LECTURE NOTES
ON

Object Oriented Programming using Java

2305CS102

M.C.A. 1st Semester

(DICA)

Prepared By

Vaseem Ghada

Assistant Professor

vaseem.ghada@darshan.ac.in

9879531513

Darshan UNIVERSITY
योगः कर्मसु कौशलम्

Table of Content

Unit-1
Introduction to Java

1.1. Evolution of Java

- Java was created by a team of programmers at sun Microsystems of U.S.A in 1991.
- This language was initially called "Oak" by James Gosling but renamed "Java" in 1995.
- Java was designed for the development of software for consumer electronics devices like TVs, VCRs, and such electronics machines.
- When the World Wide Web became popular in 1994, sun realized that Java was the perfect programming language for the Web.
- Late 1995 and early 1996 they released Java & it was instant success.

What is Java?

- Java is a programming language that:
- Is exclusively object oriented
- Has full GUI support
- Has full network support
- Is platform independent
- Executes stand-alone or "on-demand" in web browser as applets

1.2. Features of Java

- The Features of Java programming are as below

  - Simple
  - Secure
  - Portable
  - Object-oriented
  - Robust
  - Multithreaded
  - Architecture-natural
  - Interpreted
  - High performance
  - Distributed
  - Dynamic

- Simple

  - Java was designed to be easy for the professional programmer to learn and use effectively.
  - Assuming that you have some programming experience, you will not find java hard to master.
  - If you already understand the basic concepts of object-oriented programming, learning java will be even easier.

- If you are an experienced C++ programmer, moving to java will require very little effort. Because java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

- **Security**
  - Security is the benefit of java. Java system not only verifies all memory access but also ensure that no viruses are communicated with an applet.
- **Portable**
  - The most significant contribution of java over other language is its portability.
  - Java programs can be easily moved from one computer system to another.
  - Java ensures portability in two ways:

    1. Java compiler generates byte code in that can be implemented on any machine.

    2. The size of the primitive data types is machine-independent.
- **Object-Oriented**
  - Java is a true object-oriented language. All program code and data reside within object and classes.
  - The object model in java is simple and easy to extend.
- **Robust (healthy, strong)**
  - The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of java.
  - To gain reliability, java has strict compile time and run time checking for codes.
  - To better understand how java is robust, consider two main reasons for program failure: memory management mistakes and mishandled exceptional conditions.

  - 1. Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because java provides garbage collection for unused objects.)
  - 2. Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found" and they must be managed with awkward and hard to read constructs. Java helps in this area by providing object oriented exception handling. In a well-written java program, all run-time errors can and should be managed by your program.

- **Multithreaded**
  - Java was designed to meet the real-world requirement of creating interactive, networked programs.
  - To accomplish this, java supports multithreaded programming, which allows you to write programs that do many things simultaneously.
  - The java run-time system comes with an elegant yet sophisticated solution for multi process synchronization that enables you to construct smoothly running interactive systems.
  - Java's easy to use approach to multithreading allows you to think about the specific behaviour of your program, not the multitasking subsystem.

- **Architecture-Neutral**
  - A central issue of java programmers was that code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow- even on the same machine.
  - Operating system upgrades, and changes in core system resources can combine to make a program malfunction.
  - The java designer made several hard decisions in the java language and the java virtual machine in an attempt to alter this situation. Their goal was "write once; run anywhere, anytime, forever."

- **Interpreted**
  - Usually, a computer language is either compiled or interpreted. Java combines these approaches thus making java a two-stage system.
  - Java compiler translates source code into byte code instructions. Byte codes are not machine instructions and so java interpreter generates machine code that can be directly executed by the machine that is running the java program.
  - We can thus say that java is both a compiled and an interpreted language.

- **High Performance**
  - Java performance is impressive for an interpreted language, mainly due to the use of intermediate byte code.
- **Distributed**
  - Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols.
  - Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.
  - Dynamic
  - Java is capable of dynamically linking in new class libraries, methods and object.
  - Java can also determine the type of class through a query, making it possible to either dynamically link or abort the program.

1.3. JDK, JRE and JVM

1.3.1. JDK

- JDK (Java Development Kit) is a Kit that provides the environment to develop and execute(run) the Java program. JDK is a kit (or package) that includes two things.
  - Development Tools (to provide an environment to develop your java programs)
  - JRE (to execute your java program).
- JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:
  - Standard Edition Java Platform (J2SE) for client side stand alone application.
  - Enterprise Edition Java Platform(J2EE) for server side stand alone application.
  - Micro Edition Java Platform (J2ME) for mobile based application.

1.3.2. JRE

- JRE (Java Runtime Environment) is an installation package that provides an environment to only run (not develop) the java program (or application) onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system. It is used to provide the runtime environment. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

1.3.3. JVM

- JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.
- Bytecode is a highly optimized set of instructions designed to be executed by the Java Virtual Machine (JVM).
- Byte code is intermediate representation of java source code.
- Java compiler provides byte code by compiling Java Source Code.
- Extension for java class file or byte code is '.class', which is platform independent.
- JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent.

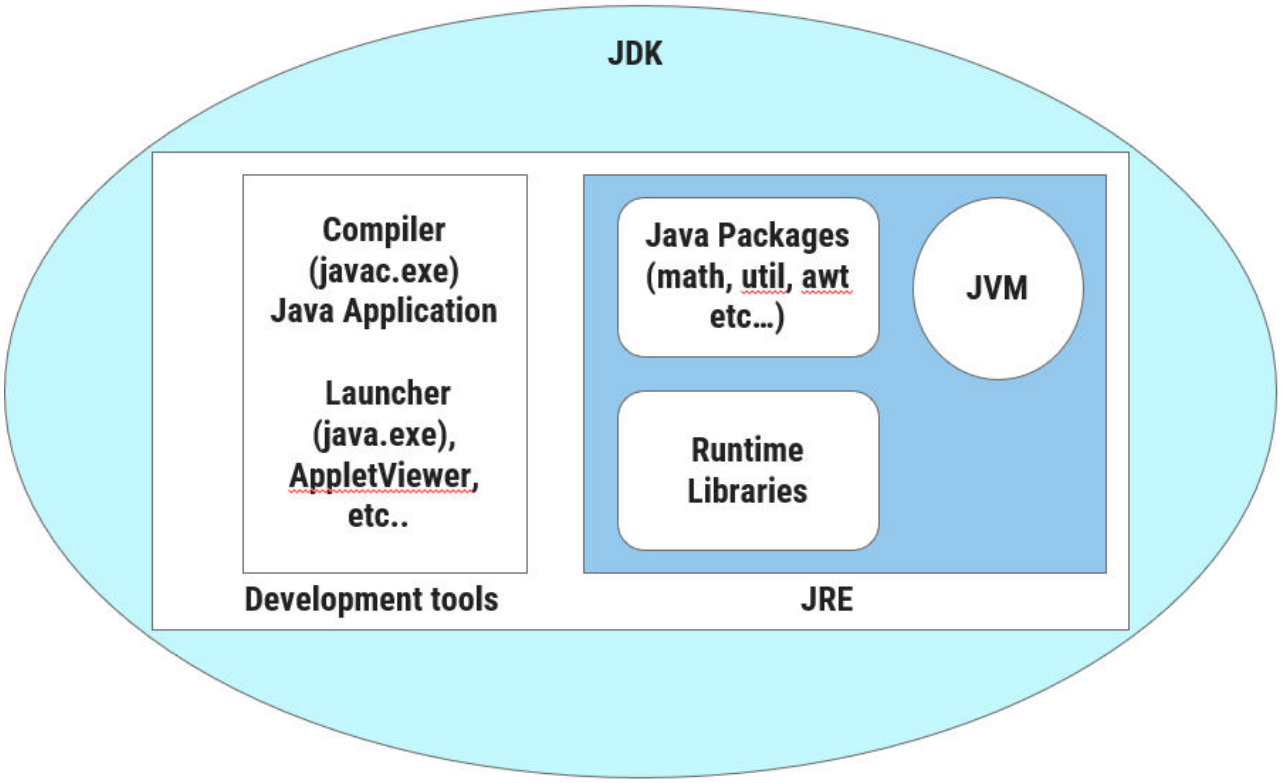

Figure 1.3. JDK, JRE and JVM

1.4. Program Structure, Compilation and Run Process.

- **Simple Java Program Structure**

```
public class Example
{
      public static void main(String args[])
      {
            System.out.println("First Example");
      }
}
```

- **class Example**
  Here name of the class is Example.
- **public static void main(String args[])**
  - public: The public keyword is an access specifier, which means that the content of the following block accessible from all other classes.
  - static: The keyword static allows main() to be called without having to instantiate a particular instance of a class.
  - void: The keyword void tells the compiler that main() does not return a value. The methods can return value.

- main(): main is a method called when a java application begins,
- String args []
  Declares a parameter named args, which is an array of instance of the class string.
  Args[] receives any command-line argument present when the program is executed.

- **System.out.println()**
  - System is predefined class that provides access to the system.
  - Out is the output stream that is connected to the console.
  - Output is accomplished by the built-in println() method. Println() displays the string which is passed to it.

- **Compilation of Java Program**

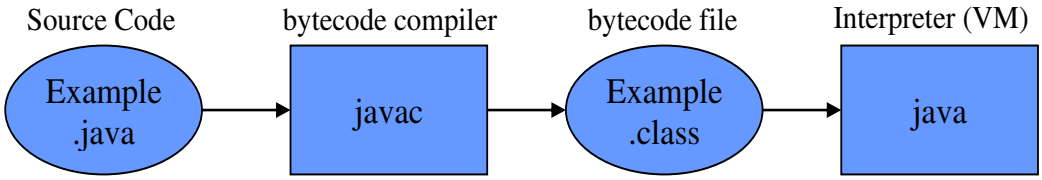

Figure 1.4. Java Program Compilation Process

- **Command 1:** Javac Example.java
  This command will compile the source file and if the compilation is successful, it will generate a file named example.class containing bytecode. Java compilers translate java program to bytecode form.
- **Command 2:** Java Example
  The command called 'java' takes the bytecode and runs the bytecode on JVM
- **Output**
  First Example

Unit-2
Basics of Java programming

2.1. Primitive data types with Variables

- Every variable has a type, every expression has a type, and every type is strictly defined.
- All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility.
- There are no automatic coercions or conversions of conflicting types as in some languages.
- The java compiler checks all expressions and parameters to ensure that the types are compatible.
- There are two types of data types
  - Primitive types
  - Non primitive types
- Primitive Types
  Java provides eight primitive types of data:
  - Byte
  - Short
  - Int
  - Long
  - Char
  - Float
  - Double
  - Boolean
- The primitive types are also commonly referred to as simple types.
- These can be put in four groups:
  - Integer
  - Floating-point numbers
  - Characters
  - Boolean
- Integer
  - Java provides four integer types: byte, short, int, long.
  - All of these are signed, positive and negative values.

Type	Size/bytes	Range
Byte	8	-128 to 127
Short	16	-32,768 to 32,767
Int	32	-2,147,483,648 to 2,147,483,647
Long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

- Floating-Point Types
  - Floating-Point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision.

- For example, calculations such as square root, or transcendental such as sine and cosine, result in a value whose precision requires a floating-point type.
- There are two kinds of floating-point types, float and double, which represent single and double-precision numbers, respectively.

Type	Size/bytes	Range
Float	32	1.4e – 045 to 3.4e + 038
Double	64	4.9e - 324 to 1.8e + 308

- Character
  - The data type used to store characters is char.

Type	Size/bytes	Range
Char	16	0 to 65,536

- Boolean
  - Java has primitive type, called boolean, for logical values.
  - It can have only one of two possible values, true or false.

Type	Size/bytes	Range
Boolean	1	True/False, Yes/No , 0/1

Variable: -

- The variable is the basic unit of storage in a java program.
- A variable is defined by the combination of identifiers, a type and an optional initialize.
- All variables have a scope, which defines their visibility and a life time.
- Naming Rules for a variable: -
  - It should start with a lowercase letter such as id, name.
  - It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
  - If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
  - Avoid using one-character variables such as x, y, z.
- Declaring a Variable: -
  - All variables must be declared before they can be used. The basic form of a variable declaration is shown here.
  - Type identifier [= value] [, identifier [=value]...];
  - Int a,b,c; // declare 3 integers
  - Byte z = 22; // initialize z
  - Char x = 'X'; // the variable x has the value 'X'

- Dynamic Initialization:
- Java allows variables to be initialized dynamically using any valid expression at the time the variable is declared.

- Example:

```
class DynamicInt
{
 public static void main(String args[])
 {
       double a= 3.0, b= 5.0;
       // c is dynamically initialized
       double c = Math.sqrt (a * a+b * b);
       System.out.println("The value of C is: - " + c);
 }
}
```

- In above example method sqrt (), is the member of the Math class.

2.2. Type Casting

- It is common to assign a value of one type to a variable of another type.
- If the two types are compatible then java will perform the conversion automatically. For example, it is possible to assign an integer value to a long variable.
- However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For example, there is no conversion defined from double to byte
- It is still possible to obtain a conversion between incompatible types.
- For that you must use a cast, which performs an explicit conversion between incompatible types.
- **Java's Automatic Type Conversion**
  - 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.
  - The int type is always large 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 not compatible with each other.
  - However, numeric types are not compatible with char or Boolean. Char and Boolean data types are not compatible with each other.
  - Java performs an automatic type conversion when storing a literal integer constant into variables of type byte, short or long.

- **Casting Incompatible Types**
  - Automatic type conversion will not fulfil all needs. For example, 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 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 conversion between two incompatible types, you must use a cast.
- A cast is 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 byte, it will be reduced modules (the reminder of an integer division by the) byte's range.

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

- A different type of conversion will occur when a floating-point value is assigned to an integer type: truncation.
- Integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.
- For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated.
- If the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.
- Example:

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

2.3. Operator

- An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.
- Operators are used in programs to manipulate data and variables.
- Java operators can be classified into a number of related categories as below:

- Arithmetic operator
- Relational operator
- Logical operator
- Assignment operators
- Increment and decrement operator
- Conditional operators
- Bitwise operators
- Special operators

2.3.1. Arithmetic Operator

- Java provides all the basic arithmetic operators.

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

- Consider Following example with the variable value A=10, B=20.

Table 2.3.1 Arithmetic Operator

Operator	Description	Example
+	Addition	A + B = 30
-	Subtraction	A - B = -10
*	Multiplication	A * B = 200
/	Division	B / A = 2
%	Modulus	B % A = 0
++	Increment	B++ = 21
--	Decrement	B-- = 19

2.3.2. Relational Operator

- For comparing two quantities, and depending on their relation, we take certain decision.
- For example, we may compare the age of two persons, or the price of two items, and so on. This comparison can be done with the help of relational operators.

Operator	Meaning

<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

- Consider Following example with the variable value A=10, B=20.

Table 2.3.2 Relational Operator

Operator	Description	Example
==	Equals	(A == B) is not true.
!=	Not Equals	(A != B) is true.
>	Greater than	(A > B) is not true.
<	Less than	(A < B) is true.
>=	Greater than equals	(A >= B) is not true.
<=	Less than equals	(A <= B) is true.

2.3.3. Logical Operator

- Java has three logical operators:

Operator	Meaning
&&	logical AND
||	logical OR
!	Logical NOT

- Consider Following example with the variable value A=10, B=20.

Table 2.3.2 Logical Operator

Operator	Description	Example
&&	Logical AND operator	(A && B) is false.
||	Called Logical OR Operator	(A || B) is true.
!	Called Logical NOT Operator	!(A && B) is true.

2.3.4. Assignment Operator

- Assignment operators are used to assign the value of an expression to a variable.
- Java has a set of 'shorthand' assignment operators which are used in the form
  var-name op= exp; which is equivalent to

var-name=var-name op (exp);

Table 2.3.4 Assignment Operator

Operator	Description	Example
=	Simple assignment operator	C = A + B will assign value of A + B into C
+=	Add AND assignment operator	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

2.3.5. Increment and Decrement Operator

- java has two increment and decrement operators:

++ and - -

The operator ++ adds 1 to the operand while - - subtracts 1. both are used in the following format:

++m; or m++;

--m or m--;

- Where: ++m is equivalent to m=m+1;
  m is equivalent to m=m-1;

2.3.6. Conditional Operator

- The character pair ? : is a ternary operator available in java. This operator is used to construct conditional expressions of the form Exp1 ? Exp2: Exp3
- Consider the following example:

```
a=10;
b=15;
```

```
x=(a>b) ? a: b;
It is same as:
if (a>b)
x=a;
else
x=b;
```

2.3.7. Bitwise Operators

- Java has a distinction of supporting special operators known as bitwise operators for manipulation of data at values of bit level.
- These operators are used for testing the bits, or shifting them to the right or left.
- Bitwise operators may not be applied to float or double.

Operator	Meaning
&	bitwise AND
!	Bitwise OR
^	Bitwise exclusive OR
~	one's complement
<<	shift left
>>	shift right

Example :

```
AND
0101 AND 0011 = 0001
OR
0101 OR 0011 = 0111
XOR
0101 XOR 0011 = 0110
NOT
NOT 0111 = 1000
0111 LEFT-SHIFT = 1110
0111 RIGHT-SHIFT = 0011
```

2.3.8. Special Operator

- Java supports some special operators of interest such as instanceof operator and member selection operator(.)
- **Instanceof Operator**
  - the instanceof is an object reference operator and returns true if the object on the left-hand side is an instance of the class given on the right-hand side. This operator allows us to determine whether the object belongs to a particular class or not.
  - For example:
    person instanceof student
  - is true if the object person belongs to the class student; otherwise, it is false.
- **Dot operator**
  - The dot operator(.) is used to access the instance variables and methods of class objects.
  - For example:
  - person1.age //reference to the variable age
  - person1.salary() //reference to the method salary ()
  - It is used to access classes and sub-packages form a package.

2.4. Operator Precedence

- As we explore the operators of the Java programming language, it may be helpful for you to know ahead of time which operators have the highest precedence. The operators in the following table are listed according to precedence order. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with relatively lower precedence. Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left.
- In general-purpose programming, certain operators tend to appear more frequently than others; for example, the assignment operator "=" is far more common than the unsigned right shift operator ">>>". With that in mind, the following discussion focuses first on the operators that you're most likely to use on a regular basis, and ends focusing on those that are less common. Each discussion is accompanied by sample code that you can compile and run. Studying its output will help reinforce what you've just learned.

Table 2.4 Operator Precedence

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

2.5. Array and its type

- An array is a group of like-typed variables that are referred by a common name.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information.

2.5.1. One-Dimensional Arrays

- A one-dimensional array is a list of like typed variables.
- To create an array, you first must create an array variable of the desired type.
- The general form of a one-dimensional array declaration is:
  type var-name[ ];
  int month_days[ ];
- The value of month_days is set to null, which represents an array with no value. To link month_days with an actual, physical array of integers, you must first allocate one using new and assign it to month_days.
- new is special character which allocates memory.
- The general form of new as it applies to one-dimensional arrays appears as follows:
  array-var=newtype[size];
- Following example allocates a 12-element array of integers and line them to month_days.
  month_days=new int[12];
- after this statement executes, month_days will refer to an array of 12 integers.
- In short, obtaining an array is a two-step process.
  1. you must declare a variable of the desired array type.
  2. you must allocate the memory that will hold the array, using new, and assign it to the array variable.
- Example:

```
class Array
{
 public static void main(String args[])
 {
       int month_days[ ];
       month_days= new int[12];
       month_days[0]=31;
       month_days[1]=28;
       month_days[2]=31
       month_days[3]=30
       month_days[4]=31
       month_days[5]=30
       month_days[6]=31
       month_days[7]=31;
       month_days[8]=30;
       month_days[9]=31;
       month_days[10]=30;
```

```
        month_days[11]=31;
        System.out.println ("April has" + month_days[3] + "days.");
    }
}
```

- it is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:
  `int month_days[ ]=new int[12];`
- Arrays can be initialized when they are declared.
- An array initializer is a list of comma-separated expressions surrounded by curly braces. The comma separates the values of the array elements.
- The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use new.
- For example,

```
Class AutoArray
{
 public static void main (String args[ ])
 {
        int month_days[]={31,28,31,30,31,30,31,31,
        30,31,30,31};
        System.out.println("April has" + month_days[3]   + "days.");
 }  }
```

2.5.2. Multidimensional Arrays

- Multidimensional arrays are actually arrays of arrays.
- To declare a multidimensional array variable, specify each additional index using another set of square brackets.
- For example, following declares two dimensional array:
  `int twoD[ ] [ ] = new int [4] [5];`
- This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int.
- Alternative Array Declaration, There is a second form that may be used to declare an array:
  `type[ ] var-name;`
- The square brackets follow the type specifier, and not the name of the array variable.
- For example, the following two declarations are equivalent:

```
int a1[ ] = new int[4];
int [ ] a1= new int[4];
char twod [ ] [ ] = new char [3] [4];
char [ ] [ ] twod = new char [3] [4];
```

- This alternative declaration form offers convenience when declaring several arrays at the same time. For example,
  `int [ ] nums1, nums2, nums3;`
- This creates 3 array variables of int type.

Unit-3
Object Oriented Programming

3.1. Object Oriented Programming Concepts.

- Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:
- Object
- Class
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

3.1.1. Object

- An object is a region of storage that defines both state & behaviour.
  - State is represented by a set of variables & the values they contain.
  - Behaviour is represented by a set of methods & the logic they implement.
- Thus, an object is a combination of a data & the code that acts upon it.
- Objects are instance of a class.
- Objects are the basic runtime entities in an object-oriented system.
- Object take up space in memory and have an associated address like a record in Pascal or structure in C.
- The arrangement of bits or the data in an object's memory space determines that object's state at given moment.
- Objects are runtime instance of some class.
  For Example:

```
Person p1,p2;
p1 = new person();
p2 = new person();
```

3.1.2. Class

- A class is a template from which objects are created. That is objects are instance of a class.
- When you create a class, you are creating a new data-type. You can use this type to declare objects of that type.
- Class defines structure and behaviour (data & code) that will be shared by a set of objects

3.1.3. Abstraction

- Hiding internal details and showing functionality is known as abstraction. For example, phone call, we don't know the internal processing.
- In Java, we use abstract class and interface to achieve abstraction.

3.1.4. Encapsulation

- Encapsulation is the mechanism that binds code and data together and keeps both safe from outside interference and misuse.
- It works as a protective wrapper that prevents the code and data form being accessed by other code defined outside the wrapper.
- Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.
- In Java, the basis of encapsulation is the class.
- A class defines the structure and behaviour that will be shared by a set of objects. Each object of a given class contains the structure and behaviour defined by the class.
- For this reason, objects are sometimes referred to as instances of a class.
- When we create a class, we will specify the code and data that constitute that class. Collectively, these elements are called members of the class.
- The data defined by the class are referred to as member variables or instance variables.
- The code that operates on that data is referred to as member methods or just method.
- Each method or variable in a class may be marked private or public.
- The public interface of a class represents everything that external users of the class need to know, or may know.
- The private methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable.

3.1.5. Inheritance

- Inheritance is the process by which object of one class acquires the properties of another class.
- It supports the concept of hierarchical classification. For example, the bird robin is a part of the class flying Bird, which is again a part of the class Bird. Each derived class shares common characteristics with the class from it is derived.
- The concept of Inheritance provides the idea of reusability.
- This means that we can add additional features to an existing class without modifying it. This is possible by deriving new class form the existing one.
- The new class have the combined feature of both the classes.
- Each subclass defines only those features that are unique to it.
- Derived class is known as sub class and main class is known as super class.

3.1.6. Polymorphism

- Polymorphism means the ability to take more than one form.
- A single function name can be used to handle different no and different types of argument.
- It plays an important role in allowing objects having different internal structures to share the same external interface.

3.2. Introduction to Class Object and Methods

3.2.1. The General Form of class

- A class is a template from which objects are created. That is objects are instance of a class.
- When you create a class, you are creating a new data-type. You can use this type to declare objects of that type.
- Class defines structure and behaviour (data & code) that will be shared by a set of objects
- A class is declared by use of the class keyword. Classes may contain data and code both.
- The general form of al class definition:

```
class ClassName
 {
       type instance variable1;
       type instance variable2;

       type methodname1 (parameter list)
       {
            body of method;
       }
       type methodname2 (parameter list)
       {
            body of method;
       }
 }
```

- The data or variable are called instance variable. The code is contained within methods.
- The method and variables defined within a class are called member of the class.
- Example:

```
class Box
{
 double width;
 double height;
 double depth;
}
class BoxDemo
{
 public static void main (String args[])
 {
       Box mybox = new Box ();
       double vol;
       mybox.width =10;
       mybox.height = 20;
       mybox.depth = 30;
       vol = mybox.width * mybox.height * mybox.depth;
       System.out.println ("Volume is: - "+vol);
 }
```

```
}
```

- Each object contains its own copy of each variable defined by the class.
- So, every Box object contains its own copy of the instance variables width, height and depth.
- To access these variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable.

3.2.2. Declaring Object

- `Box mybox; // declare ref. to object which contains null value.`
- `mybox = new Box (); // allocate a Box object.`
- General form of a new object

```
class var = new classname ();
mybox = new Box ();
  Where:
class var = variable of the class type.
classname = name of the class.
```

- The classname followed by parentheses specifies the constructor for the class.
- A constructor defines what occurs when an object of a class is created.
- Most classes explicitly define their own constructors within their class definition but if no explicit constructor is specified then java will automatically supply a default constructor.
- This is the case with Box. This is default constructor.
- Assigning Object Reference Variable

```
Box b1 = new Box ();
Box b2 = new b1;
```

- After this executes, b1 and b2 will both refer to the same object.
- The assignment of b1 to b2 did not allocate any memory or copy any part of the original object.
- It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

3.2.3. Introducing Methods

-

```
type name (parameter-list)
{
body of method
}
```

- Where:
  - Type : Specifies the type of data returned by the method. If the method does not return a value its return type must be void.
  - Name: Specifies the name of the method.
  - Parameter-list: It is a sequence of type & identifiers pairs separated by commas.

- Parameters are variables that receive the value of the argument passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.

- Methods that have a return type other than void return a value to the calling routine using the following form of the return statement; return value;
- **Types of Methods**
  - Does not return value – void
  - Returning a value
  - Method which takes parameter

- Example: Method that Does not return value – void

```
class Box
{
 double width, height, depth;
 void volume()
 {
       System.out.println("Volume is: -"+width*height*depth);
 }
}
class BoxDemo
{
 public static void main (String args[])
 {
       Box mybox1 = new Box ();
       Box mybox2 = new Box ();

       mybox1.width =10;
       mybox1.height =20;
       mybox1.depth =15;
       mybox2.width =10;
       mybox2.height =15;
       mybox2.depth =25;
       mybox1.volume ();
       mybox2.volume ();
 }
}
```

- Example: Method that Returning a value

```
class Box
{
 double width, height, depth;
 void volume()
 {
       return width*height*depth;
 }
}
class BoxDemo
{
 public static void main (String args[])
 {
```

```
        Box mybox1 = new Box ();
        Box mybox2 = new Box ();
        double vol;

        mybox1.width =10;
        mybox1.height =15;
        mybox1.depth =20;
        mybox2.width =2;
        mybox2.height =3;
        mybox2.depth =5;
        vol = mybox1.volume ();
        System.out.println ("Volume is: -"+vol);

        vol = mybox2.volume ();
        System.out.println ("Volume is: -"+vol);
  }
}
```

- Example: Method which takes parameter

```
int square ()
{
 return 10 * 10;
}
```

- It will return the square of 10 but this method is specified to only 10. If you modify the method so that it takes a parameter.

```
int square (int i)
 {
      return i * i;
 }
 int x;
 x = square (5);
```

- A parameter is a variable defined by a method that receives a value when the method is called. For example, in square (), i is a parameter.
- An argument is value that is passed to a method when it is invoked. For example in square (100), passes 100 as an argument.

```
class Box
{
 double width, height, depth;
 double volume()
 {
      return width*height*depth;
 }
 void setDim (double w, double h, double d)
 {
      width = w;
      height = h;
      depth = d;
```

```
    }
  }
  class BoxDemo
  {
   public static void main (String args[])
   {
         double vol;
         Box mybox1 = new Box ();
         Box mybox2 = new Box ();

         mybox1.setDim (10,15,25);
         mybox2.setDim (3,5,7);
         vol = mybox1.volume ();
         System.out.println ("Volume is: -"+vol);

         vol = mybox2.volume ();
         System.out.println ("Volume is: -"+vol);
   }
  }
```

3.3. Nested Class, Anonymous Inner Class

3.3.1. Nested & Inner class

- 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 is known to A, but not outside of A.
- A nested class (B) has access to the members, including private members, of class in which it is nested (A).
- However, the enclosing class does not have access to the members of the nested class.
- There are two types of nested classes:
  - Static
  - non-static (inner-class)
- Static nested class
  - A static nested class is one which has the static modifier applied because it is static, it must access the member of its enclosing class through an object. i.e. it can not refer to members of its enclosing class directly.
  - Because of this reason, static nested classes are rarely used.
- Non-Static nested class(inner-class)
  - The most imp 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 member of the outer class do.
- Thus, an inner class is fully within the scope of its enclosing class.

- The following program shows how to define and use an inner-class. The class named outer has one instance variable and method and defines one inner class called Inner.

```
class Outer
{
 int outer_x =100;
 void test()
 {
      Inner inner =  new Inner();
      inner.display();
 }
 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();
 }
}
```

- 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. method named display() is defined inside Inner. This method display outer_x on the 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.
- Inner class is known only within the scope of outer class.
- The java compiler generates an error message. If any code outside of class outer attempts to instantiate class Inner.
- An inner class has access to all of the members of its enclosing class, but the reverse is not true.
- Members of the inner class are known only within the scope of the inner class and may not be used by the outer class.

```
class Outer1
{
 int outer_x = 100;
 void test1()
 {
      Inner1 inner = new Inner1();
      inner.display();
 }
 class Inner1
```

```
    {
          //int y= 10;  // local to Inner
          void display()
          {
                System.out.println("Display outer" +outer_x);
          }
    }
    void showy ()
    {
          //System.out.println (y);  // error y is not known here.
    }
}
class InnerClassDemo1
{
    public static void main(String args[])
    {
          Outer1 outer = new Outer1();
          outer.test1();
    }
}
```

3.3.2. Anonymous Inner class

- Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.
- In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:
  - Class (may be abstract or concrete).
  - Interface
- Java anonymous inner class example using class

```
abstract class Person{
   abstract void eat();
}
class TestAnonymousInner{
    public static void main(String args[]){
     Person p=new Person(){
     void eat(){System.out.println("nice fruits");}
     };
     p.eat();
    }
}
```

- Java anonymous inner class example using interface

```
interface Eatable{
    void eat();
}
class TestAnnonymousInner1{
    public static void main(String args[]){
```

```
    Eatable e=new Eatable(){
     public void eat(){System.out.println("nice fruits");}
    };
    e.eat();
    }
   }
```

3.4. String, StringBuffer and Math class

3.4.1. String

- In java, a string is a sequence of characters. But, unlike many other languages that implement strings as character arrays, java implements strings as objects of type String.
- When we create a String object, we are creating a string that cannot be changed. That is, once a String object has been created, we cannot change the characters that comprise that string. We can perform all types of operations.
- For those cases in which a modifiable string is desired, java provides two options: StringBuffer and StringBuilder. Both hold strings that can be modified after they are created.
- The String, StringBuffer and StringBuilder classes are defined in java.lang. Thus, they are available to all programs automatically. All three implements CharSequence interface.
- **String Constructor**
- The String class support several constructors. To create an empty String, you call the default constructor.
- For example,

```
String s=new String();
this will create an instance of with no characters in it.
String s = new String("Computer Deparatment");
```

```
class StringEx
{
 public static void main(String args[])
 {
      String s1=new String("Computer Department");
      String s2;
      s2=s1 + ",Saurashtra University";
      System.out.println(s2);
 }
}
```

- **String Array**
- To create a String initialized by an array of characters, use the constructor shown here:

```
String(char chars[])
For example:
char chars[]={'j','a','v','a'};
String s=new String(chars);
String(char chars[], int startIndex, int numChars)
```

For example:

```
char chars[]={'j','a','v','a'};
String s=new String(chars,1,2);
String(String strobj)
String(byte asciiChars[])
String(byte asciiChars[], int startIndex, int numChars)
```

```
class StringArray
{
 public static void main(String args[])
 {
       char chars[]={'j','a','v','a'};
       String s1=new String(chars);
       String s2=new String(s1);
       String s3=new String(chars,1,2);
       System.out.println("String array:" +s1);
       System.out.println("object as parameter:" +s2);
       System.out.println("position:" +s3);
 }
}
```

```
class StringAsci
{
 public static void main(String args[])
 {
       byte ascii[]={65,66,67,68,69,70};
       String s1= new String(ascii);
       System.out.println(s1);

       String s2=new String(ascii,2,3);
       System.out.println(s2);
 }
}
```

- **String Operation**
- Earlier we have explicitly created a String instance form an array of character by using the new operator.
- There is an easier way to do this using a string literal.
- For each string literal in your program, java automatically constructs a String object. Thus, you can use a string literal to initialize a String object.
- for example, the following code creates two equivalent strings:

```
char chars[]={'a','b','c','d'}
String s1=new String(chars);
String s2="abcd";
```

String Concatenation

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

```
        String s1="Computer Science Department";
        String s2="Saurashtra University";
        String s3=s1+s2;
        System.out.println(s3);
    }
}
```

- **String Methods**

Table 3.4.1 String class Methods

Method Call	Task Performed
S2=s1.toLowerCase;	Conver ts the string s1 to lowercase
S2=s1.toUpperCase;	Converts the string s1 to uppercase
S2=s1.replace('x','y')	Replace all appearances of x with y.
S2=s1.trim()	Remove white spaces at the beginning and end of the string s1
S1.equals(s2)	Returns true if s1 and s2 are equal
S1.equalsIgnoreCase(s2)	Returns true if s1=s2, ignoring the case of characters
S1.length()	Gives the length of s1
S1.CharAt(n)	Gives the nth character of s1
S1.compareTo(s2)	Returns –ve if s1<s2, +ve if s1>s2, and 0 if s1=s2
S1.concat(s2)	Concatenates s1 and s2
S1.substring(n)	Gives substring starting from nth character.
S1.substring(n,m)	Gives substring starting from nth char up to mth
String.ValueOf(p)	Returns the string representation of the specified type argument.
toString()	This object (which is already a string!) is itself returned.

S1.indexOf('x')	Gives the position of the first occurrence of 'x' in the string s1
S1.indexOf('x',n)	Gives the position of 'x' that occurs after nth position in the string s1
String.ValueOf(variable)	Converts the parameter value of string representation

- Example:

```
class IndexEx
{
 public static void main(String args[])
 {
        String s1="Computer Science Department";
        int length=s1.length();
        System.out.println("s1.indexOf('c')"
 +s1.indexOf('c'));
        System.out.println("s1.lastIndexof('c')"
 +s1.lastIndexOf('c'));
 }
}
```

3.4.2. StringBuffer

- StringBuffer is a peer class of String.
- String creates strings of fixed length, while StringBuffer creates strings of flexible length that can be modified in terms of both length and content.
- We can insert characters and substrings in the middle of a string, or append another string to the end.
- StringBuffer defines these Constructor:
  StringBuffer()
  StringBuffer(int size)
  StringBuffer(String str)
  StringBuffer(CharSequence chars)

Table 3.4.2 StringBuffer Class Methods

Method Call	Task Performed
Sb.length()	Gives the current length of a StringBuffer.
Sb.capacity()	Gives the total allocated capacity (default 16)
ensureCapacity(int capacity)	It preallocate room for a certain number of characters after a StringBuffer has been constructed.

setLength(int len)	Set the length of the buffer within a String Buffer object.
charAt(int where)	Gives the value of charat
setCharAt(int where, char ch)	Set the value of a character within a StringBuffer.
S1.append(s2)	Appends the string s2 to s1 at the end
S1.insert(n,s2)	Inserts the string s2 at the position n of the string s1
S1.reverse()	Reverse the string of s1
S1.deleteCharAt(nth)	Delete the nth character of string s1
S1.delete(startIndex, endIndex)	Delete characters from start to end.

- Example:

```
class BufferEx
{
 public static void main(String args[])
 {
       StringBuffer s1=new StringBuffer("Computer Science
Department");
       StringBuffer s2=new StringBuffer("Saurashtra University");
       System.out.println("inserting value:"+s1.insert(9,s2));
       System.out.println("appending two strings:"+s1.append(s2));

 }
}
```

3.4.3. Math class

- The Java Math class provides more advanced mathematical calculations other than arithmetic operator.
- The java.lang.Math class contains methods which performs basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
- All the methods of class Math are static.
- Fields:
  - Math class comes with two important static fields
  - E: returns double value of Euler's number (i.e 2.718281828459045).
  - PI: returns double value of PI (i.e. 3.141592653589793).

Table 3.4.3.1 Math class Method

Method	Description
Math.abs()	It will return the Absolute value of the given value.
Math.max()	It returns the Largest of two values.
Math.min()	It is used to return the Smallest of two values.
Math.round()	It is used to round of the decimal numbers to the nearest value.
Math.sqrt()	It is used to return the square root of a number.
Math.cbrt()	It is used to return the cube root of a number.
Math.pow()	It returns the value of first argument raised to the power to second argument.
Math.signum()	It is used to find the sign of a given value.
Math.ceil()	It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer.
Math.copySign()	It is used to find the Absolute value of first argument along with sign specified in second argument.
Math.nextAfter()	It is used to return the floating-point number adjacent to the first argument in the direction of the second argument.
Math.nextUp()	It returns the floating-point value adjacent to d in the direction of positive infinity.
Math.nextDown()	It returns the floating-point value adjacent to d in the direction of negative infinity.
Math.floor()	It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value.
Math.floorDiv()	It is used to find the largest integer value that is less than or equal to the algebraic quotient.
Math.random()	It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

Math.rint()	It returns the double value that is closest to the given argument and equal to mathematical integer.
Math.hypot()	It returns sqrt(x^2 +y^2) without intermediate overflow or underflow.
Math.ulp()	It returns the size of an ulp of the argument.
Math.getExponent()	It is used to return the unbiased exponent used in the representation of a value.
Math.IEEEremainder()	It is used to calculate the remainder operation on two arguments as prescribed by the IEEE 754 standard and returns value.
Math.addExact()	It is used to return the sum of its arguments, throwing an exception if the result overflows an int or long.
Math.subtractExact()	It returns the difference of the arguments, throwing an exception if the result overflows an int.
Math.multiplyExact()	It is used to return the product of the arguments, throwing an exception if the result overflows an int or long.
Math.incrementExact()	It returns the argument incremented by one, throwing an exception if the result overflows an int.
Math.decrementExact()	It is used to return the argument decremented by one, throwing an exception if the result overflows an int or long.
Math.negateExact()	It is used to return the negation of the argument, throwing an exception if the result overflows an int or long.
Math.toIntExact()	It returns the value of the long argument, throwing an exception if the value overflows an int.

Table 3.4.3.2 Logarithmic Math Method

Method	Description
Math.log()	It returns the natural logarithm of a double value.

Math.log10()	It is used to return the base 10 logarithm of a double value.
Math.log1p()	It returns the natural logarithm of the sum of the argument and 1.
Math.exp()	It returns E raised to the power of a double value, where E is Euler's number and it is approximately equal to 2.71828.
Math.expm1()	It is used to calculate the power of E and subtract one from it.

Table 3.4.3.3 Trigonometric Math Method

Method	Description
Math.sin()	It is used to return the trigonometric Sine value of a Given double value.
Math.cos()	It is used to return the trigonometric Cosine value of a Given double value.
Math.tan()	It is used to return the trigonometric Tangent value of a Given double value.
Math.asin()	It is used to return the trigonometric Arc Sine value of a Given double value
Math.acos()	It is used to return the trigonometric Arc Cosine value of a Given double value.
Math.atan()	It is used to return the trigonometric Arc Tangent value of a Given double value.

Unit-4
Inheritance and Abstraction

4.1. Inheritance

- Inheritance is the process by which object of one class acquires the properties of another class.
- Inheritance allows the creation of hierarchical classifications.
- A class that is inherited is called a superclass. The class that does the inheriting is called a subclass.
- Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and add its own, unique elements.
- To inherit a class, you simply incorporate the definition of one class into another by using the extend keyword.
- The general form of class declaration that inherits a superclass is shown here.

```
class subclass-name extends superclass-name
{
      // body of class.
}
```

- You can only specify one super class for any subclass.
- Java does not support the inheritance of multiple super classes into a single subclass.
- We can create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

- Types of Inheritance
- There are five types of inheritance as listed below.
  - Single Inheritance
    In Single Inheritance one class extends another class (one parent class and one child class).
  - Multiple Inheritance
    Multiple Inheritance is one of the inheritances in Java types where one class extending more than one class. Java does not support multiple inheritance.
  - Multilevel Inheritance
    In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.
  - Hierarchical Inheritance
    In Hierarchical Inheritance, one class is inherited by many sub classes.
  - Hybrid Inheritance
    Hybrid inheritance is one of the inheritance types in Java which is a combination of Single and Multiple inheritance.

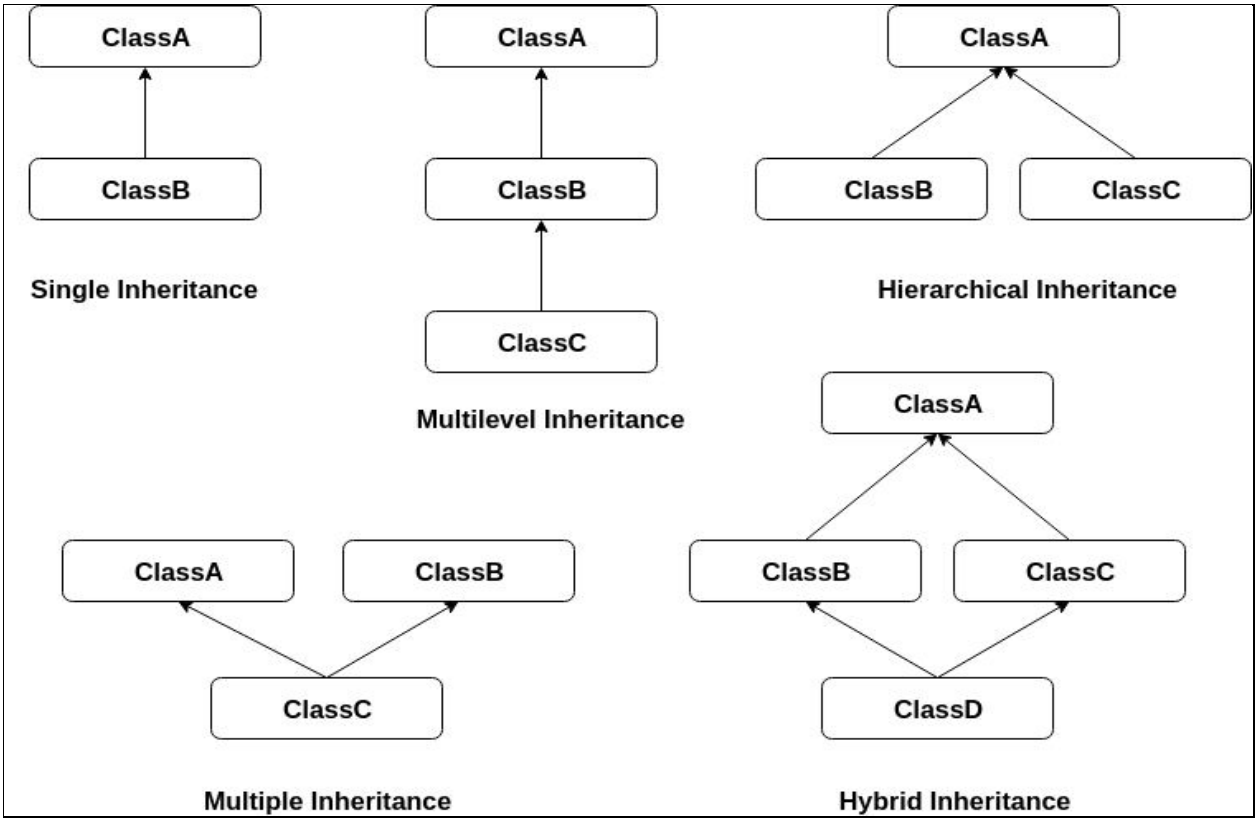

Figure 4.1 Types of Inheritance

4.2. Function Overloading and Overriding

- When two or more methods in the same class have the same name but different parameters, it's called Overloading.
- When the method signature (name and parameters) are the same in the superclass and the child class, it's called Overriding.

4.2.1. Function Overloading v/s Function Overriding

- Overriding implements Runtime Polymorphism whereas Overloading implements Compile time polymorphism.
- The method Overriding occurs between superclass and subclass. Overloading occurs between the methods in the same class.
- Overriding methods have the same signature i.e. same name and method arguments. Overloaded method names are the same but the parameters are different.
- With Overloading, the method to call is determined at the compile-time. With overriding, the method call is determined at the runtime based on the object type.
- If overriding breaks, it can cause serious issues in our program because the effect will be visible at runtime. Whereas if overloading breaks, the compile-time error will come and it's easy to fix.
- In simple words, the compiler knows which method to execute in method overloading, as soon as the compiler compiles the program. This is static binding. the compiler does not know what method to execute

during compilation. It is executed during runtime. This is dynamic binding.

4.2.2. Example of Function Overloading

```
public class OverloadExample {
 public static int area(int side) {
   //calculates and returns the area of square
   return side * side;
 }
 public static int area(int side1, int side2) {
   //calculates and returns the area of rectangle
   return side1 * side2;
 }
 public static void main(String[] args) {
   System.out.println(area(5));
   System.out.println(area(5, 2));
 }
}
```

4.2.3. Example of Function Overriding

```
class Father {
 void bowl() {
   System.out.println("Fast Bowler");
 }
}
class Child extends Father {
 void bowl() {
   System.out.println("Spin Bowler!");
 }
}
public class OverridingExample {
 public static void main(String[] args) {
   Father f = new Father();
   f.bowl();
   Father c = new Child();
   c.bowl();
 }
}
```

4.2.4. Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism in which a call to an overridden method is resolved at run time instead of compile time. This is an important concept because of how Java implements run-time polymorphism.
- Java uses the principle of 'a superclass reference variable can refer to a subclass object' to resolve calls to overridden methods at run time. When a superclass reference is used to call an overridden method, Java determines which version of the method to execute based on the type of the object being referred to at the time call.
- In other words, it is the type of object being referred to that determines which version of an overridden method will be executed.
- Advantages of dynamic method dispatch

- It allows Java to support overriding of methods, which are important for run-time polymorphism.
- It allows a class to define methods that will be shared by all its derived classes, while also allowing these sub-classes to define their specific implementation of a few or all of those methods.
- It allows subclasses to incorporate their own methods and define their implementation.

```
class Apple
{
    void display()
    {
        System.out.println("Inside Apple's display method");
    }
}

class Banana extends Apple
{
    void display()    // overriding display()
    {
        System.out.println("Inside Banana's display method");
    }
}

class Cherry extends Apple
{
    void display()    // overriding display()
    {
        System.out.println("Inside Cherry's display method");
    }
}

class Fruits_Dispatch
{
    public static void main(String args[])
    {
        Apple a  = new Apple();    // object of Apple
        Banana b = new Banana();   // object of Banana
        Cherry c = new Cherry();   // object of Cherry

        Apple ref;     // taking a reference of Apple

        ref = a;    // r refers to a object in Apple
        ref.display();    // calling Apple's version of display()

        ref = b;    // r refers to a object in Banana
        ref.display();    // calling Banana's version of display()

        ref = c;  // r refers to a object in Cherry
        ref.display();  // calling Cherry's version of display()
    }
}
```

4.3. Constructor

- A constructor in Java is a special method that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.
- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized
- There are two types of constructors in Java:
  - Default constructor (no-argument constructor)
  - Parameterized constructor

4.3.1. Example of Default Constructor

```
public class Bike {
//creating a default constructor
Bike () {System.out.println("Bike is created");}
//main method
public static void main (String args[]){
//calling a default constructor
Bike b=new Bike ();
}
}
```

4.3.2. Example of Parametrized constructor

```
public class Student{
    int id;
    String name;
    //creating a parameterized constructor
    Student (int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display (){System.out.println(id+" "+name);}

    public static void main (String args[]){
    //creating objects and passing values
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

4.3.3. Constructor Overloading

- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

```
public class Student{
    int id;
    String name;
    int age;
    //creating two arguments constructor
    Student(int i,String n){
    id = i;
    name = n;
    }
    //creating three arguments constructor
    Student(int i,String n,int a){
    id = i;
    name = n;
    age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan",25);
    s1.display();
    s2.display();
   }
}
```

4.4. Keywords: super, final, this and static

4.4.1. Super keyword:

- However, sometimes you want to create a super class that keeps the details of its implementation to itself (i.e., it keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own.
- Since encapsulation provides a solution to this problem. Whenever a subclass needs to refer to its immediate super class, it can do so by use of the keyword super.
- Super has 2 general forms
  - It calls the super class's constructor.
  - It is used to access a member of the superclass that has been hidden by member of subclass.
- [1] Super to call superclass Constructor
  - A subclass can call a constructor defined by its superclass by use of the following form of super.
  - super(parameter-list)
  - Where parameter-list specifies any parameter needed by the constructor in the superclass.
  - super() must always be the first statement executed inside a subclass constructor.
- Example:

```
class Box
{
```

```
 private double width;
 private double height;
 private double depth;
 Box(double w, double h, double d)
 {
       width=w;
       height=h;
       depth=d;
 }


class BoxWeight extends Box
{
 double weight;
 BoxWeight(double w, double h, double d, double m)
 {
       super(w,h,d);
       weight=m;
 }
class DemoSuper
{
 public static void main(String args[])
 {
       BoxWeight mybox1= new BoxWeight(10,20,15,34.3);
       double vol;
       vol=mybox1.volume();
       System.out.println("Volume of mybox1 is:"+vol);
 }
 }
```

- [2] It is used to access a member of the superclass that has been hidden by member of subclass.
  - The second form of super acts somewhat like this,except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:- Super.member
  - Member=can be either a method or an instance variable.
  - This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.
- Example:

```
class A
{
 int i;
}
class B extends A
{
 int i;  //this I hides the I in A
 B(int a, int b)
 {
       super.i=a;
       i=b;
```

```
    }
    void show()
    {
          System.out.println("I in superclass:"+super.i);
          System.out.println("I in subclass:"+i);
    }
}
class UseSuper
{
    public static void main(String args[])
    {
          B subob=new B(1,2);
          subob.show();
    }
}
```

4.4.2. Final Keyword

- The keyword final has 3 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
  - Using final to prevent Inheritance
- **[1] 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 form being overridden, specify final as a modifier at the start of its declaration.
- Methods declared as final cannot be overridden.
- Example:

```
class A
{
    final void meth()
    {
          System.out.println("this is a final method");
    }
}
class B extends A
{
    Void meth()
    {
          // error cannot 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.
- Methods declared as final can sometimes provide a performance enhancement: the compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass.

- When a small final method is called, often the java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. In lining is only an option with final methods.
- Normally, java resolves call to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.
- **[2] 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.
- Example of final class:

```
final class A
{
 ----
}
class B extends A
{
 //error
}
```

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

4.4.3. This Keyword

- 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.
- this is always a ref. to the object on which the method was invoked.
- Consider the following Example:
- //redundant use of this

```
Box(double w, double h, double d){
   this.width=w;
   this.height=h;
   this.depth=d;
}
```

- **Instance Variable Hiding:**
- It is illegal in java to declare two local variables with the same name inside the same or enclosing scopes.
- We have local variables, including formal parameters to methods, which overlap with the names of the class's instance variables.
- However, when a local variable has the same name as an instance variable, the local variable hides the instance variable.

- This is why width, height and depth were not used as the names of the parameters to the Box() constructor inside the box class.
- If they had been, them width would have referred to the formal parameter, hiding the instance variable width. While it is easier to use different names.
- this lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.
- For example, here is another version of Box(), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name.
- // use this to resolve name-space collisions.

```
Box(double width, double height, double depth)
{
 this.width=width;
 this.height=height;
 this.depth=depth;
}
```

4.4.4. Static keyword

- Sometimes you want to define a class member that will be used independently without (of) any object of that class.
- Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword static.
- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to an object.
- You can declare both methods and variables to be static.
- The most common example of a static member is main().main() is declared as static because it must be called before any object exist.
- Instance variables declared as static are actually, global variables. When objects of its class are declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.
- Method declared as static have several restrictions:
  - They can only call other static methods.
  - They must only access static data.
  - They cannot refer to this or super in any way.
- If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded.
- The following example shows a class that has a static method, static variables and static initialization block:
- As the UseStatic class is loaded, all of the static statements are run. first a is set to 3 then the static block executes and finally, b is initialized to a*4 or 12. then main() is called which calls

metho(), passing 42 to x. the 3 println() statements refer to the two static variables a and b as well as to the local variable x.

- Example:

```
class UseStatic
{
 static int a=3;
 static int b;
 static void meth(int x)
 {
       System.out.println("x=" +x);
       System.out.println("a=" +a);
       System.out.println("b=" +b);
 }
 static
 {
       System.out.println("Static block initialized");
       b=a*4;
 }
 public static void main(String args[])
 {
       meth(42);
 }
}
```

- Static block initialized
  X=42
  A=3
  B=12
- Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need to specify only name of their class followed by the dot operator.
- For Ex, if you wish to call a static method from outside its class, you can do so using the following:
- Classname.method()
  - Class name is the name of the class in which static method is declared. A static variable and method can be accessed in same way by use of the dot operator on the name of the class.

```
class StaticDemo
{
 static int a=42;
 static int b=99;
 static void callme()
 {
       System.out.println("a=" +a);
 }
}
class StaticByName
{
 public static void main(String args[])
 {
       StaticDemo.callme();
       System.out.println("b=" +StaticDemo.b);
```

```
    }
}
```

Output:

```
A=42
B=99
```

Unit-5
Package and Exception Handling

5.1. Package

- Packages are container for classes that are used to keep the class name space compartmentalized.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definition.
- Java provides a mechanism for partitioning the class name space into more manageable chunk. This mechanism is the package.
- The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class member that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each-other but not expose that knowledge to the rest of the world.
- **Defining a Package**
  - To create a package, simply include a package command as the first statement in java source file.
  - Any classes declared within that file will belong to the specified package.
  - The package statement defines a name space in which classes are stored.
  - If you omit the package statement, the class names are put into the default package, which has no name.
- General form of package statement:
  Package pkg;
- Where, pkg is the name of the package. For example, following statement creates a package called MyPackage must be stored in directory called MyPackage. The directory name must match the package name exactly.
- More than one file can include the same package statement.
- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of multileveled package statement:
  Package pkg1[.pkg2[.pkg3]];
- For example:
  Package java.awt.image;
- This package needs to be stored in java/awt/image, java\awt\image or java:awt:image on your system.
- You cannot rename a package without renaming the directory in which the classes are stored.
- **Finding Packages and CLASSPATH**
  - How does the java run-time system know where to look for packages that you create? – the answer has two parts:
  - by default, the java run-time system uses the current working directories as its starting point. Thus, if your package is in

the current directory, or a subdirectory of the current directory, it will be found.
- you can specify a directory path or paths by setting the CLASSPATH environmental variable.

5.2. Access Specifiers

- **Access Control**
- Encapsulation links data with the code that manipulates it. Encapsulation provides another important attribute: Access Control
- Through encapsulation, you can control what parts of a program can access the member of a class. By controlling access, you can prevent misuse.
- How a member can be accessed is determined by the access specifier that modifies its declaration.
- Java supplies a rich set of access specifies. Some aspects of access control are related to inheritance or packages. Let's begin by examining access control as it applies to a single class.
- Java's access specifiers are:
- **Public**
  - When a member of a class is modified by the public specifier, then that member can be accessed by any other code.
- **Private**
  - When a member of a class is specified as private, then that member can only be accessed by other members of its class.
- **Protected**
  - If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.
- **Default**
  - when a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package.
- Now, we can understand why main (), has always been preceded by the public specifier. It is called by the code outside of the program. (by java run-time system).
- When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.
- In the classes developed so far, all member of a class have used the default access mode, which is essentially public usually, you will want to restrict access to the data members of a class allowing access only through methods. Also, there will be times when you will want to define methods, which are private to a class.
- An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:

```
public int i;
private double j;
```

```
   private int myMethod (int a, char b);
```

- To understand the effects of public, private access considers the following program:

```
class Test
{
   int a;
   public int b;
   private int c;

   void SetC (int i)
   {
        c = i;
   }
   int getc()
   {
        return c;
   }
}
class AccessTest
{
   public static void main (String args[])
   {
        Test ob = new Test();
        ob.a = 10;
        ob.b = 20;
        //ob.c = 100;   // cause an error.
        ob.SetC (100);
        System.out.println("a, b and c: "+ob.a+" "+ob.b+" "+ob.getc());
   }
}
```

5.2.1. Access Protection

- Public: Anything declared public can be accessed from anywhere.
- Private: Anything declared private cannot be seen outside of its class.
- Default: when a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package.
- Protected: If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.
- A class only has two possible access level: default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.
- Following is tabular representation of Access Specifiers

	PRIVATE	DEFAULT	PROTECTED	PUBLIC
Same class	Yes	Yes	Yes	Yes
Same package Subclass	No	Yes	Yes	Yes
Same package Non-subclass	No	Yes	Yes	Yes
Different package Subclass	No	No	Yes	Yes
Different package Non-subclass	No	No	No	Yes

5.3. Exception Handling Mechanism

- An exception is an object that is generated at run-time to describe a problem encountered during the execution of a program.
- Some causes for an exception are integer division-by-zero, array index negative or out-of-bounds, illegal cast, interrupted I/O operation, unexpected end-of-file condition, missing file, incorrect number format.
- An exception is an abnormal condition that arises in a code sequence at run time or we can say an exception is a run-time error.
- In computer languages that do not support exception handling, errors must be checked and handled manually through the use of error codes.
- A java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws and finally**.
- **Try**: program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown.
- **Catch**: your code can catch this exception using catch and handle it in some rational manner.
- **Throw**: system-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw.
- **Throws**: any exception that is thrown out of a method must be specified by a throws clause.

- **Finally:** any code that absolutely must be executed before a method returns is put in a finally block.
- General form:

```
Try
{
      //block of code to monitor for errors…
}
Catch(ExceptionType1 exOb)
{
      // exception handling block
}
Finally
{
      // finally block
}
```

- The try statement contains a block of statements enclosed by braces. This is the code you want to monitor for exceptions. If a problem occurs during its executing, an exception is thrown.
- Immediately following the try block is a sequence of catch blocks. Each of these begins with the catch keyword. An argument is passed to each catch block. That argument is the exception object that contains information about the problem.
- If a problem occurs during execution of the try block, the JVM immediately stops executing the try block and looks for a catch block that can process that type of exception. Any remaining statements in the try block are not executed. The search begins at the first catch block. If the type of the exception object matches the type of the catch block parameter, those statements are executed. Otherwise, the remaining catch clauses are examined in sequence for a type match.
- When a catch block completes executing, control passes to the statements in the finally block. The java compiler ensures that the completes without problems, the finally block executed in all circumstances.
- When a try block completes without problems, the finally block executes. Even if a return statement is included in a try block, the compiler ensures that the finally block is executed before the current method returns.
- The finally block is optional. However, in some applications it can provide a useful way to relinquish resources. For example, you may wish to close files or databases at this point.
- Each try block must have at least one catch or finally block.
- Example:

```
Class ExceptionTest
{
 Public static void main(String args[])
 {
      Int a=10;
      Int b=5;
      Int c=5;
      Int x,y;
```

```
        Try
        {
              X=a/(b-c);
        }
        Catch(ArithmeticException e)
        {
              System.out.println("Division by zero");
        }
        Y=a/(b+c);
        System.out.println("y="+y);
   }
  }
  Output:
  Division by zero
  Y=1
```

- **Displaying a Description of an Exception**

  Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the excaeption. You can display this description in a println() statement by simply passing the exception as an argument. For example, the catch block in the preceding program can be rewritten like this:

```
Catch (ArithmeticException e)
{
      System.out.println("Exception:" +e);
      A=0;
}
```

5.3.1. Multiple catch Clauses for different types of exception.

- In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.
- The following example traps two different exception types:

```
Class MultiCatch
{
 Public static void main(String args[])
 {
      Try
      {
            Int a=args.length;
            System.out.println("a=" +a);
            Int b= 42/a;
            Int c[]={1};
            C[42]=99;
      }
      Catch(ArithmeticException e)
      {
            System.out.println("divide by zero:" +e);
```

```
        }
        Catch(ArrayIndexOutOfBoundsException e)
        {
              System.out.println("Array index oob:" +e);
        }
  }
  System.out.println("After try/catch block");
  }
}
```

- When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their super classes.
- This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in java, unreachable code is an error.
- For example, consider the following program:

```
Class SuperSubCatch
{
  Public static void main(String args[])
  {
        Try
        {
              Int a=0;
              Int b=42/a;
        }
        Catch(Exception e)
        {
              System.out.println("Generic Exception catch.");
        }
        // ArithmeticException is a subclass of Exception.
        Catch(ArithmeticException e)
        {
              // error-unreachable
              System.out.println("this is never reached.");
        }
  }
}
```

- If you try to compile this program, you will receive an error message stating that the second catch statement is unreachable because the exception has already been caught. Since ArithmeticException is a subclass of Exception, the first catch statement will handle all exception based errors, including ArithmeticException. This means that the second catch statement will never execute. To fix the problem, revere the order of the catch statement.
- Example:

```
class ExceptionMulti
{
  Public static void main(String args[])
  {
        Int a[]={5,10};
```

```
        Int b=5;
        Try
        {
              Int x=a[2]/b-a[1];
        }
        Catch(ArithmeticException e)
        {
              System.out.println("Division by zero");
        }
        Catch(ArrayIndexOutOfBoundsException e)
        {
              System.out.println("Array index error");
        }
        Catch(ArrayStoreException e)
        {
              System.out.println("wrong data type");
        }
        Int y=a[1]/a[0];
        System.out.println("y=" +y);
    }
  }
```

5.3.2. Throw

- We saw that an exception was generated by the JVM when certain run-time problems occurred.
- It is also possible for our program to explicitly generate an exception. This can be done with a throw statement. Its form is as follows:

```
throw object;
```

- Here, object must be of type java.lang.Throwable. Otherwise, a compiler error occurs.
- Inside a catch block, you may throw the same exception object that was provided as an argument. This can be done with the following syntax:

```
catch(ExceptionType param)
{
      throw param;
}
```

- Alternatively, you may create and throw a new exception object as follows:
- throw new ExceptionType(args);
- Here, exceptionType is the type of the exception object and args is the optional argument list for its constructor.
- When a throw statement is encountered, a search for a matching catch block begins. Any subsequent statements in the same try or catch block are not executed.

5.3.3. Throws

- When you write a method that can throw exceptions to its caller,
- A Java language keyword valid only at the end of method declarations that specifies which exceptions are not handled within the method

but rather passed up the call stack. Unhandled checked exceptions are required to be declared in a method's throws clause whereas unchecked exceptions are generally not declared.

Unit-6
IO Programming

6.1. File Class

- Java I/O (Input and Output) is used to process the input and produce the output.
- Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.
- We can perform file handling in Java by Java I/O API.
- **Stream**
  - A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.
  - InputStream – The InputStream is used to read data from a source.
  - OutputStream – The OutputStream is used for writing data to a destination.

6.1.1. Files and Directories

- The file class encapsulates information about the properties of a file or directory. These include its read and write permissions, time of last modification, and length.
- It is also possible to determine the files that are contained in a directory. This is valuable because you can build an application that navigates a directory hierarchy.
- New directories can be created, and existing files and directories may be deleted or renamed.
- The file class provides the following constructors:
  - File(String path)
  - File(String directoryPath, String filename)
  - File(File directory, String filename)
- The first form has one parameter that is the path to a file or directory.
- The second form has two parameters. These are the path to a directory and the name of a file in that directory.
- The last form also has two parameters. These are a File object for a directory and the name of a file in that directory.
- All of these constructors throw a NullPointerException if path or filename is null.
- File defines two char constants. These are separatorChar (\) and pathSeparatorChar (;).
- **Methods of File Class**

Table 6.1.1 File Class Methods

Method	Description

Boolean canRead()	Returns true if the file exists and can be read. Otherwise, returns false.
Boolean canWrite()	Returns true if the file exists and can be written. Otherwise, returns false.
Boolean delete()	Deletes the file. Returns true if the file is successfully deleted. Otherwise, returns false. Note that a directory must be empty before it can be deleted.
Boolean equals(Object obj)	Returns true if the current object and obj refer to the same file. Otherwise, return false.
Boolean exists()	Returns true if the file exists. Otherwise, returns false.
String getAbsolutePath()	Returns the absolute path to the file.
String getCanonicalPath()	Returns the canonical path to the file.
String getName()	Returns the name of the file.
String getParent()	Returns the parent of the file.
String getPath()	Returns the path to the file.
Boolean isAbsolute()	Returns true if the file path name is absolute. Otherwise, returns false.
Boolean isDirectory()	Returns true if the file is a directory. Otherwise, returns false.
Boolean isFile()	Returns true if the file is not a directory. Otherwise, returns false.
Long lastModified()	Returns the number of milliseconds between 00:00:00 GMT, January 1, 1970, and the time of last modification for this file.
Long length()	Returns the number of bytes in the file.
String[] list()	Returns the names of the files in the directory.
Boolean mkdir()	Creates a directory with the name of this file. All parent directories must already exist. Returns true if the directory was created. Otherwise, returns false.
Boolean mkdirs()	Creates a directory with the name of this file. any missing parent directories are also created. Returns true if the

	directory was created. Otherwise, returns false.
Boolean renameTo(File newName)	Renames the file or directory to newName. Returns true if the current object has been renamed to newName. Otherwise, returns flase.

- Example:

```
import java.io.*;
class FileDemo
{
 public static void main(String args[])
 {
 try
 {
  System.out.println("pathSeparatorChar=" +File.pathSeparatorChar);
  System.out.println("separatorChar=" +File.separatorChar);

  File file= new File(args[0]);
  System.out.println("getName()=" +file.getName());
  System.out.println("getParent()=" +file.getParent());
  System.out.println("getAbsolutePath()=" +file.getAbsolutePath());
  System.out.println("getCanonicalPath()="
+file.getCanonicalPath());
  System.out.println("getPath()=" +file.getPath());
      System.out.println("canRead()=" +file.canRead());
  System.out.println("canWrite()=" +file.canWrite());
 }
 catch(Exception e)
 {
       e.printStackTrace();
 }
 }
}
```

6.2. Streams, Byte Stream and Character Stream

- **Streams**
- A stream is an abstraction for a source or destination of data. (A stream is an abstraction that either produces or consumes information.)
- A stream is linked to a physical device by the Java I/O system.
- It enables you to use the same techniques to interface with different types of physical devices. For example, an input stream may read its data from a keyboard, file or memory buffer. An output stream may write its data to a monitor, file or memory buffer. – other types of devices may also be used as the source or destination for a stream.
- There are two types of streams:
  - character Streams
  - byte Streams
- **character Streams**

- It allows you to read and write characters and strings.
- An input character Stream converts bytes to Character.
- An output character Stream converts character to byte.

- **Byte Streams**
  - It allows you to read and write binary data.
  - For ex, an application that simulates the behavior of an electric circuit can write a sequence of float values to a file. These would represent the value of a signal over a time interval.

6.2.1. Character Stream Classes

- Character stream allows you to read and write characters and strings.
- An input character stream converts bytes to characters.
- An output character stream converts characters to bytes.
- Java internally represents characters according to the 16-bit Unicode encoding. However, this may not be the encoding used on a specific machine.
- Character streams translate between these two formats.
- Ability to translate between Unicode and other encodings is an important feature because it enables you to write programs that operate correctly for an international marketplace.
- Fig shows a few of the character streams provided by the java.io package.

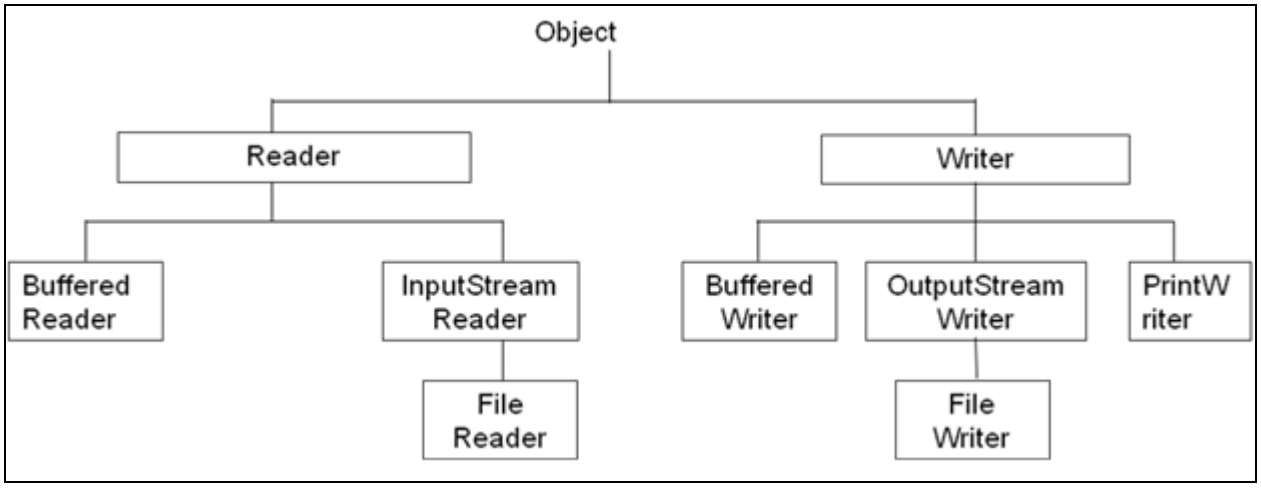

Figure 6.2.1 Character Stream Class

- **Reader Class**
- Reader stream classes are designed to read characters from the file.
- The abstract Reader class defines the functionality that is available for all character input streams.
- Reader is an abstract class and therefore, we cannot create an instance of this class. Rather, we must use the subclasses that inherit form the Reader class. i.e. Buffered Reader and Input Stream Reader.

Table 6.2.1 Reader Class Methods

Method	Description
Void close()	Closes the input stream. Further read attempts generate an IOException. Must be implemented by a subclass.
Void mark(int numChars)	Places a mark at the current point in the input stream that will remain valid until numChars characters are read.
Boolean markSupported()	Returns true if mark()/reset() are supported in this stream
Int read()	Reads a character from the stream. Waits until data is available.
Int read(char buffer[])	Attempts to read up to buffer-length characters into buffer and returns the actual number of characters that were successfully read. Waits until data is available.
Int read(char buffer[], int offset, int numChars)	Attempts to read up to numChars characters into buffer starting at buffer[offset] and returns the actual number of characters that were successfully read. Waits until data is available.
Boolean ready	Returns true if the next read() will not wait.
Void reset()	Resets the input pointer to the previously set mark.
Int skip(long numChars)	Skips over numChars bytes of input returning the number of characters actually skipped.

- InputStreamReader
- The Input Stream Reader class extends Reader. It converts a stream of bytes to a stream of characters. This is done according to the rules of a specific character encoding.
- The constructors provided by this class are as follows:
  InputStreamReader(InputStream is)
  InputStreamReader(InputStream is, String encoding)
  Where:
  Is= input stream
  Encoding = name of character encoding.
- The first form of the constructor uses the default character encoding of the user's machine.

- getEncoding() – returns the name of the character encoding. It has following syntax:

```
String getEncoding()
```

- **File Reader**
- The File Reader class extends InputStreamReader and inputs characters form a file.
- Its two most common constructors are:
  FileReader(String filepath)
  FileReader(File fileObj)
- Either can throw a FileNotFoundException.
- The program that reads a file is shown in the following listing. It accepts one command-line argument that is the name of the file to read. A FileReader object is created. Individual characters are obtained via read() and displayed via System.out.print(). The streakm is closed when all characters have been read.
- Example:

```
import java.io.*;
class FileReaderDemo
{
 public static void main(String args[])
 {
      try
      {
            //create a file reader
            FileReader fr=new FileReader(args[0]);

            //read and display characters
            int i;
            while((i=fr.read())!=-1)
            {
                  System.out.print((char)i);
            }

            //close file reader
            fr.close();
      }
      catch(Exception e)
      {
            System.out.println("Exception:"+e);
      }
 }
}
```

- **Writer**
- The writer stream classes are designed to perform all output operation on files.
- The writer class is an abstract class which acts as a base class for all the other writer stream classes.
- This base class provides support for all output operations by defining methods that are identical to those in output stream class.

Table 6.2.1.1 Writer Class Methods

Method	Description
Void close()	Closes the output stream.
Void flush()	Writes any buffered data to the physical device represented by that stream.
Void write(int c)	Writes the lower 16 bits of c to the stream.
Void write(char buffer[])	Writes the characters in buffer to the stream.
Void write(char buffer[], int index, int size)	Writes size characters form buffer starting at position index to the stream.
Void write(String s)	Writes s to the stream.
Void write(String s, int index, int size)	Writes size characters form s starting at position index to the stream.

- OutputStreamWriter
- The output Stream Writer class extends writer.
- It converts a stream of characters to a stream of bytes.
- This is done according to the rules of a specific character encoding.
- Its constructors are like this:
  OutputStreamWriter(OutputStream os)
  OutputStreamWriter(OutputStream os, String encoding)
- Here, os is the output stream and encoding is the name of a character encoding. The first form of the constructor uses the default character encoding of the user's machine.
- The getEncoding() method returns the name of he character encoding. It has this syntax:
  String getEncoding()

- **File Writer**
- The file writer class extends OuptputStreamWriter and outputs characters to a file.
- Its constructors are as follows:
  FileWriter(String filepath) throws IOException
  FileWriter(String filepath, Boolean append) throws IOException
  FileWriter(File fileObj) throws IOExcepiton
- The program that writes file is shown in the following listing. It accepts one command-line argument that is the name of the file to create. Twelve strings are written to the file by using the write() method of FileWriter.

- Example:

```
import java.io.*;
class FileWriterDemo
{
 public static void main(String args[])
 {
      try
      {
            //create a file writer
            FileWriter fw=new FileWriter(args[0]);

            //write string to file
            for(int i=0;i<12;i++)
            {
                  fw.write("Line" +i +"\n");
            }

            //close file writer
            fw.close();
      }
      catch(Exception e)
      {
            System.out.println("Exception:"+e);
      }
 }
}
```

- **Buffered Character Streams**
- There are two classes BufferedWriter and BufferedReader.
- The advantage of buffering is that the number of reads and writes to a physical device is reduce. This improves performance.
- **BufferedWriter**
- The BufferWriter class extends Writer and buffers output to a character stream. Its constructors are as follows:
  BufferedWriter(Writer w)
  BufferedWriter(Writer w, int bufSize)
- The first form creates a buffered stream using a buffer with a default size.
- In second, the size of the buffer is specified by bufsize.
- This class implements all of the methods defined by Writer. In addition, it provides the newLine() method to output a line separator. Its signature is shown below:
  Void newLine() throws IOException
- The program that writes a file is shown in the following listing. It accepts one command-line argument that is the name of the file to create. A FileWriter object is created and passed as the argument to the BufferedWriter constructor. Twelve strings are written to the file by using the write() method of BufferedWriter.
- Example:

```
import java.io.*;
```

```
class BufferedWriterDemo
{
 public static void main(String args[])
 {
      try
      {
            //create a file writer
            FileWriter fw= new FileWriter(args[0]);

            //create a buffered writer
            BufferedWriter bw= new BufferedWriter(fw);
            //write strings to the file
            for(int i=0;i<12;i++)
            {
                  bw.write("Line "+i + "\n");
            }

            //close buffered writer
            bw.close();
      }
      catch(Exception e)
      {
            System.out.println("Exception:"+e);
      }
 }
}
```

- **BufferedReader**
- The BufferedReader class extends Reader and buffers input form a character stream. Its constructors are as follows:
  BufferedReader(Reader r)
  BufferedReader(Reader r, int bufSize)
- The first form creates a buffered stream using a buffer with a default size.
- In second, the size of the buffer is specified by bufsize.
- This class implements all of the functionality defined by Reader. In addition, the readLine() method reads newline-terminated strings form a character stream. Its signature is:
  String readLine() throws IOException
- The program that reads a file is shown in the following program. It accepts one command-line argument that is the name of the file to red. A FileReader object is created and passed as the argument to the BufferedReader constructor. The readLine() method is used to obtain the individual lines in the file. Note that readLine() discards the newline character it reds. The file reader is closed after all lines have been displayed.

```
import java.io.*;
```

- Example:

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

```
    {
          try
          {
                //create a file reader
                FileReader fr=new FileReader(args[0]);
                //create a buffered reader
                BufferedReader br=new BufferedReader(fr);

                //send and display lines from file
                String s;
                while((s=br.readLine())!=null)
                System.out.println(s);

                //close file reader
                fr.close();
          }
          catch(Exception e)
          {
                System.out.println("Exception:"+e);
          }
    }
  }
```

- This example shows how to use a buffered character stream to read input form the keyboard. The program executes an infinite loop that reads a string and displays the number of characters it contains. Each string must be terminated by a newline character.
- System.in is passed as the argument to the InputStreamReader constructor. This is done because System.in is an InputStream. The InputStreamReader object is then passed as the argument to the BufferedReader constructor.
- Example:

```
import java.io.*;
class ReadConsole
{
 public static void main(String args[])
 {
      try
      {
            //create an input stream reader
            InputStreamReader isr=new InputStreamReader (System.in);
            //create a buffered reader
            BufferedReader br=new BufferedReader(isr);
            //read and process lines from console
            String s;
            while((s=br.readLine())!=null)
            {
                  System.out.println(s.length());
            }

            //close inputstream reader
```

```
                isr.close();

            }
            catch(Exception e)
            {
                System.out.println("Exception:"+e);
            }
        }
    }
```

6.2.2. Byte Streams Classes

- Byte streams allow a programmer to work with the binary data in a file.
- Figure shows a few of the byte streams provided by the java.io package. It includes OutputStream, FileOutputStream, DataOutputStream, and PrintStream classes that are used for output, while the InputStream, FileInputStream, FilterInputStream, BufferedINputStream, and DataInputStream classes are used for input.
- The OutputStream class defines the functionality that is available for all byte output streams.
- The following table summarizes the methods provided by this class.

Table 6..2 ByteStream Class's Methods

Method	Description
Void close() Throws IOException	Closes the output stream.
Void flush() Throws IOException	Flushes the output stream.
Void write(int i) Throws IOException	Writes lowest-order 8 bits of I to the stream.
Void write(byte buffer[]) throws IOException	Writes buffer to the stream.
Void write(byte buffer[], int index, int size) Throws IOException	Writes size bytes form buffer starting at position index to the stream.

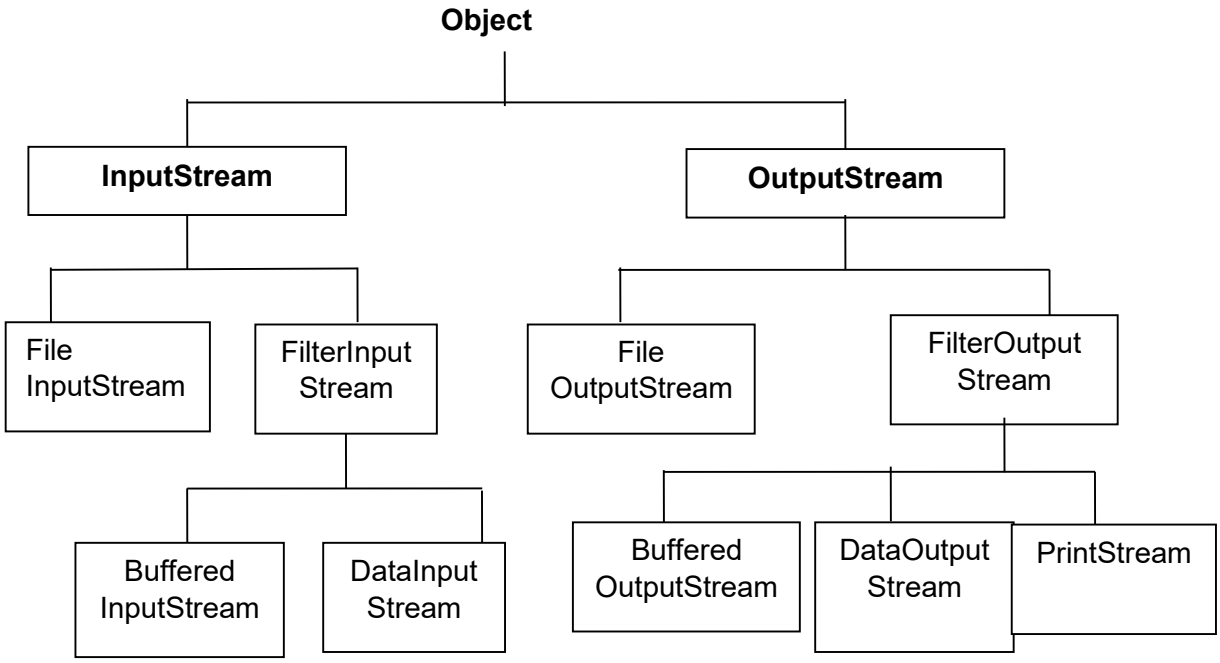

Figure: 6.2 Byte Stream classes

- **InputStream**
- The abstract InputStream class defines the functionality that is available for all byte output streams.
- The methods provided by the InputStream class are:
- **FileInputStream**
- The fileInputStream class extends INputStream and allows you to read binary data form a file. Its most commonly used constructors are as follows:
  FileInputStream(String filepath) throws FileNotFoundException
  FileInputStream(File fileObj) throws FileNotFoundExcepiton
- Here, filepath is the full path name of a file and fileObj is a File object athat describes the file.
- **FilterInputStream**
- The FilterInputStream class extends InputStream and filters an input stream. It provides this constructor:
  FilterInputStream(InputStream is)
- Here, is is the input stream to be filtered.
- You do not directly instantiate FilterInputStream. Instead, you must create a subclass to implement the desired functionality.
- **BufferedInputStream**
- The BufferedInputStream class extends FilterInputStream and buffers input form a byte stream. Its constructors are as follows:
  BufferedInputStream(InputStream is)
  BufferedInputStream(InputStream is, int bufSize)
- The first argument to both constructors is a reference to the input stream. The first form creates a buffered stream by using a buffer with a default size.

- In second, the size of the is specified by bufSize.
- The program that reads a file is shown in the following listing. It accepts one cmdline argument that is the name of the file to read. A FileInputStream object is created. Bytes are obtained via read() and displayed via System.out.println(). The stream is closed when all bytes have been read.
- Example:

```
import java.io.*;
class FileInputStreamDemo
{
 public static void main(String args[])
 {
      try
      {
            //create a file input stream
            FileInputStream fis= new FileInputStream(args[0]);
            //read and display data
            int i;
            while((i=fis.read())!=-1)
            {
                  System.out.println(i);
            }
            //close file input stream
            fis.close();
      }
      catch(Exception e)
      {
            System.out.println("Exception:"+e);
      }
 }
}
```

- The program that writes to a file is shown in the following listing. It accepts one command-line argument that is the name of the file to create.
- Example:

```
import java.io.*;
class BufferedOutputStreamDemo
{
 public static void main(String args[])
 {
      try
      {     FileOutputStream fos=new FileOutputStream(args[0]);
            BufferedOutputStream bos=new BufferedOutputStream (fos);
            //write 12 bytes to the file
            for(int i=0;i<12;i++)
            {
                  bos.write(i);
            }
            bos.close();
      }
```

```
        catch(Exception e)
        {
              System.out.println("Exception:"+e);
        }
    }
     }
```