

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.

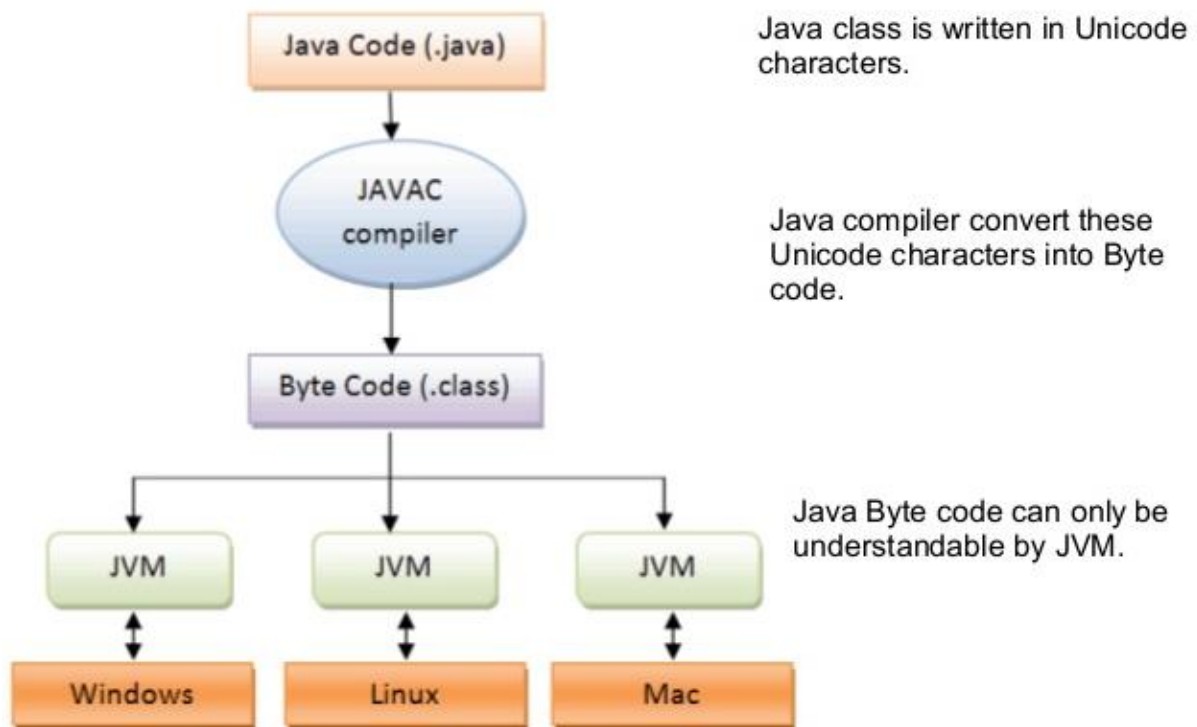
Types of Java Applications and Applets

- 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 window-based 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

\wedge (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	$(A \wedge B)$ will give 49 which is 0011 0001
\sim (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	$(\sim A)$ will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
\ll (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	$A \ll 2$ will give 240 which is 1111 0000
\gg (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	$A \gg 2$ will give 15 which is 1111
\ggg (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.	$A \ggg 2$ 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 $C = C - A$

<code>*=</code>	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	<code>C *= A</code> is equivalent to <code>C = C * A</code>
<code>/=</code>	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	<code>C /= A</code> is equivalent to <code>C = C / A</code>
<code>%=</code>	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	<code>C %= A</code> is equivalent to <code>C = C % A</code>
<code><<=</code>	Left shift AND assignment operator.	<code>C <<= 2</code> is same as <code>C = C << 2</code>
<code>>>=</code>	Right shift AND assignment operator.	<code>C >>= 2</code> is same as <code>C = C >> 2</code>
<code>&=</code>	Bitwise AND assignment operator.	<code>C &= 2</code> is same as <code>C = C & 2</code>
<code>^=</code>	bitwise exclusive OR and assignment operator.	<code>C ^= 2</code> is same as <code>C = C ^ 2</code>
<code> =</code>	bitwise inclusive OR and assignment operator.	<code>C = 2</code> is same as <code>C = C 2</code>

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-else-if 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 );  
    }  
}
```


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:

```
Access Modifier   Class Name
  ↓               ↓
public class Dog {
    String breed;
    int age;
    String color;
    void barking() {
    }
    void hungry() {
    }
    void sleeping() {
    }
}
```

Annotations in the diagram:

- An arrow points from "Access Modifier" to `public`.
- An arrow points from "Class Name" to `Dog`.
- An arrow points from "Data Members" to the data fields: `String breed;`, `int age;`, and `String color;`.
- An arrow points from "Member Functions / Methods" to the method definitions: `void barking() { ... }`, `void hungry() { ... }`, and `void sleeping() { ... }`.

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:

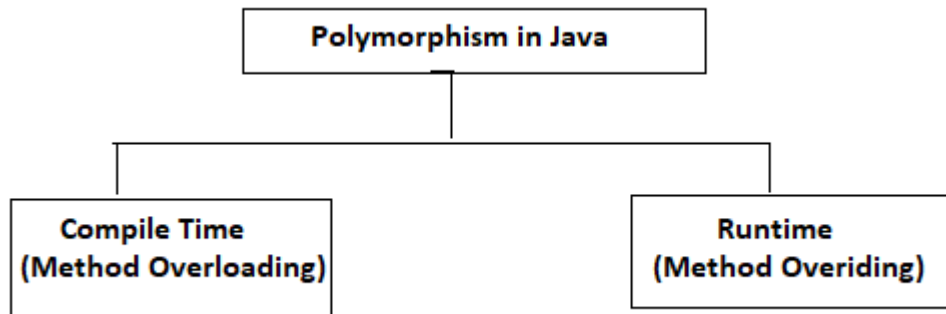
Name: Raaju Poudel
Age: 27

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 well-defined 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 *parent-child* 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
}
```

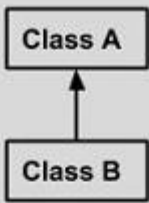
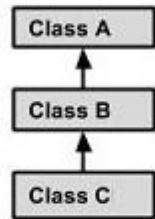
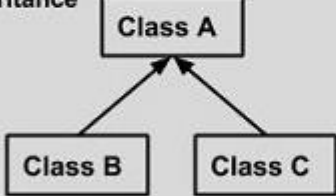
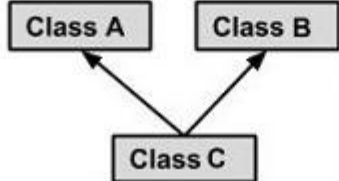
The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

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

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

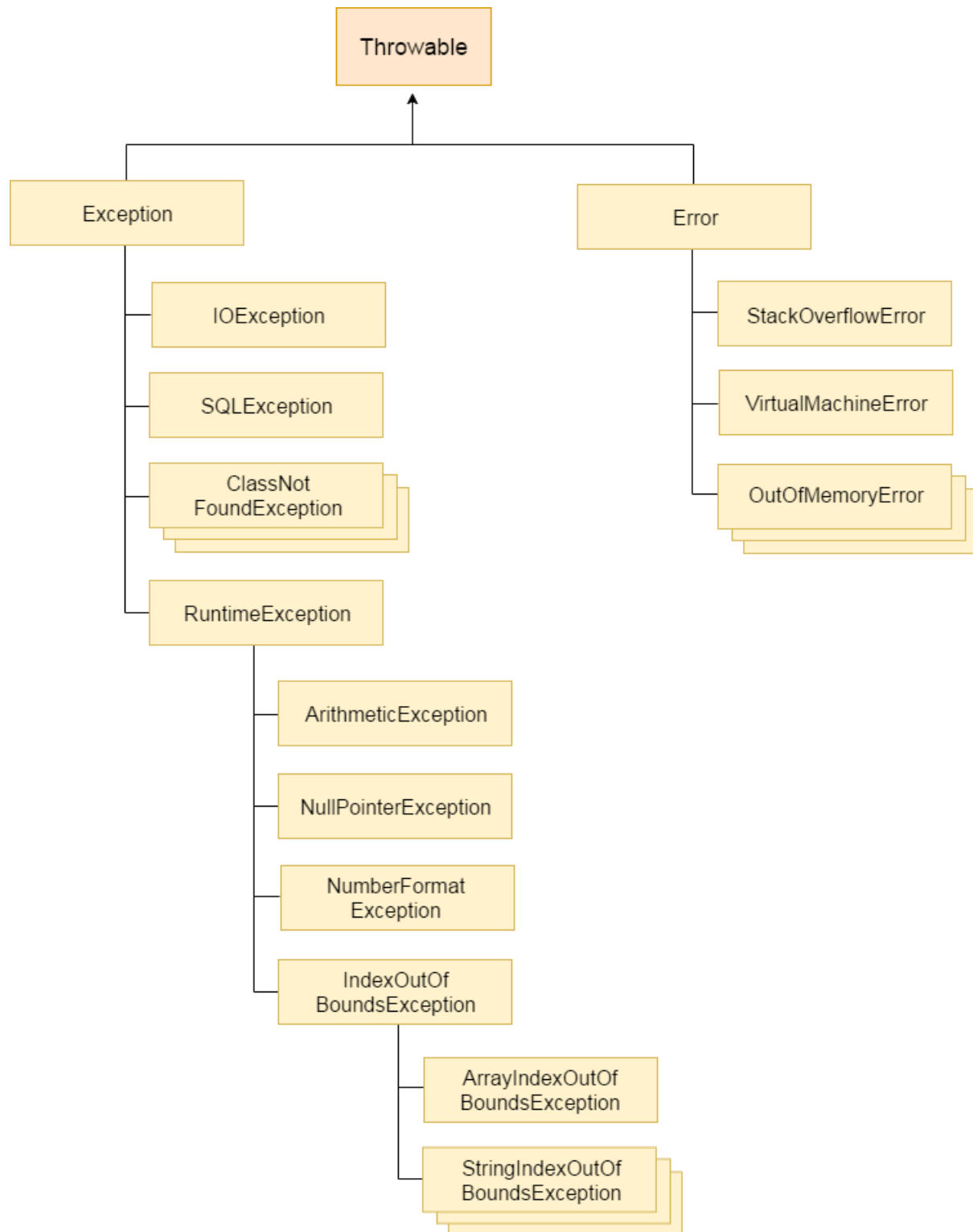
Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

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.