

Unit -1 Introduction to Java [2 Hrs.]

Java is an object-oriented, cross platform, multi-purpose programming language produced by Sun Microsystems. It is a combination of features of C and C++ with some essential additional concepts. Java is well suited for both standalone and web application development and is designed to provide solutions to most of the problems faced by users of the internet era.

It was originally designed for developing programs for set-top boxes and handheld devices, but later became a popular choice for creating web applications.

The Java syntax is similar to C++, but is strictly an object-oriented programming language. For example, most Java programs contain classes, which are used to define objects, and methods, which are assigned to individual classes. Java is also known for being stricter than C++, meaning variables and functions must be explicitly defined. This means Java source code may produce errors or "exceptions" more easily than other languages, but it also limits other types of errors that may be caused by undefined variables or unassigned types.

Unlike Windows executables (.EXE files) or Macintosh applications (.APP files), Java programs are not run directly by the operating system. Instead, Java programs are interpreted by the Java Virtual Machine, or JVM, which runs on multiple platforms. This means all Java programs are multiplatform and can run on different platforms, including Macintosh, Windows, and Unix computers. However, the JVM must be installed for Java applications or applets to run at all. Fortunately, the JVM is included as part of the Java Runtime Environment (JRE).

Oracle acquired Sun Microsystems in January, 2010. Therefore, Java is now maintained and distributed by Oracle.

Features of Java

- a) **Object-Oriented** - Java supports the features of object-oriented programming. Its object model is simple and easy to expand.
- b) **Platform independent** - C and C++ are platform dependency languages hence the application programs written in one Operating system cannot run in any other Operating system, but in platform independence language like Java application programs written in one Operating system can able to run on any Operating system.
- c) **Simple** - Java has included many features of C / C ++, which makes it easy to understand.
- d) **Secure** - Java provides a wide range of protection from viruses and malicious programs. It ensures that there will be no damage and no security will be broken.
- e) **Portable** - Java provides us the concept of portability. Running the same program with Java on different platforms is possible.

- f) **Robust** - During the development of the program, it helps us to find possible mistakes as soon as possible.
- g) **Multi-threaded** - The multithreading programming feature in Java allows you to write a program that performs several different tasks simultaneously.
- h) **Distributed** - Java is designed for distributed Internet environments as it manages the TCP/IP protocol.

The most striking feature of the language is that it is a ***platform-neutral*** language. Java is the first programming language that is not tied to any particular hardware or operating system. Programs developed in Java can be executed anywhere on any stream. We can call Java as a revolutionary technology because it has brought in a fundamental shift in how we develop and use programs. Nothing like this has happened to the software industry before.

History of Java

Java is a general-purpose, object-oriented programming language developed by Sun Microsystems of USA in 1991. Originally called ***Oak*** by **James Gosling**, one of the inventors of the language, Java was designed for the development of software for consumer electronic devices like TVs, VCRs and such other electronic machines. The goal had a strong impact on the development team to make the language simple, portable and highly reliable. The Java team which included **Patrick Naughton** discovered that the existing languages like C and C++ had limitations in terms of both reliability and portability. However, they modelled their new language Java on C and C++ but removed a number of features of C and C++ that were considered as sources of problems and thus made Java a really simple, reliable, portable and powerful language.

Java Milestones

<i>Year</i>	<i>Development</i>
1990	Sun Microsystems decided to develop special software that could be used to manipulate consumer electronic devices. A team of Sun Microsystems programmers headed by James Gosling was formed to undertake this task.
1991	After exploring the possibility of using the most popular object-oriented language C++, the team announced a new language named "Oak".
1992	The team, known as Green Project team by Sun, demonstrated the application of their new language to control a list of home appliances using a hand-held device with a tiny touch-sensitive screen.
1993	The World Wide Web (WWW) appeared on the Internet and transformed the text-based Internet into a graphical-rich environment. The Green Project team came up with the idea of developing Web applets (tiny programs) using the new language that could run on all types of computers connected to Internet.
1994	The team developed a Web browser called "HotJava" to locate and run applet programs on Internet. HotJava demonstrated the power of the new language, thus making it instantly popular among the Internet users.
1995	Oak was renamed "Java", due to some legal snags. Java is just a name and is not an acronym. Many popular companies including Netscape and Microsoft announced their support to Java.
1996	Java established itself not only as a leader for Internet programming but also as a general-purpose, object-oriented programming language. Sun releases Java Development Kit 1.0.
1997	Sun releases Java Development Kit 1.1 (JDK 1.1).
1998	Sun releases the Java 2 with version 1.2 of the Software Development Kit (SDK 1.2).
1999	Sun releases Java 2 Platform, Standard Edition (J2SE) and Enterprise Edition (J2EE).
2000	J2SE with SDK 1.3 was released.
2002	J2SE with SDK 1.4 was released.
2004	J2SE with JDK 5.0 (instead of JDK 1.5) was released. This is known as J2SE 5.0.

The Internet and Java's Place in IT

Earlier Java was only used to design and program small computing devices, but it was later adopted as one of the platform-independent programming languages, and now according to Sun, 3 billion devices run Java.

Java is one of the most important programming languages in today's IT industries.

- **JSP** - In Java, JSP (Java Server Pages) is used to create dynamic web pages, such as in PHP and ASP.
- **Applets** - Applets are another type of Java programs that are implemented on Internet browsers and are always run as part of a web document.
- **J2EE** - Java 2 Enterprise Edition is a platform-independent environment that is a set of different protocols and APIs and is used by various organizations to transfer data between each other.
- **JavaBeans** - This is a set of reusable software components that can be easily used to create new and advanced applications.
- **Mobile** - In addition to the above technology, Java is widely used in mobile devices nowadays, many types of games and applications are being made in Java.

Types of Java Applications and Applets

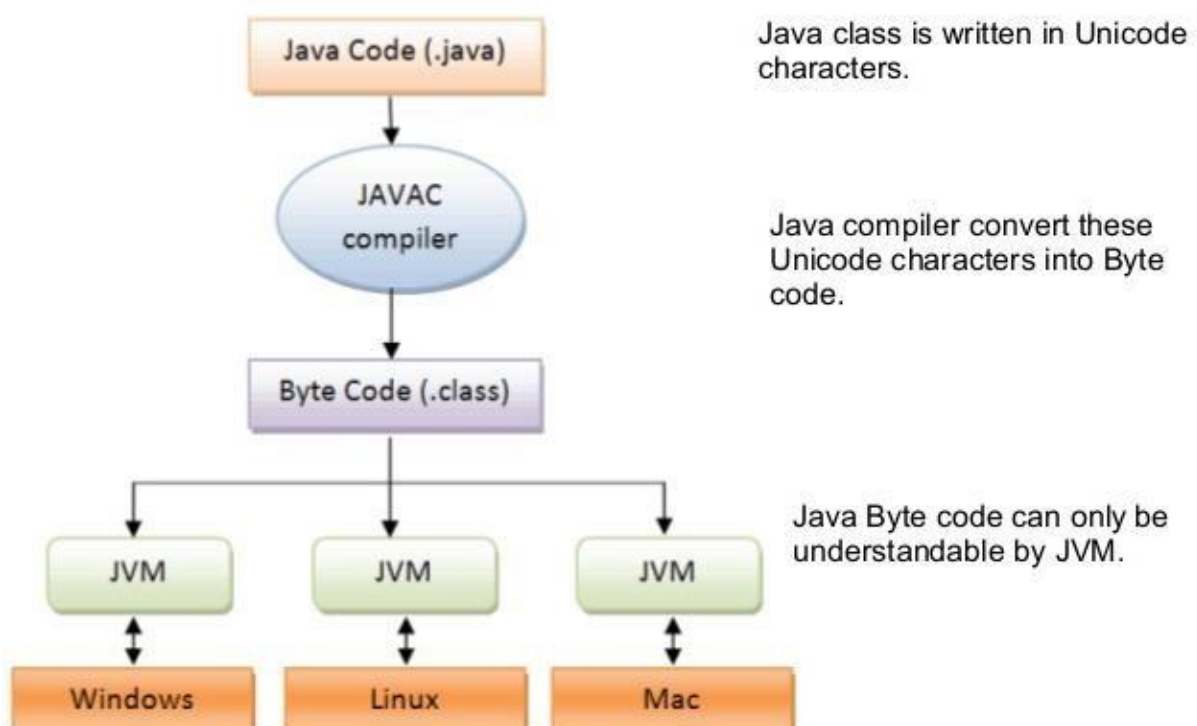
- a) **Web Application** - Java is used to create server-side web applications. Currently, Servlet, JSP, Struts, JSF, etc. technologies are used.

- b) **Standalone Application** - It is also known as the desktop application or windowbased application. An application that we need to install on every machine or server such as media player, antivirus, etc. AWT and Swing are used in java for creating standalone applications.
- c) **Enterprise Application** - An application that is distributed in nature, such as banking applications, etc. It has the advantage of the high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.
- d) **Mobile Application** - Java is used to create application software for mobile devices. Currently, Java ME is used for building applications for small devices, and also Java is a programming language for Google Android application development.

Java Virtual Machine

Java Virtual Machine, or JVM as its name suggest is a “virtual” computer that resides in the “real” computer as a software process. JVM gives Java the flexibility of platform independence.

Java code is written in .java file. This code contains one or more Java language attributes like Classes, Methods, Variable, Objects etc. Javac is used to compile this code and to generate .class file. Class file is also known as “byte code “. The name byte code is given may be because of the structure of the instruction set of Java program. Java byte code is an input to Java Virtual Machine. JVM read this code and interpret it and executes the program.



The JVM has two primary functions: to allow Java programs to run on any device or operating system (known as the "Write once, run anywhere" principle), and to manage

and optimize program memory. When Java was released in 1995, all computer programs were written to a specific operating system, and program memory was managed by the software developer. So the JVM was a revelation.



Fig-3.6 Process of compilation

The virtual machine code is not machine specific. The machine specific code (known as machine code) is generated by the Java interpreter by acting as an intermediary between the virtual machine and the real machine as shown in Fig. 3.7. Remember that the interpreter is different for different machines.



Fig. 3.7 Process of converting bytecode into machine code

Byte Code – not an Executable Code

Java bytecode is the resulting compiled object code of a Java program. This bytecode can be run in any platform which has a Java installation in it. The Java bytecode gets processed by the Java virtual machine (JVM) instead of the processor. It is the job of the JVM to make the necessary resource calls to the processor in order to run the bytecode. The Java bytecode is not completely compiled, but rather just an intermediate code sitting in the middle because it still has to be interpreted and executed by the JVM installed on the specific platform such as Windows, Mac or Linux. Upon compile, the Java source code is converted into the .class bytecode.

Procedure-Oriented vs. Object-Oriented Programming

Object-Oriented Programming (OOP) is a high-level programming language where a program is divided into small chunks called objects using the object-oriented model, hence the name. This paradigm is based on objects and classes.

- **Object** – An object is basically a self-contained entity that accumulates both data and procedures to manipulate the data. Objects are merely instances of classes.
- **Class** – A class, in simple terms, is a blueprint of an object which defines all the common properties of one or more objects that are associated with it. A class can be used to define multiple objects within a program.

The OOP paradigm mainly eyes on the data rather than the algorithm to create modules by dividing a program into data and functions that are bundled within the objects. The modules cannot be modified when a new object is added restricting any non-member function access to the data. Methods are the only way to assess the data.

Objects can communicate with each other through same member functions. This process is known as message passing. This anonymity among the objects is what makes the program secure. A programmer can create a new object from the already existing objects by taking most of its features thus making the program easy to implement and modify.

Procedure-Oriented Programming (POP) follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions. Each step is carried out in order in a systematic manner so that a computer can understand what to do. The program is divided into small parts called functions and then it follows a series of computational steps to be carried out in order. It follows a top-down approach to actually solve a problem, hence the name. Procedures correspond to functions and each function has its own purpose. Dividing the program into functions is the key to procedural programming. So a number of different functions are written in order to accomplish the tasks.

Difference between OOP and POP is shown below:

Procedure Oriented Programming	Object Oriented Programming
In POP, program is divided into small parts called functions .	In OOP, program is divided into parts called objects .
In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a real world .
POP follows Top Down approach .	OOP follows Bottom Up approach .
POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.

In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
POP does not have any proper way for hiding data so it is less secure .	OOP provides Data Hiding so provides more security .
In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Unit – 2 Tokens, Expressions and Control Structures [5 Hrs.]

Primitive Data Types

Primitive types are the most basic data types available within the Java language. There are 8 primitive data types: boolean, byte, char, short, int, long, float and double. These types serve as the building blocks of data manipulation in Java. Such types serve only one purpose — containing pure, simple values of a kind. Because these data types are defined into the Java type system by default, they come with a number of operations predefined.

Data types are divided into two groups:

- Primitive data types - includes byte, short, int, long, float, double, boolean and char
- Non-primitive(Derived) data types - such as String, Arrays and Classes

Data Type	Size	Description
byte	1 byte	Stores whole numbers from -128 to 127
short	2 bytes	Stores whole numbers from -32,768 to 32,767
int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,808
float	4 bytes	Stores fractional numbers from 3.4e-038 to 3.4e+038. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers from 1.7e-308 to 1.7e+038. Sufficient for storing 15 decimal digits
boolean	1 byte	Stores true or false values
char	2 bytes	Stores a single character/letter

User Defined Data Types

User defined data types are those that user / programmer himself defines. For example, classes, interfaces. For example:

MyClass obj

Here ***obj*** is a variable of data type ***MyClass*** and we call them reference variables as they can be used to store the reference to the object of that class.

Declaration of Variables and Assignment

In Java, variables are the names of storage locations. After designing suitable variable names, we must declare them to the compiler. Declaration does three things:

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.
3. The place of declaration (in the program) decides the scope of the variable.

A variable can be used to store a value of any data type. That is, the name has nothing to do with the type. Java allows any properly formed variable to have any declared data type. The general form of declaration of a variable is:

type variable1, variable2,....., variable

A simple method of giving value to a variable is through the assignment statement as follows: ***variableName = value;*** For Example: ***abc = 100;***
xyz = 10.2;

It is also possible to assign a value to a variable at the time of its declaration. This takes the form:

type variableName = value;

Examples:

```
int    finalValue = 100;
char   yes        = 'x';
double total      = 75.36;
```

Type Conversion and Casting

Conversion of one data type to another data type is called type casting. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an ***int*** value to a ***long*** 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 Automatic Conversions

When one type of data is assigned to another type of variable, an 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.

When these two conditions are met, a widening conversion takes place. For example, the ***int*** type is always large enough to hold all valid ***byte*** values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, there are no automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other. As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, long, or char.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a **narrowing conversion**, since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A cast is simply an explicit type conversion. It has this general form: **(target-type) value**

Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a byte, it will be reduced modulo (the remainder of an integer division by the) byte's range.

```
int a;  
byte b;  
// ... b = (byte) a;
```

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.  
class Conversion {  
    public static void main(String args[]) {  
        byte b;  
        int i = 257;  
        double d = 323.142;  
  
        System.out.println("\nConversion of int to byte.");  
        b = (byte) i;  
        System.out.println("i and b " + i + " " + b);  
  
        System.out.println("\nConversion of double to int.");  
        i = (int) d;  
        System.out.println("d and i " + d + " " + i);  
  
        System.out.println("\nConversion of double to byte.");  
        b = (byte) d;  
        System.out.println("d and b " + d + " " + b);  
    }  
}
```

This program generates the following output:

```
Conversion of int to byte.  
i and b 257 1  
  
Conversion of double to int.  
d and i 323.142 323  
  
Conversion of double to byte.  
d and b 323.142 67
```

Let's look at each conversion. When the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case. When the d is converted to an **int**, its fractional component is lost. When d is converted to a **byte**, its fractional component is lost, and the value is reduced modulo 256, which in this case is 67.

Garbage Collection

Java garbage collection is the process by which Java programs perform automatic memory management. Java programs compile to bytecode that can be run on a Java Virtual Machine, or JVM for short. When Java programs run on the JVM, objects are created on the heap, which is a portion of memory dedicated to the program. Eventually, some objects will no longer be needed. The garbage collector finds these unused objects and deletes them to free up memory.

In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing **OutOfMemoryErrors**.

But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects. Main objective of Garbage Collector is to free heap memory by destroying unreachable objects.

Generally, an object becomes eligible for garbage collection in Java on following cases:

1. All references to that object explicitly set to null e.g. object = null
2. The object is created inside a block and reference goes out scope once control exit that block.
3. Parent object set to null if an object holds the reference to another object and when you set container object's reference null, child or contained object automatically becomes eligible for garbage collection.

Operators in Java

An operator, in Java, is a special symbols performing specific operations on one, two or three operands and then returning a result.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators. Assume integer variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0

++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

Relational Operators

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

Bitwise Operators

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows –

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators.
Assume integer variable A holds 60 and variable B holds 13 then –

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
(bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

Logical Operators

The following table lists the logical operators.
Assume Boolean Variables A holds true and variable B holds false, then –

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
(logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

Assignment Operators

Following are the assignment operators supported by Java language –

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	$C = A + B$ will assign value of $A + B$ into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	$C += A$ is equivalent to $C = C + A$
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	$C -= A$ is equivalent to $C = C - A$
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	$C *= A$ is equivalent to $C = C * A$
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	$C /= A$ is equivalent to $C = C / A$
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	$C \% = A$ is equivalent to $C = C \% A$
<<=	Left shift AND assignment operator.	$C <<= 2$ is same as $C = C << 2$
>>=	Right shift AND assignment operator.	$C >>= 2$ is same as $C = C >> 2$
&=	Bitwise AND assignment operator.	$C \&= 2$ is same as $C = C \& 2$
^=	bitwise exclusive OR and assignment operator.	$C \wedge= 2$ is same as $C = C \wedge 2$
=	bitwise inclusive OR and assignment operator.	$C = 2$ is same as $C = C 2$

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 operator is written as –

```
variable x = (expression) ? value if true : value if false
```

Following is an example –

```
public class Test {  
  
    public static void main(String args[]) {  
        int a, b;  
        a = 10;  
        b = (a == 1) ? 20: 30;  
        System.out.println( "Value of b is : " + b );  
  
        b = (a == 10) ? 20: 30;  
        System.out.println( "Value of b is : " + b );  
    }  
}
```

Output

```
Value of b is : 30  
Value of b is : 20
```

Control Statements

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump.

- **Selection statements** allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- **Iteration statements** enable program execution to repeat one or more statements (that is, iteration statements form loops).
- **Jump statements** allow your program to execute in a nonlinear fashion.

Java's Selection Statements

Java supports two selection statements: if and switch. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

if

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the if statement:

```
if (condition) statement1;  
else statement2;
```

The if works like this: If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```

int a, b;
// ...
if(a < b) a = 0;
else b = 0;

```

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero. In no case are they both set to zero.

Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example:

```

if(i == 10) {
    if(j < 20) a = b;
    if(k > 100) c = d; // this if is
    else a = c;       // associated with this else
}
else a = d;           // this else refers to if(i == 10)

```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-elseif ladder. It looks like this:

```

if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
.
.
.
else
    statement;

```

Here is a program that uses an if-else-if ladder to determine which season a particular month is in.


```
// Demonstrate if-else-if statements.
class IfElse {
    public static void main(String args[]) {
        int month = 4; // April
        String season;

        if(month == 12 || month == 1 || month == 2)
            season = "Winter";
        else if(month == 3 || month == 4 || month == 5)
            season = "Spring";
        else if(month == 6 || month == 7 || month == 8)
            season = "Summer";
        else if(month == 9 || month == 10 || month == 11)
            season = "Autumn";
        else
            season = "Bogus Month";

        System.out.println("April is in the " + season + ".");
    }
}
```

Here is the output produced by the program:

```
April is in the Spring.
```

switch

The switch statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
switch (expression) {
    case value1:
        // statement sequence
        break;
    case value2:
        // statement sequence
        break;
    .
    .
    .
    case valueN:
        // statement sequence
        break;
    default:
        // default statement sequence
}
```

Here is a simple example that uses a switch statement:

```
// A simple example of the switch.
class SampleSwitch {
    public static void main(String args[]) {
        for(int i=0; i<6; i++)
            switch(i) {
                case 0:
                    System.out.println("i is zero.");
                    break;
                case 1:
                    System.out.println("i is one.");
                    break;
                case 2:
                    System.out.println("i is two.");
                    break;
                case 3:
                    System.out.println("i is three.");
                    break;
                default:
                    System.out.println("i is greater than 3.");
            }
    }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater than 3.
i is greater than 3.
```

Iteration Statements

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. A loop statement allows us to execute a statement or group of statements multiple times.

for loop

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to be executed a specific number of times. A for loop is useful when you know how many times a task is to be repeated.

The syntax of a for loop is –

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

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

```

public class Test {

    public static void main(String args[]) {

        for(int x = 10; x < 20; x = x + 1) {
            System.out.print("value of x : " + x );
            System.out.print("\n");
        }
    }
}

```

Output

```

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

```

while Loop

A while loop statement in Java programming language repeatedly executes a target statement as long as a given condition is true. The syntax of a while loop is –

```

while(Boolean_expression) {

    // Statements

}

```

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

Example:

```

public class Test {

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

        while( x < 20 ) {
            System.out.print("value of x : " + x );
            x++;
            System.out.print("\n");
        }
    }
}

```

Output

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

do while loop

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Following is the syntax of a do...while loop –

```
do {
    // Statements
}while(Boolean_expression);
```

Example

```
public class Test {

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

        do {
            System.out.print("value of x : " + x );
            x++;
            System.out.print("\n");
        }while( x < 20 );
    }
}
```

Output

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

Nested Loops

Like all other programming languages, Java allows loops to be nested. That is, one loop may be inside another. For example, here is a program that nests for loops:

```
// Loops may be nested.
class Nested {
    public static void main(String args[]) {
        int i, j;

        for(i=0; i<10; i++) {
            for(j=i; j<10; j++)
                System.out.print(".");
            System.out.println();
        }
    }
}
```

The output produced by this program is shown here:

```
.....
.....
.....
.....
.....
.....
.....
....
...
..
.
```

Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

Using break

In Java, the break statement has three uses. First, as you have seen, it terminates a statement sequence in a switch statement. Second, it can be used to exit a loop. Third, it can be used as a “civilized” form of goto.

Using break to Exit a Loop

By using break, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a break statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
    public static void main(String args[]) {
        for(int i=0; i<100; i++) {
            if(i == 10) break; // terminate loop if i is 10
            System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

As you can see, although the for loop is designed to run from 0 to 99, the break statement causes it to terminate early, when *i* equals 10.

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop’s end. The continue statement performs such an action.

In a while and do-while loops, a continue statement causes control to be transferred directly to the conditional expression that controls the loop. In a for loop, control goes first to the

iteration portion of the for statement and then to the conditional expression. For all three loops, any intermediate code is bypassed.

Here is an example program that uses continue to cause two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
    public static void main(String args[]) {
        for(int i=0; i<10; i++) {
            System.out.print(i + " ");
            if (i%2 == 0) continue;
            System.out.println("");
        }
    }
}
```

This code uses the % operator to check if i is even. If it is, the loop continues without printing a newline. Here is the output from this program:

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

Using return

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

Example:

```
class A{
    int a,b,sum;
    public int add() {
        a=10;
        b=15;
        sum=a+b;
        return sum;
    }
}

class B {
    public static void main(String[] args) {
        A obj=new A();
        int res=obj.add();
        System.out.println("Sum of two numbers="+res);
    }
}
```

Output:

Sum of two numbers=25

Unit – 3 Object Oriented Programming Concepts [9 Hrs.]

Fundamentals of a Class

A class, in the context of Java, are templates that are used to create objects, and to define object data types and methods. Core properties include the data types and methods that may be used by the object. All class objects should have the basic class properties. Classes are categories, and objects are items within each category.

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers** : A class can be public or has default access.
2. **Class name**: The name should begin with a initial letter (capitalized by convention).
3. **Superclass(if any)**: The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any)**: A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body**: The class body surrounded by braces, {}.

General form of a class is shown below:



Java Object

A Java object is a combination of data and procedures working on the available data. An object has a state and behavior. The state of an object is stored in fields (variables), while methods

(functions) display the object's behavior. Objects are created from templates known as classes. In Java, an object is created using the keyword "new". Object is an instance of a class. There are three steps to creating a Java object:

1. Declaration of the object
2. Instantiation of the object
3. Initialization of the object

When a Java object is declared, a name is associated with that object. The object is instantiated so that memory space can be allocated. Initialization is the process of assigning a proper initial value to this allocated space. The properties of Java objects include:

- One can only interact with the object through its methods. Hence, internal details are hidden.
- When coding, an existing object may be reused.
- When a program's operation is hindered by a particular object, that object can be easily removed and replaced.

A new object `t` from the class "tree" is created using the following syntax: **Tree
t = new Tree ();**

Below example shows the implementation of class and object.

```
class Box{
    double width,height,depth,vol; //data members
    void getvolume() { //method
        width=10;
        height=20;
        depth=11.5;
        vol=width*height*depth;
    }
    void display() { //method
        System.out.println("Volume of a box is "+vol);
    }
}

class Volume {
    public static void main(String[] args) {
        Box obj=new Box(); //creating instance of a class Box
        obj.getvolume(); //calling member function using object
        obj.display();
    }
}
```

Output

Volume of a box is 2300.0

Method That Returns Value

```
class A{
    int a,b,sum;
    public int add() {
        a=10;
        b=15;
        sum=a+b;
        return sum;
    }
}

class B {
    public static void main(String[] args) {
        A obj=new A();
        int res=obj.add();
        System.out.println("Sum of two numbers="+res);
    }
}
```

Output:

Sum of two numbers=25

```
class Box{
    double width,height,depth,vol; //data members
    void getvolume() { //method
        width=10;
        height=20;
        depth=11.5;
    }
    double calculate() { //method
        vol=width*height*depth;
        return vol;
    }
}

class Volume {
    public static void main(String[] args) {
        Box obj=new Box(); //creating instance of a class Box
        obj.getvolume(); //calling member function using object
        double res=obj.calculate(); //calling
        System.out.println("Volume of a box is "+res);
    }
}
```

Output:

Volume of a box is 2300.0

Method That Takes Parameters

Passing by value

Actual parameter expressions that are passed to a method are evaluated and a value is derived. Then this value is stored in a location and then it becomes the formal parameter to the invoked method. This mechanism is called pass by value and Java uses it.

```
class Box{
    double vol; //data members
    void getvolume(double width, double height, double depth) { //method
        vol=width*height*depth;
    }
    void display() { //method
        System.out.println("Volume of a box is "+vol);
    }
}

class Volume {
    public static void main(String[] args) {
        Box obj=new Box(); //creating instance of a class Box
        obj.getvolume(10,20,11.5); //calling member function using object
        obj.display(); //calling
    }
}
```

Output:

Volume of a box is 2300.0

Another Example

```

class Box{
    double width,height,depth,vol; //data members
    void getdata(double w, double h, double d) { //method
        width=w;
        height=h;
        depth=d;
    }
    double calculate() { //method
        vol=width*height*depth;
        return vol;
    }
}

class Volume {
    public static void main(String[] args) {
        Box obj=new Box(); //creating instance of a class Box
        obj.getdata(10,20,11.5); //calling member function using object
        double res=obj.calculate(); //calling
        System.out.println("Volume of a box is "+res);
    }
}

```

Output:

Volume of a box is 2300.

Passing by Reference

We know that in C, we can pass arguments by reference using pointers, and same can be done in C++ using references.

Java is pass by value and it is not possible to pass primitives by reference in Java. Also integer class is immutable in java and java objects are references that are passed by value. So an integer object points to the exact same object as in the caller and but no changes can be made to the object which gets reflected in the caller function.

Constructors

A *constructor* in Java is a block of code similar to a method that's called when an instance of an object is created. Here are the key differences between a constructor and a method:

- A constructor doesn't have a return type.
- The name of the constructor must be the same as the name of the class.
- Unlike methods, constructors are not considered members of a class.
- A constructor is called automatically when a new instance of an object is created.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used. Following is the syntax of a constructor –

```

class ClassName {
    ClassName() {
    }
}

```

Default Constructor

The *default constructor* is a constructor that is automatically generated in the absence of explicit constructors (i.e. no user defined constructor). The automatically provided constructor is called sometimes a *nullary* constructor.

Constructor with no Arguments

```
class Box{
    double l,b,h,vol;
    Box(){
        l=10;
        b=5;
        h=3.3;
    }
    void calculate() {
        vol=l*b*h;
        System.out.println("Volume of a box is "+vol);
    }
}

public class Constructor {
    public static void main(String[] args) {
        Box obj=new Box();
        obj.calculate();
    }
}
```

Output:

Volume of a box is 165.0

Parameterized Constructor

The constructor which has parameters or arguments is known as parameterized constructor. In this type of constructor, we should supply/pass arguments while defining object of a class. The values of arguments are assigned to data members of the class.

```
class Box{
    double l,b,h,vol;
    Box(double x, double y, double z){
        l=x;
        b=y;
        h=z;
    }
    void calculate() {
        vol=l*b*h;
        System.out.println("Volume of a box is "+vol);
    }
}

public class Constructor {
    public static void main(String[] args) {
        Box obj=new Box(10,5,3.3);
        obj.calculate();
    }
}
```

Output:

Volume of a box is 165.0

The this Keyword

Keyword `this` is a reference variable in Java that refers to the current object.

The various usages of 'THIS' keyword in Java are as follows:

- It can be used to refer instance variable of current class
- It can be used to invoke or initiate current class constructor
- It can be passed as an argument in the method call
- It can be passed as argument in the constructor call
- It can be used to return the current class instance

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines `this` keyword. `this` can be used inside any method to refer to the current object. That is, `this` is always a reference to the object on which the method was invoked. You can use `this` anywhere a reference to an object of the current class' type is permitted.

```
// A redundant use of this.
Box(double w, double h, double d) {
    this.width = w;
    this.height = h;
    this.depth = d;
}

//Java code for using 'this' keyword to
//refer current class instance variables
class Test
{
    int a;
    int b;

    // Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    void display()
    {
        //Displaying value of variables a and b
        System.out.println("a = " + a + " b = " + b);
    }

    public static void main(String[] args)
    {
        Test object = new Test(10, 20);
        object.display();
    }
}
```

Output:

```
a = 10  b = 20
```

Four Main Features of Object Oriented Programming:

1. Abstraction
2. Encapsulation
3. Polymorphism
4. Inheritance

Abstraction

Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it. In Java, abstraction is achieved using Abstract classes and interfaces.

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain **abstract methods**, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must** be declared abstract.
- If a class is declared abstract, it cannot be **instantiated** (We cannot create object).
- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

```

abstract class Box{
    int l,b,h,vol;
    void getdata() {
        l=10;
        b=5;
        h=2;
    }
    void calculate() {
        vol=l*b*h;
        System.out.println("Volume of a box is "+vol);
    }
}

public class Abstraction extends Box{
    public static void main(String[] args) {
        Abstraction obj=new Abstraction();
        obj.getdata();
        obj.calculate();
    }
}

```

Output:

Volume of a box is 100

Encapsulation

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

To achieve encapsulation in Java –

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

Benefits of Encapsulation

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.


```

class Test{
    private String name;
    private int age;

    public String getname() {
        return name;
    }
    public int getage() {
        return age;
    }

    public void setname(String nm){
        name=nm;
    }
    public void setage(int ag) {
        age=ag;
    }
}

public class Encapsulation {
    public static void main(String args) {
        Test obj=new Test();
        obj.setname("Raaju Poudel");
        obj.setage(27);
        System.out.println("Name: "+obj.getname());
        System.out.println("Age: "+obj.getage());
    }
}

```

Output:

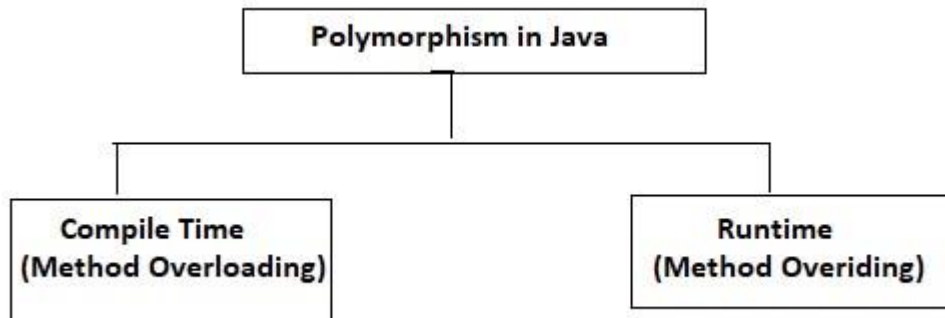
Name: Raaju Poudel
Age: 27

Here, we cannot access private data members name and age directly. So we have created public getter and setter methods to access private data members.

Polymorphism

Polymorphism in Java is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.



Method Overloading

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists. In order to overload a method, the argument lists of the methods must differ in either of these:

a) Number of parameters.

add(int, int) add(int,
int, int)

b) Data type of parameters.

add(int, int) add(int,
float)

c) Sequence of Data type of parameters.

add(int, float) add(float,
int)

**Example of
method overloading:**

```

class Overload
{
    void demo (int a)
    {
        System.out.println ("a: " + a);
    }
    void demo (int a, int b)
    {
        System.out.println ("a and b: " + a + "," + b);
    }
    double demo(double a) {
        System.out.println("double a: " + a);
        return a*a;
    }
}
class MethodOverloading
{
    public static void main (String args [])
    {
        Overload Obj = new Overload();
        double result;
        Obj .demo(10);
        Obj .demo(10, 20);
        result = Obj .demo(5.5);
        System.out.println("O/P : " + result);
    }
}

```

Output:

```

a: 10
a and b: 10,20
double a: 5.5
O/P : 30.25

```

Here the method **demo()** is overloaded 3 times: first method has **1 int** parameter, second method has **2 int** parameters and third one is having **double** parameter. Which method is to be called is determined by the arguments we pass while calling methods. This happens at compile time so this type of polymorphism is known as compile time polymorphism.

Method Overriding

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a

method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method.

Let's take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat (). Boy class is giving its own implementation to the eat () method or in other words it is overriding the eat () method.

```
class Human{
    //Overridden method
    public void eat()
    {
        System.out.println("Human is eating");
    }
}
class Boy extends Human{
    //Overriding method
    public void eat(){
        System.out.println("Boy is eating");
    }
    public static void main( String args[]) {
        Boy obj = new Boy();
        //This will call the child class version of eat()
        obj.eat();
    }
}
```

Output:

Boy is eating

Rules of method overriding in Java

1. **Argument list:** The argument list of overriding method (method of child class) must match the Overridden method (the method of parent class). The data types of the arguments and their sequence should exactly match.
2. **Access Modifier** of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier, because all of these three access modifiers are more restrictive than public.

Another Example:

Animal.java

```
public class Animal{  
    public void sound(){  
        System.out.println("Animal is making a sound");  
    }  
}
```

Horse.java

```
class Horse extends Animal{  
    @Override  
    public void sound(){  
        System.out.println("Neigh");  
    }  
    public static void main(String args[]){  
        Animal obj = new Horse();  
        obj.sound();  
    }  
}
```

Output:

```
Neigh
```

Recursion

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 factorial is $1 \times 2 \times 3$, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
    // this is a recursive method
    int fact(int n) {
        int result;

        if(n==1) return 1;
        result = fact(n-1) * n;
        return result;
    }
}

class Recursion {
    public static void main(String args[]) {
        Factorial f = new Factorial();

        System.out.println("Factorial of 3 is " + f.fact(3));
        System.out.println("Factorial of 4 is " + f.fact(4));
        System.out.println("Factorial of 5 is " + f.fact(5));
    }
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

Introducing 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.

There are two types of nested classes: *static* and *non-static*. A static nested class is one that has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

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.

The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test()**, and defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
    int outer_x = 100;

    void test() {
        Inner inner = new Inner();
        inner.display();
    }

    // this is an inner class

    class Inner {
        void display() {
            System.out.println("display: outer_x = " + outer_x);
        }
    }
}

class InnerClassDemo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        outer.test();
    }
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display()** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main()** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test()** method. That method creates an instance of class **Inner** and the **display()** method is called.

Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: access control. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a welldefined set of methods, you can prevent the misuse of that data.

Java's access specifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved.

Unit – 4 Inheritance & Packaging [3 Hrs.]

Inheritance is an important pillar of OOP (Object Oriented Programming). It is the mechanism in java by which one class is allow to **inherit** the features (**fields and methods**) of another class.

The process by which one class acquires the properties (data members) and functionalities(methods) of another class is called **inheritance**. The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class. The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parentchild* relationship. Inheritance is used in java for the following:

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms used in Inheritance ○ **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. ○ **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.

- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

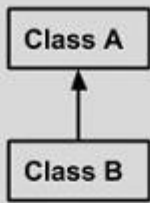
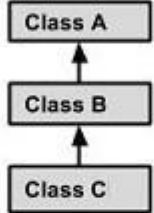
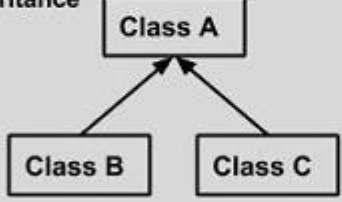
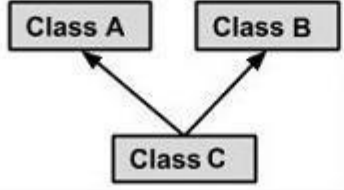
The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

Types of inheritance in java

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.

Single Inheritance  <pre> graph BT B[Class B] --> A[Class A] </pre>	<pre> public class A { } public class B extends A { } </pre>
Multi Level Inheritance  <pre> graph BT C[Class C] --> B[Class B] B --> A[Class A] </pre>	<pre> public class A {} public class B extends A {.....} public class C extends B {.....} </pre>
Hierarchical Inheritance  <pre> graph BT B[Class B] --> A[Class A] C[Class C] --> A </pre>	<pre> public class A {} public class B extends A {.....} public class C extends A {.....} </pre>
Multiple Inheritance  <pre> graph BT C[Class C] --> A[Class A] C --> B[Class B] </pre>	<pre> public class A {} public class B {.....} public class C extends A,B { } // Java does not support mutiple Inheritance </pre>

1. Single Inheritance

Single Inheritance refers to a child and parent class relationship where a class extends the another class.

```

class Animal{ void
    eat(){
        System.out.println("eating...");
    }
}
class Dog extends Animal{ void
    bark(){
        System.out.println("barking...");
    } } class TestInheritance{ public
static void main(String args[]){ Dog d=new
Dog(); d.bark();
    d.eat();
}
}

```

Output:

```
barking... eating...
```

2. Multilevel Inheritance

Multilevel inheritance refers to a child and parent class relationship where a class extends the child class. For example, class C extends class B and class B extends class A.

```
class Animal{ void
    eat(){
        System.out.println("eating...");
    }
}
class Dog extends Animal{ void
    bark(){
        System.out.println("barking...");
    }
}
class BabyDog extends Dog{ void
    weep(){
        System.out.println("weeping...");
    } } class TestInheritance2{ public
static void main(String args[]){ BabyDog
d=new BabyDog(); d.weep();
    d.bark();
    d.eat();
}
}
```

Output

```
weeping...
barking... eating...
```

3. Hierarchical Inheritance

Hierarchical inheritance refers to a child and parent class relationship where more than one classes extends the same class. For example, classes B, C & D extends the same class A.

```
class Animal{ void
    eat(){
        System.out.println("eating...");
    }
}
```

```

    }
}
class Dog extends Animal{ void
    bark(){
        System.out.println("barking...");
    }
}
class Cat extends Animal{
    void meow(){
        System.out.println("meowing..."); } }
class TestInheritance3{ public static void
main(String args[]){ Cat c=new Cat();
c.meow();
c.eat();
//c.bark(); //Error //To access property of class Dog create object of class Dog
Dog d=new Dog(); d.bark();
}
}

```

Output

```

weeping...
barking... eating...

```

4. Multiple Inheritance

When one class extends more than one classes then this is called multiple inheritance. For example: Class C extends class A and B then this type of inheritance is known as multiple inheritance.

Java doesn't allow multiple inheritance. We can use **interfaces** instead of **classes** to achieve the same purpose.

Interface in Java

Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body).

- ❖ Interfaces specify what a class must do and not how. It is the blueprint of the class.
- ❖ An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move (). So it specifies a set of methods that the class has to implement.
- ❖ If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.

Syntax

```
interface <interface_name> {
```

```
// declare constant fields
// declare methods that abstract      // by default.
}
```

To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

Why do we use interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.

```
// A simple interface  interface
Player
{
    final int id =
10;        int move();
}
```

Example of Interface

```

// Java program to demonstrate working of
// interface.
import java.io.*;

// A simple interface
interface in1
{
    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}

// A class that implements interface.
class testClass implements in1
{
    // Implementing the capabilities of
    // interface.
    public void display()
    {
        System.out.println("Geek");
    }

    // Driver Code
    public static void main (String[] args)
    {
        testClass t = new testClass();
        t.display();
        System.out.println(a);
    }
}

```

Output:

Geek

10

Use of Interface to achieve multiple inheritance

Let us say we have two interfaces A & B and two classes C & D. Then we can achieve multiple inheritance using interfaces as follows:

```
interface A{ } interface B{ } class  
C{ } class D extends C implements  
A,B { }
```

Example Program

```
interface interface1{  
    void display1();  
}  
  
interface interface2{  
    void display2();  
}  
  
class A{  
    void display3() {  
        System.out.println("I am inside class");  
    }  
}  
class B extends A implements interface1,interface2{  
    public void display1() {  
        System.out.println("I am inside interface1");  
    }  
    public void display2() {  
        System.out.println("I am inside interface2");  
    }  
}  
  
public class MultiLevel {  
    public static void main(String[] args) {  
        B obj=new B();  
        obj.display1();  
        obj.display2();  
        obj.display3();  
    }  
}
```

Output

```
I am inside interface1  
I am inside interface2  
I am inside class
```

Another Example

```

interface interface1{
    int a=20,b=10;
    void add();
}

class Ax{
    int diff;
    void subtract(int x,int y) {
        diff=x-y;
        System.out.println("Subtraction= "+diff);
    }
}

class Bx extends Ax implements interface1{
    int sum;
    public void add() {
        sum=a+b;
        System.out.println("Addition= "+sum);
    }
}

public class Multilevel {
    public static void main(String[] args) {
        Bx obj=new Bx();
        obj.add();
        obj.subtract(50, 10);
    }
}

```

Output

Addition= 30

Subtraction= 40

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.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
    void callme() {
        System.out.println("Inside A's callme method");
    }
}

class B extends A {
    // override callme()
    void callme() {
        System.out.println("Inside B's callme method");
    }
}

class C extends A {
    // override callme()
    void callme() {
        System.out.println("Inside C's callme method");
    }
}

class Dispatch {
    public static void main(String args[]) {
        A a = new A(); // object of type A
        B b = new B(); // object of type B
        C c = new C(); // object of type C
        A r; // obtain a reference of type A

        r = a; // r refers to an A object
        r.callme(); // calls A's version of callme

        r = b; // r refers to a B object
        r.callme(); // calls B's version of callme

        r = c; // r refers to a C object
        r.callme(); // calls C's version of callme
    }
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one **superclass** called **A** and **two subclasses** of it, called **B** and **C**. **Subclasses B and C** override **callme()** declared in A. Inside the **main()** method, objects of type **A, B, and C** are declared. Also, a reference of **type A, called r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme()**

). As the output shows, the version of **callme()** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A's callme()** method.

Super Keyword

The **super** keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

1. Use of super with variables: This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

```
/* Base class vehicle */
class Vehicle
{
    int maxSpeed = 120;
}

/* sub class Car extending vehicle */
class Car extends Vehicle
{
    int maxSpeed = 180;

    void display()
    {
        /* print maxSpeed of base class (vehicle) */
        System.out.println("Maximum Speed: " + super.maxSpeed);
    }
}

/* Driver program to test */
class Test
{
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```

Output:

Maximum Speed: 120

In the above example, both base class and subclass have a member **maxSpeed**. We could access **maxSpeed** of base class in subclass using super keyword.

2. Use of super with methods: This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

```
/* Base class Person */
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
}

/* Subclass Student */
class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }

    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}

/* Driver program to test */
class Test
{
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

Output

This is student class

This is person class

In the above example, we have seen that if we only call method message () then, the current class message () is invoked but with the use of super keyword, message () of superclass could also be invoked.

3. Use of super with constructors: super keyword can also be used to access the parent class constructor. One more important thing is that, "super" can call both parametric as well

as non-parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

```
/* superclass Person */
class Person
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

/* subclass Student extending the Person class */
class Student extends Person
{
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}

/* Driver program to test*/
class Test
{
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

Output:

Person class Constructor
Student class Constructor

In the above example we have called the superclass constructor using keyword 'super' via subclass constructor.

Abstract and Final Classes

Abstract class has been discussed already in **Unit-3 (Abstraction)**.

The keyword **final** has **three** uses. First, it can be used to create the equivalent of a named constant. The other two uses of final apply to inheritance.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

```
class A {
    final void meth() {
        System.out.println("This is a final method.");
    }
}

class B extends A {
    void meth() { // ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

Because **meth ()** is declared as **final**, it cannot be overridden in B. If you attempt to do so, a compile-time error will result.

Using final to Prevent Inheritance

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

Here is an example of a final class:

```
final class A {
    // ...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
    // ...
}
```

As the comments imply, it is illegal for **B** to inherit **A** Since **A** is declared as final.

The Object Class

There is one special class, **Object**, defined by Java. All other classes are subclasses of **Object**. That is, **Object** is a superclass of all other classes. This means that a reference variable of type **Object** can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type **Object** can also refer to any array.

Object defines the following methods, which means that they are available in every object.

Method	Purpose
Object clone()	Creates a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize()	Called before an unused object is recycled.
Class getClass()	Obtains the class of an object at run time.
int hashCode()	Returns the hash code associated with the invoking object.
void notify()	Resumes execution of a thread waiting on the invoking object.
void notifyAll()	Resumes execution of all threads waiting on the invoking object.
String toString()	Returns a string that describes the object.
void wait() void wait(long milliseconds) void wait(long milliseconds, int nanoseconds)	Waits on another thread of execution.

The methods **getClass()**, **notify()**, **notifyAll()**, and **wait()** are declared as **final**. You may **override** the others. However, notice two methods now: **equals()** and **toString()**. The **equals()** method compares the contents of two objects. It returns **true** if the objects are equivalent, and **false** otherwise.

Package

A **java package** is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, **built-in package** and **user-defined package**. There are many **built-in** packages such as **java**, **lang**, **awt**, **javax**, **swing**, **net**, **io**, **util**, **sql** etc.

Advantage of Java Package

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collision.

The **package** keyword is used to create a package in java.

```
//save as Simple.java
package mypack; public
class Simple{
```

```

    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}

```

There are three ways to access the package from outside the package.

1. import package.*; 2. import package.classname;
3. fully qualified name.

If you use **package.*** then all the classes and interfaces of this package will be accessible but not subpackages.

The **import keyword** is used to make the classes and interface of another package accessible to the current package.

```
//save by A.java
```

```

package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}

```

```
//save by B.java
```

```

package mypack;
import pack.A;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}

```

```
Output:Hello
```

Subpackage in java

Package inside the package is called the **subpackage**.

```

package java.util.Scanner;  class
Simple{
    public static void main(String args[]){
        Scanner scan=new Scanner(System.in);

```

```
        System.out.println("Enter a number");
int a = scan.nextInt();
        System.out.println("Inputted number is: "+a);
    } }
```

Unit – 5 Handling Error/Exceptions [2 Hrs.]

An **exception** is an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time, that disrupts the normal flow of the program's instructions.

Error vs Exception

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

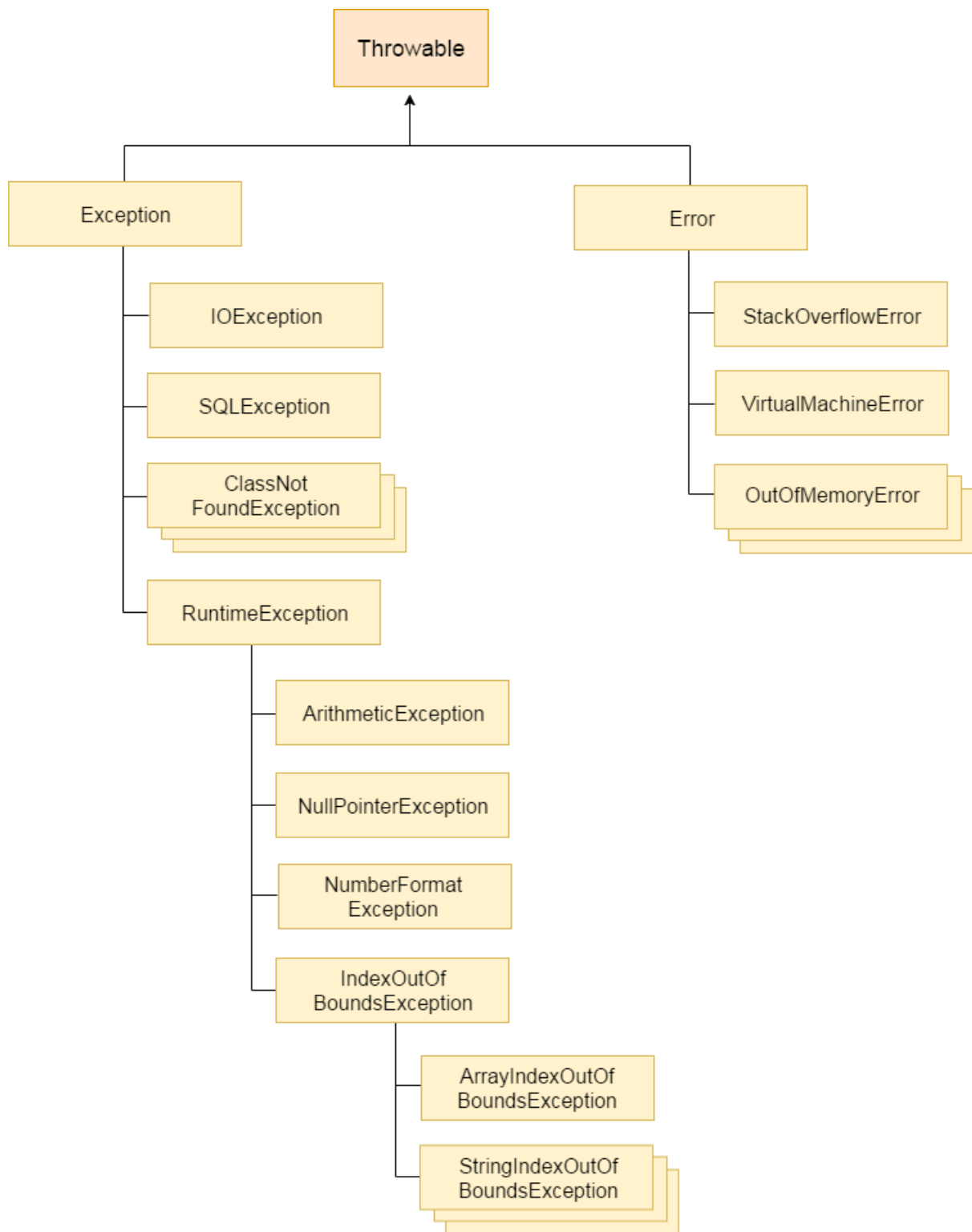
Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Based on these, we have three categories of Exceptions. You need to understand them to know how exception handling works in Java.

- **Checked exceptions** – A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored; the programmer should take care of (handle) these exceptions.
- **Unchecked exceptions** – An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.
- **Errors** – These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

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. A hierarchy of Java Exception classes are given below:



Java Exception Keywords

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
---------	-------------

try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.

```

public class JavaExceptionExample{
    public static void main(String args[]){
        try{
            //code that may raise exception
            int data=100/0;
        }catch(ArithmeticException e) {
            System.out.println(e);
        }
        //rest code of the program
        System.out.println("rest of the code...");
    }
}

```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero rest
of the code...
```

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;  
System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";  
int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];  
a[10]=50; //ArrayIndexOutOfBoundsException
```

Java finally block

Java finally block is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not. Java finally block follows try or catch block.

```
public class TestFinallyBlock2{  
    public static void main(String args[]){  
        try{  
            int data=25/0;  
            System.out.println(data);  
        }  
        catch(ArithmeticException e){  
            System.out.println(e);  
        }  
        finally{  
            System.out.println("finally block is always executed");  
        }  
        System.out.println("rest of the code...");  
    }  
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by
zero          finally block is always executed          rest of the
code...
```

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

```
public class MultipleCatchBlock1 {

    public static void main(String[] args) {
        try{
int a[]=new int[5];
a[5]=30/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException occurs");
        }
        catch(Exception e)
        {
            System.out.println("Parent Exception occurs");
        }
        System.out.println("rest of the code");
    }
}
```

Output:

```
Arithmetic Exception occurs rest
of the code
```

```
public class MultipleCatchBlock2 {

    public static void main(String[] args) {
        try{
int a[]=new int[5];
```

```

        System.out.println(a[10]);
    }
    catch(ArithmeticException e)
    {
        System.out.println("Arithmetic Exception occurs");
    }
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.out.println("ArrayIndexOutOfBoundsException occurs");
    }
    catch(Exception e)
    {
        System.out.println("Parent Exception occurs");
    }
    System.out.println("rest of the code");
}
}

```

Output:

```

ArrayIndexOutOfBoundsException occurs rest
of the code

```

Java Nested try block

The try block within a try block is known as nested try block in java. Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```

....
try
{
    statement          1;
    statement 2;
    try
    {
        statement          1;
        statement 2;
    }
    catch(Exception e)
{
    }
}
catch(Exception e)

```

```
{  
}  
....
```

Example:

```
class Excep6{    public static void  
main(String args[]){    try{    try{  
    System.out.println("going to divide");  
    int b =39/0;  
    }catch(ArithmeticException e){  
        System.out.println(e);  
    }  
    try{    int  
a[]=new int[5];  
a[5]=4;  
    }catch(ArrayIndexOutOfBoundsException e){  
        System.out.println(e);  
    }  
  
    System.out.println("other statement);  
}catch(Exception e){  
    System.out.println("handeled");  
}  
  
    System.out.println("normal flow..");  
}  
}
```

Java throw keyword

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

The syntax of java throw keyword is given below.

```
throw new IOException("sorry device error);
```

Example:

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class TestThrow1{    static void
validate(int age){        if(age<18)        throw
new ArithmeticException("not valid");    else
    System.out.println("welcome to vote");
}
    public static void main(String args[]){
validate(13);
    System.out.println("rest of the code...");
}
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException: not valid
```

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing checkup before the code being used.

Syntax of java throws return_type method_name() **throws**

```
exception_class_name{
//method code
}
```

Example of Java throws

```
import java.io.IOException;    class Testthrows1{    void
m()throws IOException{        throw new IOException("device
error");//checked exception
}
```

```

    void n()throws IOException{
m();    }    void p(){    try{
n();
    }catch(Exception e){
        System.out.println("exception handled");
    }
}
    public static void main(String args[]){
Testthrows1 obj=new Testthrows1();
obj.p();
    System.out.println("normal flow...");
}
}

```

Output:

```

exception handled normal
flow...

```

Difference between throw and throws in Java

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Unit – 5 Handling Strings [2 Hrs.]

String Creation

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:


```
char[] ch={'j','a','v','a','t','p','o','i','n','t'};
String s=new String(ch);
```

is same as:

```
String s="javatpoint";
```

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

```
String s="welcome";
```

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
String s1="Welcome";
```

```
String s2="Welcome";//It doesn't create a new instance
```

2) By new keyword

```
String s=new String("Welcome");//creates two objects and one reference variable
```

Java String Example

```
public class StringExample{ public
static void main(String args[]){
String s1="java";//creating string by java string literal char
ch[]={ 's','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output

```
java
strings
example
```

String Length

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

The following fragment prints "3", since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' }; String
s = new String(chars);
System.out.println(s.length());
```

Concatenation of String

The following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string "He is 9 years old."

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the `+` to concatenate them. Here is an example:

```
// Using concatenation to prevent long lines. class
ConCat {
    public static void main(String args[]) {
        String longStr = "This could have been " +
            "a very long line that would have " +
            "wrapped around. But string concatenation " +
            "prevents this.";
        System.out.println(longStr);
    }
}
```

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example: `int age = 9;`

```
String s = "He is " + age + " years old.";
System.out.println(s);
```

In this case, `age` is an `int` rather than another `String`, but the output produced is the same as before. This is because the `int` value in `age` is automatically converted into its string representation within a `String` object. This string is then concatenated as before.

Be careful when you mix other types of operations with string concatenation expressions, however, you might get surprising results. Consider the following:

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

This fragment displays **four: 22** rather than the **four: 4**

To complete the integer addition first, you must use parentheses, like this:

```
String s = "four: " + (2 + 2);
```

Now `s` contains the string "**four: 4**".

Conversion of String

- ❖ Conversion of **any type to String** use,
 - a. `String.valueOf()`
 - b. `toString()`
- ❖ Conversion of **String to any type** use,
 - a. `any_type.parse()`

Converting any type to String:

```
public class Conversion {
    public static void main(String[] args) {
        int a=10;        double b=5.5;
        //Converting other types to String
        String s1=String.valueOf(a);
        String s2=String.valueOf(b);
        //concatination of two strings
        String s3=s1+s2;
        System.out.println(s3);
    }
}
```

Output:

105.5

Converting String to any type:

```
public class Conversion {
    public static void main(String[] args) {
        String s1="10";
        String s2="11.5";
        //converting String to Integer
        int a=Integer.parseInt(s1);
        //converting String to Double
        double b=Double.parseDouble(s2);
        double res=(double)a+b;
        System.out.println("Result= "+res);
    }
}
```

Output

Result= 21.5

Changing case of String

The method **toLowerCase()** converts all the characters in a string from uppercase to lowercase. The **toUpperCase()** method converts all the characters in a string from lowercase to uppercase. Non alphabetical characters, such as digits, are unaffected.

Here is an example that uses **toLowerCase()** and **toUpperCase()**:

```
// Demonstrate toUpperCase() and toLowerCase().
class ChangeCase {
```

```

    public static void main(String args[])
    {
        String s = "This is a test.";
        System.out.println("Original: " + s);
        String upper = s.toUpperCase();
        String lower = s.toLowerCase();
        System.out.println("Uppercase: " + upper);
        System.out.println("Lowercase: " + lower);
    }
}

```

Output

```

Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.

```

Character Extraction

The String class provides a number of ways in which characters can be extracted from a String object. Although the characters that comprise a string within a String object cannot be indexed as if they were a character array, many of the String methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt()

To extract a single character from a String, you can refer directly to an individual character via the **charAt()** method. It has this general form:

```
char charAt(int where)
```

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. **charAt()** returns the character at the specified location. For example, `char ch; ch = "abc".charAt(1);` assigns the value **"b"** to **ch**.

getChars()

If you need to extract more than one character at a time, you can use the **getChars()** method. It has this general form:

```
void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)
```

Here, **sourceStart** specifies the index of the beginning of the substring, and **sourceEnd** specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from **sourceStart** through **sourceEnd-1**. The array that will receive the

characters – is specified by **target**. The index within target at which the substring will be copied is passed in **targetStart**. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

```
class getCharsDemo {
    public static void main(String args[]) {
        String s = "This is a demo of the getChars method.";
        int start = 10;
        int end = 14;
        char buf[] = new char[end - start];
        s.getChars(start, end, buf, 0);
        System.out.println(buf);
    }
}
```

Output

demo

String Comparison

There are three ways to compare string in java:

1. By equals() method
2. By == operator
3. By compareTo() method

1) String compare by equals() method

The String equals() method compares the original content of the string. It compares values of string for equality. String class provides two methods:

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

```
class Teststringcomparison1{ public
static void main(String args[]){
    String s1="Sachin";
    String s2="Sachin";
    String s3=new String("Sachin");
    String s4="Saurav";
    System.out.println(s1.equals(s2));//true
    System.out.println(s1.equals(s3));//true
    System.out.println(s1.equals(s4));//false
}
}
```

Output

```
true
true
false
```

```
class Teststringcomparison2{ public
static void main(String args[]){
    String s1="Sachin";
    String s2="SACHIN";
    System.out.println(s1.equals(s2));//false
    System.out.println(s1.equalsIgnoreCase(s2));//true
}
}
```

Output:

```
false
true
```

2) String compare by compareTo() method

The java string compareTo() method compares the given string with current string lexicographically. It returns positive number, negative number or 0.

It compares strings on the basis of Unicode value of each character in the strings.

if s1 > s2, it returns positive number **if**
s1 < s2, it returns negative number **if**
s1 == s2, it returns 0

```
public class CompareToExample{ public
static void main(String args[]){
    String s1="hello";
    String s2="hello";
    String s3="meklo";
    System.out.println(s1.compareTo(s2));//0 because both are equal
    System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"
}}
```

Output

```
0
-5
```

Searching Strings

The String class provides two methods that allow you to search a string for a specified character or substring:

- **indexOf()** Searches for the first occurrence of a character or substring.
- **lastIndexOf()** Searches for the last occurrence of a character or substring

To search for the first occurrence of a character, use `int indexOf(int ch)`

To search for the last occurrence of a character, use `int lastIndexOf(int ch)`

Here, **ch** is the character being sought.

To search for the first or last occurrence of a substring, use `int indexOf(String str)` `int lastIndexOf(String str)` Here, **str** specifies the substring.

You can specify a starting point for the search using these forms:

```
int indexOf(int ch, int startIndex) int
lastIndexOf(int ch, int startIndex) int
indexOf(String str, int startIndex) int
lastIndexOf(String str, int startIndex)
```

Here, **startIndex** specifies the index at which point the search begins. For **indexOf()**, the search runs from **startIndex** to the end of the string. For **lastIndexOf()**, the search runs from **startIndex** to zero.

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {      public static void main(String
args[]) {                String s = "Now is the time for all
good men " +
    "to come to the aid of their country.";
    System.out.println(s);
    System.out.println("indexOf(t) = " +
s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " +
s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
s.lastIndexOf("the", 60));
    }
}
```

Output

Now is the time for all good men to come to the aid of their country.
 indexOf(t) = 7 lastIndexOf(t) = 65 indexOf(the) = 7

```
lastIndexOf(the) = 55 indexOf(t, 10) = 11 lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

substring()

The **java string substring()** method returns a part of the string.

We pass begin index and end index number position in the java substring method where start index is inclusive and end index is exclusive. In other words, start index starts from 0 whereas end index starts from 1.

There are two types of substring methods in java string.

```
substring(int startIndex) and
substring(int startIndex, int endIndex)
```

Examples **public class** SubstringExample{
 public static void main(String args[]){
 String s1="javatpoint";
 System.out.println(s1.substring(2,4));*//returns va*
 System.out.println(s1.substring(2));*//returns vatpoint }*

Output

```
va
vatpoint
```

```
public class SubstringExample2 {
public static void main(String[] args) {
    String s1="Javatpoint";
    String substr = s1.substring(0); // Starts with 0 and goes to end
    System.out.println(substr);
    String substr2 = s1.substring(5,10); // Starts from 5 and goes to 10
    System.out.println(substr2);
    String substr3 = s1.substring(5,15); // Returns Exception
}
}
```

Output

```
Javatpoint
point
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException: begin 5, end 15, length
10
```


concat()

You can concatenate two strings using **concat()**, shown here: `String`

```
concat(String str)
```

This method creates a new object that contains the invoking string with the contents of **str** appended to the end. **concat()** performs the same function as **+**. For example,

```
String s1 = "one";
```

```
String s2 = s1.concat("two");
```

 puts the string "onetwo" into **s2**. It generates the same result as the following sequence:

```
String s1 = "one";
```

```
String s2 = s1 + "two";
```

replace()

The **java string replace()** method returns a string replacing all the old char or CharSequence to new char or CharSequence.

There are two type of replace methods in java string.

1. `replace(char oldChar, char newChar)` and
2. `replace(CharSequence target, CharSequence replacement)`

Java String replace(char old, char new) method example

```
public class ReplaceExample1{ public static
void main(String args[]){ String
s1="javatpoint is a very good website";
String replaceString=s1.replace('a','e');//replaces all occurrences of 'a' to 'e'
System.out.println(replaceString);
}}
```

Output jevetpoint is e very
good website

Java String replace(CharSequence target, CharSequence replacement) method example

```
public class ReplaceExample2{ public static void main(String args[]){
String s1="my name is khan my name is java";
String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "
was"
System.out.println(replaceString);
}}
```

Output my name was khan my
name was java

trim()

The **java string trim()** method eliminates leading and trailing spaces. The unicode value of space character is '\u0020'. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.

```
public class StringTrimExample {
public static void main(String[] args) {
String s1 = " hello java string ";
System.out.println(s1.length());
```

```

        System.out.println(s1); //Without trim()
        String tr = s1.trim();
        System.out.println(tr.length());
        System.out.println(tr); //With trim()
    }
}

```

Output

```

22      hello java string
17      hello java string

```

String Buffer

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class

Method	Description
append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.

insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
reverse()	is used to reverse the string.
capacity()	is used to return the current capacity.
ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
charAt(int index)	is used to return the character at the specified position.
length()	is used to return the length of the string i.e. total number of characters.
substring(int beginIndex)	is used to return the substring from the specified beginIndex.
substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

```

class StringBufferExample{  public static void
    main(String args[]){  StringBuffer sb=new
        StringBuffer("Hello ");  sb.append("Java");//now
        original string is changed
        System.out.println(sb);//prints Hello Java
    }
}

```

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```

class StringBufferExample2{  public static void
    main(String args[]){  StringBuffer sb=new

```

```

        StringBuffer("Hello "); sb.insert(1,"Java");//now
        original string is changed
        System.out.println(sb);//prints HJavaello
    }
}

```

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```

class StringBufferExample3{ public static void
    main(String args[]){ StringBuffer
        sb=new StringBuffer("Hello");
        sb.replace(1,3,"Java");
        System.out.println(sb);//prints HJavallo  }
}

```

4) StringBuffer delete() and deleteCharAt() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```

class StringBufferExample4{ public static void
    main(String args[]){ StringBuffer
        sb=new StringBuffer("Hello");
        sb.delete(1,3);
        System.out.println(sb);//prints Hlo  }
}

// Demonstrate delete() and deleteCharAt()
class deleteDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("This is a test.");

        sb.delete(4, 7);
        System.out.println("After delete: " + sb);

        sb.deleteCharAt(0);
        System.out.println("After deleteCharAt: " + sb);
    }
}

```

The following output is produced:

```

After delete: This a test.
After deleteCharAt: his a test.

```

5) StringBuffer reverse() method

The reverse() method of StringBuiler class reverses the current string.

```

class StringBufferExample5{ public static void
    main(String args[]){ StringBuffer
        sb=new StringBuffer("Hello");
        sb.reverse();
    }
}

```

```

        System.out.println(sb);//prints olleH
    }
}

```

6) charAt() and setCharAt()

The value of a single character can be obtained from a StringBuffer via the **charAt()** method. You can set the value of a character within a StringBuffer using **setCharAt()**.

```

// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");
        System.out.println("buffer before = " + sb);
        System.out.println("charAt(1) before = " + sb.charAt(1));
        sb.setCharAt(1, 'i');
        sb.setLength(2);
        System.out.println("buffer after = " + sb);
        System.out.println("charAt(1) after = " + sb.charAt(1));
    }
}

```

Here is the output generated by this program:

```

buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i

```

7) setLength()

To set the length of the buffer within a StringBuffer object, use **setLength()**. Its general form is shown here:

```
void setLength(int len)
```

Here, **len** specifies the length of the buffer. This value must be nonnegative.

8) ensureCapacity()

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

ensureCapacity() has this general form: **void**

```
ensureCapacity(int capacity)
```

Here, **capacity** specifies the size of the buffer.

9) length() and capacity()

The current length of a StringBuffer can be found via the **length()** method, while the total allocated capacity can be found through the **capacity()** method. They have the following general forms: **int length()** **int capacity()**

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("Hello");

        System.out.println("buffer = " + sb);
        System.out.println("length = " + sb.length());
        System.out.println("capacity = " + sb.capacity());
    }
}
```

Output buffer
= Hello length
= 5
capacity = 21

Since **sb** is initialized with the string “Hello” when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

Unit – 7 Threads [3 Hrs.]

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

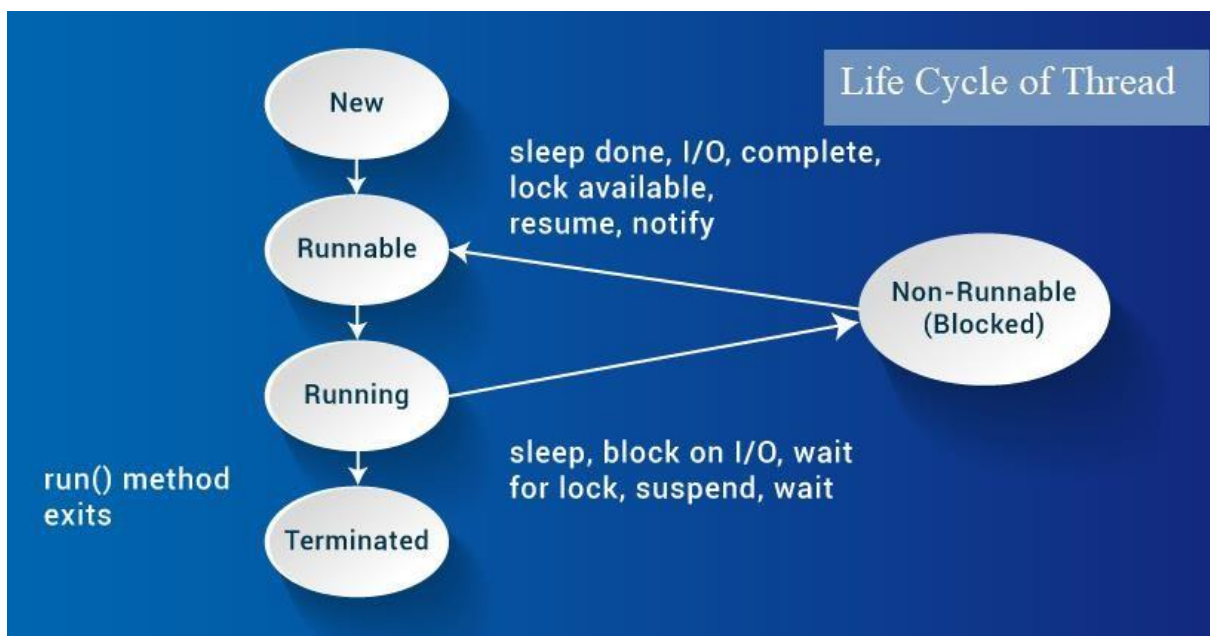
A thread is actually a lightweight process. Unlike many other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread and each thread defines a separate path of the execution. Thus, multithreading is a specialized form of multitasking.

The Java Thread Model

The Java run-time system depends on threads for many things. Threads reduce inefficiency by preventing the waste of CPU cycles.

Threads exist in several states:

- **New** - When we create an instance of Thread class, a thread is in a new state.
- **Running** - The Java thread is in running state.
- **Suspended** - A running thread can be **suspended**, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
- **Blocked** - A Java thread can be blocked when waiting for a resource.
- **Terminated** - A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.



Threads can be created by using two mechanisms:

1. Extending the Thread class
2. Implementing the Runnable Interface

Thread creation by extending the Thread class

We create a class that extends the **java.lang.Thread** class. This class overrides the **run()** method available in the Thread class. A thread begins its life inside **run()** method. We create an object of our new class and call **start()** method to start the execution of a thread. **start()** invokes the **run()** method on the **Thread** object.

```
public class MyClass extends Thread {      public void run() {
    System.out.println("MyClass running");
}
}
```

To create and start the above thread:

```
MyClass t1 = new MyClass (); t1.start();
//or
//new MyClass().start();
```

Example Program:

```
class ThreadA extends Thread{
public void run() {
for(int i=1;i<=5;i++) {
    System.out.println("Running thread "+i+" From Class A");
}
    System.out.println("Exit from Class A");
}
}
```

```
class ThreadB extends Thread{
public void run() {
for(int j=1;j<=5;j++) {
    System.out.println("Running thread "+j+" From Class B");
}
    System.out.println("Exit from Class B");
}
}
```

```
class ThreadC extends Thread{
public void run() {
for(int k=1;k<=5;k++) {
    System.out.println("Running thread "+k+" From Class C");
}
    System.out.println("Exit from Class C");
}
}
```



```

public class ThreadExample {
    public static void main(String[] args) {
        new ThreadA().start();
        ThreadB().start();
        ThreadC().start();
    }
}

```

Output

```

Running thread 1 From Class A
Running thread 2 From Class A
Running thread 3 From Class A
Running thread 4 From Class A
Running thread 5 From Class A Exit
from Class A
Running thread 1 From Class B
Running thread 2 From Class B
Running thread 3 From Class B
Running thread 4 From Class B
Running thread 5 From Class B
Exit from Class B
Running thread 1 From Class C
Running thread 2 From Class C
Running thread 3 From Class C
Running thread 4 From Class C
Running thread 5 From Class C
Exit from Class C

```

We have simply initiated three new threads and started them. They are running concurrently on their own. Note that the output from the threads are not specially sequential. They do not follow any specific order. They are running independently of one another and each executes whenever it has a chance. Remember, once the threads are started, we cannot decide with certainty the order in which they may execute statements.

Thread creation by implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement Runnable interface, a class need only implement a single method called **run()**, which is declared like this:

```

public class MyClass implements Runnable { public
void run() {
System.out.println("MyClass running");
}
}

```

```
}
```

To execute the **run()** method by a thread, pass an instance of **MyClass** to a Thread in its constructor. Here is how that is done:

```
//MyClass obj=new MyClass();  
//Thread t1=new Thread(obj);  
//t1.start();  
//or  
Thread t1 = new Thread(new MyClass ()); t1.start();
```

Example Program

```
class ThreadA implements Runnable{  
    public void run() {  
        for(int i=1;i<=5;i++) {  
            System.out.println("Running thread "+i+" From Class A");  
        }  
        System.out.println("Exit from Class A");  
    }  
}
```

```
class ThreadB implements Runnable{  
    public void run() {  
        for(int j=1;j<=5;j++) {  
            System.out.println("Running thread "+j+" From Class B");  
        }  
        System.out.println("Exit from Class B");  
    }  
}
```

```
class ThreadC implements Runnable{  
    public void run() {  
        for(int k=1;k<=5;k++) {  
            System.out.println("Running thread "+k+" From Class C");  
        }  
        System.out.println("Exit from Class C");  
    }  
}
```

```
public class ThreadExample {  
    public static void main(String[] args) {  
        Thread t1=new Thread(new ThreadA());  
        Thread t2=new Thread(new ThreadB());  
        Thread t3=new Thread(new ThreadC());  
        t1.start();    t2.start();  
        t3.start();  
    }  
}
```

```
}  
}
```

Output

```
Running thread 1 From Class B  
Running thread 1 From Class C  
Running thread 2 From Class C  
Running thread 3 From Class C  
Running thread 4 From Class C  
Running thread 5 From Class C  
Running thread 1 From Class A  
Running thread 2 From Class A Exit  
from Class C  
Running thread 2 From Class B  
Running thread 3 From Class A  
Running thread 4 From Class A  
Running thread 5 From Class A  
Exit from Class A  
Running thread 3 From Class B  
Running thread 4 From Class B  
Running thread 5 From Class B  
Exit from Class B
```

Note: Output can be different on each run because thread doesn't follow sequential order.

Stopping and Blocking a Thread

Stopping a Thread

Whenever we want to stop a thread from running further, we may do so by calling **stop()** method, like:

```
aThread.stop();
```

This statement causes the thread to move to the dead state. A thread will also move to the dead state automatically when it reaches the end of its method. The **stop()** method may be used when the premature death of a thread is desired.

Blocking a Thread

A thread can also be temporarily suspended or blocked from entering into the runnable state by using either of the following thread methods:

```
sleep()      //blocked for a specified time  
suspend()    // blocked until further orders wait()  
//blocked until certain condition orders
```

```
//Program to demonstrate stop() and sleep().  
class ThreadA extends Thread{    public void  
run() {                          for(int i=1;i<=5;i++) {  
                                System.out.println("Running thread "+i+" From Class A");
```

```

        if(i==3) stop();
    }
    System.out.println("Exit from Class A");
}}
class ThreadB extends Thread{
public void run() {
for(int j=1;j<=5;j++) {
    System.out.println("Running thread "+j+" From Class B");
    if(j==2)
        try {
            sleep(1000); //in milliseconds
            //sleep must be surrounded with try/catch
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    System.out.println("Exit from Class B");
}
}

class ThreadC extends Thread{
public void run() {
for(int k=1;k<=5;k++) {
    System.out.println("Running thread "+k+" From Class C");
}
    System.out.println("Exit from Class C");
}
}

public class ThreadExample {
    public static void main(String[] args) {
new ThreadA().start();        new
ThreadB().start();            new
ThreadC().start();
    } }

```

Output

```

Running thread 1 From Class A
Running thread 2 From Class A
Running thread 3 From Class A
Running thread 1 From Class B
Running thread 2 From Class B
Running thread 1 From Class C
Running thread 2 From Class C
Running thread 3 From Class C
Running thread 4 From Class C
Running thread 5 From Class C
Exit from Class C

```

```
Running thread 3 From Class B
Running thread 4 From Class B
Running thread 5 From Class B
Exit from Class B
```

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, 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.

Java permits us to set the priority of a thread using the **setPriority()** method as follows:

ThreadName.setPriority (intNumber)

The **intNumber** is an integer value to which the thread's priority is set. The Thread class defines several priority constants:

```
MIN_PRIORITY    = 1
NORM_PRIORITY   = 5
MAX_PRIORITY    = 10
```

The **intNumber** may assume one of these constants or any value between **1** and **10**. Note that the default setting is **NORM_PRIORITY**.

```
//Program to demonstrate thread priority.
class ThreadA extends Thread{    public
void run() {                    for(int
i=1;i<=5;i++) {
    System.out.println("Running thread "+i+" From Class A");
    }
    System.out.println("Exit from Class A");
}
}

class ThreadB extends Thread{
public void run() {
for(int j=1;j<=5;j++) {
    System.out.println("Running thread "+j+" From Class B");
    }
    System.out.println("Exit from Class B");
}
}
```

```

class ThreadC extends Thread{
public void run() {
for(int k=1;k<=5;k++) {
    System.out.println("Running thread "+k+" From Class C");
}
    System.out.println("Exit from Class C");
}
}

public class ThreadExample {
    public static void main(String[] args) {
        ThreadA t1=new ThreadA();
        ThreadB t2=new ThreadB();
        ThreadC t3=new ThreadC();
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(t1.getPriority()+1);
        //or t2.setPriority(2)
        t3.setPriority(Thread.MAX_PRIORITY);
        t1.start();    t2.start();
        t3.start();
    }
}

```

Note that although the **threadA** started first, the higher priority **threadB** has preempted it and started printing the output first. Immediately, the **threadC** that has been assigned the highest priority takes control over the other two threads. The **threadA** is the last to complete.

Synchronization in Thread

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results. So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```

class AThread {

```

```

        synchronized void deposit(int amt) {
            System.out.println("Deposit Completed with Rs. "+amt);
        }
        synchronized void withdraw(int amt) {
            System.out.println("Withdraw Completed with Rs. "+amt);
        }
    }
}

```

```

public class Synchronization{    public
static void main(String[] args) {
    AThread obj=new AThread();

        new Thread() {
public void run() {
            obj.deposit(15000);
        }
    }.start();

        new Thread() {
public void run() {
            obj.withdraw(10000);
        }
    }.start();

    }
}

```

Output

Deposit Completed with Rs. 15000
 Withdraw Completed with Rs. 10000

Inter Thread Communication

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

- wait() ○ notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

2) notify() method

Wakes up a single thread that is waiting. If any threads are waiting on this object, one of them is chosen to be awakened.

3) notifyAll() method

Wakes up all threads that are waiting.

```
class AThread{    int
amount=10000;
    synchronized void withdraw(int amt) {
        System.out.println("Going to Withdraw...");
        if(amount<amt) {
            System.out.println("Less balance; waiting for deposit...");
            try {
                wait();
            } catch (Exception e) {
                System.out.println(e);
            }
        }
        amount-=amt;
        System.out.println("Withdraw Completed with Rs. "+amt);
        System.out.println("Balance is Rs. "+amount);
    }

    synchronized void deposit(int amt) {
        System.out.println("Going to Deposit...");
        amount+=amt;
        System.out.println("Deposit Completed with Rs. "+amt);
        System.out.println("Balance is Rs. "+amount);
        notify();
    }
}

public class Synchronization{    public
static void main(String[] args) {
    AThread obj=new AThread();

    new Thread() {
public void run() {
obj.withdraw(15000);
}
}.start();

    new Thread() {
public void run() {
```



```

        obj.deposit(10000);
    }
    }.start();
}
}

```

Output

```

Going to Withdraw...
Less balance; waiting for deposit...
Going to Deposit...
Deposit Completed with Rs. 10000
Balance is Rs. 20000
Withdraw Completed with Rs. 15000
Balance is Rs. 5000

```

Deadlock

synchronized keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock.

It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called **Deadlock**. Below is a simple example of Deadlock condition.

Figure - 1

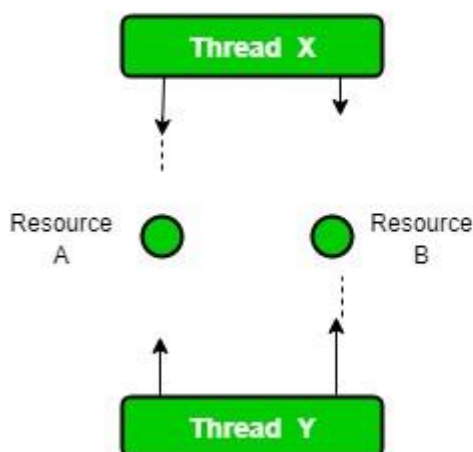
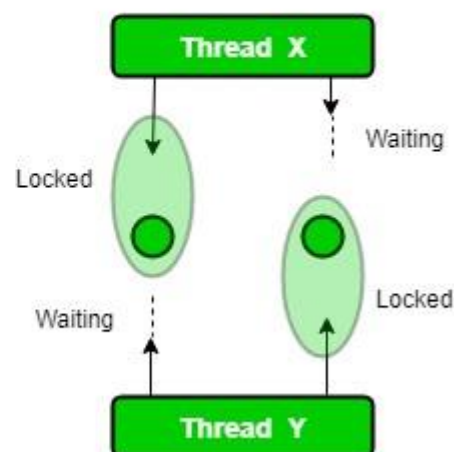


Figure - 2



Important Points :

- If threads are waiting for each other to finish, then the condition is known as Deadlock.
- Deadlock condition is a complex condition which occurs only in case of multiple threads.
- Deadlock condition can break our code at run time and can destroy business logic.
- We should avoid this condition as much as we can.

```

class MyThread{
    String resource1 = "BCA";

```

```

String resource2 = "Third Semester";

void MethodA() {
    synchronized (resource1) {
        System.out.println("Thread 1: locked resource 1");

        try { Thread.sleep(100);} catch (Exception e) {}

        synchronized (resource2) {
            System.out.println("Thread 1: locked resource 2");
        }
    }
}

void MethodB() {
    synchronized (resource2) {
        System.out.println("Thread 2: locked resource 2");

        try { Thread.sleep(100);} catch (Exception e) {}

        synchronized (resource1) {
            System.out.println("Thread 2: locked resource 1");
        }
    }
}

}

public class Deadlock {
    public static void main(String[] args) {
        MyThread obj=new MyThread();
        new Thread() {
            public void run() {
                obj.MethodA();
            }
        }.start();
        new Thread() {
            public void run() {
                obj.MethodB();
            }
        }.start();
    }
}

```

Output

```

Thread 1: locked resource 1
Thread 2: locked resource 2

```

Unit – 9 Understanding Core Packages [3 Hrs.]

Java.lang.Math Class in Java

Math Class methods helps to perform the numeric operations like square, square root, cube, cube root, exponential and trigonometric operations.

Methods of lang.math class :

Method Name	Description
1. abs()	java.lang.Math.abs() method returns the absolute value of any type of argument passed.
2. acos()	java.lang.Math.acos() method returns the arc cosine value of the passed argument. arc cosine is inverse cosine of the argument passed. $\text{acos}(\text{arg}) = \cos^{-1}$ of arg
3. asin()	java.lang.Math.asin() method returns the arc sine value of the method argument passed. arc sine is inverse sine of the argument passed. $\text{asin}(\text{arg}) = \sin^{-1}$ of arg
4. sqrt()	java.lang.Math.sqrt() returns the square root of a value of type double passed to it as argument.
5. cbrt()	java.lang.Math.cbrt() method returns the cube root of the passed argument.

6. floor()	java.lang.Math.floor() method returns the floor value of an argument i.e. the closest integer value which is either less or equal to the the passed argument. eg : 101.23 has floor value = 101.
7. log()	java.lang.Math.log() method returns the logarithmic value of the passed argument.
8. ceil()	java.lang.Math.ceil(double a) method returns the smallest possible value which is either greater or equal to the argument passed. The returned value is a mathematical integer.
9. atan()	java.lang.Math.atan() method returns returns the arc tangent of the method argument value. arc tan is inverse tan of the argument passed. atan(arg) = tan inverse of arg

Example of lang.math with its methods

```
import java.lang.Math;
public class Math_Test {
    public static void main(String[] args) {

        //abs method
        int value1=-101;
        System.out.println("Absolute value = "+Math.abs(value1));

        //acos,asin,atan
        //value with be NaN(Not a Number) if >1
        double value2=0.7;
        System.out.println("ACos value = "+Math.acos(value2));
        System.out.println("ASin value = "+Math.asin(value2));
        System.out.println("ATan value = "+Math.atan(value2));

        //square, cube root
        System.out.println("Square root= "+Math.sqrt(25));
        System.out.println("Cube root= "+Math.cbrt(125));

        //floor,ceil
        double value3=77.11;
        System.out.println("Floor value= "+Math.floor(value3));
        System.out.println("Ceil value= "+Math.ceil(value3));

    }
}
```

Output:

```
Absolute value = 101
ACos value = 0.7953988301841436
ASin value = 0.775397496610753
ATan value = 0.6107259643892086
Square root= 5.0
Cube root= 5.0
Floor value= 77.0
Ceil value= 78.0
|
```

Wrapper Class & Associated Methods

A Wrapper class is a class whose object wraps or contains a primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store a primitive data types.

In other words, we can wrap a primitive value into a wrapper class object.

Primitive Data types and their Corresponding Wrapper class

Primitive Data Type	Wrapper Class
char	Character
byte	Byte
short	Short
long	Integer
float	Float
double	Double
boolean	Boolean

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as [ArrayList](#) and [Vector](#), store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Wrapping & Unwrapping Example

```

public class Wrap_Unwrap {
    public static void main(String[] args) {
        //wrapping int to Integer
        int a=10;
        Integer obji=new Integer(a);
        //wrapping double to Double
        double d=10.5;
        Double objd=new Double(d);
        //wrapping char to Character
        char c='x';
        Character objc=new Character(c);

        System.out.println("After Wrapping...");
        System.out.println("Integer= "+obji);
        System.out.println("Double= "+objd);
        System.out.println("Character= "+objc);

        //unwrapping Integer to int
        int a1=obji;
        //unwrapping Double to double
        double d1=objd;
        //unwrapping Character to char
        char c1=objc;

        System.out.println("After UnWrapping...");
        System.out.println("int= "+a1);
        System.out.println("double= "+d1);
        System.out.println("char= "+c1);
    }
}

```

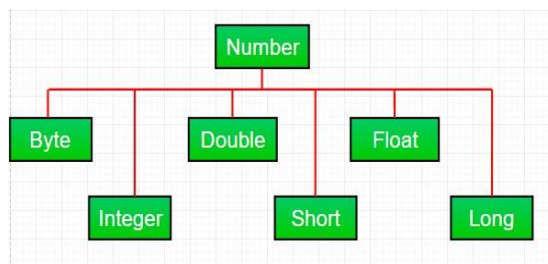
```

After Wrapping...
Integer= 10
Double= 10.5
Character= x
After UnWrapping...
int= 10
double= 10.5
char= x

```

Number Class

There are mainly six sub-classes under Number class. These sub-classes define some useful methods which are used frequently while dealing with numbers.



Associated Methods

```

byte byteValue()
short shortValue()
int intValue()
long longValue()
float floatValue()
double doubleValue()

```

```

public class Test
{
    public static void main(String[] args)
    {
        // Creating a Double Class object with value "6.9685"
        Double d = new Double("6.9685");

        // Converting this Double(Number) object to
        // different primitive data types
        byte b = d.byteValue();
        short s = d.shortValue();
        int i = d.intValue();
        long l = d.longValue();
        float f = d.floatValue();
        double d1 = d.doubleValue();

        System.out.println("value of d after converting it to byte : " + b);
        System.out.println("value of d after converting it to short : " + s);
        System.out.println("value of d after converting it to int : " + i);
        System.out.println("value of d after converting it to long : " + l);
        System.out.println("value of d after converting it to float : " + f);
        System.out.println("value of d after converting it to double : " + d1)
    }
}

```

```

value of d after converting it to byte : 6
value of d after converting it to short : 6
value of d after converting it to int : 6
value of d after converting it to long : 6
value of d after converting it to float : 6.9685
value of d after converting it to double : 6.9685

```

Methods Associated with Double

Method Name	Description	Example
1. toString() :	Returns the string corresponding to the double value.	double b = 55.05; Double x = new Double(b); String s=Double.toString(x);
2. valueOf() :	returns the Double object initialised with the value provided.	String s="45.55"; Double d=Double.valueOf(s);
3. parseDouble() :	returns double value by parsing the string. Differs from valueOf() as it returns a primitive double value and valueOf() return Double object.	String s="45.55"; Double d=Double.parseDouble(s);

4. equals() :	Used to compare the equality of two Double objects. This methods returns true if both the objects contains same double value.	If(double1.equals(double2)) //true else // false
5. compareTo()	Used to compare two Double objects for numerical equality. Returns a value less than 0,value greater than 0 for less than,equal to and greater than.	If(double1.compareTo(double2)) //true else // false
6. compare() :	Used to compare two primitive double values for numerical equality.	If(compare(double1,double2)==0) //equal
7. byteValue() :	returns a byte value corresponding to this Double Object.	byte b=double1.byteValue();
8. shortValue() :	returns a short value corresponding to this Double Object.	short s=double1.shortValue();
9. intValue() :	returns a int value corresponding to this	int i=double1.shortValue();
	Double Object.	
10. longValue() :	returns a long value corresponding to this Double Object.	long l=double1.longValue();
11. doubleValue() :	returns a double value corresponding to this Double Object.	double d=double1.doubleValue();
12. floatValue() :	returns a float value corresponding to this Double Object.	float f=double1.floatValue();

Methods Associated with Float

Method Name	Description	Example
1. toString() :	Returns the string corresponding to the float value.	float b = 55.05; Float x = new Float (b); String s= Float.toString(x);
2. valueOf() :	returns the Float object initialised with the value provided.	String s="45.55"; Float d= Float.valueOf(s);
3. parseFloat() :	returns float value by parsing the string. Differs from valueOf() as it returns a primitive float value and valueOf() return Float object.	String s="45.55"; Float d= Float.parseDouble(s);
4. equals() :	Used to compare the equality of two Float objects. This methods returns true if both the objects contains same float value.	If(float1.equals(float2)) //true else // false
5. compareTo()	Used to compare two Float objects for numerical equality. Returns a value less than 0,0,value greater than 0 for less than,equal to and greater than.	If(float1.compareTo(float2)) //true else // false
6. compare() :	Used to compare two primitive float values for numerical equality.	If(compare(float1,float2)==0) //equal
7. byteValue() :	returns a byte value corresponding to this Float Object.	byte b=float1.byteValue();
8. shortValue() :	returns a short value corresponding to this Float Object.	short s= float1.shortValue();

9. intValue() :	returns a int value corresponding to this Float Object.	int i= float1.shortValue();
10. longValue() :	returns a long value corresponding to this Float Object.	long l= float1.longValue();
11. doubleValue() :	returns a double value corresponding to this Float Object.	float d= float1.doubleValue();
12. floatValue() :	returns a float value corresponding to this Float Object.	float f= float1.floatValue();

Methods Associated with Integer

1. **toString()**
2. **toHexString() :** Returns the string corresponding to the int value in hexadecimal form, that is it returns a string representing the int value in hex characters-[0-9][a-f].
3. **toOctalString() :** Returns the string corresponding to the int value in octal form, that is it returns a string representing the int value in octal characters-[0-7].
4. **toBinaryString() :** Returns the string corresponding to the int value in binary digits, that is it returns a string representing the int value in hex characters-[0/1].
5. **valueOf()**
6. **parseInt()**
7. **getInteger() :** returns the Integer object representing the value associated with the given system property or null if it does not exist.
8. **byteValue()**
9. **intValue()**
10. **doubleValue()**
11. **shortValue()**
12. **longValue()**
13. **floatValue()**
14. **equals()**
15. **compareTo()** 16. **compare()**

Methods Associated with Character

1. **boolean isLetter(char ch) :** This method is used to determine whether the specified char value(ch) is a letter or not. The method will return true if it is letter([A-Z],[a-z]), otherwise return false.

```
// Java program to demonstrate isLetter() method
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(Character.isLetter('A'));

        System.out.println(Character.isLetter('0'));
    }
}
```

Output:

```
true
false
```

2. **boolean isDigit(char ch)** : This method is used to determine whether the specified char value(ch) is a digit or not.

```
// Java program to demonstrate isDigit() method
public class Test
{
    public static void main(String[] args)
    {
        // print false as A is character
        System.out.println(Character.isDigit('A'));

        System.out.println(Character.isDigit('0'));
    }
}
```

Output:

```
false
true
```

3. **boolean isWhitespace(char ch)** : It determines whether the specified char value(ch) is white space. A whitespace includes space, tab, or new line.

```
// Java program to demonstrate isWhitespace() method
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(Character.isWhitespace('A'));
        System.out.println(Character.isWhitespace(' '));
        System.out.println(Character.isWhitespace('\n'));
        System.out.println(Character.isWhitespace('\t'));

        //ASCII value of tab
        System.out.println(Character.isWhitespace(9));

        System.out.println(Character.isWhitespace('9'));
    }
}
```

```
false
true
true
true
true
true
false
```

4. **boolean isUpperCase(char ch)** : It determines whether the specified char value(ch) is uppercase or not.
5. **boolean isLowerCase(char ch)** : It determines whether the specified char value(ch) is lowercase or not.
6. **char toUpperCase(char ch)** : It returns the uppercase of the specified char value(ch).
7. **char toLowerCase(char ch)** : It returns the lowercase of the specified char value(ch).

```
// Java program to demonstrate toLowerCase() method
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(Character.toLowerCase('A'));
        System.out.println(Character.toLowerCase(65));
        System.out.println(Character.toLowerCase(48));
    }
}
```

Output:

```
a
97
48
```

8. **toString(char ch)** : It returns a String class object representing the specified character value(ch) i.e a one-character string.

```
// Java program to demonstrate toString() method
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(Character.toString('x'));
        System.out.println(Character.toString('Y'));
    }
}
```

Output:

```
x
y
```

Methods Associated with Boolean

1. **parseBoolean(String s)** : This method parses the string argument as a boolean. The boolean returned represents the value true if the string argument is not null and is equal, ignoring case, to the string "true", otherwise return false.
 boolean b1 = Boolean.parseBoolean("True");
 System.out.println(b1); //Output: true
2. **booleanValue()** : This method returns the value of this Boolean object as a boolean primitive.
 boolean b1 = Boolean.booleanValue("True");
 System.out.println(b1); //Output: true
3. **valueOf(boolean b)** : This method returns a Boolean instance representing the specified boolean value. If the specified boolean value is true, it returns Boolean.TRUE or if it is false, then this method returns Boolean.FALSE.
 boolean b1 = true;
 Boolean b3 = Boolean.valueOf(b1);
 System.out.println(b3); //Output: true
4. **valueOf(String s)** : This method returns a Boolean with a value represented by the specified string 's'. The Boolean returned represents a true value if the string argument is not null and is equal, ignoring case, to the string "true".
5. **toString(boolean b)** : This method returns a String object representing the specified boolean. If the specified boolean is true, then the string "true" will be returned, otherwise the string "false" will be returned.

6. **toString()** : This method returns a String object representing this Boolean's value. If this object represents the value true, a string equal to "true" is returned. Otherwise, the string "false" is returned.
7. **equals(Object obj)** : This method returns true iff the argument is not null and is a Boolean object that represents the same boolean value as this object.
8. **compareTo(Boolean b)** : This method "compares" this Boolean instance with passed argument 'b'.
9. **compare(boolean x, boolean y)** : This method is used to "compares" primitives boolean variables.

Methods Associated with Other Wrapper Classes

//Same as Double,Float and Integer.....

Classes Associated with Java.util Package

Vector Class

The Vector class implements a growable array of objects. Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index.

Methods in Vector:

1. **add(element)**: This method is used to add elements in vector.
2. **get(index)**: This method is used to retrieve elements on the basis of index number.
3. **isEmpty()**: This method tests if this vector has no components.
4. **remove(index)**: This method removes a single element on the basis of index number.
5. **clear()**: This method clear all the elements from vector.
6. **size()**: This method returns the number of components in this vector.
7. **removeAllElements()**: This method removes all components from this vector and sets its size to zero.
8. **indexOf(element)**: This method returns first occurrence of given element or -1 if element is not present in vector.
9. **lastIndexOf(element)**: This method returns the last occurrence of given element or 1 if element is not present in vector.

```

import java.util.Vector;
public class Vector_Example {
    public static void main(String[] args) {
        Vector<Integer> vec=new Vector<Integer>();
        //insert
        int i;
        for(i=1;i<=10;i++) {
            vec.add(i);
        }

        //display
        System.out.println(vec);
        System.out.println("Item at index 5= "+vec.get(5));

        //remove
        vec.remove(5); //index
        System.out.println("After removing item from index 5");
        System.out.println(vec);

        //size
        System.out.println("Size of vector is: "+vec.size());

        //clear
        vec.clear();
        System.out.println("After Clearing vector object");
        System.out.println(vec);
    }
}

```

Output:

```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Item at index 5= 6
After removing item from index 5
[1, 2, 3, 4, 5, 7, 8, 9, 10] Size
of vector is: 9
After Clearing vector object
[]

```

Stack Class

Java Collection framework provides a Stack class which models and implements Stack data structure. The class is based on the basic principle of last-in-first-out. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek.

Methods in Stack class

1. **push(element)** : Pushes an element on the top of the stack.
2. **pop()** : Removes and returns the top element of the stack. An 'EmptyStackException' exception is thrown if we call pop() when the invoking stack is empty.
3. **peek()** : Returns the element on the top of the stack, but does not remove it.
4. **empty()** : It returns true if nothing is on the top of the stack. Else, returns false.

5. **search(element)** : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

```
import java.util.Stack;
public class Stack_Example {
    public static void main(String[] args) {
        Stack<Integer> st=new Stack<Integer>();
        //push
        int i;
        for(i=1;i<=10;i++) {
            st.push(i);
        }
        System.out.println("After Insertion in Stack");
        System.out.println(st);

        //pop
        st.pop();
        st.pop();
        System.out.println("After Deletion from Stack");
        System.out.println(st);

        //empty, peek and search
        if(st.empty()) {
            System.out.println("Stack is empty");
        }else {
            System.out.println("Stack is not empty");
            System.out.println("Peek Item= "+st.peek());
            System.out.println("Item Postion of 5 = "+st.search(5));
        }
    }
}
```

Output:

```
After Insertion in Stack
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
After Deletion from Stack
[1, 2, 3, 4, 5, 6, 7, 8]
Stack is not empty
Peek Item= 8
Item Postion of 5 = 4
```

Hashtable Class

This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.

To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

- It is similar to HashMap, but is synchronised.
- Hashtable stores key/value pair in hash table.

- In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Methods in Hashtable:

1. **put(key,value):** This method is used to add elements in hashtable.
2. **get(key):** This method is used to retrieve elements on the basis of key.
3. **isEmpty():** This method tests if this hashtable has no components.
4. **remove(key):** This method removes a single element on the basis of key.
5. **clear():** This method clear all the elements from hashtable.
6. **size():** This method returns the number of components in this hashtable.
7. **removeAllElements():** This method removes all components from this hashtable and sets its size to zero.

```
import java.util.*;
public class Dictionary_HashTable {
    public static void main(String[] args) {

        // create a new hashtable
        Hashtable hash = new Hashtable();
        // put some elements
        hash.put(1, "beer");
        hash.put(2, "tea");
        hash.put(5, "Coffee");
        System.out.println(hash);
        System.out.println("Getting value of key 5: "+hash.get(5));
        //remove
        hash.remove(2);
        System.out.println(hash);
    }
}
```

Output:

```
{5=Coffee, 2=tea, 1=beer}
Getting value of key 5: Coffee
{5=Coffee, 1=beer}
```

Dictionary Class

Dictionary is an abstract class, representing a key-value relation and works similar to a map. Given a key you can store values and when needed can retrieve the value back using its key. Thus, it is a list of key-value pair.

Methods in Dictionary:

1. **put(key,value):** This method is used to add elements in Dictionary.
2. **get(key):** This method is used to retrieve elements on the basis of key.
3. **isEmpty():** This method tests if this Dictionary has no components.
4. **remove(key):** This method removes a single element on the basis of key.
5. **clear():** This method clear all the elements from Dictionary.
6. **size():** This method returns the number of components in this Dictionary.

7. **removeAllElements():** This method removes all components from this Dictionary and sets its size to zero.

```
import java.util.*;
public class Dictionary_Example {
    public static void main(String[] args)
    {
        // Initializing a Dictionary
        Dictionary geek = new Hashtable();

        // put() method
        geek.put("123", "Code");
        geek.put("456", "Program");

        // get() method :
        System.out.println("\nValue at key = 6 : " + geek.get("6"));
        System.out.println("Value at key = 456 : " + geek.get("123"));

        // isEmpty() method :
        System.out.println("\nThere is no key-value pair : " + geek.isEmpty() + "\n");

        // remove() method :
        System.out.println("\nRemove : " + geek.remove("123"));
        System.out.println("Check the value of removed key : " + geek.get("123"));

        System.out.println("\nSize of Dictionary : " + geek.size());
    }
}
```

Output:

Value at key = 6 : null
Value at key = 456 : Code

There is no key-value pair : false

Remove : Code
Check the value of removed key : null
Size of Dictionary : 1

Enumerations in Java

Enumerations serve the purpose of representing a group of named constants in a programming language. For example the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).

Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay fixed for all time.

```
// A simple enum example where enum is declared
// outside any class (Note enum keyword instead
// class keyword)
enum Color
{
    RED, GREEN, BLUE;
}

public class Test
{
    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

Output :

RED

Random Number Generation in Java

```
import java.util.Random;

public class RandomNumber {
    public static void main(String[] args) {
        Random rand = new Random();
        // Obtain a number between [0 - 49].
        int n = rand.nextInt(100);
        System.out.println("Random Number now is: "+n);
    }
}
```

Output:

Random Number now is: 71

Unit – 10 Holding Collection of Data [3 Hrs.]

Arrays and Collection Classes/Interfaces

An array is a group of like-typed variables that are referred to by a common name. Arrays in Java work differently than they do in C/C++. Following are some important point about Java arrays.

- In Java all arrays are dynamically allocated.
- Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using sizeof.
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- Java array can be also be used as a static field, a local variable or a method parameter.
- The **size** of an array must be specified by an int value and not long or short.

Array Declaration

1. `int[] arr = new int[20];`
2. `int[] arr = new int[]{1,2,3,4,5};`
3. `int[] arr = {1,2,3,4,5};`

Example1

```
public class MyArray {  
    public static void main(String[] args) {  
        int a[]=new int[10];  
        int i;  
        for( i=0;i<10;i++) {  
            a[i]=i;  
        }  
        for(i=0;i<10;i++) {  
            System.out.println(a[i]);  
        }  
    }  
}
```

0
1
2
3
4
5
6
7
8
9

Example2

```
import java.util.Scanner;
public class Array_Example {
//Program to find maximum/minimum number in array
    public static void main(String[] args) {
        int a[]=new int[5]; //size 5
        int i,max,min;
        Scanner scan=new Scanner(System.in);

        System.out.println("Enter 5 numbers");
        for(i=0;i<5;i++) {
            a[i]=scan.nextInt();
        }

        max=a[0];
        min=a[0];
        for(i=1;i<5;i++) {
            if(a[i]>max)
                max=a[i];
        }
        for(i=1;i<5;i++) {
            if(a[i]<min)
                min=a[i];
        }
        System.out.println("Maximum value is: "+max);
        System.out.println("Minimum value is: "+min);
    }
}
```

Output:

Enter 5 numbers

20

15

16

10

25

Maximum value is: 25

Minimum value is: 10

Collection Classes/Interfaces

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main “root” interfaces of Java collection classes.

Advantages of Collection Framework:

Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were Arrays or Vectors or Hashtables. All of these collections had no common interface.

Accessing elements of these Data Structures was a hassle as each had a different method (and syntax) for accessing its members.

1. **Consistent API** : The API has a basic set of interfaces like Collection, Set, List, or Map. All classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have *some* common set of methods.
2. **Reduces programming effort**: A programmer doesn't have to worry about the design of Collection, and he can focus on its best use in his program.
3. **Increases program speed and quality**: Increases performance by providing highperformance implementations of useful data structures and algorithms.

Collection Interfaces

1. Map Interface

The java.util.Map interface represents a mapping between a key and a value. Few characteristics of the Map Interface are:

1. A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value like the HashMap and LinkedHashMap, but some do not like the TreeMap.
2. The order of a map depends on specific implementations, e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not.
3. There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, TreeMap and LinkedHashMap.

Methods in Map Interface:

1. **put(key, value)**: This method is used to insert an entry in this map.
2. **putAll(Map map)**: This method is used to insert the specified map in this map.
3. **remove(key)**: This method is used to delete an entry for the specified key.
4. **get(key)**: This method is used to return the value for the specified key.
5. **containsKey(key)**: This method is used to search the specified key from this map.
6. **clear()**: This method clear all the elements.
7. **size()**: This method returns the number of components.

Example Program:

```
import java.util.HashMap;
import java.util.Map;
public class Map_Interface {
    public static void main(String[] args)
    {
        // Initializing a Map
        Map<String,String> geek = new HashMap<String,String>();

        // put() method
        geek.put("123", "Code");
        geek.put("456", "Program");

        // get() method :
        System.out.println("\nValue at key = 6 : " + geek.get("6"));
        System.out.println("Value at key = 456 : " + geek.get("123"));

        // isEmpty() method :
        System.out.println("\nThere is no key-value pair : " + geek.isEmpty() + "\n");

        // remove() method :
        System.out.println("\nRemove : " + geek.remove("123"));
        System.out.println("Check the value of removed key : " + geek.get("123"));

        System.out.println("\nSize of Dictionary : " + geek.size());
    }
}
```

Output:

```
Value at key = 6 : null
Value at key = 456 : Code

There is no key-value pair : false

Remove : Code
Check the value of removed key : null

Size of Map : 1
```

2. List Interface

The Java.util.List is a child interface of Collection. It is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order, it allows positional access and insertion of elements. List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

List is an interface, and the instances of List can be created in the following ways:

```
List a = new ArrayList();  
List b = new LinkedList();  
List c = new Vector();  
List d = new Stack();
```

Methods in List Interface:

1. **add(element):** This method is used to add elements in list.
2. **get(index):** This method is used to retrieve elements on the basis of index number.
3. **isEmpty():** This method tests if this list has no components.
4. **remove(index):** This method removes a single element on the basis of index number.
5. **clear():** This method clear all the elements from list.
6. **size():** This method returns the number of components in this list.
7. **removeAllElements():** This method removes all components from this list and sets its size to zero.
8. **indexOf(element):** This method returns first occurrence of given element or -1 if element is not present in list.
9. **lastIndexOf(element):** This method returns the last occurrence of given element or 1 if element is not present in list.

Example Program


```

import java.util.*;
public class List_Example {
    public static void main(String[] args) {
        List<Integer> list=new ArrayList<Integer>();
        //adding items
        list.add(10);
        list.add(20);
        list.add(30);
        //retrieving items
        System.out.println(list);
        System.out.println("Item at index 1 is: "+list.get(1));
        //removing item
        list.remove(2); //item from index 2 is removed
        System.out.println("After Removing Item");
        System.out.println(list);
        //getting size
        System.out.println("Size of list is: "+list.size());
        //removing all item
        list.clear();
        System.out.println("List after removing all items: ");
        System.out.println(list);
    }
}

```

Output:

```

[10, 20, 30]
Item at index 1 is: 20
After Removing Item
[10, 20]
Size of list is: 2
List after removing all items: []

```

3. Set Interface

- Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored.
- Basically, Set is implemented by HashSet, LinkedHashSet or TreeSet (sorted representation).
- Set has various methods to add, remove clear, size, etc to enhance the usage of this interface

```

import java.util.*;
public class Set_example
{
    public static void main(String[]
args)
    {
        // Set deonstration using HashSet
        Set<String> hash_Set = new
HashSet<String>();
        hash_Set.add("For");
        hash_Set.add("Example");
        hash_Set.add("Set");
        hash_Set.add("Geeks");
        System.out.print("Set output without the duplicates");
    }
}

```

```

        System.out.println(hash_Set);
        // Set deonstration using TreeSet
        System.out.print("Sorted Set after passing into TreeSet");
        Set<String> tree_Set = new TreeSet<String>(hash_Set);
        System.out.println(tree_Set);
    }
}

```

Output:

Set output without the duplicates[Geeks, Example, For, Set]

Sorted Set after passing into TreeSet[Example, For, Geeks, Set]

Note: As we can see the duplicate entry “Geeks” is ignored in the final output, Set interface doesn’t allow duplicate entries.

Methods in Set Interface:

1. **add(element):** This method is used to add elements in set.
2. **get(index):** This method is used to retrieve elements on the basis of index number.
3. **isEmpty():** This method tests if this set has no components.
4. **remove(index):** This method removes a single element on the basis of index number.
5. **clear():** This method clear all the elements from set.
6. **size():** This method returns the number of components in this set.
7. **removeAllElements():** This method removes all components from this set and sets its size to zero.
8. **indexOf(element):** This method returns first occurrence of given element or -1 if element is not present in set.
9. **lastIndexOf(element):** This method returns the last occurrence of given element or 1 if element is not present in set.

Collection Classes

1. Array List

The ArrayList class extends AbstractList and implements the List interface. ArrayList supports dynamic arrays that can grow as needed.

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

Methods of ArrayList

add(), get(), remove(), size(), clear(), isEmpty(), indexOf(), lastIndexOf(), contains(), get()

```

import java.util.ArrayList;
public class ArrayList_Example {
    public static void main(String args[]) {
        // create an array list
        ArrayList<String> al = new ArrayList<String>();
        System.out.println("Initial size of al: " + al.size());

        // add elements to the array list
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        al.add(1, "A2");
        System.out.println("Size of al after additions: " + al.size());

        // display the array list
        System.out.println("Contents of al: " + al);

        // Remove elements from the array list
        al.remove("F");
        al.remove(2);
        System.out.println("Size of al after deletions: " + al.size());
        System.out.println("Contents of al: " + al);
    }
}

```

Output:

```

Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]

```

2. Linked List

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at the contiguous location, the elements are linked using pointers.

//Methods are same as arraylist

```

import java.util.LinkedList;
public class ArrayList_Example {
    public static void main(String args[]) {
        // create an array list
        LinkedList<String> ll = new LinkedList<String>();
        System.out.println("Initial size of list: " + ll.size());

        // add elements to the array list
        ll.add("C");
        ll.add("A");
        ll.add("E");
        ll.add("B");
        ll.add("D");
        ll.add("F");
        ll.add(1, "A2");
        System.out.println("Size of list after additions: " + ll.size());

        // display the array list
        System.out.println("Contents of list: " + ll);

        // Remove elements from the array list
        ll.remove("F");
        ll.remove(2);
        System.out.println("Size of list after deletions: " + ll.size());
        System.out.println("Contents of list: " + ll);
    }
}

```

Output

```

Initial size of list: 0
Size of list after additions: 7
Contents of list: [C, A2, A, E, B, D, F]
Size of list after deletions: 5
Contents of list: [C, A2, E, B, D]

```

3. Hash Map

HashMap is a part of Java's collection since Java 1.2. It provides the basic implementation of Map interface of Java. It stores the data in (Key, Value) pairs. To access a value one must know its key. HashMap is known as Hash Map because it uses a technique called Hashing. Hashing is a technique of converting a large String to small String that represents the same String. A shorter value helps in indexing and faster searches.

//Methods is same as Hash set

```

import java.util.HashMap;

public class HashMap_Example {
    public static void main(String[] args)
    {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("vishal", 10);
        map.put("sachin", 30);
        map.put("vaibhav", 20);

        System.out.println(map);
        System.out.println("Size of map is:- " + map.size());

        if (map.containsKey("vishal"))
        {
            Integer a = map.get("vishal");
            System.out.println("value for key \"vishal\" is:- " + a);
        }

        map.clear();
        System.out.println("After Clearing Map: ");
        System.out.println(map);
    }
}

```

Output:

```

{vaibhav=20, vishal=10, sachin=30}
Size of map is:- 3 value for key
"vishal" is:- 10
After Clearing Map:
{}

```

4. Hash Set

The HashSet class implements the Set interface, backed by a hash table which is actually a HashMap instance. Few important features of HashSet are:

- Implements Set Interface.
- Underlying data structure for HashSet is hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in same order.
- Objects are inserted based on their hash key.
- NULL elements are allowed in HashSet.

//Methods are similar to map set interface

```

import java.util.HashSet;
public class HashSet_Example {
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();

        // Adding elements into HashSet
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add("India");// adding duplicate elements

        // Displaying the HashSet
        System.out.println(h);
        System.out.println("List contains India or not:" +
                           h.contains("India"));

        // Removing items from HashSet using remove()
        h.remove("Australia");
        System.out.println("List after removing Australia:\n"+h);
    }
}

```

Output:

```

[South Africa, Australia, India]
List contains India or not:true
List after removing Australia:
[South Africa, India]

```

5. Tree Set

TreeSet is one of the most important implementations of the SortedSet interface in Java that uses a Tree for storage. The ordering of the elements is maintained by a set using their natural ordering whether or not an explicit comparator is provided. This must be consistent with equals if it is to correctly implement the Set interface.

Few important features of TreeSet are as follows:

- TreeSet implements the SortedSet interface so duplicate values are not allowed.
- Objects in a TreeSet are stored in a sorted and ascending order.
- TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
- TreeSet does not allow to insert Heterogeneous objects. It will throw classCastException at Runtime if trying to add heterogeneous objects.
- TreeSet serves as an excellent choice for storing large amounts of sorted information which are supposed to be accessed quickly because of its faster access and retrieval time.

//Methods are same as set interface


```

import java.util.TreeSet;

public class TreeSet_Example {
    public static void main(String[] args)
    {
        TreeSet<String> h = new TreeSet<String>();

        // Adding elements into TreeSet
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add("India");// adding duplicate elements

        // Displaying the TreeSet
        System.out.println(h);
        System.out.println("List contains India or not:" +
                           h.contains("India"));

        // Removing items from TreeSet using remove()
        h.remove("Australia");
        System.out.println("List after removing Australia:\n"+h);
    }
}

```

Output:

```

[Australia, India, South Africa] List
contains India or not:true
List after removing Australia:
[India, South Africa]

```

Accessing Collections using Iterator

```
import java.util.HashSet;
import java.util.Iterator;

public class Iterator_Example {
    public static void main(String[] args)
    {
        HashSet<String> h = new HashSet<String>();
        // Adding elements into HashSet using add()
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add("India");// adding duplicate elements

        // Iterating over hash set items
        System.out.println("Iterating over list:");
        Iterator<String> i = h.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

Output:

Iterating over list:
South Africa
Australia
India

Comparator in Java

Comparator in Java are very useful for sorting the collection of objects. Java provides some inbuilt methods to sort primitive types array or Wrapper classes array or list.

```
import java.util.ArrayList;
import java.util.Collections;

public class Comparator_Example {
    public static void main(String args[]) {
        ArrayList<String> list = new ArrayList<String>();
        list.add("Ram");
        list.add("Shyam");
        list.add("Hari");
        Collections.sort(list); // Sorts the array list
        System.out.println(list);
    }
}
```

Output:

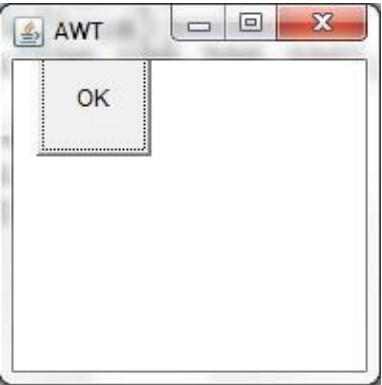

[Hari, Ram, Shyam]

Unit – 11 Java Applications [8 Hrs.]

About AWT & Swing

AWT and Swing both are used to create GUI interface in Java. Both of them are used to perform almost same work, still they differ from each other.

Following are the differences between AWT and Swing:

AWT	Swing
AWT stands for Abstract Window Toolkit.	Swing is a part of Java Foundation Class (JFC).
AWT components are heavy weight.	Swing components are light weight.
AWT components are platform dependent so there look and feel changes according to OS.	Swing components are platform independent so there look and feel remains constant.
AWT components are not very good in look and feel as compared to Swing components. See the button in below image, its look is not good as button created using Swing. 	Swing components are better in look and feel as compared to AWT. See the button in below image, its look is better than button created using AWT. 

JFrame(a top level window in Swing)

Whenever you create a graphical user interface with Java Swing functionality, you will need a container for your application. In the case of Swing, this container is called a JFrame. All GUI applications require a JFrame. In fact, some Applets even use a JFrame. Why?

You can't build a house without a foundation. The same is true in Java: Without a container in which to put all other elements, you won't have a GUI application. In other words, the JFrame is required as the foundation or base container for all other graphical components.

Java Swing applications can be run on any system that supports Java. These applications are lightweight. This means that don't take up much space or use many system resources.

JFrame is a class in Java and has its own methods and constructors. Methods are functions that impact the JFrame, such as setting the size or visibility. Constructors are run when the instance is created: One constructor can create a blank JFrame, while another can create it with a default title.

When a new JFrame is created, you actually create an instance of the JFrame class. You can create an empty one, or one with a title. If you pass a string into the constructor, a title is created as follows:

```
1. JFrame f = new JFrame();
2. // Or overload the constructor and give it a title:
3. JFrame f2 = new JFrame("The Twilight Zone");
```

JPanel in Swing

The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class. It doesn't have title bar.

```
JFrame f= new JFrame("Panel Example");
JPanel panel=new JPanel(); JLabel
lbl=new JLabel("Testing label");
panel.add(lbl); f.add(panel);
```

Swing Components and Containers

A component is an independent visual control. Swing Framework contains a large set of components which provide rich functionalities and allow high level of customization. They all are derived from JComponent class. All these components are lightweight components. This class provides some common functionality like pluggable look and feel, support for accessibility, drag and drop, layout, etc.

A container holds a group of components. It provides a space where a component can be managed and displayed. Containers are of two types:

1. **Top level Containers** ○ It inherits Component and Container of AWT. ○ It cannot be contained within other containers. ○ Heavyweight.
 - Example: JFrame, JDialog, JApplet
2. **Lightweight Containers** ○ It inherits JComponent class.
 - It is a general purpose container. ○ It can be used to organize related components together.
 - Example: JPanel

JButton

JButton is in implementation of a push button. It is used to trigger an action if the user clicks on it. JButton can display a text, an icon, or both.

```
JButton btn=new JButton("Click Me");
```

JLabel

Label is a simple component for displaying text, images or both. It does not react to input events.

```
JLabel lbl=new JLabel("This is a label");
```

TextField

TextField is a text component that allows editing of a single line of non-formatted text.

```
TextField txt=new TextField();
```

CheckBox

CheckBox is a box with a label that has two states: on and off. If the check box is selected, it is represented by a tick in a box. A check box can be used to show or hide a splashscreen at startup, toggle visibility of a toolbar etc.

```
JCheckBox chk1=new JCheckBox("BCA");
```

```
JCheckBox chk2=new JCheckBox("BBA");
```

RadioButton

RadioButton allows the user to select a single exclusive choice from a group of options. It is used with the ButtonGroup component.

```
JRadioButton rad1=new JRadioButton ("Male");
```

```
JRadioButton red2=new JRadioButton ("Female");
```

```
ButtonGroup grp=new ButtonGroup();
```

```
grp.add(rad1); grp.add(rad2);
```

ComboBox

JComboBox is a component that combines a button or editable field and a drop-down list. The user can select a value from the drop-down list, which appears at the user's request. If you make the combo box editable, then the combo box includes an editable field into which the user can type a value.

```
String arr[]={ "BCA", "BBA", "MCA", "MBA" };
```

```
JComboBox cmb=new JComboBox(arr);
```

```
//or
```

```
JComboBox<String> cmb=new JComboBox<String>();
```

```
cmb.addItem("BCA"); cmb.addItem("BBA");
```

```
cmb.addItem("MCA"); cmb.addItem("MBA");
```

JList

JList is a component that displays a list of objects. It allows the user to select one or more items.

```
String arr[]={"BCA","BBA","MCA","MBA"};
```

```
JList cmb=new JList(arr);
```

JTextArea

A JTextArea is a multiline text area that displays plain text. It is lightweight component for working with text. The component does not handle scrolling. For this task, we use JScrollPane component.

```
JTextArea txt=new JTextArea();
```

JTable

The JTable class is a part of Java Swing Package and is generally used to display or edit twodimensional data that is having both rows and columns. It is similar to a spreadsheet. This arranges data in a tabular form.

```
// Data to be displayed in the JTable
String[][] data = {
    { "Kundan Kumar Jha", "4031", "CSE" },
    { "Anand Jha", "6014", "IT" }
};

// Column Names
String[] columnNames = { "Name", "Roll Number", "Department" };
// Initializing the JTable
JTable j = new JTable(data, columnNames);
```

JMenu

The JMenuBar class is used to display menubar on the window or frame. It may have several menus. The object of JMenu class is a pull down menu component which is displayed from the menu bar. It inherits the JMenuItem class.

The object of JMenuItem class adds a simple labeled menu item. The items used in a menu must belong to the JMenuItem or any of its subclass.

```
JMenuBar mb=new JMenuBar();
JMenu menu1=new JMenu("Courses");
JMenuItem item1=new JMenuItem("BCA");
JMenuItem item2=new
JMenuItem("BBA"); menu1.add(item1);
menu1.add(item2); mb.add(menu1);
```

Dialog Boxes in Swing

Dialog windows or dialogs are an indispensable part of most modern GUI applications. A dialog is defined as a conversation between two or more persons. In a computer application a dialog is a window which is used to "talk" to the application. A dialog is used to input data, modify data, change the application settings etc. Dialogs are important means of communication between a user and a computer program.

In Java Swing, we can create two kinds of dialogs: **standard dialogs and custom dialogs**. Custom dialogs are created by programmers. They are based on the `JDialog` class. Standard dialogs are predefined dialogs available in the Swing toolkit, for example the `JColorChooser` or the `JFileChooser`. These are dialogs for common programming tasks like showing text, receiving input, loading and saving files. They save programmer's time and enhance using some standard behaviour.

Modal and Modeless Dialog

There are two basic types of dialogs: modal and modeless. Modal dialogs block input to other top-level windows. Modeless dialogs allow input to other windows. An open file dialog is a good example of a modal dialog. While choosing a file to open, no other operation should be permitted. A typical modeless dialog is a find text dialog. It is handy to have the ability to move the cursor in the text control and define, where to start the finding of the particular text.

Standard Dialog Example – Message Box

Message dialogs are simple dialogs that provide information to the user. Message dialogs are created with the `JOptionPane.showMessageDialog()` method.

```
JOptionPane.showMessageDialog(null,"This is a test message");
```

Custom Dialog Creation

```
JDialog jd=new JDialog();  
jd.setTitle("This is a Test Dialog");  
JLabel lbl=new JLabel("Do you want to exit? "); jd.add(lbl);  
JButton yes=new JButton("Yes"); jd.add(yes);
```

Layout Management

In Java swing, Layout manager is used to position all its components, with setting properties, such as the size, the shape and the arrangement. Following are the different types of layout managers:

1. Flow Layout
2. Border layout
3. Grid Layout

Flow Layout

The FlowLayout arranges the components in a directional flow, either from left to right or from right to left. Normally all components are set to one row, according to the order of different components. If all components cannot be fit into one row, it will start a new row and fit the rest in.

```
import javax.swing.*; import
java.awt.FlowLayout;

public class FlowLayoutExample {
    public static void main(String[] args)
    {
        // Create and set up a frame window
        JFrame frame = new JFrame("Layout");
        // Define new buttons
        JButton jb1 = new JButton("Button 1");
        JButton jb2 = new JButton("Button 2");
        JButton jb3 = new JButton("Button 3");
        // Define the panel to hold the buttons
        JPanel panel = new JPanel();
        panel.setLayout(new FlowLayout()); //setting flowlayout
        panel.add(jb1);          panel.add(jb2);          panel.add(jb3);

        // Set the window to be visible as the default to be false
        frame.add(panel);          frame.pack();
        frame.setVisible(true);
    }
}
```



Border Layout

A BorderLayout lays out a container, arranging its components to fit into five regions: NORTH, SOUTH, EAST, WEST and CENTER. For each region, it may contain no more than one component. When adding different components, you need to specify the orientation of it to be the one of the five regions.

For BorderLayout, it can be constructed like below:

- **BorderLayout():** construct a border layout with no gaps between components.
 - **BorderLayout(int hgap, int vgap):** construct a border layout with specified gaps between components.
- ```
// Define new buttons with different regions
```

```

JButton jb1 = new JButton("NORTH");
JButton jb2 = new JButton("SOUTH");
JButton jb3 = new JButton("WEST");
JButton jb4 = new JButton("EAST");
JButton jb5 = new JButton("CENTER");
// Define the panel to hold the buttons
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
panel.add(jb1, BorderLayout.NORTH);
panel.add(jb2, BorderLayout.SOUTH);
panel.add(jb3, BorderLayout.WEST);
panel.add(jb4, BorderLayout.EAST);
panel.add(jb5, BorderLayout.CENTER);

```



## Grid Layout

The GridLayout manager is used to lay out the components in a rectangle grid, which has been divided into equal-sized rectangles and one component is placed in each rectangle. It can be constructed with the following methods:

- **GridLayout():** construct a grid layout with one column per component in a single row.
- **GridLayout(int row, int col):** construct a grid layout with specified numbers of rows and columns.
- **GridLayout(int row, int col, int hgap, int vgap):** construct a grid layout with specified rows, columns and gaps between components.

```

// Define new buttons
JButton jb1 = new JButton("Button 1");
JButton jb2 = new JButton("Button 2");
JButton jb3 = new JButton("Button 3");
JButton jb4 = new JButton("Button 4");
JButton jb5 = new JButton("Button 5");

// Define the panel to hold the buttons
JPanel panel = new JPanel();
panel.setLayout(new GridLayout(3, 2));
panel.add(jb1); panel.add(jb2);
panel.add(jb3); panel.add(jb4);
panel.add(jb5);

```



## **MDI using JDesktop Pane and JInternal Frame**

MDI- Multi Document Interface

The JDesktopPane class, can be used to create "multi-document" applications. A multidocument application can have many windows included in it. The JDesktopPane is a container which is used to create a multiple-document interface or a virtual desktop. The JFrame inside the desktop becomes JInternalFrame. JInternalFrame is used just like the JFrame but is added to JDesktopPane object.

```
JDesktopPane jd=new JDesktopPane();
```

```
JInternalFrame frame1=new JInternalFrame("Internal
Frame1",true,true,true,true);
frame1.setLayout(null);
frame1.setSize(200,100);
frame1.setVisible(true);
```

```
JInternalFrame frame2=new JInternalFrame("Internal
Frame2",true,true,true,true);
frame2.setLayout(null);
frame2.setSize(200,100);
frame2.setVisible(true);
```

```
JInternalFrame frame3=new JInternalFrame("Internal
Frame3",true,true,true,true);
frame3.setLayout(null);
frame3.setSize(200,100);
frame3.setVisible(true);
jd.add(frame1);
jd.add(frame2);
jd.add(frame3);
```



## **Event Handling**

Any program that uses GUI (graphical user interface) such as Java application written for windows, is event driven. Event describes the change in state of any object. For Example: Pressing a button, entering a character in Textbox, Clicking or Dragging a mouse, etc.

Event handling has three main components,

- **Events:** An event is a change in state of an object.
- **Events Source:** Event source is an object that generates an event.
- **Listeners:** A listener is an object that listens to the event. A listener gets notified when an event occurs.

### **How Events are handled?**

A source generates an Event and send it to one or more listeners registered with the source. Once event is received by the listener, they process the event and then return. Events are supported by a number of Java packages, like **java.util**, **java.awt** and **java.awt.event**.

### **Important Event Classes and Interface**

| Event Classes      | Description                                                                                                    | Listener Interface |
|--------------------|----------------------------------------------------------------------------------------------------------------|--------------------|
| <b>ActionEvent</b> | generated when button is pressed, menu-item is selected, list-item is double clicked                           | ActionListener     |
| <b>MouseEvent</b>  | generated when mouse is dragged, moved,clicked,pressed or released and also when it enters or exit a component | MouseListener      |
| <b>KeyEvent</b>    | generated when input is received from keyboard                                                                 | KeyListener        |
| <b>ItemEvent</b>   | generated when check-box or list item is clicked                                                               | ItemListener       |
| <b>TextEvent</b>   | generated when value of textarea or textfield is changed                                                       | TextListener       |

|                        |                                                                                           |                        |
|------------------------|-------------------------------------------------------------------------------------------|------------------------|
| <b>MouseEvent</b>      | generated when mouse wheel is moved                                                       | MouseListener          |
| <b>WindowEvent</b>     | generated when window is activated, deactivated, deiconified, iconified, opened or closed | WindowListener         |
| <b>ComponentEvent</b>  | generated when component is hidden, moved, resized or set visible                         | ComponentEventListener |
| <b>ContainerEvent</b>  | generated when component is added or removed from container                               | ContainerListener      |
| <b>AdjustmentEvent</b> | generated when scroll bar is manipulated                                                  | AdjustmentListener     |
| <b>FocusEvent</b>      | generated when component gains or loses keyboard focus                                    | FocusListener          |

### Example of ActionListener

```

JButton btn=new JButton("Click Here"); btn.addActionListener(new
ActionListener(){
 public void actionPerformed(ActionEvent ae){
 //event handling code here
 }
});

```

### Example of ItemListener

```

JComboBox cmb=new JComboBox(arr); cmb.addItemListener(new
ItemListener(){ public void actionPerformed(ItemEvent ie){
 if
 (ie.getStateChange() == ItemEvent.SELECTED) //event
 handling code here
 }
});

```

### Keyboard and Mouse Events

The Java **MouseListener** is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

```

void mouseClicked(MouseEvent e);
void mouseEntered(MouseEvent e);
void mouseExited(MouseEvent e); void
mousePressed(MouseEvent e);
void mouseReleased(MouseEvent e);

```

**Mouse Event Example** **import** java.awt.\*; **import** java.awt.event.\*; **public class**

```

MouseListenerExample extends Frame implements MouseListener

```

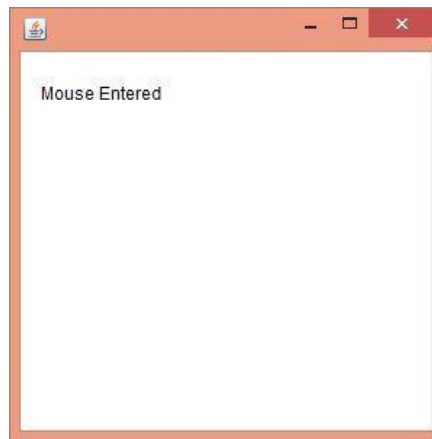
```

{
 Label l;
 MouseListenerExample(){
 addMouseListener(this);

 l=new Label();
 l.setBounds(20,50,100,20);
 add(l);
 setSize(300,300);
 setLayout(null); setVisible(true);
 }

 public void mouseClicked(MouseEvent e) {
 l.setText("Mouse Clicked"); }
 public void mouseEntered(MouseEvent e) {
 l.setText("Mouse Entered"); }
 public void mouseExited(MouseEvent e) {
 l.setText("Mouse Exited"); }
 public void mousePressed(MouseEvent e) {
 l.setText("Mouse Pressed"); }
 public void mouseReleased(MouseEvent e) {
 l.setText("Mouse Released"); }
 public static void main(String[] args) {
 new MouseListenerExample();
 }
}

```



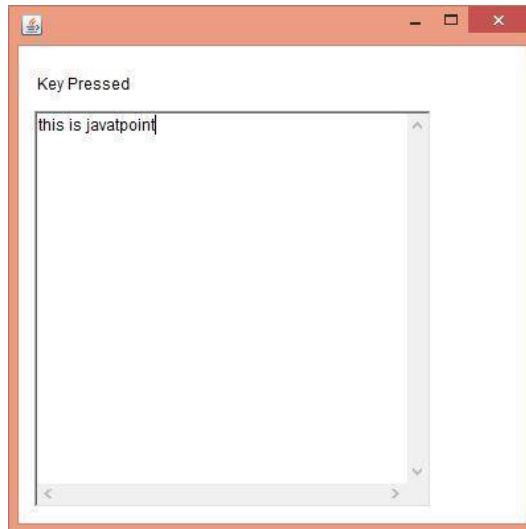
The Java **KeyListener** is notified whenever you change the state of key. It is notified against **KeyEvent**. The **KeyListener** interface is found in **java.awt.event** package. It has three methods.

```
void keyPressed(KeyEvent e); void
keyReleased(KeyEvent e); void
keyTyped(KeyEvent e);
```

#### Key Event Example **import**

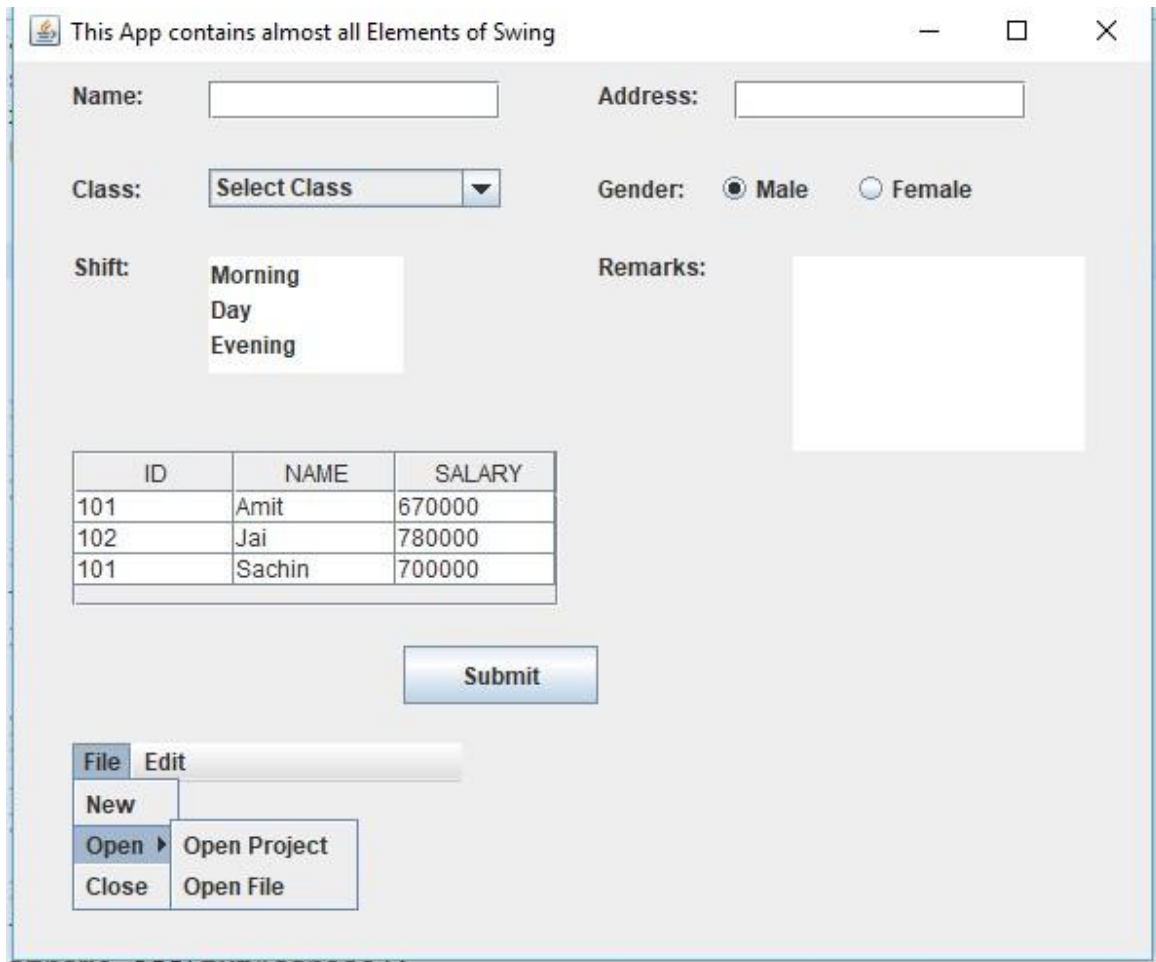
```
java.awt.*;
import
java.awt.event.*;
public class KeyListenerExample extends Frame implements KeyListener{
 Label l;
 TextArea area;
 KeyListenerExample(){
l=new Label();
 l.setBounds(20,50,100,20);
 area=new TextArea();
 area.setBounds(20,80,300, 300);
 area.addKeyListener(this);
 add(l);add(area);
 setSize(400,400);
 setLayout(null); setVisible(true);
 }
 public void keyPressed(KeyEvent e) {
l.setText("Key Pressed"); }
 public void keyReleased(KeyEvent e) {
l.setText("Key Released"); }
 public void keyTyped(KeyEvent e) {
l.setText("Key Typed");
 }
}
```

```
public static void main(String[] args) {
 new KeyListenerExample();
}
```



## Java Swing Examples

### Example 1



```
import java.awt.event.*;
import javax.swing.*; public
class SwingExample{
 SwingExample(){
 JFrame jframe=new JFrame("This App contains almost all
 Elements of Swing");
 jframe.setSize(600, 500);
 jframe.setLocationRelativeTo(null);
 jframe.setLayout(null);
 jframe.setVisible(true);
```

```
 //Name
 JLabel lblName=new JLabel("Name: ");
 lblName.setBounds(30, 12, 150, 10);
 jframe.add(lblName);
```

```
 JTextField txtName=new JTextField();
 txtName.setBounds(100, 10, 150, 20);
 jframe.add(txtName);
```

```
 //Address
```

```

 JLabel lblAddress=new JLabel("Address: ");
 lblAddress.setBounds(300, 12, 150, 10);
 jframe.add(lblAddress);

 JTextField txtAddress=new JTextField();
 txtAddress.setBounds(370, 10, 150, 20);
 jframe.add(txtAddress);

 //Class
 JLabel lblClass=new JLabel("Class: ");
 lblClass.setBounds(30, 60, 150, 10);
 jframe.add(lblClass);

 JComboBox<String> cmbClass=new JComboBox<String>();
 cmbClass.addItem("Select Class");
 cmbClass.addItem("BCA"); cmbClass.addItem("BBA");
 cmbClass.addItem("MCA");
 cmbClass.addItem("MBA");

 /*String[] clas= {"BCA","BBA"};
 JComboBox cmbClass=new JComboBox(clas);*/

 cmbClass.setBounds(100, 55, 150, 20);
 //cmbClass.setEditable(true);
 jframe.add(cmbClass);

 //Gender
 JLabel lblSex=new JLabel("Gender: ");
 lblSex.setBounds(300,60,80,10); jframe.add(lblSex);

 ButtonGroup group1=new ButtonGroup();
 JRadioButton chkMale=new JRadioButton("Male",true);
 chkMale.setBounds(360, 55, 70, 20); jframe.add(chkMale);
 group1.add(chkMale);

 JRadioButton chkFemale=new
 JRadioButton("Female"); chkFemale.setBounds(430,
 55, 70, 20); jframe.add(chkFemale);
 group1.add(chkFemale);

 //Shift
 JLabel lblShift=new JLabel("Shift: ");
 lblShift.setBounds(30,100,80,10);
 jframe.add(lblShift);

 String[] shift= {"Morning","Day","Evening"};
 JList list1=new JList(shift);
 list1.setBounds(100, 100, 100, 60);
 jframe.add(list1);

```

```

 //Remarks
 JLabel lblRemarks=new JLabel("Remarks: ");
 lblRemarks.setBounds(300,100,80,10);
 jframe.add(lblRemarks);

 JTextArea txtArea=new JTextArea();
 txtArea.setBounds(400, 100, 150, 100);
 jframe.add(txtArea);

 //JTable
 String data[][]={ {"101","Amit","670000"},
 {"102","Jai","780000"},
 {"101","Sachin","700000"} };
 String column[]={"ID","NAME","SALARY"};
 JTable jt=new JTable(data,column);
 JScrollPane sp=new JScrollPane(jt);
 sp.setBounds(30,200,250,80);
 jframe.add(sp);

 JButton click=new JButton("Submit");
 click.setBounds(200,300,100,30); jframe.add(click);

 //creating menu
 JMenuBar mb=new JMenuBar(); mb.setBounds(30, 350,
 200, 20);

 JMenu menu1=new JMenu("File");
 JMenuItem item1,item3;
 item1=new JMenuItem("New"); JMenu
 item2=new JMenu("Open"); JMenuItem
 i1,i2; i1=new JMenuItem("Open
 Project");
 i2=new JMenuItem("Open
 File"); item2.add(i1);
 item2.add(i2); item3=new
 JMenuItem("Close");
 menu1.add(item1);
 menu1.add(item2);
 menu1.add(item3);
 mb.add(menu1);

 JMenu menu2=new
 JMenu("Edit"); JMenuItem it1,it2;
 it1=new JMenuItem("Copy");
 it2=new JMenuItem("Paste");
 menu2.add(it1);
 menu2.add(it2);
 mb.add(menu2);
 jframe.add(mb);

```



```

click.addActionListener(new
ActionListener() {
public void
actionPerformed(ActionEvent ae)
{
 //get textfield
 String name=txtName.getText();

 //get combo
 String clas=cmbClass.getSelectedItem().toString();

 //get radio + checkbox (same process)
 String gender="";
 if(chkMale.isSelected())
gender=chkMale.getText();
 else
 gender=chkFemale.getText();

 //get list
 String shift=list1.getSelectedValuesList().toString();

 //get textarea
 String remarks=txtArea.getText();
 JOptionPane.showMessageDialog(null, "Remarks: "+remarks);
}
});

//can use action listener also
cmbClass.addItemListener(new ItemListener() {
void itemStateChanged(ItemEvent e) {
if(e.getStateChange()==ItemEvent.SELECTED)
JOptionPane.showMessageDialog(null,
cmbClass.getSelectedItem().toString());
}
});

// listener on menu item //can use
item listener also item1.addActionListener(new
ActionListener() {
public void actionPerformed(ActionEvent ae) {
JOptionPane.showMessageDialog(null,
item1.getText().toString()+" Clicked!");
}
});
}

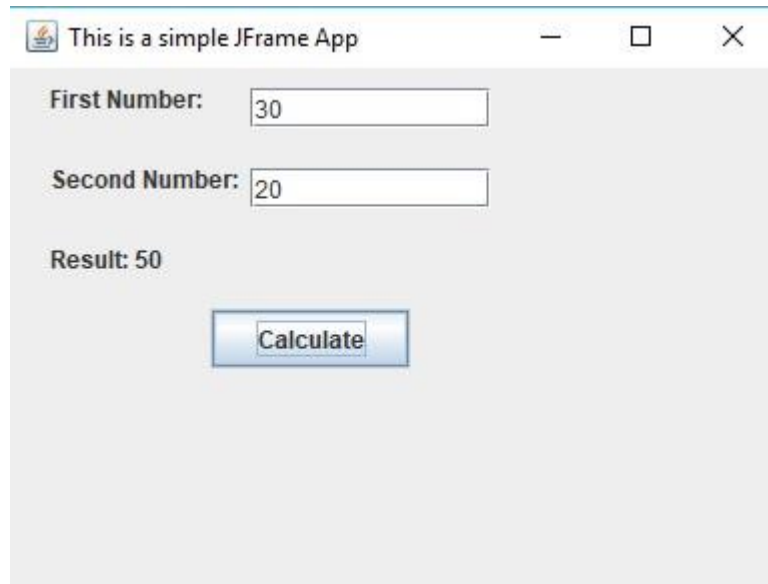
```

```

 public static void main(String[] args) {
 SwingUtilities.invokeLater(new Runnable() {
 public void run() {
 new SwingExample();
 }
 });
 }
}

```

## Example 2



```

import javax.swing.*; import
java.awt.event.*; public
class SwingDemo {

 SwingDemo(){
 JFrame jframe=new JFrame("This is a simple JFrame App");
 jframe.setSize(400, 300);
 jframe.setLocationRelativeTo(null);
 jframe.getContentPane().setLayout(null);
 jframe.setVisible(true);

 JLabel lbl1=new JLabel("First Number:");
 lbl1.setBounds(20, 10, 100, 10);
 jframe.add(lbl1);

 JTextField txt1=new JTextField();
 txt1.setBounds(120, 10, 120, 20);
 jframe.add(txt1);
 }
}

```

```

 JLabel lbl2=new JLabel("Second Number:");
 lbl2.setBounds(20, 50, 100, 10);
 jframe.add(lbl2);

 JTextField txt2=new JTextField();
 txt2.setBounds(120, 50, 120, 20);
 jframe.add(txt2);

 JLabel lbl3=new JLabel("Result: ");
 //lbl3.setText("Result: ");
 lbl3.setBounds(20,80,100,30);
 jframe.add(lbl3);

 JButton btn=new JButton("Calculate");
 btn.setBounds(100, 120, 100, 30);
 jframe.add(btn);

 btn.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent ae) {

 String first1=txt1.getText().toString();
 String second1=txt2.getText().toString();
 int a,b,c;
 a=Integer.parseInt(first1);
 b=Integer.parseInt(second1);
 c=a+b;
 lbl3.setText("Result: "+c);

 //JOptionPane.showMessageDialog(null, "Addition= "+c);

 }
 });
 }

 public static void main(String[] args)
 {
 SwingUtilities.invokeLater(new Runnable() {
 public void run() {
 new SwingDemo();
 }
 });
 }
}

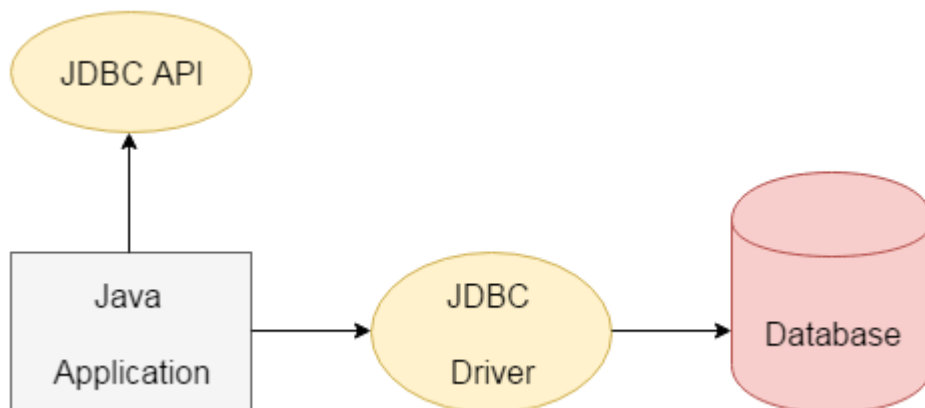
```

## **Unit – 13 Java Database Connectivity [2 Hrs.]**

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database. There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver, ○  
Native Driver,
- Network Protocol Driver, and
- Thin Driver

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



We can use JDBC API to handle database using Java program and can perform the following activities:

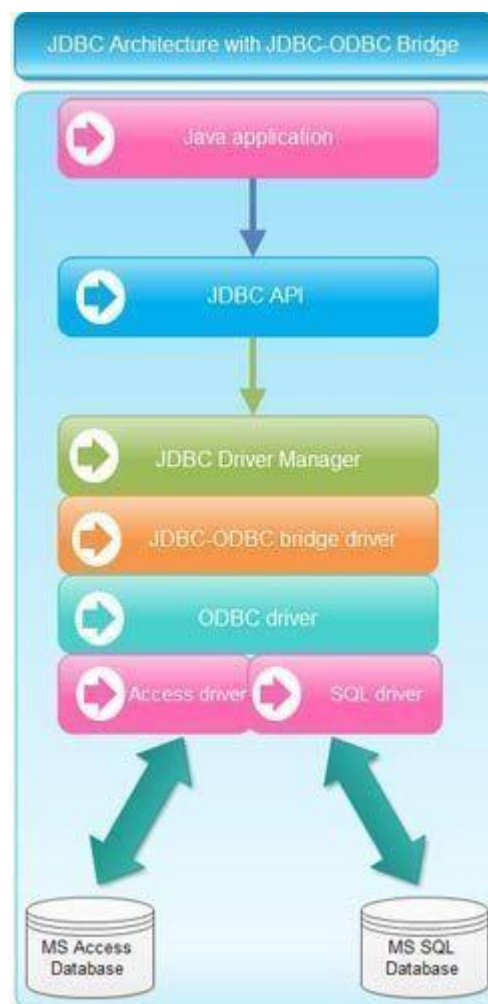
1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

## **JDBC-ODBC Bridge**

Microsoft's ODBC (Open Database Connectivity) is the most commonly used driver to connect to the database as it can connect to almost all databases on most of the platforms. However, ODBC uses the concept of pointers and other constructs that are not supported by Java.

Therefore, JDBC-ODBC bridge driver was developed which translates the JDBC API to the ODBC API and vice versa. This bridge acts as interface that enables all DBMS which support ODBC (Open Data Base Connectivity) to interact with Java Applications. JDBC-ODBC bridge is implemented as a class file and a native library. The name of the class file is JdbcOdbc.class.

Figure shows the JDBC application architecture in which a front-end application uses JDBC API for interacting with JDBC Driver Manager. Here, JDBC Driver Manager is the backbone of JDBC architecture. It acts as interface that connects a Java application to the driver specified in the Java program. Next, is the JDBC-ODBC bridge which helps the JDBC to access ODBC data sources.



## **Making ODBC Connection**

```
//JDBC-ODBC Bridge (But removed from Java 8)
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
String url = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb,
 *.accdb)};DBQ=" + "D:\\mydb.accdb";
Connection conn = DriverManager.getConnection(url);
```

### Example-1

```
import java.sql.*;

public class DB_Test {
 public static void main(String[] args) {
 try {
 Class.forName("com.mysql.jdbc.Driver");
 Connection conn=DriverManager.getConnection
 ("jdbc:mysql://localhost:3306/bca","root","");
 //database="", username=root and password=""

 System.out.println("Database connected");
 Statement st=conn.createStatement();

 //clearing data
 String sql="DELETE FROM student";
 st.execute(sql);

 //inserting data
 String sql1="INSERT INTO
 student(sid,name,address,contact) VALUES
 ('1','Raaju','Btm','9862612723'),
 ('2','Ram','Ktm','8811111111)";
 st.execute(sql1);
 System.out.println("Data Inserted Successfully\n");

 //retrieving data
 System.out.println("Data Before Update and Delete");
 String sql4="SELECT * FROM student";
 ResultSet rs=st.executeQuery(sql4);
```

```

System.out.println("Sid\t"+"Name\t"+"address\t"+"Contact");
 while(rs.next()) {
 String sid=rs.getString(1);
 String name=rs.getString(2);
 String address=rs.getString(3);
 String contact=rs.getString(4);

System.out.println(sid+"\t"+name+"\t"+address+"\t"+contact);
 }

 //updating data
 String sql2="UPDATE student SET
 name='Hari',address='Ktm' WHERE sid='1'";
 st.execute(sql2);
 System.out.println("\nData Updated Successfully"); //deleting
data
 String sql3="DELETE FROM student WHERE sid='2'";
 st.execute(sql3);
 System.out.println("Data Deleted Successfully\n");

 //retrieving data
 System.out.println("Data After Update and Delete");
 String sql5="SELECT * FROM student";
 ResultSet rs1=st.executeQuery(sql5);

 System.out.println("Sid\t"+"Name\t"+"address\t"+"Contact");
 while(rs1.next()) {
 String sid=rs1.getString(1);
 String name=rs1.getString(2);
 String address=rs1.getString(3);
 String contact=rs1.getString(4);

System.out.println(sid+"\t"+name+"\t"+address+"\t"+contact);
 }

 }catch(Exception e) {
 System.out.println(e);
 }
}
}

```

### **Output**

Database connected

Data Inserted Successfully

Data Before Update and Delete

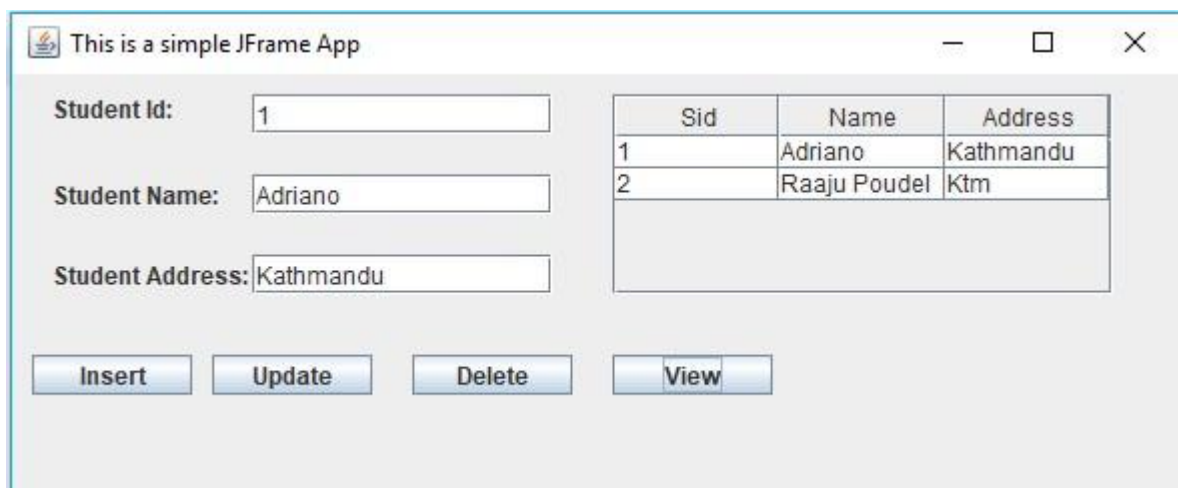
| Sid | Name  | address | Contact    |
|-----|-------|---------|------------|
| 1   | Raaju | Btm     | 9862612723 |
| 2   | Ram   | Ktm     | 8811111111 |

Data Updated Successfully  
Data Deleted Successfully

Data After Update and Delete  

| Sid | Name | address | Contact    |
|-----|------|---------|------------|
| 1   | Hari | Ktm     | 9862612723 |

## Example-2



```
import javax.swing.*; import
javax.swing.table.*; import
java.awt.event.*; import
java.sql.*; public class
SwingDatabase {

 Connection conn;
 Statement st;
 //creating connection
 void getConnection() {
 try {
 Class.forName("com.mysql.jdbc.Driver");
 conn = DriverManager.getConnection
 ("jdbc:mysql://localhost:3306/bca","root","");
 st=conn.createStatement();
 } catch (Exception e) {
 JOptionPane.showMessageDialog(null, e);
 }
 }
}
```



```

 }
}

SwingDatabase(){
getConnection();
 JFrame jframe=new JFrame("This is a simple JFrame App");
 jframe.setSize(600, 250);
jframe.setLocationRelativeTo(null);
jframe.getContentPane().setLayout(null);
jframe.setVisible(true);

 JLabel lbl1=new JLabel("Student Id:");
 lbl1.setBounds(20, 12, 100, 10); jframe.add(lbl1);

 JTextField txt1=new JTextField();
 txt1.setBounds(120, 10, 150, 20);
 jframe.add(txt1);

 JLabel lbl2=new JLabel("Student Name:");
 lbl2.setBounds(20, 55, 100, 10);
 jframe.add(lbl2);

 JTextField txt2=new JTextField();
 txt2.setBounds(120, 50, 150, 20);
 jframe.add(txt2);

 JLabel lbl3=new JLabel("Student Address: ");
 lbl3.setBounds(20,85,120,30); jframe.add(lbl3);

 JTextField txt3=new JTextField();
 txt3.setBounds(120, 90, 150, 20);
 jframe.add(txt3);

 JButton insert=new JButton("Insert");
 insert.setBounds(10, 140, 80, 20); jframe.add(insert);

 JButton update=new JButton("Update");
 update.setBounds(100, 140, 80, 20); jframe.add(update);

 JButton delete=new JButton("Delete");
 delete.setBounds(200, 140, 80, 20); jframe.add(delete);

 JButton view=new JButton("View");
 view.setBounds(300, 140, 80, 20);
 jframe.add(view);

```

```

//insert
insert.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
String id=txt1.getText().toString();
String name=txt2.getText().toString();
String address=txt3.getText().toString();
try {
String sql="INSERT INTO student
(sid,name,address)
VALUES('"+id+"','"+name+"','"+address+"')";
st.execute(sql);
JOptionPane.showMessageDialog
(null, "Data Inserted Successfully");
}
catch(Exception e) {
JOptionPane.showMessageDialog(null, e);
}
}
});

```

```

//update
update.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
String id=txt1.getText().toString();
String name=txt2.getText().toString();
String address=txt3.getText().toString();
try {
String sql="UPDATE student SET
name='"+name+"',address='"+address+"' WHERE
sid='"+id+"'";
st.execute(sql);
JOptionPane.showMessageDialog
(null, "Data Updated Successfully");
} catch(Exception e) {
JOptionPane.showMessageDialog(null, e);
}
}
});

```

```

//delete
delete.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
String id=txt1.getText().toString();
try {

```

```

 String sql="DELETE FROM student
 WHERE sid='"+id+"'";
 st.execute(sql);
 JOptionPane.showMessageDialog
 (null, "Data Deleted Successfully");
 }catch(Exception e) {
 JOptionPane.showMessageDialog(null, e);
 }

 }

});

//view
view.addActionListener(new ActionListener() {
 public void actionPerformed(ActionEvent ae) {
 //plotting in JTable
 String[] columns= {"Sid","Name","Address"};
 JTable jt=new JTable();
 DefaultTableModel model = new DefaultTableModel();
 jt.setModel(model);
 model.setColumnIdentifiers(columns);
 try {
 String sql="SELECT * FROM student";
 ResultSet rs=st.executeQuery(sql);
 while(rs.next()) {
 String id=rs.getString(1);
 String name=rs.getString(2);
 String address=rs.getString(3);
 model.addRow(new
 String[]{id,name,address});
 }
 JScrollPane sp=new JScrollPane(jt);
 sp.setBounds(300, 10, 250, 100);
 jframe.add(sp);

 }catch(Exception e) {
 JOptionPane.showMessageDialog(null, e);
 }

 }

});

}

public static void main(String[] args)
{
 SwingUtilities.invokeLater(new Runnable() {

```

```

 public void run() {
 new SwingDatabase();
 }
 });
}
}

```

## **Unit – 12 Introduction to Java Applets [1 Hr.]**

### **Definition**

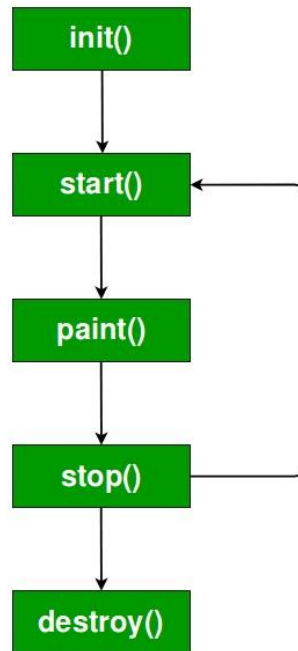
Applet is a Java program that can be embedded into a web page. It runs inside the web browser and works at client side. Applet is embedded in a HTML page using the APPLET or OBJECT tag and hosted on a web server.

Applets are used to make the web site more dynamic and entertaining.

Following are the features of Java Applet:

1. All applets are sub-classes (either directly or indirectly) of [java.applet.Applet](#) class.
2. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
3. In general, execution of an applet does not begin at main() method.
4. Output of an applet window is not performed by **System.out.println()**. Rather it is handled with various AWT methods, such as **drawString()**.

### **Life cycle of an applet:**



When an applet begins, the following methods are called, in this sequence:

1. `init( )`
2. `start( )`
3. `paint( )`

When an applet is terminated, the following sequence of method calls takes place:

1. `stop( )`
2. `destroy( )`

1. **`init( )`** : The **`init( )`** method is the first method to be called. This is where you should initialize variables. This method is called **only once** during the run time of your applet.
2. **`start( )`** : The **`start( )`** method is called after **`init( )`** . It is also called to restart an applet after it has been stopped. Note that **`init( )`** is called once i.e. when the first time an applet is loaded whereas **`start( )`** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **`start( )`** .
3. **`paint( )`** : The **`paint( )`** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **`paint( )`** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **`paint( )`** is called. The **`paint( )`** method has one parameter of type [Graphics](#). This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
4. **`stop( )`** : The **`stop( )`** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **`stop( )`** is called, the applet is probably running. You should use **`stop( )`** to suspend threads that

don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

5. **destroy( )** : The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

### Creating a simple applet program:

```
import java.applet.Applet; import
java.awt.Graphics; public class
Applet1 extends Applet {
 // Overriding paint() method
 @Override
 public void paint(Graphics g)
 {
 g.drawString("Hello World", 20, 20);
 }
}
```

### Output

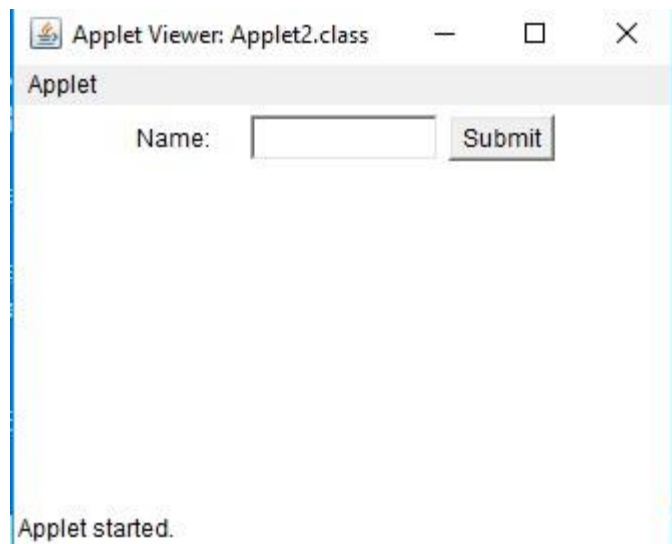


Applet started.

### Adding Controls to Applet

```
import
java.applet.*; import java.awt.*;
public class Applet2 extends Applet {
 public void init() {
 //adding label
 Label lbl=new Label("Name: ");
 add(lbl);
 //adding textfield
 TextField txt=new TextField(10);
 add(txt);
 //adding button
 Button btn=new Button("Submit");
 add(btn);
 }
}
```

### Output



### Example Program – Program to add two Numbers

```
import java.applet.*; import java.awt.*;
import java.awt.event.*; public class
Applet2 extends Applet { public void init()
{
 Label lbl1=new Label("First Number: ");
add(lbl1);
 TextField txt1=new TextField(10);
add(txt1);
 Label lbl2=new Label("Second Number: ");
add(lbl2);
 TextField txt2=new TextField(10); add(txt2);
 Label lbl3=new Label("Result: ");
add(lbl3);

 Button btn=new Button("Submit");
add(btn);

 btn.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent ae) {
 int first=Integer.parseInt
 (txt1.getText().toString());
 int second=Integer.parseInt
 (txt2.getText().toString());
 int res=first+second;
 lbl3.setText("Result: "+res);
 }
});
}
```

## Output



## Unit – 8 I/O and Streams [2 Hrs.]

### java.io Package

This package provides for system input and output through data streams, serialization and the file system. Unless otherwise noted, passing a null argument to a constructor or method in any class or interface in this package will cause a **NullPointerException** to be thrown. Some important classes of this package are:

- BufferedInputStream
- BufferedOutputStream
- BufferedReader
- BufferedWriter
- File
- FileDescriptor
- FileInputStream
- FileOutputStream
- FilePermission
- FileReader and FileWriter
- StringBufferInputStream



- StringReader
- StringWriter
- Writer

## Reading and Writing a file

### 1. Reading a File

```
import java.io.*;
public class Read {
 public static void main(String[] args) {
 try {
 FileReader reader=new
 FileReader("C:\\Users\\Raazu\\Desktop\\myfile.txt");
 BufferedReader buffer=new BufferedReader(reader);
 String line=buffer.readLine();
 if(line!=null) {
 System.out.println(line);
 }else {
 System.out.println("No Contents to Display");
 }
 reader.close();
 }catch(Exception e) {
 System.out.println(e);
 }
 }
}
```

### 2. Writing to a File

```
import java.io.*; public
class Write {
 public static void main(String[] args) {
 try {
 FileWriter writer=new
 FileWriter("C:\\Users\\Raazu\\Desktop\\myfile.txt");
 writer.write("This is a content written in file.");
 writer.write(" This is another content");
 System.out.println("Contents written successfully !");
 writer.close();
 }catch(Exception e) {
 System.out.println(e);
 }
 }
}
```

## Stream Classes

A stream is a method to sequentially access a file. I/O Stream means an input source or output destination representing different types of sources e.g. disk files. The java.io package provides classes that allow you to convert between Unicode character streams and byte streams of non-Unicode text.

- **Stream** – A sequence of data.
- **Input Stream:** reads data from source.
- **Output Stream:** writes data to destination.

## Character Stream

In Java, characters are stored using Unicode conventions. Character stream automatically allows us to read/write data character by character. For example, **FileReader** and **FileWriter** are character streams used to read from source and write to destination.

```
import java.io.*;
public class Character_Stream {
 //reading and writing character stream
 public static void main(String[] args) {
 try {
 //writing data to a file
 FileWriter writer=new
 FileWriter("C:\\Users\\Raazu\\Desktop\\bca.txt");
 BufferedWriter wbuffer=new BufferedWriter(writer);
 wbuffer.write("This is a content.");
 wbuffer.write("This is another content.");
 System.out.println("Contents are written successfully !");
 wbuffer.close();
 writer.close();
 //read
 operation
 FileReader reader=new
 FileReader("C:\\Users\\Raazu\\Desktop\\bca.txt");
 BufferedReader rbuffer=new BufferedReader(reader);
 String data=rbuffer.readLine();
 System.out.println(data);
 rbuffer.close();
 reader.close();

 }catch(Exception e) {
 System.out.println(e);
 }
 }
}
```

## **Byte Stream**

Byte streams process data byte by byte (8 bits). For example FileInputStream is used to read from source and FileOutputStream to write to the destination.

```
import java.io.*; public
class Byte_Stream {
 public static void main(String[] args) {
 try {
 //write
 FileOutputStream output=new
 FileOutputStream("C:\\Users\\Raazu\\Desktop\\bca.txt");
 int content=1011011;
 output.write((byte) content);
 System.out.println
 ("Byte Stream written successfully !");
 output.close();

 //read
 FileInputStream input=new
 FileInputStream("C:\\Users\\Raazu\\Desktop\\bca.txt");
 int data=input.read();
 System.out.println((char) data);
 input.close();

 }catch(Exception e) {
 System.out.println(e);
 }
 }
}
```

### **When to use Character Stream over Byte Stream?**

- In Java, characters are stored using Unicode conventions. Character stream is useful when we want to process text files. These text files can be processed character by character. A character size is typically 16 bits.

### **When to use Byte Stream over Character Stream?**

- Byte oriented reads byte by byte. A byte stream is suitable for processing raw data like binary files.

## **Serialization Interface**

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that objects of these classes may get the certain capability. The Cloneable and Remote are also marker interfaces.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implement the **java.io.Serializable** interface by default.

Let's see the example given below:

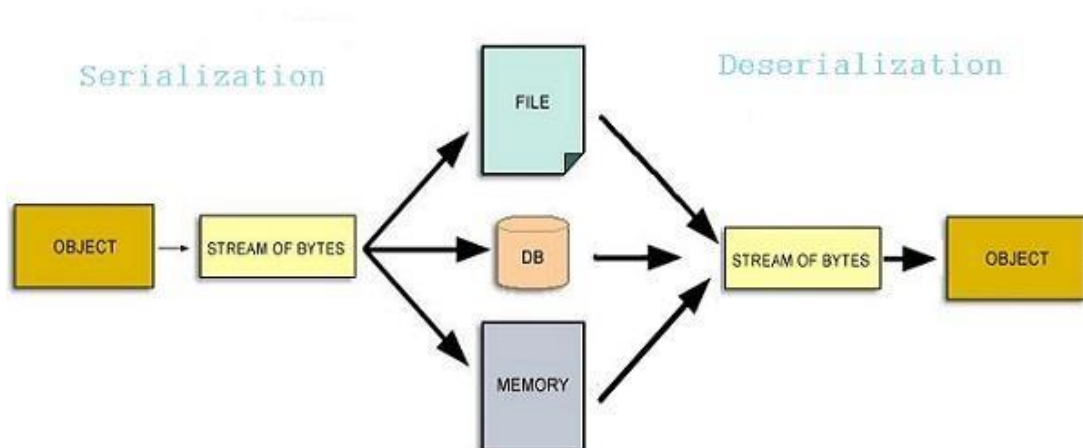
```
import java.io.Serializable; public class
Student implements Serializable{ int id;
 String name; public Student(int id,
 String name) { this.id = id;
 this.name = name;
 }
}
```

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

## Serialization and Deserialization

**Serialization** is a mechanism of converting the state of an object into a byte stream.

**Deserialization** is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

## Advantages of Serialization

1. To save/persist state of an object.
2. To travel an object across a network. Serialization Example

```
import java.io.*;
class Studentinfo implements Serializable
{
 String name;
 int rid;
 static String contact;
 Studentinfo(String n, int r, String c)
```

```

 {
 this.name =
n; this.rid = r;
this.contact = c;
 }
 }
 public class MyTest {
 public static void main(String[] args) {
 try
 {
 Studentinfo si = new Studentinfo("Abhi", 104, "110044");
 FileOutputStream fos = new FileOutputStream("student.ser");
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject(si); oos.close(); fos.close();
 }
 catch (Exception e)
 {
 e.printStackTrace();
 }
 }
 }
}

```

Object of Studentinfo class is serialized using **writeObject()** method and written to **student.ser** file.

### Deserialization Example

```

import java.io * ;
class DeserializationTest
{
 public static void main(String[] args)
 {
 studentinfo si=null ;
 try
 {
 FileInputStream fis = new FileInputStream("student.ser");
 ObjectOutputStream ois = new ObjectOutputStream(fis); si
 = (studentinfo)ois.readObject();
 }

 catch (Exception e)
 {
 e.printStackTrace();
 }
 System.out.println(si.name);
 System.out.println(si.rid);
 System.out.println(si.contact);
 }
}

```

### Output:

Abhi  
104  
Null

Contact field is null because, it was marked as static and as we have discussed earlier static fields does not get serialized.

**NOTE:** Static members are never serialized because they are connected to class not object of class.

-----