
- By creating the object of the **JFrame** class (association)

Extending (inheriting) the JFrame class:

182. Program:

```
import javax.swing.*;
class SwingDemo1 extends JFrame {
    public static void main(String args[]) {
        System.out.println("Swing Frame Window.");
        new SwingDemo1();
        System.out.println("Thank you.");
    }
    SwingDemo1() {
        JButton btn = new JButton("Click Me");
        btn.setBounds(50, 100, 120, 40);
        add(btn);
        setTitle("Swing Frame Window");
        setSize(800, 600);
        setLayout(null);
        setVisible(true);
    }
}
```

Creating object (association) of the JFrame class:

184. Program:

```
import javax.swing.*;
class SwingDemo2 {
    public static void main(String args[]) {
        System.out.println("Swing Frame Window.");
        new SwingDemo2();
        System.out.println("Thank you.");
    }
    SwingDemo2() {
        JButton btn = new JButton("Click Me");
        btn.setBounds(50, 100, 120, 40);
        JFrame objF = new JFrame();
        objF.add(btn);
        objF.setTitle("Swing Frame Window.");
        objF.setSize(800, 600);
        objF.setLayout(null);
        objF.setVisible(true);
    }
}
```

Swing Components/Controls:

Every user interface considers three main aspects:

- UI Elements:

These are the core visual elements the user eventually sees and interacts with. Swing provides a huge list of widely used and common elements varying from basic to complex.

- Layouts:

They define how UI elements should be organized on the screen and provide a final look and feel to the GUI (Graphical User Interface).

- Behavior:

These are the events which occur when the user interacts with UI elements.

Event Handling:

Action performed by the user is known as an event, or changing the state of an object is also known as an event. Eg: Click on a button, dragging mouse, entering text in an input (text) field, closing the frame, etc.

The same **java.awt.event** package provides many event classes and event listener interfaces for event handling just like the AWT. Perhaps Swing uses the same event handling mechanism that is found in the AWT.

185. Program:

```
import javax.swing.*;
import java.awt.event.*;
class EventHandling1 extends JFrame implements ActionListener {
    JTextField txt1;
    JLabel lbl1;
    public static void main(String[]args) {
        new EventHandling1();
    }
    EventHandling1() {
```

```
        txt1 = new JTextField();
        txt1.setBounds(80, 80, 200, 20);
        JButton btn1 = new JButton("Click Me");
        btn1.setBounds(80, 120, 100, 30);
        lbl1 = new JLabel("Enter your name :");
        lbl1.setBounds(80, 170, 200, 30);
        btn1.addActionListener(this);
        add(txt1);
        add(btn1);
        add(lbl1);
        setTitle("Swing Frame Window and Button Click Event");
        setSize(800, 600);
        setLayout(null);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public void actionPerformed(ActionEvent ae) {
        String strName = txt1.getText();
        lbl1.setText("Hello! " + strName);
    }
}
```

186. Program:

```
import javax.swing.*;
import java.awt.event.*;
class EventHandling2 extends JFrame {
    JTextField txt1, txt2;
    JLabel lbl1, lbl2, lblResult;
    public static void main(String[]args) {
        new EventHandling2();
    }
    EventHandling2() {
        lbl1 = new JLabel("Number 1 : ");
        lbl1.setBounds(120, 80, 70, 30);
        txt1 = new JTextField();
        txt1.setBounds(195, 80, 200, 30);
        lbl2 = new JLabel("Number 2 : ");
        lbl2.setBounds(120, 120, 70, 30);
        txt2 = new JTextField();
        txt2.setBounds(195, 120, 200, 30);
        JButton btnA = new JButton("Addition");
        btnA.setBounds(30, 160, 120, 40);
        btnA.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                double no1 = Double.parseDouble(txt1.getText());
                double no2 = Double.parseDouble(txt2.getText());
                double add = no1 + no2;
                lblResult.setText("Addition : " + add);
```

```
        }
    });
    JButton btnS = new JButton("Subtraction");
    btnS.setBounds(160, 160, 120, 40);
    btnS.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            double no1 = Double.parseDouble(txt1.getText());
            double no2 = Double.parseDouble(txt2.getText());
            double sub = no1 - no2;
            lblResult.setText("Subtraction : " + sub);
        }
    });
    JButton btnM = new JButton("Multiplication");
    btnM.setBounds(290, 160, 120, 40);
    btnM.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            double no1 = Double.parseDouble(txt1.getText());
            double no2 = Double.parseDouble(txt2.getText());
            double mul = no1 * no2;
            lblResult.setText("Multiplication : " + mul);
        }
    });
    JButton btnD = new JButton("Division");
    btnD.setBounds(420, 160, 120, 40);
    btnD.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            double no1 = Double.parseDouble(txt1.getText());
            double no2 = Double.parseDouble(txt2.getText());
            double div = no1 / no2;
            lblResult.setText("Division : " + div);
        }
    });
    lblResult = new JLabel("Result");
    lblResult.setBounds(80, 210, 150, 30);
    add(lbl1);
    add(txt1);
    add(lbl2);
    add(txt2);
    add(btnA);
    add(btnS);
    add(btnM);
    add(btnD);
    add(lblResult);
    setTitle("Calculator Application");
    setSize(800, 600);
    setLayout(null);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    }
}
```