As per U.G.C. Guidelines and also on the basis of revised syllabus of
Kavayitri Bahinabai Chaudhari North Maharashtra University
with effect from Academic Year 2023-24. Also useful for all Universities.

SEM IV ▪ BCA-403

Bachelor in Computer Application

JAVA PROGRAMMING

Text Book Development Committee

- C O O R D I N A T O R -

Dr. B. H. Barhate
Vice-Principal,
Bhusawal Arts, Commerce and P.O. Nahta Science College, Bhusawal.

- A U T H O R S -

Sanjay E. Pate
Department of Computer Science,
Nanasaheb Y.N.Chavan Arts, Science and Commerce College, Chalisgaon.

Prashant M. Savdekar
Head, Department of BCA,
DNCVPS Shirish Madhukrrao Chaudhari College, Jalgaon.

Prashant
Prashant Publications

JAVA PROGRAMMING

SEM IV ▪ BCA-403

Publisher | Printer
Prashant Publications
3, Pratap Nagar, Sant Dnyaneshwar Mandir Rd,
Near Nutan Maratha College, Jalgaon 425001.

Phone | Website | E-mail
0257-2235520, 0257-2232800, 9665626717
www.prashantpublications.com
prashantpublication.jal@gmail.com

Distributor
Prashant Book House
17, Stadium Shopping Centre, Opp. SBI, Jalgaon.
Ph. 0257-2227231

Available at all leading booksellers
Jalgaon | Dhule | Nadurbar

First Edition, January, 2024

ISBN 978-81-19120-69-7

Price ₹ 85/-

PREFACE

JAVA PROGRAMMING is a Simple version for S.Y.B.C.A. students, published by Prashant Publication.

This text is in accordance with the new syllabus CBCS-2023 recommended by the Kavayitri Bahainabai Chaudhari North Maharashtra University, Jalgaon, which has been serving the need of S.Y.B.C.A. Computer Science students from various colleges. This text is also useful for the student of Engineering, B.Sc. (Information Technology and Computer Science), M.Sc, M.C.A. B.B.M., M.B.M. other different Computer courses.

We are extremely grateful to Prof. Dr. S.R.Kolhe, Professor, Director, School of Computer Sciences and Chairman, Board of Studies, Kavayitri Bahinabai Chaudhari North Maharashtra University, Jalgaon for his valuable guidance.

We are obligated to Principals Dr.S.R.Jadhav, Nanasaheb Y. N. Chavan Arts, Science and Commerce College, Chalisgaon, Dr. P. R. Chaudhari, Shirish Madhukarrao Chaudhari Arts, Science and Commerce College, Jalgaon and Librarians and staff of respective colleges for their encouragement.

We are very much thankful to Shri. Rangrao Patil, Shri. Pradeep Patil of Prashant Publications, who has shown extreme co-operation during the preparation of this book, for getting the book published in time and providing an opportunity to be a part of this book.

We welcome any suggestions aimed at enhancing the content of this book, and we look forward to constructive feedback from our readers.

- Authors

S Y L L A B U S

Bachelor in Computer Applications (BCA)

Java Programming | Sem. IV | BCA-403

w.e.f. Academic Year 2023-24 (Semester System 60 + 40 Pattern)

Unit 1 : Introduction to JAVA **(10 L, 15 M)**
History and Features of Java, JDK, JRE, JIT, Data Types, Variables, Types of Comments, Operators, Control Structures and Loops, Compiling and running Java programs using command line and Editors, command line arguments Accepting Input from Console (Using Buffered Reader, Scanner Classes), Arrays.

Unit 2 : Objects and Classes **(10 L, 15 M)**
Introduction to classes and Objects, Defining Your Own Classes, Access Specifies, Data members and Methods, Constructors and its types, Overloading, Creating Packages, String functions, Date and Time functions.

Unit 3 : Inheritance and Interface in JAVA **(10 L, 15 M)**
Inheritance Basics, Function Overriding and Polymorphism, Use of super and this keywords, final keyword with respect to functions and classes, Interfaces, Abstract class and abstract method, Wrapper class.

Unit 4 : Exception handling in JAVA **(10 L, 15 M)**
Exception handling fundamentals, Types of Exceptions, Use of try-catch-finally, Creating user defined exceptions.

Unit 5 : User Interface using AWT and Swing **(10 L, 15 M)**
What is AWT, Swing, difference between AWT and Swing, Layout managers, Event Handling, Event sources, Listeners, Mouse and Keyboard event handling Components – JButton, JLabel, JText, JTextArea, JCheckBox, JRadioButton.

Unit 6 : Streams and Files in JAVA **(10 L, 15 M)**
Using the File Class, Stream classes, Byte Stream Class, Character Stream Class, and Create/Read/Write File.

CONTENTS

- Chapter - 1 7
  Introduction to JAVA
- Chapter - 2 29
  Objects and Classes
- Chapter - 3 44
  Inheritance and Interface in JAVA
- Chapter - 4 59
  Exception handling in JAVA
- Chapter - 5 65
  User Interface using AWT and Swing
- Chapter - 6 93
  Streams and Files in JAVA

1. Chapter

Introduction to JAVA

S y l l a b u s :

History and Features of Java, JDK, JRE, JIT, Data Types, Variables, Types of Comments, Operators, Control Structures and Loops, Compiling and running Java programs using command line and Editors, command line arguments Accepting Input from Console (Using Buffered Reader, Scanner Classes), Arrays.

(10 L, 15 M)

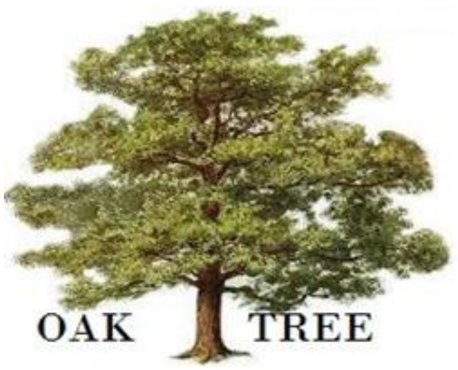
History of Java

Java history is interesting to know. The history of java starts from Green Team. Java team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc.

For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describes the history of java.

- **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc.
- Initially it was designed for small, embedded systems in electronic appliances like set-top boxes
- Firstly, it was called **"Green talk"** by James Gosling and file extension was .gt.
- After that, it was called **Oak** and was developed as a part of the Green project.



- **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.
- In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.
- Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

Why they chooses java name for java language?

- The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.
- According to James Gosling "Java was one of the top choices along with **Silk**". Since java was so unique, most of the team members preferred java.
- Java is an island of Indonesia where first coffee was produced (called java coffee).
- Notice that Java is just a name not an acronym.
- Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
- In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.
- JDK 1.0 released in (January 23, 1996).

What is Java?

- Java is a **programming language and a platform**.
- **Platform** -Any hardware or software environment in which a program runs, known as a platform. Since Java has its own Runtime Environment (JRE) and API, it is called platform.

Where it is used?

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

1. Desktop Applications such as acrobat reader, media player, antivirus etc.
2. Web Applications such as irctc.co.in, javatpoint.com etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games etc.

Types of Java Applications

There are mainly 4 type of applications that can be created using java:

1) **Standalone Application -** It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2) **Web Application -** An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

3) **Enterprise Application -** An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

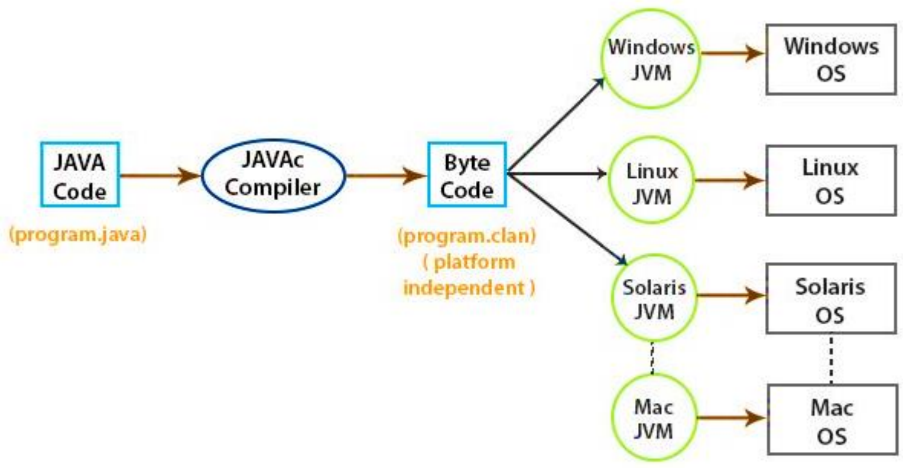
4) **Mobile Application -** An application that is created for mobile devices. Currently **Android** and Java ME are used for creating mobile applications.

Features of Java

1. **Platform Independent:** Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, or macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa.

   Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of bytecode. That is why we call java a platform-independent language.

Platform Independence in Java



2. **Object-oriented :** Java is an object-oriented programming language. Everything in Java is an object. Java is purely an object oriented language due to the absence of global scope, Everything in java is an object, all the program codes and data resides within classes and objects. It comes with an extensive set of classes, arranged in packages, object model in java in sample and easy to extend.

   Basic concepts of OOPs are:

   Object

   Class

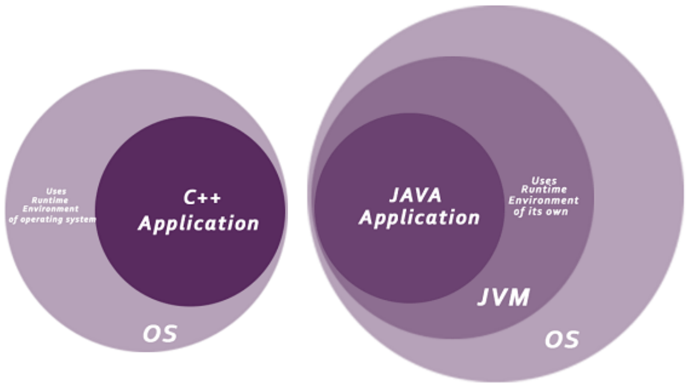
   Inheritance

   Polymorphism

   Abstraction

   Encapsulation

3. **Secured :** Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:
   - No explicit pointer
   - Java Programs run inside a virtual machine sandbox
   - **Class loader:** Class loader in Java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package **for** the classes of the local file system from those that are imported from network sources.
   - **Bytecode Verifier:** It **checks** the code fragments for illegal code that can violate access rights to objects.


Uses Runtime Environment of operating system
C++ Application
OS
JAVA Application
Uses Runtime Environment of its own
JVM
OS

- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

4. **Robust:** Java language is robust which means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

5. **Portable :** The WORA (Write Once Run Anywhere) concept and platform independent feature make Java portable. Now using the Java programming language, developers can yield the same result on any machine, by writing code only once. The reason behind this is JVM and bytecode.

   Suppose you wrote any code in Java, then that code is first converted to equivalent bytecode which is only readable by JVM. We have different versions of JVM for different platforms.

   Windows machines have their own version of JVM, linux has its own and macOS has its own version of JVM. So if you distribute your bytecode to any machine, the JVM of that machine would translate the bytecode into respective machine code.

6. **Distributed :** In Java, we can split a program into many parts and store these parts on different computers. A Java programmer sitting on a machine can access another program running on the other machine.

   This feature in Java gives the advantage of distributed programming, which is very helpful when we develop large projects. Java helps us to achieve this by providing the concept of RMI (Remote Method Invocation) and EJB (Enterprise JavaBeans).

Java comes with an extensive library of classes for interacting, using TCP/IP protocols such as HTTP and FTP, which makes creating network connections much easier than in C/C++.

7. **Multithreaded:** A thread is an independent path of execution within a program, executing concurrently. Multithreaded means handling multiple tasks simultaneously or executing multiple portions (functions) of the same program in parallel.

   The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

8. **Simple:** Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun Microsystem, Java language is a simple programming language because:
   - Java syntax is based on C++ (so easier for programmers to learn it after C++).
   - Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
   - There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

9. **Architecture-neutral:** Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

   In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

10. **High Performance:** The performance of Java is impressive for an interpreted language because of its intermediate bytecode.

    Java provides high performance with the use of **"JIT – Just In Time compiler"**, in which the compiler compiles the code on-demand basis, that is, it compiles only that method which is being called. This saves time and makes it more efficient.

    Java architecture is also designed in such a way that it reduces overheads during runtime. The inclusion of multithreading enhances the overall execution speed of Java programs.

11. **Dynamic and Extensible:** Java is dynamic and extensible means with the help of OOPs, we can add classes and add new methods to classes, creating new classes through subclasses. This makes it easier for us to **expand** our own classes and even **modify** them.

12. **Multithreading in Java:** Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part

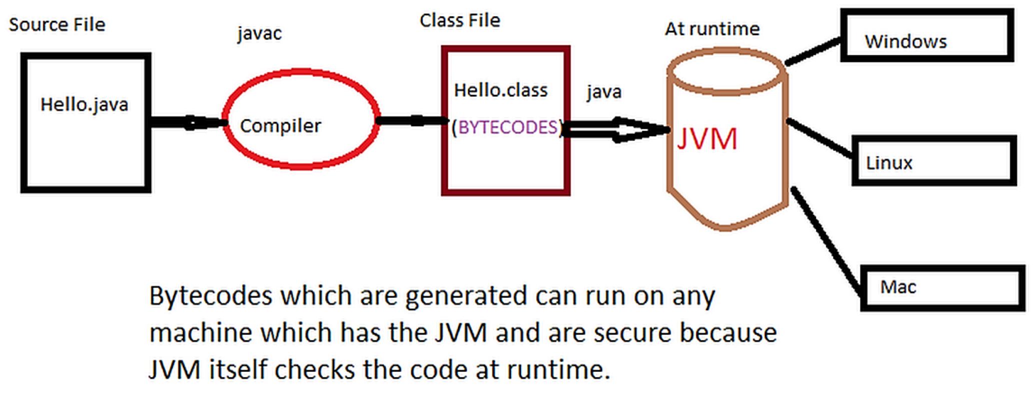
of such program is called a thread. So, threads are light-weight processes within a process.

Multithreading saves time as you can perform multiple operations together.

The threads are independent, so it does not block the user to perform multiple operations at the same time and also, if an exception occurs in a single thread, it does not affect other threads.

JVM

- Actually, JVM is an interpreter for Byte code.
- The details of the JVM will differ from platform to platform, but all interpret the same Java Byte code according to machine/platform.
- The Byte code which are generated by the compiler will be tested by the JVM on the execution of the program or we can say every Java Program is under the control of the JVM which checks the code on the runtime many times for viruses and any malicious.
- The Byte code generated by the compiler are also supported on any machine which has the JVM which makes Java a platform independent language.


Source File
Hello.java
javac
Compiler
Class File
Hello.class
(BYTECODES)
java
At runtime
JVM
Windows
Linux
Mac
Bytecodes which are generated can run on any machine which has the JVM and are secure because JVM itself checks the code at runtime.

The JVM performs following operation:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

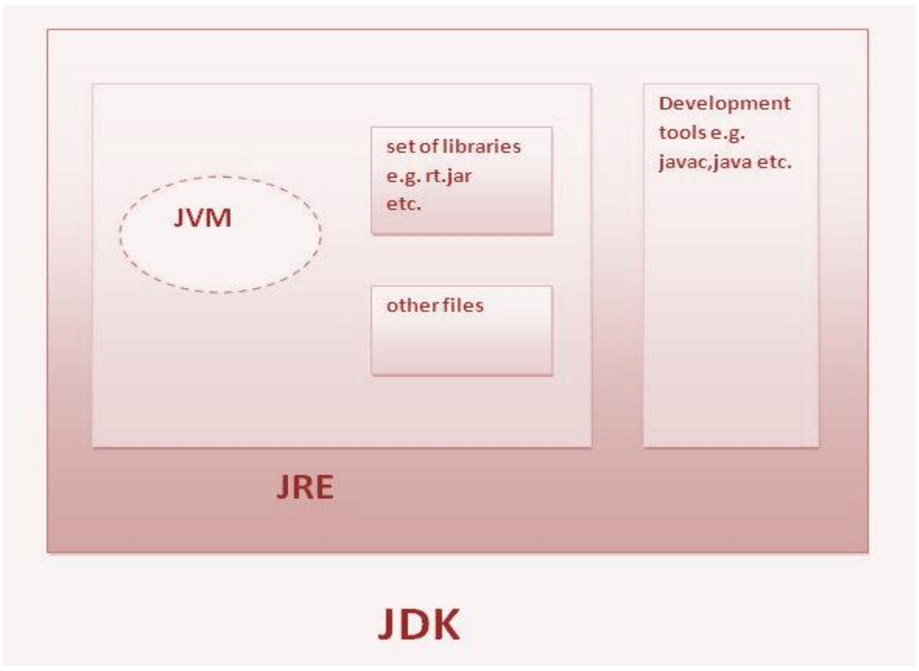
JRE is an acronym for Java Runtime Environment. It is used to provide runtime environment.

It is the implementation of JVM. It physically exists. It contains set of libraries + other files that JVM uses at runtime.

Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.

JDK

JDK is an acronym for Java Development Kit. It physically exists. It contains JRE + development tools.


JVM
set of libraries e.g. rt.jar etc.
other files
Development tools e.g. javac,java etc.
JRE
JDK

Variable and Data type in Java

There are three types of variables: local, instance and static.

There are two types of data types in java, primitive and non-primitive.

Variable

Variables in Java are fundamental elements that act as containers to store data values. They serve as named storage locations in your program, where each variable is associated with a specific **data type** and a value. The variable's name is a unique identifier used to refer to and manipulate its stored value within the code.

```
int data=50;        //Here data is variable
```

Types of Variable

There are three types of variables in java

- Local variable
- Instance variable
- Static variable

Local Variable

A variable that is declared inside the method is called local variable.

Instance Variable

A variable that is declared inside the class but outside the method is called instance variable. It is not declared as static.

Static variable

A variable that is declared as static is called static variable. It cannot be local.

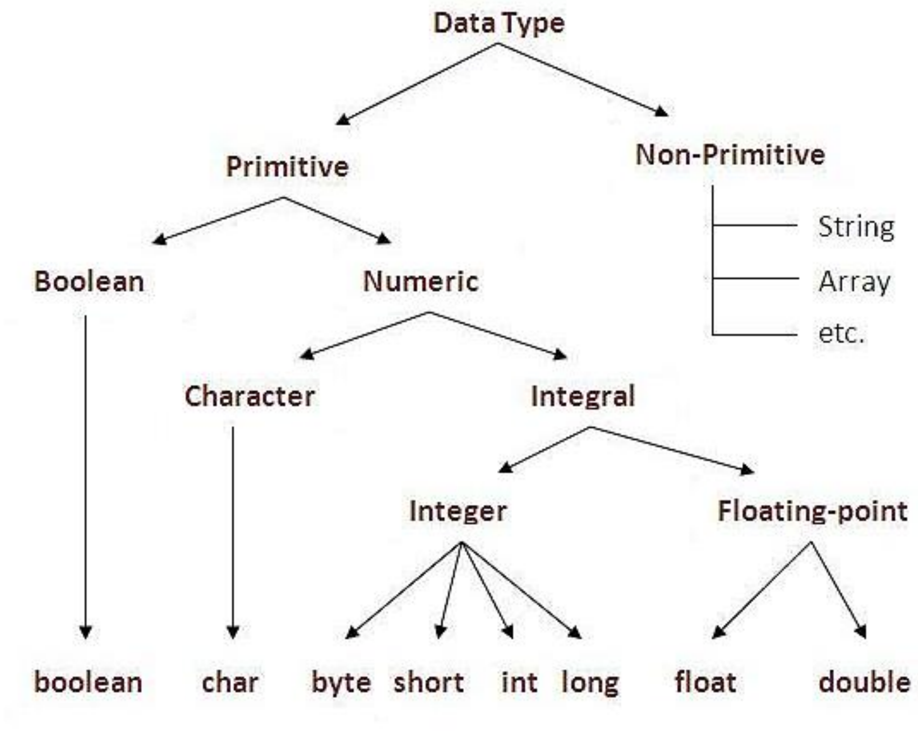
Example to understand the types of variables

```
class A{
            int data=50;            //instance variable
            static int m=100;       //static variable
            void method(){
                  int n=90;         //local variable
            }
      }//end of class
```

Data Types in Java

In java, there are two types of data types

- primitive data types
- non-primitive data types



Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. Let us now look into detail about the eight primitive data types.

Byte:

- Byte data type is an 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100 , byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int.
- Default value is 0.
- Example: short s = 10000, short r = -20000

int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: long a = 100000L, int b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values, generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letter A ='A'

Rules of variables:

- The name of a variable can start with a letter, an underscore "_", or the dollar sign $. The name cannot start with a digit. If the name starts with an underscore, the second character must be an alphabetical letter
- After the first character, the name of the variable can include letters, digits (0, 1, 2, 3, 4, 5, 6, 7, 8, or 9), or underscores in any combination
- The name of a variable cannot be one of the words that the Java languages has reserved for its own use. A reserved word is also called a keyword. This means that you cannot use one of the following keywords to name your variable:

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

Non-Primitive Data Types

- Non-primitive data types are called **reference types** because they refer to objects.
- The main difference between **primitive** and **non-primitive** data types are:
- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be null.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc. You will learn more about these in a later chapter.

Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

1. Arithmetic Operators
2. Relational Operators
3. Bitwise Operators
4. Logical Operators
5. Assignment Operators
6. Conditional Operator (? :)
7. Unary Operator

1. The Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

2. The Relational Operators:

There are following relational operators supported by Java language

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

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

3. The Bitwise Operators:

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

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

```
a = 0011 1100
b = 0000 1101
-----------------
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a  = 1100 0011
```

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

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

4. The Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

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

5. The Assignment Operators:

There are following assignment operators supported by Java language:

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

6. Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

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

Following is the example:

This would produce the following result:

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

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

7. Unary Operator

Increment operator is used to increment a value by 1. There are two varieties of increment operator:

- **Post-Increment:** Value is first used for computing the result and then incremented.
- **Pre-Increment:** Value is incremented first and then the result is computed.

Decrement operator is used for decrementing the value by 1. There are two varieties of decrement operators.

- **Post-decrement:** Value is first used for computing the result and then decremented.
- **Pre-decrement:** Value is decremented first and then the result is computed.

First, let’s look at a code snippet using the pre-increment unary operator:

```
int operand = 1;
++operand;                          // operand = 2
int number = ++operand;             // operand = 3, number = 3
```

Next, let’s have a look at the code snippet using the pre-decrement one:

```
int operand = 2;
--operand;                              // operand = 1
int number = --operand;                 // operand = 0, number = 0
```

Let’s look at a sample code snippet using the post-increment operator:

```
int operand = 1;
operand++;                              // operand = 2
int number = operand++;                 // operand = 3, number = 2
```

Also, let’s have a look at the post-decrement one:

```
int operand = 2;
operand--;                              //operand = 1
int number = operand--;                 // operand = 0, number 1
```

Control Structures and Loops

Control structures are programming blocks that can change the path we take through those instructions.

There are three kinds of control structures:

- Conditional Branches, which we use for choosing between two or more paths. There are three types in Java: *if/else/else if*, *ternary operator* and *switch*.
- Loops that are used to iterate through multiple values/objects and repeatedly run specific code blocks. The basic loop types in Java are *for*, *while* and *do while*.
- Branching Statements, which are used to alter the flow of control in loops. There are two types in Java: *break* and *continue*.

Creating hello java example

```
import java.io.*;                                       // Importing classes from packages
public class Hello {                                    // Main class
     public static void main(String[] args)             // Main driver method
     {
          System.out.println ("Welcome to Java Hello");  // Print statement
     }
}
```

Save this file as Simple.java

```
To compile:    javac Hello.java
To execute:    java Hello
```

/* Comments */	The compiler ignores comment block. Comment can be used anywhere in the program to add info about the program or code block, which will be helpful for developers to understand the existing code in the future easily.

import java.io.*	This means all the classes of io package can be imported. Java io package provides a set of input and output streams for reading and writing data to files or other input or output sources.
public class Hello	• This creates a class called Hello. • All class names must start with a capital letter. • The public word means that it is accessible from any other classes.
Braces	Two curly brackets {...} are used to group all the commands, so it is known that the commands belong to that class or method.
public static void main	• When the main method is declared public, it means that it can also be used by code outside of its class, due to which the main method is declared public. • The word static used when we want to access a method without creating its object, as we call the main method, before creating any class objects. • The word void indicates that a method does not return a value. main() is declared as void because it does not return a value. • main is a method; this is a starting point of a Java program.
String[] args	It is an array where each element of it is a string, which has been named as "args". If your Java program is run through the console, you can pass the input parameter, and main() method takes it as input.
System.out.println();	This statement is used to print text on the screen as output, where the system is a predefined class, and out is an object of the PrintWriter class defined in the system. The method println prints the text on the screen with a new line. You can also use print() method instead of println() method. All Java statement ends with a semicolon.

Program 2:

```
public class SumOfNumbers1
    {
            public static void main(String args[])
            {
                    int n1 = 225, n2 = 115, sum;
                    sum = n1 + n2;
                    System.out.println("The sum of numbers is: "+sum);
            }
    }
```

Java Command Line Arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

To run this java program, you must pass at least one argument from the command prompt.

```
class Command Line Example{
        public static void main(String args[]){
             System.out.println("Your first argument is: "+args[0]);
             }
}
         compile by > javac CommandLineExample.java
         run by > java CommandLineExample hello
```

Output: Your first argument is: hello

Ways to read input from console in Java

1. **Using Buffered Reader Class**

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Test {
        public static void main(String[] args) throws IOException
        {
    // Enter data using BufferReader
    BufferedReader reader=new BufferedReader (new InputStreamReader (System.in));

     // Reading data using readLine
     String name = reader.readLine ();

     // Printing the read line
     System.out.println (name);
    }
}
```

Input:

```
Hello Java
```

Output:

```
Hello Java
```

2. Using Scanner Class

The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however, it is also can be used to read input from the user in the command line.

```
import java.util.Scanner;
class GetInputFromUser {
        public static void main(String args[])
        {
                // Using Scanner for Getting Input from User
                Scanner in = new Scanner(System.in);
                String s = in.nextLine();
                System.out.println("You entered string " + s);
                int a = in.nextInt();
                System.out.println("You entered integer " + a);
                float b = in.nextFloat();
                System.out.println("You entered float " + b);
        }
}
```

Input:

```
Hello Java
12
3.4
```

Output:

```
You entered string Hello Java
You entered integer 12
You entered float 3.4
```

Java Arrays

An array is a collection of similar types of data.

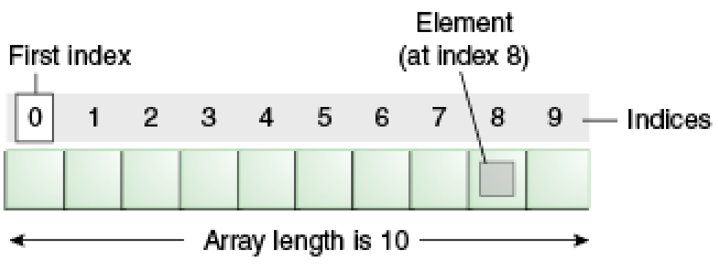
Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the size of operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Clone able interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



In Java, here is how we can declare an array.

dataType[] arrayName;

dataType - it can be primitive data types like int, char, double, byte, etc. or Java objects

arrayName - it is an identifier

```
// declare an array
double[] data;                  // allocate memory
data = new double[10];
OR
double[] data = new double[10];
```

Here, the array can store **10** elements. We can also say that the **size or length** of the array is 10.

```
//declare and initialize and array
int[] age = {12, 4, 5, 2, 5};
//Java Program to illustrate how to declare, instantiate, initialize and traverse the Java array.

class Testarray{
public static void main(String args[]){
            int a[]=new int[5];//declaration and instantiation
            a[0]=10;//initialization
            a[1]=20;
            a[2]=70;
            a[3]=40;
            a[4]=50;
            //traversing array
            for(int i=0;i<a.length;i++)//length is the property of array
            System.out.println(a[i]);
      }
}
```

Output:

```
            10
            20
            70
            40
            50
```

❒❒❒

2. Chapter

Objects and Classes

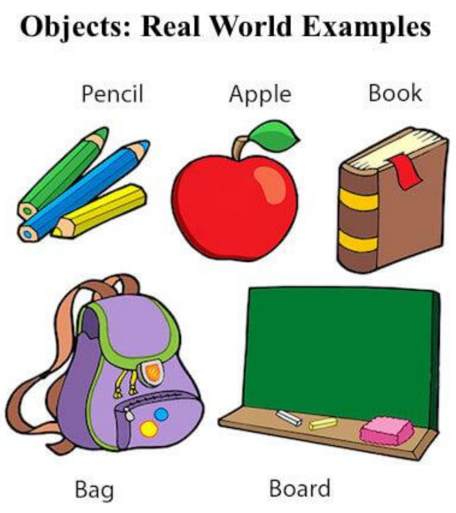
S y l l a b u s ::

Introduction to classes and Objects, Defining Your Own Classes, Access Specifies, Data members and Methods, Constructors and its types, Overloading, Creating Packages, String functions, Date and Time functions.

(10 L, 15 M)

Introduction to classes and Objects

What is an object in Java



An entity that has state and behaviour is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.

- **Behaviour:** represents the behaviour (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

What is a class in Java

A class is a group of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax:

```
access_modifier class<class_name>
{
   data member;
   method;
   constructor;
   nested class;
   interface;
}
```

Example of the class program in Java

```
class Student {
       int id; // data member (also instance variable)
       String name; // data member (also instance variable)
       public static void main(String args[])
      {
                 Student s1 = new Student(); // creating an object of
                              // Student
                 System.out.println(s1.id);
                 System.out.println(s1.name);
      }
}
```

Access Specifies

In Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element.

When a class or method or variable does not have an access specifier associated with it, we assume it is having default access.

There are four types of access modifiers available in Java:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Types of Data Members

We know that a class is a collection of data members and methods. In java programming we have two types of data members they are

1. Instance/non-static data members
2. Static data members

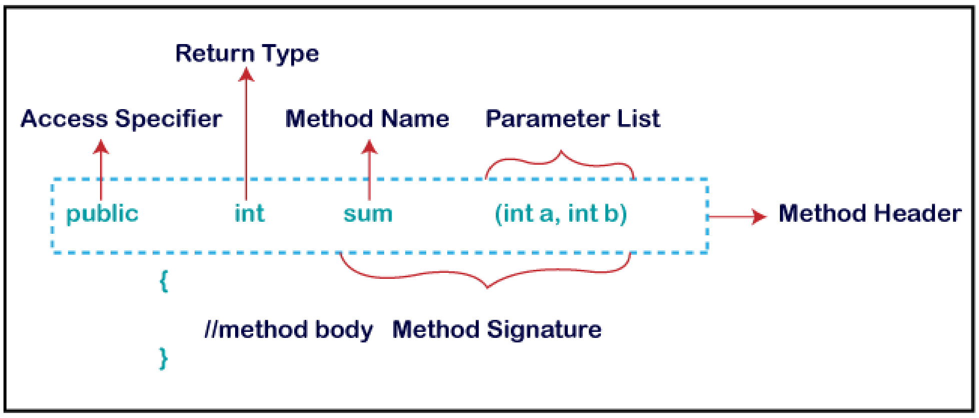
Instance Data members	**Static Data members**
1. Instance data members are those whose memory space is created each and every time whenever an object is created.	1. Static data members are those whose memory space is created only once, whenever the class is loaded in the main memory irrespective of no of objects are created.
2. Instance data members are always meant for storing specific values.	2. Static data members are meant for storing common values.
3. Programmatically instance data members declaration should not be preceded by a keyword static.	3. Programmatically static data member declaration must be preceded by **static** keyword.
Syntax:-Data type v1, v2, v3 ………. Vn;	**Syntax: -** static data type v1, v2, v3 …. Vn;

4. Each and every instance data member must be access with respective **object name**.	4. Each and every static data member must be access with respective **class name**.
Ex: Objname.instance data member name	**Ex:-** Class name.static data member name
5. Instance data members are also known as **Object Level Data Members** because they depend on object name and independent from class name.	5. Static data member are also known as **Class Level Data Members** because they depends on class name and independent from object name
Ex: - int pin; int stno; string sname;	**Ex: -** static String cname; static String Cap India;

Types of Methods

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again.

Method Declaration



In java programming we have two types of methods they are

1. Instance/ non –static methods
2. Static methods

Instance methods: -

1. Instance methods are always recommended to perform **repeated operations** like reading records from the file, reading records from the data base etc…
2. Programmatically instance methods definitions should not be preceded by a keyword static.

Syntax:-

Return type method name (list of parameters if any)

```
{
        Block of statements;
}
```

3. Each and every instance method must be access with respective **Object name.**
4. Result of instance method is **not sharable**

Ex: -

```
void getTotal()
{
        Tot = m1 +  m2 + m3;
}
```

Static methods:-

1. Static methods are always which are recommended to perform **one operation** like opening the files, obtaining a data base connection etc…
2. Programmatically static methods definitions must be preceded by **static** keyword.

   Syntax: -

```
 Static Return type method name (list of parameters if any)
{
        Block of statements;
}
```

3. Each and every static method must be accessed with respect **class name.**
4. Result set of static method always **sharable**

   Ex: -

```
Public static void main ()
{
        S1.getTotal();
        S2.getTotal();
}
```

Java Constructors

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Rules for creating Java constructor

1. Constructor name must be the same as its class name.
2. A Java constructor cannot be abstract, static, final, and synchronized.
3. Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
4. Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

Types of Java constructors

There are two types of constructors in Java:

Default constructor (no-arg constructor)

Parameterized constructor

1. **Default constructor (no-arg constructor)**

   A constructor is called "Default Constructor" when it doesn't have any parameter.

   Program:

```
        class Main {
        private String name;
         // constructor
        Main() {
        System.out.println("Constructor Called:");
        name = "Programiz";
        }
public static void main(String[] args) {
         // constructor is invoked while
        // creating an object of the Main class
        Main obj = new Main();
        System.out.println("The name is " + obj.name);
        }
}
```

   Output:

```
Constructor Called:
The name is Programiz
```

   In the above example, we have created a constructor named Main(). Inside the constructor, we are initializing the value of the name variable.

   Notice the statement of creating an object of the Main class.

```
        Main obj = new Main();
```

   Here, when the object is created, the Main() constructor is called. And, the value of the name variable is initialized.

Hence, the program prints the value of the name variables as Programiz.

2. **Parameterized Constructor**

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

```
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,"Suresh");
        Student s2 = new Student(222,"Ramesh");
         //calling method to display the values of object
            s1.display();
            s2.display();
        }
}
```

Output:

```
111 Suresh
222 Ramesh
```

Constructor Overloading in Java

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

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.

Example of Constructor Overloading

```
//Java program to overload constructors
class Student{
            int id;
            String name;
            int age;
            //creating two arg constructor
            Student(int i,String n){
                    id = i;
                    name = n;
            }
            //creating three arg 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,"Kiran");
                    Student s2 = new Student(222,"Aryan",25);
                    s1.display();
                    s2.display();
            }
      }
```

Output:

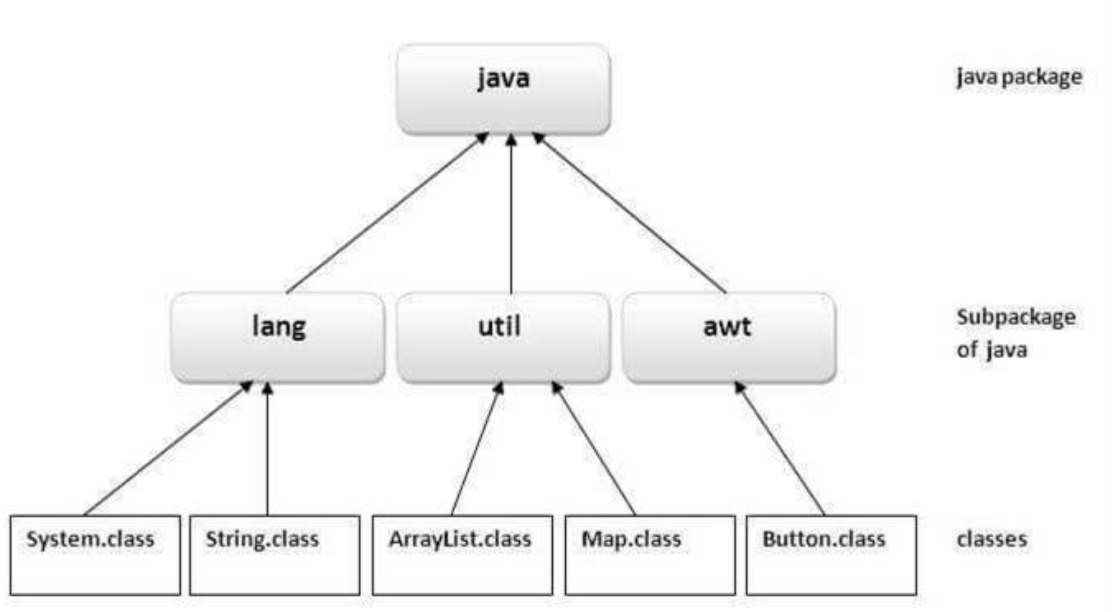
```
111 Kiran 0
222 Aryan 25
```

Packages

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

Advantage of Java Package

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



Simple example of java package

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

```
//save as Simple.java
package mypack;
public class Simple{
        public static void main(String args[]){
                System.out.println("Welcome to package");
        }
}
```

How to compile java package?

If you are not using any IDE, you need to follow the **syntax** given below:

```
javac -d directory javafilename
```

For example

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

```
To Compile: javac –d . Simple.java
To Run: java mypack.Simple
```

```
Output: Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

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

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

1) Using packagename.*

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

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

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.*;
class B{
public static void main(String args[]){
```

```
A obj = new A();
obj.msg();
}
}
```

Output: Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java
package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
import pack.A;
class B{
        public static void main(String args[]){
A obj = new A();
obj.msg();
        }
}
```

Output: Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
```

```
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Output: Hello

What is String in Java?

String is a sequence of characters. But in Java, string is an object that represents a sequence of characters.

The java.lang.String class is used to create a string object.

An array of characters works same as Java string. For example:

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

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class

implements *Serializable*, *Comparable* and *CharSequence* interfaces.

How to create a string object?

There are two ways to create String object:

1. By string literal
2. By new keyword

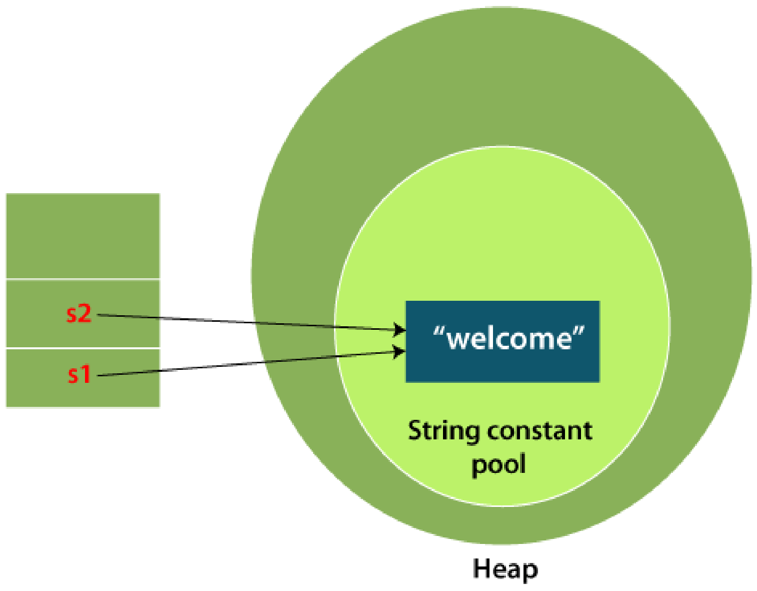
1) String Literal

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

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

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

```
String s1="Welcome";
String s2="Welcome";//It doesn't create a new instance
```



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Why Java uses the concept of String literal?

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2) By new keyword

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

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

StringExample.java

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

Output:

```
java
strings
example
```

The above code, converts a ***char*** array into a **String** object. And displays the String objects ***s1, s2***, and ***s3*** on console using ***println()*** method.

Java Date and Time

Java provides the Date class available in java.util package, this class encapsulates the current date and time.

The Date class supports two constructors as shown in the following table.

Sr. No.	Constructor & Description
1	**Date( )** This constructor initializes the object with the current date and time.
2	**Date(long millisec)** This constructor accepts an argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970.

Getting Current Date and Time

This is a very easy method to get current date and time in Java. You can use a simple Date object with toString() method to print the current date and time as follows −

Example

```
import java.util.Date;
public class DateDemo {
        public static void main(String args[]) {
                Date date = new Date();                 // Instantiate a Date object
                System.out.println(date.toString());    // display time and date using toString()
      }
}
```

Output

```
      on May 04 09:51:52 CDT 2009
```

❒❒❒

3. Chapter

Inheritance and Interface in JAVA

S y l l a b u s :

Inheritance Basics, Function Overriding and Polymorphism, Use of super and this keywords, final keyword with respect to functions and classes, Interfaces, Abstract class and abstract method, Wrapper class.

(10 L, 15 M)

Inheritance Basics

Inheritance in Java is a concept that acquires the properties from one class to other classes; for example, the relationship between father and son. Inheritance in Java is a process of acquiring all the behaviours of a parent object.

- **subclass** (child) - the class that inherits from another class
- **superclass** (parent) - the class being inherited from

Why Do We Need Java Inheritance?

- **Code Reusability:** The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- **Method Overriding: Method Overriding** is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
- **Abstraction:** The concept of abstract where we do not have to provide all details is achieved through inheritance. **Abstraction** only shows the functionality to the user.

How to Use Inheritance in Java?

The extends keyword is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, “extends” refers to increased functionality.

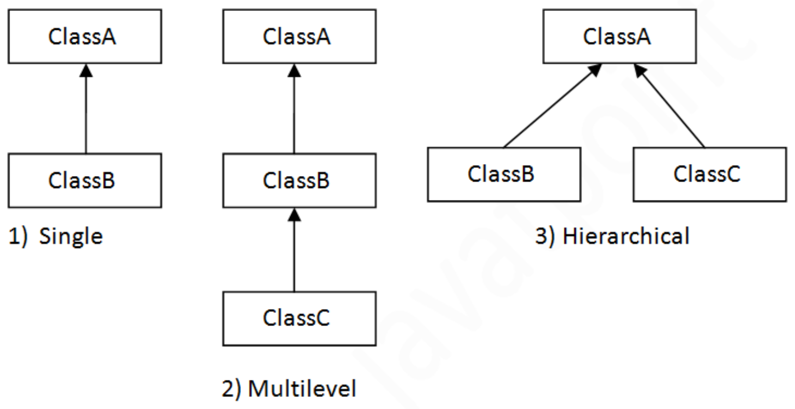
Syntax :

```
class derived-class extends base-class
{
        //methods and fields
}
```

Types of inheritance in java

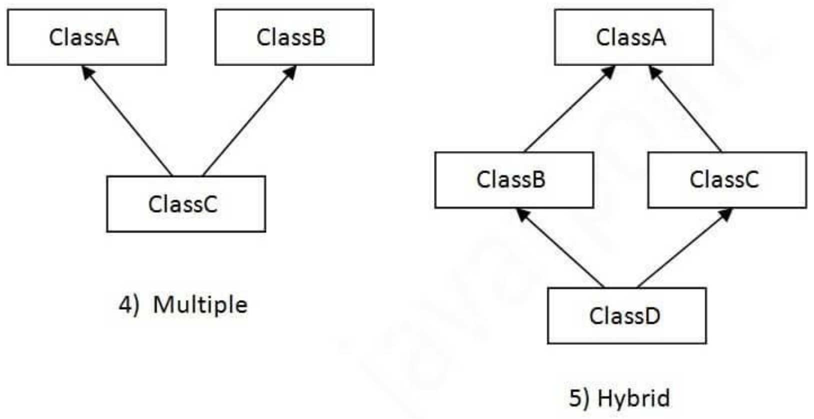
On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



Single Inheritance Example

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

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

Output:

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

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as multilevel inheritance. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

File: TestInheritance2.java

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

Output:

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

Hierarchical Inheritance

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.

File: TestInheritance3.java

```
    class Animal{
                void eat(){System.out.println("eating...");}
    }
    class Dog extends Animal{
                void bark() {System.out.println("barking...");}
    }
    class Cat extends Animal{
                void meow(){System.out.println("meowing...");}
    }
    class TestInheritance3{
        public static void main(String args[]){
                Cat c=new Cat();
                c.meow();
                c.eat();
                //c.bark();//C.T.Error
        }
    }
    Output:
        meowing...
        eating...
```

Q) Why multiple inheritance is not supported in java?

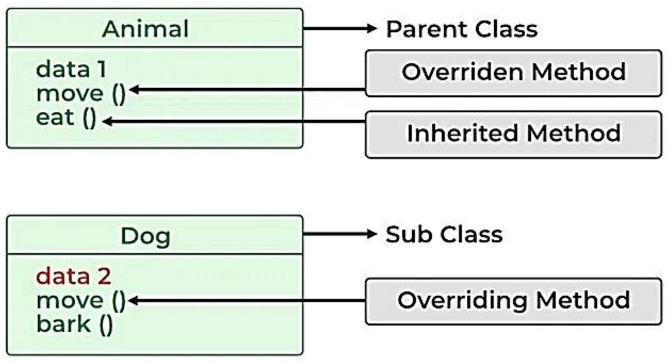
To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

Overriding in Java

In Java, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, the same parameters or signature, and the same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.



Method overriding is one of the ways by which Java achieves **Run Time Polymorphism.** The version of a method that is executed will be determined by the object that is used to invoke it.

Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

```
// Java program to demonstrate method overriding in java
class Parent {                                          // Base Class
     void show()
    {
            System.out.println("Parent's show()");
    }
    }
class Child extends Parent {                          // // Driver class
// This method overrides show() of Parent
    void show()                                                   // @Override
  {
    System.out.println("Child's show()");
  }
}
class Main {
  public static void main(String[] args)
  {
      // If a Parent type reference refers to a Parent object, then Parent's show is called
      Parent obj1 = new Parent();
      obj1.show();
       // If a Parent type reference refers to a Child object Child's show()is called. This is
      called RUN TIME POLYMORPHISM.
      Parent obj2 = new Child();
      obj2.show();
  }
}
```

Output

```
Parent's show()
Child's show()
```

Super Keyword in Java Overriding

A common question that arises while performing overriding in Java is:

Can we access the method of the superclass after overriding?

Well, the answer is Yes. To access the method of the superclass from the subclass, we use the super keyword.

```
class Animal {
        public void displayInfo() {
        System.out.println("I am an animal.");
        }
}
  class Dog extends Animal {
  public void displayInfo() {
    super.displayInfo();
    System.out.println("I am a dog.");
  }
}
class Main {
   public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.displayInfo();
  }
}
```

Output:

```
I am an animal.
I am a dog.
```

Explanation:

In the above example, the subclass Dog overrides the method displayInfo() of the superclass Animal.

When we call the method displayInfo() using the d1 object of the Dog subclass, the method inside the Dog subclass is called; the method inside the superclass is not called.

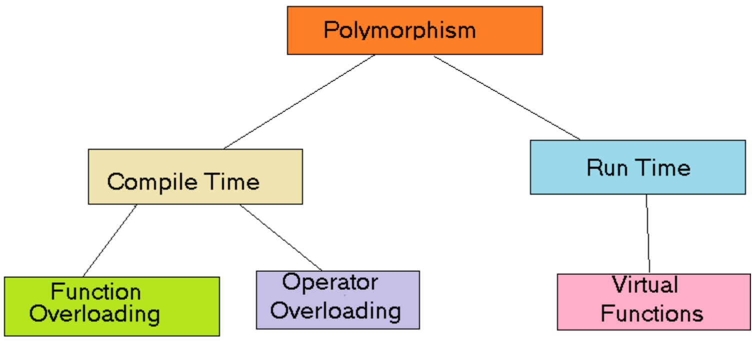
Inside displayInfo() of the Dog subclass, we have used super.displayInfo() to call displayInfo() of the superclass.

Polymorphism in Java

The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

Polymorphism explores how to create and use two methods with the same name to execute two different functionalities–like adding two functions with the same name but that accept different parameters.

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



Compile-Time Polymorphism in Java

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

Method Overloading

When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by changes in the number of arguments or/and a change in the type of arguments.

Example 1:

```
// Java Program for Method overloading By using Different Types of Arguments
class Helper {
        static int Multiply(int a, int b)
{
        return a * b;
}
static double Multiply(double a, double b)
 {
        return a * b;
  }
}
class GFG {
  public static void main(String[] args)
  {
            System.out.println(Helper.Multiply(2, 4));
            System.out.println(Helper.Multiply(5.5, 6.3));
   }
}
```

Output

```
            8
            34.65
```

Runtime Polymorphism in Java

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. **Method overriding**, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// Java Program for Method Overriding
class Parent {
            void Print()
      {
            System.out.println("parent class");
      }
}
class subclass1 extends Parent {
      void Print() { System.out.println("subclass1"); }
}
class subclass2 extends Parent {
       void Print()
   {
            System.out.println("subclass2");
    }
}
class GFG {
   public static void main(String[] args)
  {
            Parent a;
            a = new subclass1();
            a.Print();
            a = new subclass2();
            a.Print();
        }
    }
```

Output

```
            subclass1
            subclass2
```

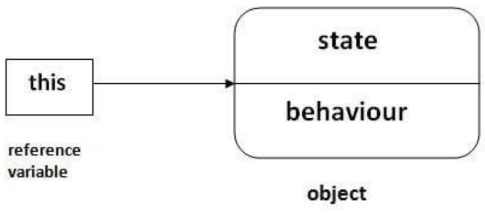
This keywords in java

The this keyword refers to the current object in a method or constructor.

The most common use of the this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).

This can also be used to:

- Invoke current class constructor
- Invoke current class method
- Return the current class object
- Pass an argument in the method call
- Pass an argument in the constructor call



```
public class Main {
  int x;
  public Main(int x) {                    // Constructor with a parameter
            this.x = x;
     }
   public static void main(String[] args) {
            Main myObj = new Main(5);
            System.out.println("Value of x = " + myObj.x);
    }
}
```

Output:

```
Value of x = 5
```

Note:

1. *If you omit the keyword in the example above, the output would be "0" instead of "5".*
2. *In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.*
3. *If local variables(formal arguments) and instance variables are different, there is no need to use this keyword.*

Interfaces in Java

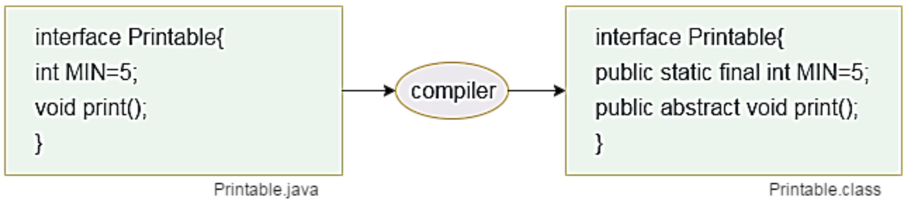
An **Interface in Java** programming language is defined as an abstract type used to specify the behavior of a class. An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods.

The interface in Java is *a* mechanism to achieve **abstraction**. There can be only abstract methods in the Java interface, not the method body. It is used to achieve abstraction and **multiple inheritances in Java using Interface.**

Syntax:

```
interface <interface_name>{
                 // declare constant fields
                // declare methods that abstract
                // by default.
}
```

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.

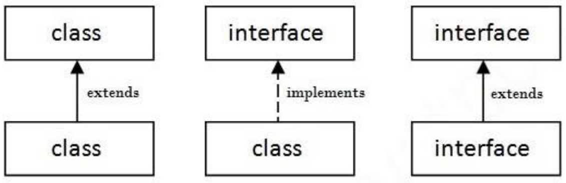


```
interface Printable{
int MIN=5;
void print();
}
```

```
interface Printable{
public static final int MIN=5;
public abstract void print();
}
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



```
interface printable{
                    void print();
                }
class A6 implements printable{
        public void print(){
                    System.out.println("Hello");
                    }
         public static void main(String args[]){
                    A6 obj = new A6();
                    obj.print();
          }
}
```

Output:

```
          Hello
```

Abstract Classes and Methods

- Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
  Abstraction can be achieved with either **abstract classes** or **interfaces** (which you will learn more about in the next chapter).
- The **abstract** keyword is a non-access modifier, used for classes and methods:
- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

```
// Abstract class
abstract class Animal {
public abstract void animalSound();          // Abstract method (does not have a body)
public void sleep() {                        // Regular method
          System.out.println("Zzz");
       }
}
```

```
class Pig extends Animal {                    // Subclass (inherit from Animal)
public void animalSound() {
        // The body of animalSound() is provided here
       System.out.println("The pig says: wee wee");
       }
 }
 class Main {
  public static void main(String[] args) {
       Pig myPig = new Pig(); // Create a Pig object
       myPig.animalSound();
       myPig.sleep();
       }
}
```

Output:

```
The pig says: wee wee
Zzz
```

Example of Abstract class that has an abstract method

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
          abstract void run();
    }
class Honda4 extends Bike{
void run(){System.out.println("running safely");}
public static void main(String args[]){
          Bike obj = new Honda4();
          obj.run();
    }
}
```

Output:

```
running safely
```

Wrapper classes in Java

The **wrapper class in Java** provides the mechanism to convert primitive into object and object into primitive.

Use of Wrapper classes in Java

Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

1. **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
2. **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
3. **Synchronization:** Java synchronization works with objects in Multithreading.
4. **java.util package:** The java.util package provides the utility classes to deal with objects.
5. **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Primitive Type	**Wrapper class**
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects  Autoboxing example of int to Integer
public class WrapperExample1{
    public static void main(String args[]){
            //Converting int into Integer
            int a=20;
            Integer i=Integer.valueOf(a);      //converting int into Integer explicitly
            Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
            System.out.println(a+" "+i+" "+j);
    }
}
```

Output:

```
    20 20 20
```

❒❒❒

4. Chapter

Exception handling in JAVA

S y l l a b u s :

Exception handling fundamentals, Types of Exceptions, Use of try-catch-finally, Creating user defined exceptions..

(10 L, 15 M)

Exception handling fundamentals

Exception Handling is a mechanism to handle exception at runtime. Exception is a condition that occurs during program execution and lead to program termination abnormally. There can be several reasons that can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc.

Suppose we run a program to read data from a file and if the file is not available then the program will stop execution and terminate the program by reporting the exception message.

The problem with the exception is, it terminates the program and skip rest of the execution that means if a program have 100 lines of code and at line 10 an exception occur then program will terminate immediately by skipping execution of rest 90 lines of code.

To handle this problem, we use exception handling that avoid program termination and continue the execution by skipping exception code.

Java exception handling provides a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Lets understand exception with an example. When we don't handle the exceptions, it leads to unexpected program termination. In this program, an ArithmeticException will be throw due to divide by zero.

```
class UncaughtException
{
 public static void main(String args[])
 {
  int a = 0;
  int b = 7/a;     // Divide by zero, will lead to exception
 }
}
```

This will lead to an exception at runtime, hence the Java run-time system will construct an exception and then throw it. As we don't have any mechanism for handling exception in the above program, hence the default handler (JVM) will handle the exception and will print the details of the exception on the terminal.

A Java Exception is an object that describes the exception that occurs in a program. When an exceptional events occurs in java, an exception is said to be thrown. The code that's responsible for doing something about the exception is called an exception handler.

Types of Exceptions

In Java, exceptions broadly can be categories into checked exception, unchecked exception and error based on the nature of exception.

Checked Exception :

The exception that can be predicted by the JVM at the compile time for example : File that need to be opened is not found, SQLException etc. These type of exceptions must be checked at compile time.

Unchecked Exception :

Unchecked exceptions are the class that extends RuntimeException class. Unchecked exception are ignored at compile time and checked at runtime. For example : ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.

Error :

Errors are typically ignored in code because you can rarely do anything about an error. For example, if stack overflow occurs, an error will arise. This type of error cannot be handled in the code.

Java provides controls to handle exception in the program. These controls are listed below.

Try : It is used to enclose the suspected code.

Catch: It acts as exception handler.

Finally: It is used to execute necessary code.

Throw: It throws the exception explicitly.

Throws: It informs for the possible exception

Try and Catch in java

Try and catch both are Java keywords and used for exception handling. The try block is used to enclose the suspected code. Suspected code is a code that may raise an exception during program execution.

For example, if a code raise arithmetic exception due to divide by zero then we can wrap that code into the try block.

```
try{
  int a = 10;
  int b = 0
  int c = a/b; // exception
}
```

The catch block also known as handler is used to handle the exception. It handles the exception thrown by the code enclosed into the try block. Try block must provide a catch handler or a finally block.

The catch block must be used after the try block only. We can also use multiple catch block with a single try block.

```
try{
  int a = 10;
  int b = 0
  int c = a/b; // exception
}catch(ArithmeticException e){
  System.out.println(e);
}
```

To declare try catch block, a general syntax is given below

```
try
{

}
catch(ExceptionClass ec)
{
}.
```

Exception handling is done by transferring the execution of a program to an appropriate exception handler (catch block) when exception occurs.

Creating and Catching Exception

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java. All the exception types are subclasses of the built in class Throwable.

Arithmetic Exception

It is thrown when an exceptional condition has occurred in an arithmetic operation.

ArrayIndexOutOfBoundsException

It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

ClassNotFoundException

This Exception is raised when we try to access a class whose definition is not found

FileNotFoundException

This Exception is raised when a file is not accessible or does not open.

IOException

It is thrown when an input-output operation failed or interrupted

InterruptedException

It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.

NoSuchFieldException

It is thrown when a class does not contain the field (or variable) specified

NoSuchMethodException

It is thrown when accessing a method which is not found.

NullPointerException

This exception is raised when referring to the members of a null object. Null represents nothing

NumberFormatException

This exception is raised when a method could not convert a string into a numeric format.

RuntimeException

This represents any exception which occurs during runtime.

StringIndexOutOfBoundsException

It is thrown by String class methods to indicate that an index is either negative than the size of the string

e.g.Let’s see some of the examples of different classes.

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

```
    {
        try {
            int a = 30, b = 0;
            int c = a/b;  // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by 0");         }     } }
class NullPointer_Demo
{
    public static void main(String args[])
    {
        try {
            String a = null; //null value
            System.out.println(a.charAt(0));
        } catch(NullPointerException e) {
            System.out.println("NullPointerException..");
        }
    }
}
```

```
class NullPointer_Demo
{
    public static void main(String args[])
    {
        try {
            String a = null; //null value
            System.out.println(a.charAt(0));
        } catch(NullPointerException e) {
            System.out.println("NullPointerException..");
        }
    }
}
```

:: QUESTIONS ::

1. How the exceptions are handled in java?
2. Explain exception handling mechanism in java?
3. What is difference between error and exception in java.
4. Explain the hierarchy of exceptions in java?
5. What is the difference between throw, throws and Throwable in java?
6. How to throw the exception in java.
7. Explain multiple catch block concept in java.
8. How can an exception be thrown manually by a programmer?
9. Explain types of Exception.
10. Explain try, catch &finally block in exception with example program.
11. How to use try-catch-finally in java? Can we use try,catch or finally block alone in java?

5. Chapter

User Interface using AWT and Swing

S y l l a b u s :

What is AWT, Swing, difference between AWT and Swing, Layout managers, Event Handling, Event sources, Listeners, Mouse and Keyboard event handling Components – JButton, JLabel, JText, JTextArea, JCheckBox, JRadioButton.

(10 L, 15 M)

Java AWT Tutorial

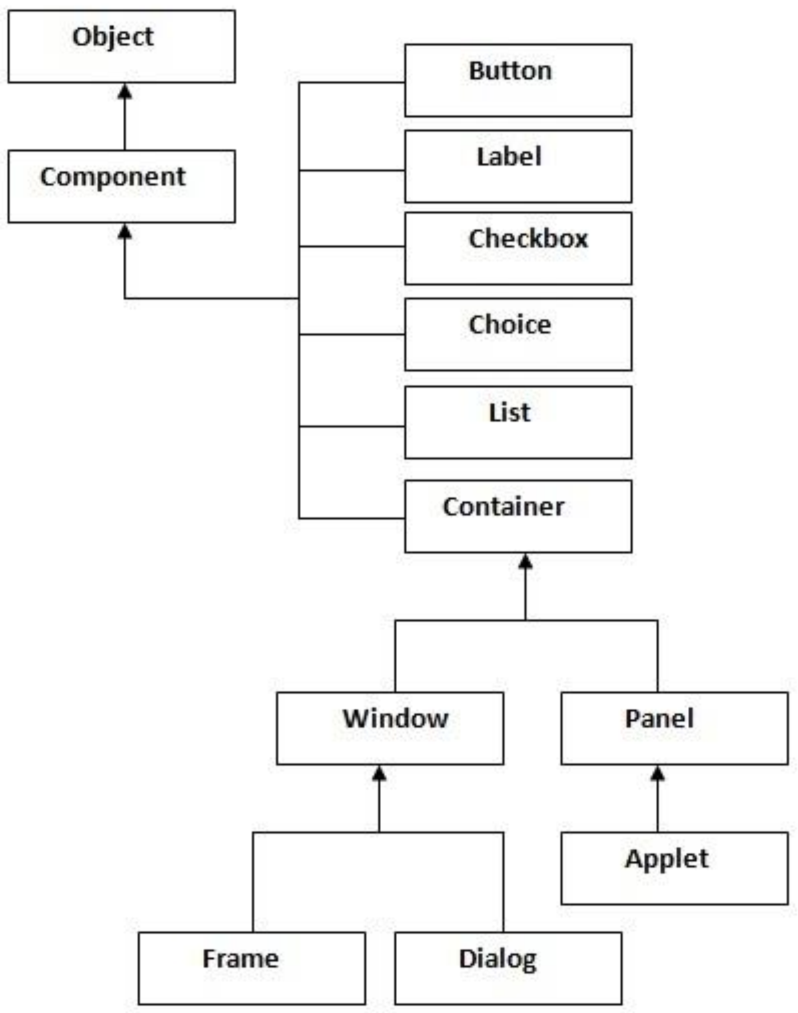
Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Some Java Fundamentals of the AWT

Concept of window and advanced components in java

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from Panel, which is used by applets, and those derived from Frame, which creates a standard window. Much of the functionality of these windows is derived from their parent classes.

Component

At the top of the AWT hierarchy is the Component class. Component is an abstract class that encapsulates all of the attributes of a visual component.

All user interface elements that are displayed on the screen and that interact with the user are subclasses of Component.

It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.

A Component object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container

The Container class is a subclass of Component. It has additional methods that allow other Component objects to be nested within it.

Other Container objects can be stored inside of a Container (since they are themselves instances of Component). This makes for a multileveled containment system.

A container is responsible for laying out (that is, positioning) any components that it contains.

Panel

The Panel class is a concrete subclass of Container. It doesn't add any new methods; it simply implements Container. Panel is the superclass for Applet. When screen output is directed to an applet, it is drawn on the surface of a Panel object.

A Panel is a window that does not contain a title bar, menu bar, or border. When you run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a Panel object by its add( ) method inherited from Container.

Once these components have been added, you can position and resize them manually using the setLocation( ), setSize( ), or setBounds( ) methods defined by Component.

Window

The Window class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop. Generally, we don't create
Window objects directly. Instead, we use a subclass of Window called Frame.

Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of Window and has a title bar, menu bar, borders, and resizing corners.

Canvas

It is not part of the hierarchy for applet or frame windows. Canvas encapsulates a blank window upon which you can draw.

Useful Methods of Component class

Method	**Description**
public void add(Component c)	inserts a component on this component.
public void set Size (int width, int height)	sets the size (width and height) of the component.

public void set Layout (Layout Manager m)	defines the layout manager for the component.
public void set Visible (boolean status)	changes the visibility of the component, by default false.

Java AWT Example

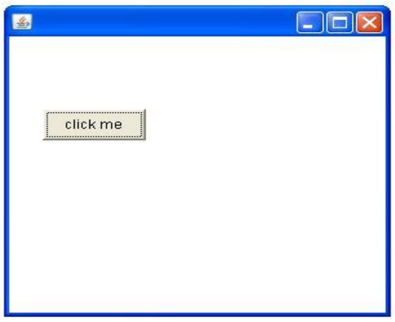
To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

1. By extending Frame class (inheritance)
2. By creating the object of Frame class (association)

AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

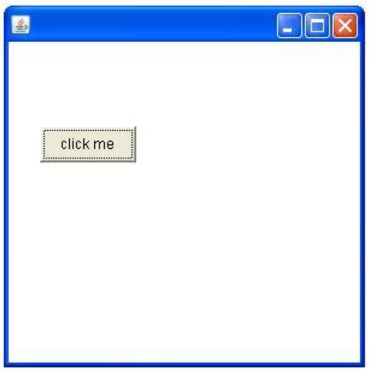
```
import java.awt.*;
    class First extends Frame{
        First(){
                Button b=new Button("click me");
                b.setBounds(30,100,80,30);// setting button position
                add(b);//adding button into frame
                setSize(300,300);//frame size 300 width and 300 height
                setLayout(null);//no layout manager
                setVisible(true);//now frame will be visible, by default not visible
        }
        public static void main(String args[]){
                First f=new First();
        }
    }
```



AWT Example by Association

Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First2{
    First2(){
            Frame f=new Frame();
            Button b=new Button("click me");
            b.setBounds(30,50,80,30);
            f.add(b);
            f.setSize(300,300);
            f.setLayout(null);
            f.setVisible(true);
    }
    public static void main(String args[]){
            First2 f=new First2();
    }
}
```



Java Swing:

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

In Swing, classes that represent GUI components have names beginning with the letter “J”.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Altogether there are more than 250 new classes and 75 interfaces in Swing — twice as many as in AWT.

AWT	Swing
AWT components are rectangular in shape.	swing components can be of any shape.
We cannot use image on AWT components.	We can use images on label or button in swing.
AWT components are less powerful and flexible.	Swing components are more powerful and flexible.
AWT components are platform dependent.	Swing components are not platform dependent.
AWT components are used to perform simple task.	Swing components are used to perform complex task.
This components are termed as heavy weight components.	This components are termed as light weight components.
These components are contained in awt package.	These components are contained in swing package.
In AWT we use paint() method for handling painting and graphics.	In Swing we use paint Component() method for handling painting and graphics.

Java Layout Managers:

The Layout Managers are used to arrange components in a particular manner. The Java Layout Managers facilitates us to control the positioning and size of the components in GUI forms. Layout Manager is an interface that is implemented by all the classes of layout managers. There are the following classes that represent the layout managers:

1. **Flow Layout**

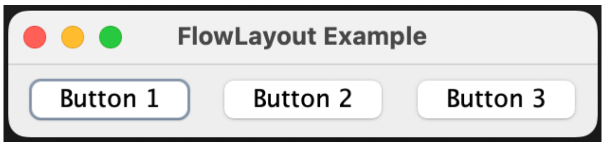
   Flow Layout is a simple layout manager that arranges components in a row, left to right, wrapping to the next line as needed. It is ideal for scenarios where components need to maintain their natural sizes and maintain a flow-like structure.

   FlowLayoutExample.java

```
import javax.swing.*;
import java.awt.*;
public class FlowLayoutExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("FlowLayout Example");
        frame.setLayout(new FlowLayout());
        frame.add(new JButton("Button 1"));
```

```
                frame.add(new JButton("Button 2"));
                frame.add(new JButton("Button 3"));
                frame.pack();
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
}
```

Output:



2. Border Layout

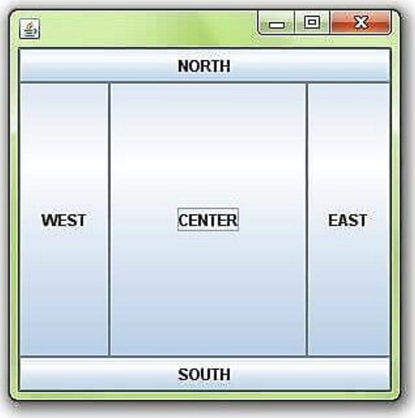
Border Layout divides the container into five regions: NORTH, SOUTH, EAST, WEST, and CENTER. Components can be added to these regions, and they will occupy the available space accordingly. This layout manager is suitable for creating interfaces with distinct sections, such as a title bar, content area, and status bar.

BorderLayoutExample.java

```
import java.awt.*;
import javax.swing.*;
public class Border
{
        JFrame f;
        Border()
{
         f = new JFrame();
                // creating buttons
        JButton b1 = new JButton("NORTH");; // the button will be labeled as NORTH
        JButton b2 = new JButton("SOUTH");; // the button will be labeled as SOUTH
        JButton b3 = new JButton("EAST");; // the button will be labeled as EAST
        JButton b4 = new JButton("WEST");; // the button will be labeled as WEST
       JButton b5 = new JButton("CENTER");; // the button will be labeled as CENTER
        f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Direction
        f.add(b2, BorderLayout.SOUTH);  // b2 will be placed in the South Direction
```

```
            f.add(b3, BorderLayout.EAST);  // b2 will be placed in the East Direction
            f.add(b4, BorderLayout.WEST);  // b2 will be placed in the West Direction
            f.add(b5, BorderLayout.CENTER);  // b2 will be placed in the Center
            f.setSize(300, 300);
             f.setVisible(true);
    }
    public static void main(String[] args) {
    new Border();
    }
}
```

Output:



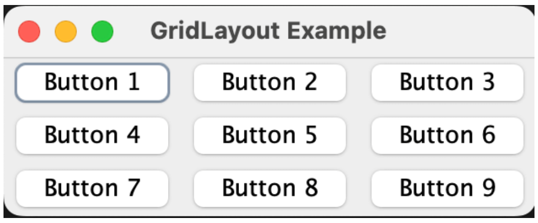
3. Grid Layout

Grid Layout arranges components in a grid with a specified number of rows and columns. Each cell in the grid can hold a component. This layout manager is ideal for creating a uniform grid of components, such as a calculator or a game board.

GridLayoutExample.java

```
import javax.swing.*;
import java.awt.*;
public class GridLayoutExample {
        public static void main(String[] args) {
                JFrame frame = new JFrame("GridLayout Example");
                frame.setLayout(new GridLayout(3, 3));
                for (int i = 1; i <= 9; i++) {
                        frame.add(new JButton("Button " + i));
                }
                frame.pack();
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
}
```

Output:



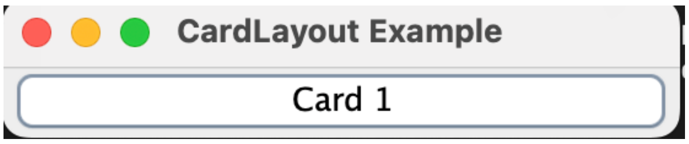
4. Card Layout

Card Layout allows components to be stacked on top of each other, like a deck of cards. Only one component is visible at a time, and you can switch between components using methods like next() and previous(). This layout is useful for creating wizards or multi-step processes.

```
CardLayoutExample.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class CardLayoutExample {
public static void main(String[] args) {
         JFrame frame = new JFrame("CardLayout Example");
      CardLayout cardLayout = new CardLayout();
      JPanel cardPanel = new JPanel(cardLayout);
      JButton button1 = new JButton("Card 1");
      JButton button2 = new JButton("Card 2");
      JButton button3 = new JButton("Card 3");
      cardPanel.add(button1, "Card 1");
      cardPanel.add(button2, "Card 2");
      cardPanel.add(button3, "Card 3");
      frame.add(cardPanel);
```

```
        frame.pack();    //frame. pack() is used to size the JFrame automatically to the size
                         of the widgets within the page.
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        button1.addActionListener(e -> cardLayout.show(cardPanel, "Card 2"));
        button2.addActionListener(e -> cardLayout.show(cardPanel, "Card 3"));
        button3.addActionListener(e -> cardLayout.show(cardPanel, "Card 1"));
        }
}
```

Output:



5. Group Layout

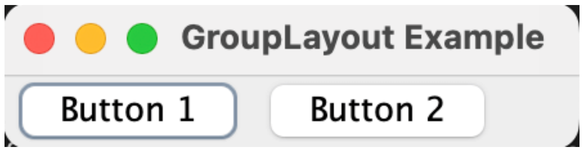
Group Layout is a versatile and complex layout manager that provides precise control over the positioning and sizing of components. It arranges components in a hierarchical manner using groups. Group Layout is commonly used in GUI builders like the one in NetBeans IDE.

GroupLayoutExample.java

```
import javax.swing.*;
public class GroupLayoutExample {
public static void main(String[] args) {
    JFrame frame = new JFrame("GroupLayout Example");
    JPanel panel = new JPanel();
    GroupLayout layout = new GroupLayout(panel);
    panel.setLayout(layout);
    JButton button1 = new JButton("Button 1");
    JButton button2 = new JButton("Button 2");
    layout.setHorizontalGroup(layout.createSequentialGroup()
      .addComponent(button1)
            .addComponent(button2));
    layout.setVerticalGroup(layout.createParallelGroup()
       .addComponent(button1)
```

```
                    .addComponent(button2));
        frame.add(panel);
    frame.pack();
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    }
```

Output:



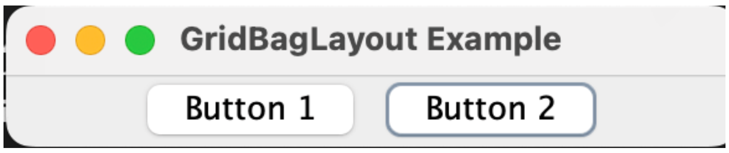
6. Grid Bag Layout

Grid Bag Layout is a powerful layout manager that allows you to create complex layouts by specifying constraints for each component. It arranges components in a grid, but unlike Grid Layout, it allows components to span multiple rows and columns and have varying sizes.

GridBagLayoutExample.java

```
import javax.swing.*;
import java.awt.*;
public class GridBagLayoutExample {
    public static void main(String[] args) {
      JFrame frame = new JFrame("GridBagLayout Example");
JPanel panel = new JPanel(new GridBagLayout());
GridBagConstraints constraints = new GridBagConstraints();
constraints.fill = GridBagConstraints.HORIZONTAL;
JButton button1 = new JButton("Button 1");
JButton button2 = new JButton("Button 2");
constraints.gridx = 0;
constraints.gridy = 0;
panel.add(button1, constraints);
constraints.gridx = 1;
panel.add(button2, constraints);
frame.add(panel);
frame.pack();
frame.setVisible(true);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
}
```

Output:



Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	**Listener Interfaces**
Action Event	Action Listener
Mouse Event	Mouse Listener and Mouse Motion Listener
Mouse Wheel Event	Mouse Wheel Listener
Key Event	Key Listener
Item Event	Item Listener
Tex tEvent	Text Listener
Adjustment Event	Adjustment Listener
Window Event	Window Listener
Component Event	Component Listener
Container Event	Container Listener
Focus Event	Focus Listener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. **Register the component with the Listener**

   Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **Menu Item**
  - public void addActionListener(ActionListener a){}
- **Text Field**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **Text Area**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

Java Event Handling Code

We can put the event handling code into one of the following places:

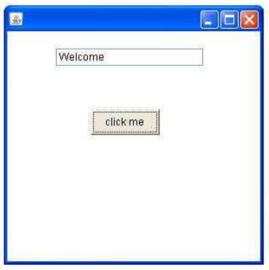
1. Within class
2. Other class
3. Anonymous class

Java event handling by implementing ActionListener

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
  TextField tf;
  AEvent(){
      //create components
      tf=new TextField();
      tf.setBounds(60,50,170,20);
      Button b=new Button("click me");
      b.setBounds(100,120,80,30);
      //register listener
```

```
                b.addActionListener(this);//passing current instance
                //add components and set size, layout and visibility
                add(b);add(tf);
                setSize(300,300);
                setLayout(null);
            setVisible(true);
        }
        public void actionPerformed(ActionEvent e){
                tf.setText("Welcome");
        }
        public static void main(String args[]){
                new AEvent();
        }
    }
```

public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.



Skip Ad

2) Java event handling by outer class

```
import java.awt.*;
import java.awt.event.*;
class AEvent2 extends Frame{
    TextFieldtf;
    AEvent2(){
        //create components
        tf=new TextField();
        tf.setBounds(60,50,170,20);
        Button b=new Button("click me");
        b.setBounds(100,120,80,30);
```

```
            //register listener
            Outer o=new Outer(this);
            b.addActionListener(o);//passing outer class instance
            //add components and set size, layout and visibility
            add(b);add(tf);
            setSize(300,300);
            setLayout(null);
            setVisible(true);
         }
       public static void main(String args[]){
            new AEvent2();
      }
}
```

Second sub program

```
import java.awt.event.*;
class Outer implements ActionListener{
            AEvent2 obj;
            Outer(AEvent2 obj){
                        this.obj=obj;
            }
            public void actionPerformed(ActionEvent e){
                        obj.tf.setText("welcome");
            }
}
```

3) Java event handling by anonymous class

```
import java.awt.*;
import java.awt.event.*;
class AEvent3 extends Frame{
       TextFieldtf;
      AEvent3(){
            tf=new TextField();
            tf.setBounds(60,50,170,20);
            Button b=new Button("click me");
```

```
                b.setBounds(50,120,80,30);

                b.addActionListener(new ActionListener(){
                        public void actionPerformed(){
                                tf.setText("hello");
                        }
                });
            add(b);add(tf);
            setSize(300,300);
            setLayout(null);
             setVisible(true);
        }
        public static void main(String args[]){
            new AEvent3();
        }
    }
```

Mouse Listener

The Java Mouse Listener is notified whenever you change the state of mouse. It is notified against Mouse Event. The Mouse Listener interface is found in java.awt.event package. It has five methods.

There are five abstract functions that represent these five events. The abstract functions are:

1. void mouseReleased(MouseEvent e) : Mouse key is released
2. void mouseClicked(MouseEvent e) : Mouse key is pressed/released
3. void mouseExited(MouseEvent e) : Mouse exited the component
4. void mouseEntered(MouseEvent e) : Mouse entered the component
5. void mousepressed(MouseEvent e) : Mouse key is pressed

There are two types of events that MouseMotionListener can generate. There are two abstract functions that represent these five events. The abstract functions are:

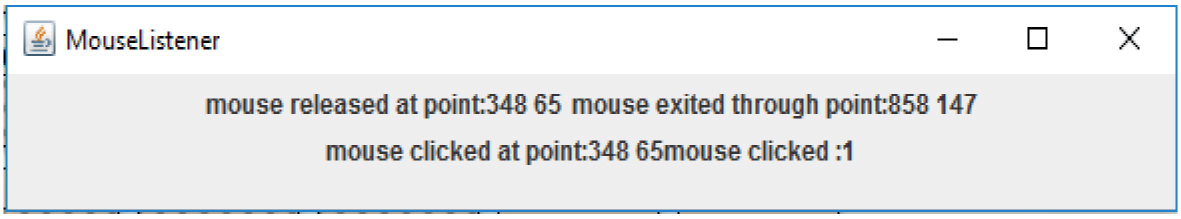
1. void mouseDragged(MouseEvent e) : Invoked when a mouse button is pressed in the component and dragged. Events are passed until the user releases the mouse button.
2. void mouseMoved(MouseEvent e) : invoked when the mouse cursor is moved from one point to another within the component, without pressing any mouse buttons.

```
// Java program to handle MouseListener events
import java.awt.*;
```

```
   import java.awt.event.*;
   import javax.swing.*;
   class Mouse extends Frame implements MouseListener {
   Mouse()
   {
   }
   public static void main(String[] args)
   {
          JFrame f = new JFrame("MouseListener");
         f.setSize(600, 100);
          f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
          JPanel p = new JPanel();
          p.setLayout(new FlowLayout());
          label1 = new JLabel("no event  ");
          label2 = new JLabel("no event  ");
          label3 = new JLabel("no event  ");
          Mouse m = new Mouse();
          f.addMouseListener(m);
          p.add(label1);
          p.add(label2);
          p.add(label3);
          f.add(p);
          f.show();
      }
public void mousePressed(MouseEvent e)
{
      label1.setText("mouse pressed at point:"
          + e.getX() + " " + e.getY());
}
public void mouseReleased(MouseEvent e)
{
      label1.setText("mouse released at point:"
          + e.getX() + " " + e.getY());
}
public void mouseExited(MouseEvent e)
{
        label2.setText("mouse exited through point:"
              + e.getX() + " " + e.getY());
}
public void mouseEntered(MouseEvent e)
{
   label2.setText("mouse entered at point:"
              + e.getX() + " " + e.getY());
}
 public void mouseClicked(MouseEvent e)
{

       label3.setText("mouse clicked at point:"
              + e.getX() + " "
              + e.getY() + "mouse clicked :" + e.getClickCount());
       }
   }
```

Output



```
mouse released at point:348 65  mouse exited through point:858 147
        mouse clicked at point:348 65mouse clicked :1
```

Java Key Listener Interface

The **Java Key Listener is notified whenever you change the state of key.** It is notified against Key Event. The Key Listener interface is found in java.awt.event package, and it has three methods.

Interface declaration

Following is the declaration for **java.awt.event.KeyListener** interface:

public interface KeyListener **extends** EventListener

The signature of 3 methods found in KeyListener interface are given below:

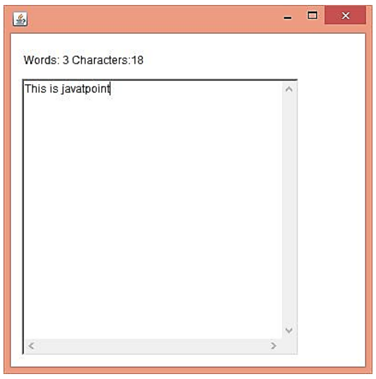
Sr.No.	Method name	Description
1.	public abstract void keyPressed (KeyEvent e);	It is invoked when a key has been pressed.
2.	public abstract void keyReleased (KeyEvent e);	It is invoked when a key has been released.
3.	public abstract void keyTyped (KeyEvent e);	It is invoked when a key has been typed.

KeyListenerExample2.java

```
/ importing the necessary libraries
import java.awt.*;
import java.awt.event.*;
// class which inherits Frame class and implements KeyListener interface
public class KeyListenerExample2 extends Frame implements KeyListener {
                  Label l;     // object of Label and TextArea
                 TextArea area;
                 KeyListenerExample2() {    // class constructor
                             l = new Label();   // creating the label
                 l.setBounds (20, 50, 200, 20);    // setting the location of
                 area = new TextArea();    label  // creating the text area
                 area.setBounds (20, 80, 300, 300);    // setting location of text area
                 area.addKeyListener(this);   // adding KeyListener to the text area
                 add(l);               // adding label and text area to frame
                 add(area);
                             setSize (400, 400);    // setting size, layout and visibility of frame
                             setLayout (null);
                             setVisible (true);
                 }
                 // even if we do not define the interface methods, we need to override them
                    public void keyPressed(KeyEvent e) {}
                 // overriding the keyReleased() method of KeyListener interface
                    public void keyReleased (KeyEvent e) {
                 // defining a string which is fetched by the getText() method of TextArea class
                    String text = area.getText();
```

```
            // splitting the string in words
              String words[] = text.split ("\\s");
            // printing the number of words and characters of the string
              l.setText ("Words: " + words.length + " Characters:" + text.length());
            }
            public void keyTyped(KeyEvent e) {}
            public static void main(String[] args) {
                     new KeyListenerExample2();
        }
}
```

Output:



```
Words: 3 Characters:18
This is javatpoint
```

JButton class

- The JButton class provides the functionality of a push button.
- JButton allows an icon, a string, or both to be associated with the push button.
- JButton are the subclass of the AbstractButton Class, which extends JComponent.

Commonly used Constructors:

- **JButton():** creates a button with no text and icon.
- **JButton(String s):** creates a button with the specified text.
- **JButton(Icon i):** creates a button with the specified icon object.

Commonly used Methods of AbstractButton class:

1) public void setText(String s): is used to set specified text on button.
2) public String getText(): is used to return the text of the button.
3) public void setEnabled(boolean b): is used to enable or disable the button.
4) public void setIcon(Icon b): is used to set the specified Icon on the button.
5) public Icon getIcon(): is used to get the Icon of the button.
6) public void setMnemonic(int a): is used to set the mnemonic on the button.

```
JMenu file = new JMenu("File");
file.setMnemonic(KeyEvent.VK_F);
```

We create a menu object. The menus can be accessed via the keybord as well. To bind a menu to a particular key, we use the setMnemonic method. In our case, the menu can be opened with the ALT + F shortcut.

7) public void addActionListener(ActionListener a): is used to add the action listener to this object.

Example of displaying image on the button:

Program :

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

import java.awt.*;
/*
<applet code="JButtonDemo.class" height=500 width=500>
</applet>
*/
public class JButtonDemo extends JApplet implements ActionListener
{
        JButton jbo,jbw,jbg;
        Container content = getContentPane();
public void init()
{
FlowLayout f1 = new FlowLayout();
        setLayout(f1);
        jbo = new JButton("Orange");
        jbw = new JButton("White");
        jbg = new JButton("Green");
        jbo.addActionListener(this);
        jbw.addActionListener(this);
        jbg.addActionListener(this);
        add(jbo);
        add(jbw);
        add(jbg);
        }
        public void actionPerformed(ActionEvent ae)
        {
                if(ae.getSource()==jbo )
        {
                content.setBackground(Color.orange);
        }
         else if(ae.getSource()==jbw)
        {
               content.setBackground(Color.white);
        }
        else if(ae.getSource()==jbg)
        {
                content.setBackground(Color.green);
        }
      }
   }
```

JRadioButton class

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

Commonly used Constructors of JRadioButton class:

- **JRadioButton():** creates an unselected radio button with no text.
- **JRadioButton(String s):** creates an unselected radio button with specified text.
- **JRadioButton(String s, boolean selected):** creates a radio button with the specified text and selected status.

Commonly used Methods of AbstractButton class:

1) public void setText(String s): is used to set specified text on button.
2) public String getText(): is used to return the text of the button.
3) public void setEnabled(boolean b): is used to enable or disable the button.
4) public void setIcon(Icon b): is used to set the specified Icon on the button.
5) public Icon getIcon(): is used to get the Icon of the button.
6) public void setMnemonic(int a): is used to set the mnemonic on the button.

7) public void addActionListener(ActionListener a): is used to add the action listener to this object.

Example of JRadioButton class:

```
import javax.swing.*;
public class Radio {
JFrame f;
Radio(){
        f=new JFrame();
        JRadioButton r1=new JRadioButton("A) Male");
        JRadioButton r2=new JRadioButton("B) FeMale");
        r1.setBounds(50,100,70,30);
        r2.setBounds(50,150,70,30);
        ButtonGroup bg=new ButtonGroup();
        bg.add(r1);bg.add(r2);
        f.add(r1);f.add(r2);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
        }
public static void main(String[] args) {
new Radio();
}
}
```

JTextField:

The JTextField is acomponent which represents a Tex-box which can be added to the Jframe .

The constructor of JTextfield is :

```
JTextField tf1=new JTextField(); //the Defualt Constructor
JTextField tf2=new JTextField(20); //with size
JTextField tf3=new JTextField("hello", 10); //with size & default text
```

You can set the text or retrieve it in the same way as a JLabel

```
tf.setText("New Text"); /* or */ tf.setText(txt);
String str = tf.getText ();
```

Program:

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class JTextFieldDemo
{
        JTextField jtf1;
public JTextFieldDemo()
{
        JFrame jfrm = new JFrame("JTextField");
        jtf1 = new JTextField ("Enter Text First");
        jtf1.setBounds(10,30,50,5);
        jfrm.setVisible(true);
        jfrm.setSize(500,500);
        jfrm.setVisible(true);
        jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        jfrm.add(jtf1);
}
public static void main(String args[])
 {
        JTextFieldDemo obj = new JTextFieldDemo();
  }
}
```

JTextArea class (Swing Tutorial):

The JTextArea class is used to create a text area. It is a multiline area that displays the plain text only.

Commonly used Constructors:

- **JTextArea():** creates a text area that displays no text initially.
- **JTextArea(String s):** creates a text area that displays specified text initially.
- **JTextArea(int row, int column):** creates a text area with the specified number of rows and columns that displays no text initially..
- **JTextArea(String s, int row, int column):** creates a text area with the specified number of rows and columns that displays specified text.

Commonly used methods of JTextArea class:

1) public void setRows(int rows): is used to set specified number of rows.
2) public void setColumns(int cols):: is used to set specified number of columns.
3) public void setFont(Font f): is used to set the specified font.
4) public void insert(String s, int position): is used to insert the specified text on the specified position.
5) public void append(String s): is used to append the given text to the end of the document.

Example of JTextArea class:

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

public class TArea {
  JTextArea area;
  JFrame f;
      TArea(){
      f=new JFrame();
      area=new JTextArea(300,300);
      area.setBounds(10,30,300,300);
      area.setBackground(Color.red);
      area.setForeground(Color.white);
      f.add(area);
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
  }
    public static void main(String[] args) {
      new TArea();
   }
}
```

Java JLabel

The object of JLabel class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly. It inherits JComponent class.

JLabel class declaration

Let's see the declaration for javax.swing.JLabel class.

public class JLabel **extends** JComponent **implements** SwingConstants, Accessible

Commonly used Constructors:

Constructor	**Description**
JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	Creates a JLabel instance with the specified text.
JLabel(Icon i)	Creates a JLabel instance with the specified image.
JLabel(String s, Icon i, int horizontalAlignment)	Creates a JLabel instance with the specified text, image, and horizontal alignment.

Java JCheckBox

The JCheckBox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a CheckBox changes its state from "on" to "off" or from "off" to "on ".It inherits JToggleButton class.

JCheckBox class declaration

Let's see the declaration for javax.swing.JCheckBox class.

public class JCheckBox **extends** JToggleButton **implements** Accessible

Commonly used Constructors:

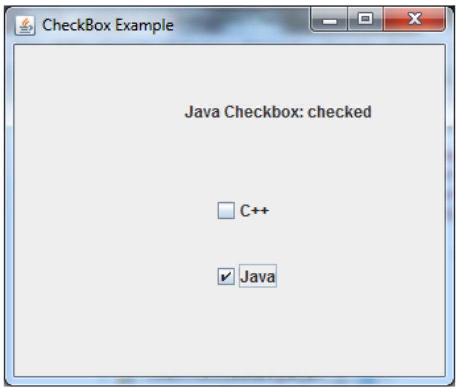
Constructor	**Description**
JJCheckBox()	Creates an initially unselected check box button with no text, no icon.
JChechBox(String s)	Creates an initially unselected check box with text.
JCheckBox(String text, boolean selected)	Creates a check box with text and specifies whether or not it is initially selected.
JCheckBox(Action a)	Creates a check box where properties are taken from the Action supplied.

Java JCheckBox Example with ItemListener

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

```
public class CheckBoxExample
{
    CheckBoxExample(){
      JFrame f= new JFrame("CheckBox Example");
      final JLabel label = new JLabel();
      label.setHorizontalAlignment(JLabel.CENTER);
      label.setSize(400,100);
      JCheckBox checkbox1 = new JCheckBox("C++");
      checkbox1.setBounds(150,100, 50,50);
      JCheckBox checkbox2 = new JCheckBox("Java");
      checkbox2.setBounds(150,150, 50,50);
      f.add(checkbox1); f.add(checkbox2); f.add(label);
      checkbox1.addItemListener(new ItemListener() {
          public void itemStateChanged(ItemEvent e) {
            label.setText("C++ Checkbox: "
            + (e.getStateChange()==1?"checked":"unchecked"));
          }
        });
      checkbox2.addItemListener(new ItemListener() {
          public void itemStateChanged(ItemEvent e) {
            label.setText("Java Checkbox: "
            + (e.getStateChange()==1?"checked":"unchecked"));
          }
        });
      f.setSize(400,400);
      f.setLayout(null);
      f.setVisible(true);
    }
public static void main(String args[])
{
   new CheckBoxExample();
}
}
```

Output:



```
Java Checkbox: checked
```

❒❒❒

6. Chapter

Streams and Files in JAVA

S y l l a b u s :

Using the File Class, Stream classes, Byte Stream Class, Character Stream Class, and Create/Read/Write File..

(10 L, 15 M)

Streams

Java defines a common way of handling all input/output devices, namely as producers/consumers of sequences of data (characters). The concept of **stream** represents a *sequence of data generated by a input device or consumed by an output device*.

Example of input streams :

i) keyboard

ii) file

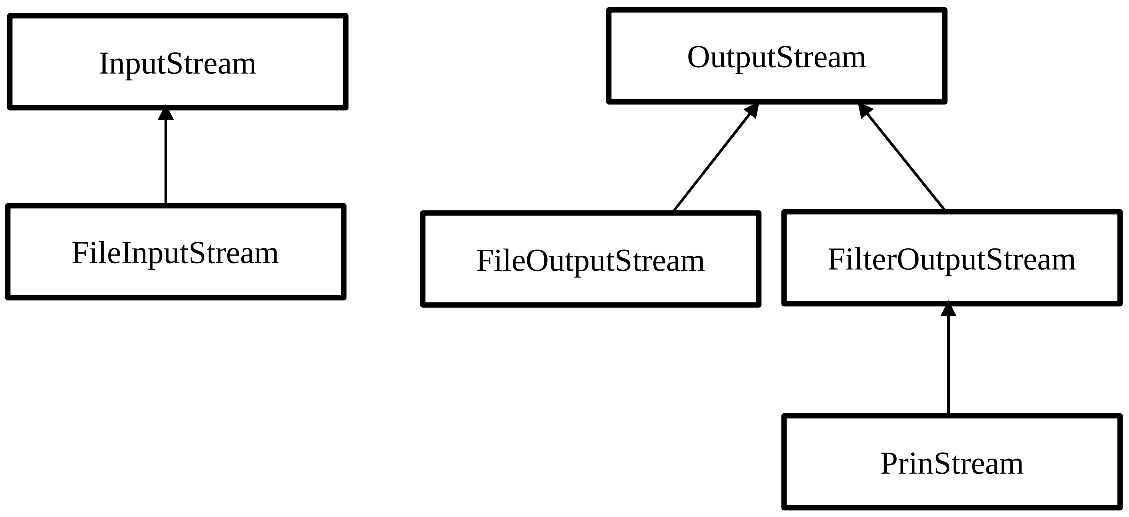
iii) Internet resource Example of

iv) output streams:

v) video

vi) file

All classes used to define the various streams are derived from the classes InputStream and OutputStream according to the following diagram.



This common handling of all input/output devices allow us to define the readers for the input channels and the writers for the output channels independently from the specific device used to realize the input/output.

Input streams

Keyboard

We use the predefined object System.in of the class InputStream.

Example:

InputStream is = System.in;

File

We create an object of the class FileInputStream (subclass ofInputStream) associated to the File object that identifies the file to open for reading.

Example:

FileInputStream is = new FileInputStream("data.txt");

We could also create a File object starting from the filename, and use such an object to create the

FileInputStream:

File f = new File("data.txt");

FileInputStream is = new FileInputStream(f);

Output streams

Video

We use the predefined object System.out of the class PrintStream (subclass of OutputStream).

Example:

OutputStream os = System.out;

File

We create an object of the class FileOutputStream (subclass ofOutputStream) associated to the File object that identifies the file to open for writing.

Example:

FileOutputStream os = new FileOutputStream("data.txt");

Note: It is not possible to write to an Internet resource by simply using an output stream.

Reading from an input stream and writing to an output stream :

The operations for reading from input streams and writing to output streams are standardized, i.e., they do not depend on the particular stream being used. This means that we can use the same Java statements to read strings from a keyboard, from a file, or from the Internet.

Example 1: reading from an input stream

```
InputStream is = ...; // keyboard, file or Internet
InputStreamReader isr = new InputStreamReader(is);
BufferedReader br = new BufferedReader(isr);
String line = br.readLine();
```

Example 2: writing to an output stream (1)

```
OutputStream os = ...; // video or file
PrintWriter pw = new PrintWriter(os);
pw.println("Hello");
```

Example 3: writing to an output stream (2)

```
OutputStream os = ...; // video or file
PrintStream ps = new PrintStream(os); ps.println("Hello");
```

Files :

Files are the most important mechanism for storing data permanently on mass-storage devices. Permanently means that the data is not lost when the machine is switched off. Files can contain: data in a format that can be interpreted by programs, but not easily by humans (binary files); alphanumeric characters, codified in a standard way (e.g., using ASCII or Unicode), and directly readable by a human user (text files. Text files are normally organized in a sequence of lines, each containing a sequence of characters and ending with a special character (usually the newline character). Consider, for example, a Java program stored in a file on the hard-disk. In this unit we will deal only with text files.

Each file is characterized by a name and a directory in which the file is placed (one may consider the whole path that allows on to find the file on the hard-disk as part of the name of the file). The most important operations on files are: creation, reading from, writing to, renaming, deleting. All these operations can be performed through the operating system (or other application programs), or through suitable operations in a Java program.

To execute reading and writing operations on a file, we must open the file before doing the operations, and close it after we are finished operating on it. Opening a file means to indicate to the operating system that we want to operate on a file from within a Java program, and the operating system verifies whether such operations are possible and allowed. There are two ways of opening a file: opening for reading and opening for writing, which cause a different behavior of the operating system. For example, two files may be opened at the same time by two applications for reading, but not for writing; or a file on a CD may be opened for reading, but not for writing. In many programming languages (including Java), opening a file for writing means actually to create a new file. Closing a file means to indicate to the operating system that the file that was previously opened is not being used anymore by the program. Closing a file also has the effect of ensuring that the data written on the file are effectively transferred to the hard-disk.

Writing Text Files

To write a string of text on a file we have to do the following :

1. Open the file for writing by creating an object of the class FileWriter associated to the name of the file, and creating an object of the class PrintWriter associated to the FileWriter object just created;
2. Write text on the file by using the print and println methods of the PrintWriter object;
3. Close the file when we are finished writing to it.

The Java statements that realize the three phases described above are:

```
// 1. opening the file for writing (creation of the file) FileWriter f = new FileWriter("test.txt");
PrintWriter out = new PrintWriter(f);

// 2. writing text on the file out.println("some text to write to the file");

// 3. closing the output channel and the file out.close();
f.close();
```

Random Access File Stream

The Java.io.RandomAccessFile class file behaves like a large array of bytes stored in the file system.Instances of this class support both reading and writing to a random access file.

Constructor :

o RandomAccessFile(File, String)

Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.

o RandomAccessFile(String, String)

Creates a random access file stream to read from, and optionally to write to, a file with the specified name.

Methods :

o close()

Closes this random access file stream and releases any system resources associated with the stream.

o getFD()

Returns the opaque file descriptor object associated with this stream.

o getFilePointer()

Returns the current offset in this file.

o length()

Returns the length of this file.

o read()

Reads a byte of data from this file.

o read(byte[])

Reads up to b.length bytes of data from this file into an array of bytes.

o read(byte[], int, int)

Reads up to len bytes of data from this file into an array of bytes.

o readBoolean()

Reads a boolean from this file.

o readByte()

Reads a signed 8-bit value from this file.

o readChar()

Reads a Unicode character from this file.

o readDouble()

Reads a double from this file.

o readFloat()

Reads a float from this file.

- readFully(byte[])

Reads b.length bytes from this file into the byte array.

- readFully(byte[], int, int)

Reads exactly len bytes from this file into the byte array.

- readInt()

Reads a signed 32-bit integer from this file.

- readLine()

Reads the next line of text from this file.

- readLong()

Reads a signed 64-bit integer from this file.

- readShort()

Reads a signed 16-bit number from this file.

- readUnsignedByte()

Reads an unsigned 8-bit number from this file.

- readUnsignedShort()

Reads an unsigned 16-bit number from this file.

- readUTF()

Reads in a string from this file.

- seek(long)

Sets the offset from the beginning of this file at which the next read or write occurs.

- skipBytes(int)

Skips exactly n bytes of input.

- write(byte[])

Writes b.length bytes from the specified byte array starting at offset off to this file.

- write(byte[], int, int)

Writes len bytes from the specified byte array starting at offset off to this file.

- write(int)

Writes the specified byte to this file.

- writeBoolean(boolean)

Writes a boolean to the file as a 1-byte value.

- writeByte(int)

Writes a byte to the file as a 1-byte value.

- writeBytes(String)

Writes the string to the file as a sequence of bytes.

- writeChar(int)

Writes a char to the file as a 2-byte value, high byte first.

- writeChars(String)

Writes a string to the file as a sequence of characters.

- o writeDouble(double)

Converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an 8-byte quantity, high byte first.

- o writeFloat(float)

Converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a 4-byte quantity, high byte first.

- o writeInt(int)

Writes an int to the file as four bytes, high byte first.

- o writeLong(long)

Writes a long to the file as eight bytes, high byte first.

- o writeShort(int)

Writes a short to the file as two bytes, high byte first.

- o writeUTF(String)

Writes a string to the file using UTF-8 encoding in a machine-independent manner.

In RandomAccessFile, while instantiating default mode is read only . But we can provide different mode. These modes are

"r"   : File is open for read only.

"rw" : File is open for read and write both.

"rws" : Same as rw mode. It also supports to update file content synchronously to device storage.

"rwd" : Same as rw mode that also supports reduced number of IO operation.

e.g.

```
import java.util.*;
import java.io.*;
public class TestClass{
  public static void main(String[] args) {
    try {
        RandomAccessFile raFile =new RandomAccessFile("D://test.txt","rw");
        raFile.write("Java Tutorial".getBytes()); //add the content
        raFile.seek(raFile.getFilePointer()-8); //set pointer backward -8 characters
        raFile.write("File Class Tutorial ".getBytes()); //write the text where pointer is
        raFile.seek(0); //set pointer to start of file
        int i;
      while((i= raFile.read())!=-1){
        System.out.print((char)i);
      }
    }
    catch (IOException e){
    e.printStackTrace();
    }   } }
```

String Tokenizer

The StringTokenizer class of java.util package, allows an application to break a string into tokens. Most programmers opt to use StringTokenizer in tokenizing a string than StreamTokenizer because the methods are much simpler than the latter. The StringTokenizer methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments.An instance of StringTokenizer behaves in one of two ways, depending on whether it was created with the returnDelims flag having the value true or false:

If the flag is false, delimiter characters serve to separate tokens. A token is a maximal sequence of consecutive characters that are not delimiters. If the flag is true, delimiter characters are themselves considered to be tokens. A token is thus either one delimiter character, or a maximal sequence of consecutive characters that are not delimiters. A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed. A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

```
tringTokenizer st = new StringTokenizer("this is a test String");
```

From the above code, we basically just create a new StringTokenizer object and make use of one of the class constructor to initialize the String to be tokenized. There are multiple constructors available for StringTokenizer, one of it accepts a delimiter apart from the input String. Since we didn't specify the delimiter, automatically the default delimiters are " tnrf": the space character, the tab character, the newline character, the carriage-return character, and the form-feed character.

The hasMoreTokens() method returns true if and only if there is at least one token in the string after the current position; false otherwise.

```
import java.util.StringTokenizer;
/*
 * This example source code demonstrates the use of
 * hasMoreTokens() method of StringTokenizer class.
 *
 */
public class StringTokenizerHasMoreTokensExample {

    public static void main(String[] args) {

            //declare a new String with tokens delimited by |
            String input = "Beth|Ryan|Grace|Travis|Vince";

            /*
             * initialize our StringTokenizer using input
             * as source String and | as the delimiter
             */

            StringTokenizer st = new StringTokenizer(input,"|");

            // print all tokens
            while(st.hasMoreTokens()){
                    System.out.println(st.nextToken());
            }

    }
}
```

Object Stream

To work with the I/O of objects Java provided the support of Object stream. Object stream classes implements either ObjectInput or the ObjectOutput interfaces. ObjectInput is the subinterface of DataInput and the ObjectOutput is the subinterface of DataOutput interface.

ObjectInput : ObjectInput is an interface which is a subinterface of DataInput interface that facilitate for reading the objects.

Methods of ObjectInput

available() : This method is used to find out the readable number of bytes.
int available() throws IOException

close() : This method is used to close the input stream.

void close() throws IOException

read() : This method is used to read the data's byte.
int read() throws IOException

read(byte[] b) : This method is used to read into an array of bytes.
int read(byte[] b) throws IOException

read(byte[] b, int off, int len) : This method is used to read into an array of bytes.
int read(byte[] b, int off, int len) throws IOException

readObject() : This method is used to read the object that support java.io.Serializable interface and also returns the object.

:: QUESTIONS ::

1. Explain the concept of streams.
2. Explain java stream classes.
3. Explain the hierarchy of input stream classes.
4. Explain the hierarchy of output stream classes.
5. Explain the hierarchy of reader stream classes.
6. Explain the hierarchy of writer stream classes.
7. How to create file in java. Explain in brief.
8. What is difference between inputStream & Reader classes.
9. What is difference between OutputStream &Writer classes.
10. Explain String tokenizer concept.
11. Explain RandomAccessFile class in java.
12. Explain the hierarchy of data stream classes.

Notes...

Notes...