Name of Institute: IITE Indus University Ahmedabad

Name of Faculty: Mr. Milan Bhadaliya

Course code: CE0421

Course name: Core Java Programming

Pre-requisites: C, C++

Credit points: 4

Offered Semester: IV

Course Coordinator

Full name: Mr. Milan Bhadaliya

Department with sitting location: CSE department, 4th floor Bhanwar Building.

Email: milanbhadaliya.ce@indusuni.ac.in

Consultation times: 3.00 P.M. to 5.00 P.M. (Monday to Friday)

Course lecturers

Full Name: Prof. Milan Bhadaliya

Department with sitting location: CSE department, 4th floor Bhanwar Building.

Email: milanbhadaliya.ce@indusuni.ac.in

Consultation times: 3.00 P.M. to 5.00 P.M. (Monday to Friday)

Course lecturers

Full Name: Prof. Madhvi Bera

Department with sitting location: CSE department, 4th floor Bhanwar Building.

Email: madhvibera.ce@indusuni.ac.in

Consultation times: 3.00 P.M. to 5.00 P.M. (Monday to Friday)

Students will be contacted throughout the session via mail with important information relating to this course.

Course Objectives:

1. To understand object oriented programming concepts and implement in java.
2. Comprehend building blocks of OOPs language, inheritance, package and interfaces.
3. Identify exception handling methods.
4. Implement multi-threading in object oriented programs.
5. Prepare UML diagrams for software system
6. To enhance the programming skills of students into field of Java Programming and to create their interest in the same field

Course Outcomes (CO)

After successful completion of the course, student will able to:

1. Apply the object oriented concepts for the given problem and able to do work in OOP Concept technology.
2. Use and create packages in a java program and manage project web classes in proper order.
3. Create applet application as per customer requirement and develop skill in desktop application development.
4. Use exceptions, threads, collections, logs of Java for the given problem. So they can able design user friendly application.
5. Use graphical user interface using applet in Java programs and able to work in GUI design requirement in industry.
6. Understand and know about basic knowledge of different java framework and able to select future way of interested framework.

Course Outline

UNIT-I **[12 hours]**

Basics of Java: Features of Java, Byte Code and Java Virtual Machine, JDK, Data types, Operator, Control Statements – If, else, nested if, if-else ladders, Switch, while, do-while, for, for-each, break, continue.

Array and String: Single and Multidimensional Array, String class, StringBuffer class, Operations on string, Command line argument, Use of Wrapper Class.

Classes, Objects and Methods: Class, Object, Object reference, Constructor, Constructor Overloading, Method Overloading, Recursion, Passing and Returning object form Method, new operator, this and static keyword, finalize() method, Access control, modifiers, Nested class, Inner class, Anonymous inner class, Abstract class.

UNIT-II **[12 hours]**

Inheritance and Interfaces:

Use of Inheritance, Inheriting Data members and Methods, constructor in inheritance, Multilevel Inheritance – method overriding Handle multilevel constructors – super keyword, Stop Inheritance-Final keywords, Creation and Implementation of an interface, Interface reference, instanceof operator, Interface inheritance, Dynamic method dispatch, Understanding of Java Object Class, Comparison between Abstract Class and interface, Understanding of System.out.println –statement

Package: Use of Package, CLASSPATH, Import statement, Static import, Access control.

Exception Handling: Exception and Error, Use of try, catch, throw, throws and finally, Built in Exception, Custom exception, Throwable Class.

UNIT-III **[12 hours]**

Networking with java.net:

InetAddress class, Socket class, DatagramSocket class, DatagramPacket class.

IO Programming: Introduction to Stream, Byte Stream, Character stream, Readers and Writers, File Class, File InputStream, File Output Stream, InputStreamReader, OutputStreamWriter, FileReader, FileWriter, Buffered Reader.

Collection Classes: List, AbstractList, ArrayList, LinkedList, Enumeration, Vector, Properties, Introuduction to Java.util package.

UNIT-IV **[12 hours]**

Multithreaded Programming: Use of Multithread programming, Thread class and Runnable interface, Thread priority, Thread synchronization, Thread communication, Deadlock.

Generics: Generics Fundamentals, Bounded Types, Using wildcard arguments & bounded wildcards, Generic methods, constructors, class hierarchies & Interfaces.

Applets: Applet basics, complete skeleton, initialization & termination, repainting, using status window & passing parameters to applets.

Method of delivery

Chalk and Board, PowerPoint presentation, self-study material.

Study time

3 Hours/week Theory

2 Hours/week Practical

CO-PO Mapping (PO: Program Outcomes)

1 Program Outcomes (PO's)

Engineering Graduates will be able to:

- **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **PO2. Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- **PO3. Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **PO4. Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis, and interpretation of data, and synthesis of the information to provide valid conclusions.
- **PO5. Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

- **PO6**. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

- **PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

- **PO8. Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

- **PO9. Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

- **PO10. Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

- **PO11. Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

- **PO12. Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change

2. Programme Specific Outcome

- **PSO1. Basics of Computer System**: Should able to understand the principles and working of computer systems. Students can assess the hardware and software aspects of computer systems.

- **PSO2. Program Design**: Design and develop computer programs in the areas related to algorithms, networking, web design, cloud computing, IoT and data analytics.

- **PSO3.Software Development**: Should able to understand the structure and development methodologies of software systems with the use of a various programming languages and open source platforms.

COURSE OUTCOME (CO) and PROGRAM OUTCOME (PO) Matrix

(1-Low, 2-Medium, 3- High)

CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C0 1	2	2	3	-	3	-	-	-	3	-	3	3
C0 2	1	1	2	2	-	-	-	-	-	-	2	-
C0 3	1	2	3	1	2	-	-	-	2	1	2	2
C0 4	-	1	2	2	3	-	-	-	1	-	1	-
C0 5	-	2	2	2	3	-	-	-	-	-	2	2
C0 6	-	2	1	2	2	-	-	-	2	-	2	2
CE0421	**2.0**	**1.66**	**2.16**	**2.0**	**2.16**	**-**	**-**	**-**	**2.0**	**1.0**	**2.0**	**2.25**

COURSE OUTCOME and PROGRAM SPECIFIC OUTCOME Matrix

Core Java Programming(CE0421)

CO	PSO1	PSO2
CO 1	1	3
CO2	-	2
CO 3	2	3
CO 4	2	2
CO 5	3	2
CO 6	2	2
CE0421	**2.0**	**2.33**

Blooms Taxonomy and Knowledge retention (For reference)

(Blooms taxonomy has been given for reference)

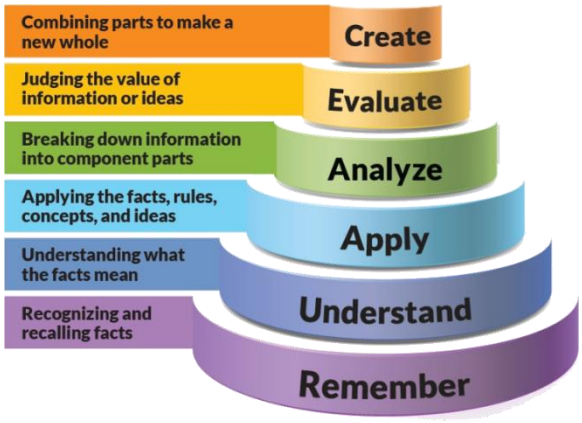


Figure 1: Blooms Taxonomy

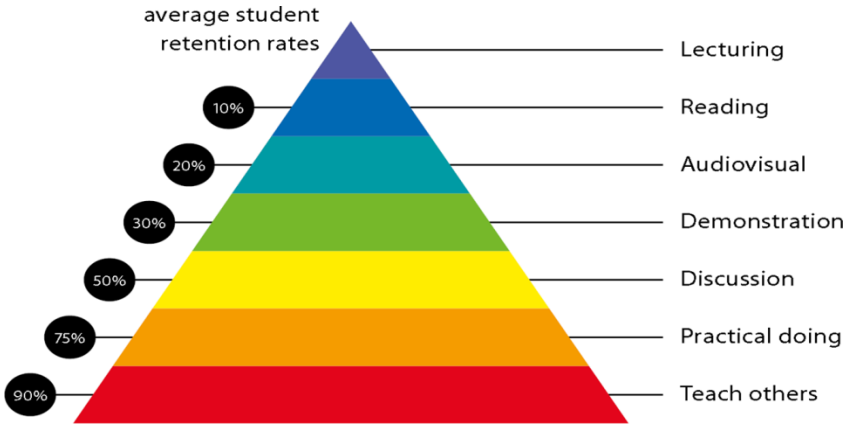


Figure 2: Knowledge retention

Practical work:

Week No.	Class Activity	List of Practical
01	Lab 1	1. Write a program to display "Welcome to Java World". 2. Write a program to find whether the number is prime or not. 3. Write a program to find a greater number among given three numbers using a) ternary operator b) nested if. 4. Write a program to print the Fibonacci series.
02	Lab 2	1. Write a program to find the average of n numbers stored in an Array.
03	Lab 3	1. WAP to replace substring with other substring in the given string. 2. WAP that to sort given strings into alphabetical order. 3. Create a String Buffer with some default string. Append any string to i^{th} position of original string and display the modified string. Also display the reverse of modified string.
04	Lab 4	1.1) WAP that declares a class named Person. It should have instance variables to record name, age and salary. Use new operator to create a Person object. Set and display its instance variables. 1.2) Add a constructor to the Person class developed above. 2. The employee list for a company contains employee code, name, designation and basic pay. The employee is given HRA of 10% of the basic and DA of 45% of the basic pay. The total pay of the employee is calculated as Basic pay+HRA+ DA. Write a class to define the details of the employee. Write a constructor to assign the required initial values. Add a method to calculate HRA, DA and Total pay and print them out. Write another class with a main method. Create objects for three different employees and calculate the HRA, DA and total pay.
05	Lab 5	1. Write a program which defines base class Employee having three data members, namely name[30], emp_numb and gender and two methods namely input_data() and show_data(). Derive a class SalariedEmployee from Employee and adds a new data member, namely salary. It also adds two member methods, namely allowance (if gender is female HRA=0.1 *salary else 0.09* salary. DA= 0.05*salary) and increment (salary= salary+0.1*salary). Display the gross salary in main class. (Tip: Use super to call base class's constructor0).

		2. WAP that illustrates method overriding. Class A3 is extended by Class B3. Each of these classes defines a hello(string s) method that outputs the string "A3: Hello From" or "B3: Hello From" respectively. Use the concept Dynamic Method Dispatch and keyword **super**.
06	Lab 6	1. Write an abstract class shape, which defines abstract method area. Derive class circle from shape. It has data member radius and implementation for area function. Derive class Triangle from shape. It has data members height, base and implementation for area function. Derive class Square from shape. It has data member side and implementation for area function. In main class, use dynamic method dispatch in order to call correct version of method. 2. Create an interface Shape2D which declares a getArea() method. Point 3D contains coordinates of a point. The abstract class Shape declares abstract display() method and is extended by Circle class. it implements the Shape2D interface. The Shapes class instantiates this class and exercises its methods.
07	Lab 7	1. Create a package "employee" and define a Class Employee having three data members, name, emp_num, and gender and two methods- input_data and show_data. Inherit class SalariedEmployee from this class and keep it in package "employee". Add new variable salary and methods allowance (if female hra=0.1* salary else 0.09* salary. DA= 0.05*salary) and increment (salary= salary+0.01 * salary). Calculate gross salary in main class defined in the same package.
08	Lab 8	1. WAP using try catch block. User should enter two command line arguments. If only one argument is entered then exception should be caught. In case of two command line arguments, if fist is divided by second and if second command line argument is 0 then catch the appropriate exception. 2. Define an exception called "NoMatchException" that is thrown when a string is not equal to "India". Write a program that uses this exception.
09	Lab 9	1. The program to creates and run the following three threads. The first thread prints the letter 'a' 100 times. The second thread prints the letter 'b' 100 times. The third thread prints the integer 1 to 100. 2. Write the thread program -1using Runnable interface.
10	Lab 10	1. Write a program that takes two files names (source and destination) as command line argument. Copy source file's content to destination file. Use character stream class. Also do same using byte stream and buffer stream. 2.Write a program which generates random integers and stores them in a file named "rand.dat". The program then reads the integers from the file and displays on the screen.

11	Lab 11	Write the program that demonstrate the use of Stack, Vector and ArrayList classes
12	Lab 12	1. Write a Network program that client sends the data as redius of circle to server and server received that data and send the resultant area of circle to requested client.
13	Lab 13	Write a program to count occurrence of character in a string
Practicals Beyond syllabus		
14	Lab 14	Write a programms to create simple calculator using applet
15	Lab 15	Write programms to create student registration form in applet and store data in database.

Lecture/Tutorial times

Attendance Requirements

The University norms states that it is the responsibility of students to attend all lectures, tutorials, seminars and practical work as stipulated in the course outline. Minimum attendance requirement as per university norms is compulsory for being eligible for semester examinations.

Text Books:

1. Java Fundamentals, A comprehensive introduction by Herbert Schildt, Dale Skrien, McGraw Hill Education, First Edition, 2013, ISBN 13:978125900659

Reference Books:

1) Programming with Java A Primer – E.Balaguruswamy, McGrawhill, 4th Edition, 2009,ISBN- 9780070141698
2) The Complete Reference, Java 2 Herbert Schildt, TMH, 7th Edition, 2007, ISBN: 978-0-07- 163177-8

3) Core Java Volume-I Fundamentals Horstmann & Cornell, - Pearson Education, 8th Edition, 2008, ISBN -9780132354769
4) Object Oriented Modeling and Design with UML Michael Blaha and James Rambaugh – Pearson Publication, 2nd Edition, 2005, ISBN - 9780131968592
5) UML Distilled: A Brief Guide to the Standard Object Modeling Language by Martin Fowle, 3rd Edition, 2004, ISBN -0321193687

Web Resources:

1. OOP, Basic of Java:
   http://www.nptelvideos.com/java/java_video_lectures_tutorials.php
2. Exceptions and Functions:
   http://www.nptelvideos.com/java/java_video_lectures_tutorials.php?pn=1
3. Multithreading:
   http://www.learnerstv.com/Free-Computers-Video-lectures-ltv006-Page1.htm
4. Networking Basics:

   http://nptel.ac.in/courses/106105084/

ASSESSMENT GUIDELINES

Your final course mark will be calculated from the following:

CIE-Theory (60 Marks)		**CIE-Practical (60 Marks)**	
Class Regularity	10 Marks	Minor Project	20 Marks
Assignment	10 Marks	Viva / Quiz	20 Marks
Mid Semester Exam	40 Marks	Practical File + Performance	20 Marks
ESE-Theory (40 Marks)		**ESE-Practical (40 Marks)**	
Total: 100 Marks		**Total: 100 Marks**	

SUPPLEMENTARY ASSESSMENT

Students who receive an overall mark less than 40% in internal component or less than 40% in the end semester will be considered for supplementary assessment in the respective components (i.e internal component or end semester) of semester concerned. Students must make themselves available during the supplementary examination period to take up the respective components (internal component or end semester) and need to obtain the required minimum 40% marks to clear the concerned components.

Practical Work Report/Laboratory Report:

A report on the practical work is due the subsequent week after completion of the class by each group.

Late Work

Late assignments will not be accepted without supporting documentation. Late submission of the reports will result in a deduction of -% of the maximum mark per calendar day

Format

All assignments must be presented in a neat, legible format with all information sources correctly referenced. **Assignment material handed in throughout the session that is not neat and legible will not be marked and will be returned to the student.**

Retention of Written Work

Written assessment work will be retained by the Course coordinator/lecturer for two weeks after marking to be collected by the students.

University and Faculty Policies

Students should make themselves aware of the University and/or Faculty Policies regarding plagiarism, special consideration, supplementary examinations and other educational issues and student matters.

Plagiarism - Plagiarism is not acceptable and may result in the imposition of severe penalties. Plagiarism is the use of another person's work, or idea, as if it is his or her own - if you have any doubts at all on what constitutes plagiarism, please consult your Course coordinator or lecturer. Plagiarism will be penalized severely.

Do not copy the work of other students.

Do not share your work with other students (except where required for a group activity or assessment.

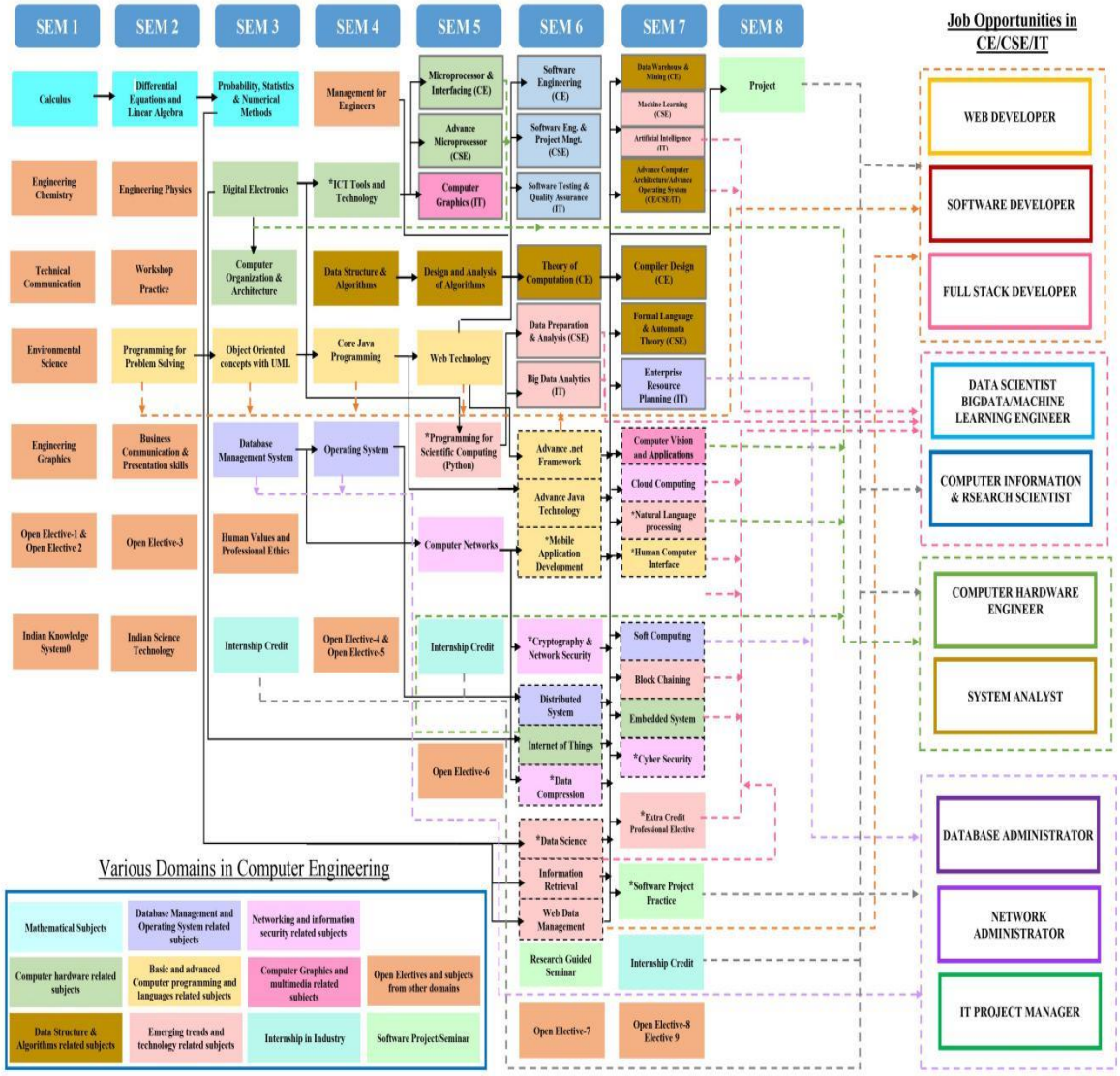
Course schedule

Week #	Topic & contents	CO Addressed	Teaching Learning Activity (TLA)
Week 1	**Basics of Java**: Features of Java, Byte Code and Java Virtual Machine, JDK, Data types, Operator, Control Statements – If , else, nested if, if-else ladders, Switch, while, do-while, for, for-each, break, continue.	I	Chalk & Board, Discussion
Week 2	**Array and String**: Single and Multidimensional Array, String class, StringBuffer class, Operations on string, Command line argument, Use of Wrapper Class.	I	Presentation, Chalk & Board
Week 3	**Classes, Objects and Methods**: Class, Object, Object reference, Constructor, Constructor Overloading, Method Overloading, Recursion, Passing and Returning object form Method, new operator, this and static keyword, finalize() method, Access control, modifiers, Nested class, Inner class, Anonymous inner class, Abstract class.	I	Presentation, Chalk & Board
Week 4	**Inheritance and Interfaces**: Use of Inheritance, Inheriting Data members and Methods, constructor in inheritance, Multilevel Inheritance –	II	Presentation, Chalk & Board

		method overriding Handle multilevel constructors – super keyword, Stop Inheritance - Final keywords, Creation and Implementation of an interface, Interface reference, instanceof operator, Interface inheritance, Dynamic method dispatch ,Understanding of Java Object Class, Comparison between Abstract Class and interface, Understanding of System.out.println –statement		
	Week 5	**Package**: Use of Package, CLASSPATH, Import statement, Static import, Access control.	II	Presentation, Chalk & Board
	Week 6	**Exception Handling**: Exception and Error, Use of try, catch, throw, throws and finally, Built in Exception, Custom exception, Throwable Class.	IV	Model presentation
	Week 7	**Networking with java.net**: InetAddress class, Socket class, DatagramSocket class, DatagramPacket class.	III	Presentation, Chalk & Board, Demonstration
	Week 8	**IO Programming**: Introduction to Stream, Byte Stream, Character stream, Readers and Writers, File Class, File InputStream, File Output Stream, InputStreamReader, OutputStreamWriter, FileReader, FileWriter, Buffered Reader.	III,IV	Presentation, Chalk & Board, Demonstration

	Week 9	**Collection Classes**: List, AbstractList, ArrayList, LinkedList, Enumeration, Vector, Properties, Introuduction to Java.util package.	II	Presentation, Chalk & Board
	Week 10	**Multithreaded Programming**: Use of Multithread programming, Thread class and Runnable interface, Thread priority, Thread synchronization, Thread communication, Deadlock.	IV	Presentation, Chalk & Board
	Week 11	**Generics**: Generics Fundamentals, Bounded Types, Using wildcard arguments & bounded wirldcards, Generic methods, constructors, class hierarchies & Interfaces.	IV	Presentation, Chalk & Board
	Week 12	**Applets**: Applet basics, complete skeleton, initialization & termination, repainting, Using status window & passing parameters to applets.	V,VI	Presentation, Chalk & Board

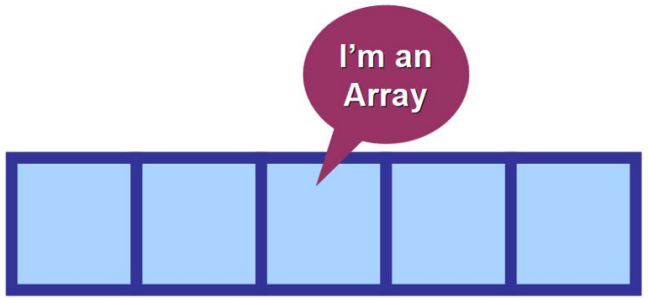
COMPUTER ENGINEERING DEPARTMENT COURSE DEPENDANCY CHART



Array

Why Array?

- Very often we need to deal with relatively large set of data.
- E.g.
  - Percentage of all the students of the college. (May be in thousands)
  - Age of all the citizens of the city. (May be lakhs)
- We need to declare thousands or lakhs of the variable to store the data which is practically not possible.
- We need a solution to store more data in a single variable.
- **Array** is the most appropriate way to handle such data.
- As per English Dictionary, "***Array means collection or group or arrangement in a specific order.***"



Array

- **An array is a fixed size sequential collection of elements of same data type grouped under single variable name.**

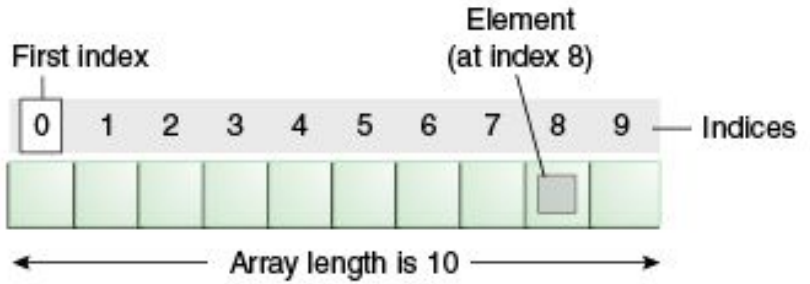
```
int percentage[10];
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Fixed Size	Sequential	Same Data type	Single Variable Name
The size of an array is fixed at the time of declaration which cannot be changed later on. Here **array size** is **10**.	All the elements of an array are stored in a consecutive blocks in a memory. **10** (0 to 9)	Data type of all the elements of an array is same which is defined at the time of declaration. Here **data type** is **int**	All the elements of an array will be referred through common name. Here **array name** is **percentage**

Array declaration

- Normal Variable Declaration: int **a**;
- Array Variable Declaration: int **b**[10];
- Individual value or data stored in an array is known as an **element of an array**.
- Positioning / indexing of an elements in an array always **starts with 0 not 1**.
  - If 10 elements in an array then index is 0 to 9
  - If 100 elements in an array then index is 0 to 99
  - If 35 elements in an array then index is 0 to 34
- Variable **a** stores 1 integer number where as variable **b** stores 10 integer numbers which can be accessed as b[0], b[1], b[2], b[3], b[4], b[5], b[6], b[7], b[8] and b[9]



Array

- Important point about Java array.
  - An array is **derived** datatype.
  - An array is **dynamically** allocated.
  - The individual elements of an array is refereed by their **index/subscript** value.
  - The **subscript** for an array always begins with **0**.

35	13	28	106	35	42	5	83	97	14
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

One-Dimensional Array

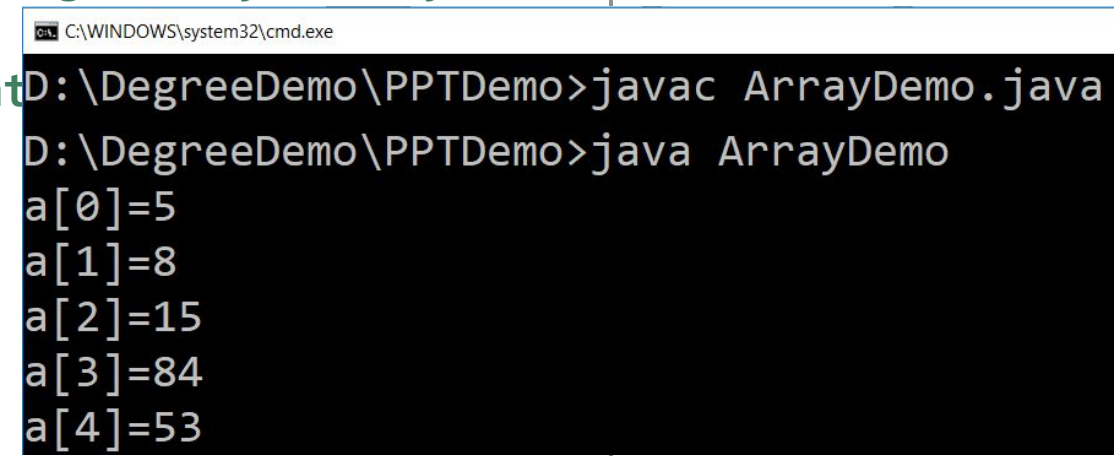
- An array using **one subscript** to represent the **list of elements** is called **one dimensional array**.
- A One-dimensional array is essentially a **list** of **like-typed variables.**
- Array declaration: type var-name[];

  Example: int student_marks[];
- Above example will represent array with no value (null).
- To link **student_marks** with actual array of integers, we must allocate one using ***new*** keyword.

  Example: int student_marks[] = ***new*** int[20];

Example (Array)

```
public class ArrayDemo{
    public static void main(String[] args) {
        int a[]; // or int[] a
        // till now it is null as it does not assigned any memory

        a = new int[5]; // here we actually creat
        a[0] = 5;
        a[1] = 8;
        a[2] = 15;
        a[3] = 84;
        a[4] = 53;

        /* in java we use length property to determine the length
         * of an array, unlike c where we used sizeof function */
        for (int i = 0; i < a.length; i++) {
            System.out.println("a["+i+"]="+a[i]);
        }
    }
}
```



WAP to store 5 numbers in an array and print them

```
1.  import java.util.*;
2.  class ArrayDemo1{
3.  public static void main (String[] args){
4.        int i, n;
5.        int[] a=new int[5];
6.        Scanner sc = new Scanner(System.in);
7.        System.out.print("enter Array Length:");
8.        n = sc.nextInt();
9.        for(i=0; i<n; i++) {
10.         System.out.print("enter a["+i+"]:");
11.         a[i] = sc.nextInt();
12.       }
13.       for(i=0; i<n; i++)
14.           System.out.println(a[i]);
15.   }
16. }
```

Output:

```
enter Array
Length:5
enter a[0]:1
enter a[1]:2
enter a[2]:4
enter a[3]:5
enter a[4]:6
1
2
4
5
6
```

WAP to print elements of an array in reverse order

```
1.  import java.util.*;
2.  public class RevArray{
3.  public static void  main(String[] args) {
4.    int i, n;
5.    int[] a;
6.    Scanner sc=new Scanner(System.in);
7.    System.out.print("Enter Size of an Array:");
8.    n=sc.nextInt();
9.    a=new int[n];
10.   for(i=0; i<n; i++){
11.       System.out.print("enter a["+i+"]:");
12.       a[i]=sc.nextInt();
13.   }
14.   System.out.println("Reverse Array");
15.   for(i=n-1; i>=0; i--)
16.       System.out.println(a[i]);
17.  }
18. }
```

Output:

```
Enter Size of an
Array:5
enter a[0]:1
enter a[1]:2
enter a[2]:3
enter a[3]:4
enter a[4]:5
Reverse Array
5
4
3
2
1
```

WAP to count positive number, negative number and zero from an array of n size

```
1.  import java.util.*;
2.  class ArrayDemo1{
3.  public static void main (String[] args){
4.       int n,pos=0,neg=0,z=0;
5.       int[] a=new int[5];
6.       Scanner sc = new Scanner(System.in);
7.       System.out.print("enter Array Length:");
8.       n = sc.nextInt();
9.       for(int i=0; i<n; i++) {
10.           System.out.print("enter a["+i+"]:");
11.           a[i] = sc.nextInt();
12.           if(a[i]>0)
13.             pos++;
14.           else if(a[i]<0)
15.             neg++;
16.           else
17.             z++;
18.      }
19.      System.out.println("Positive no="+pos);
20.      System.out.println("Negative no="+neg);
21.      System.out.println("Zero no="+z);
22. }}
```

Output:

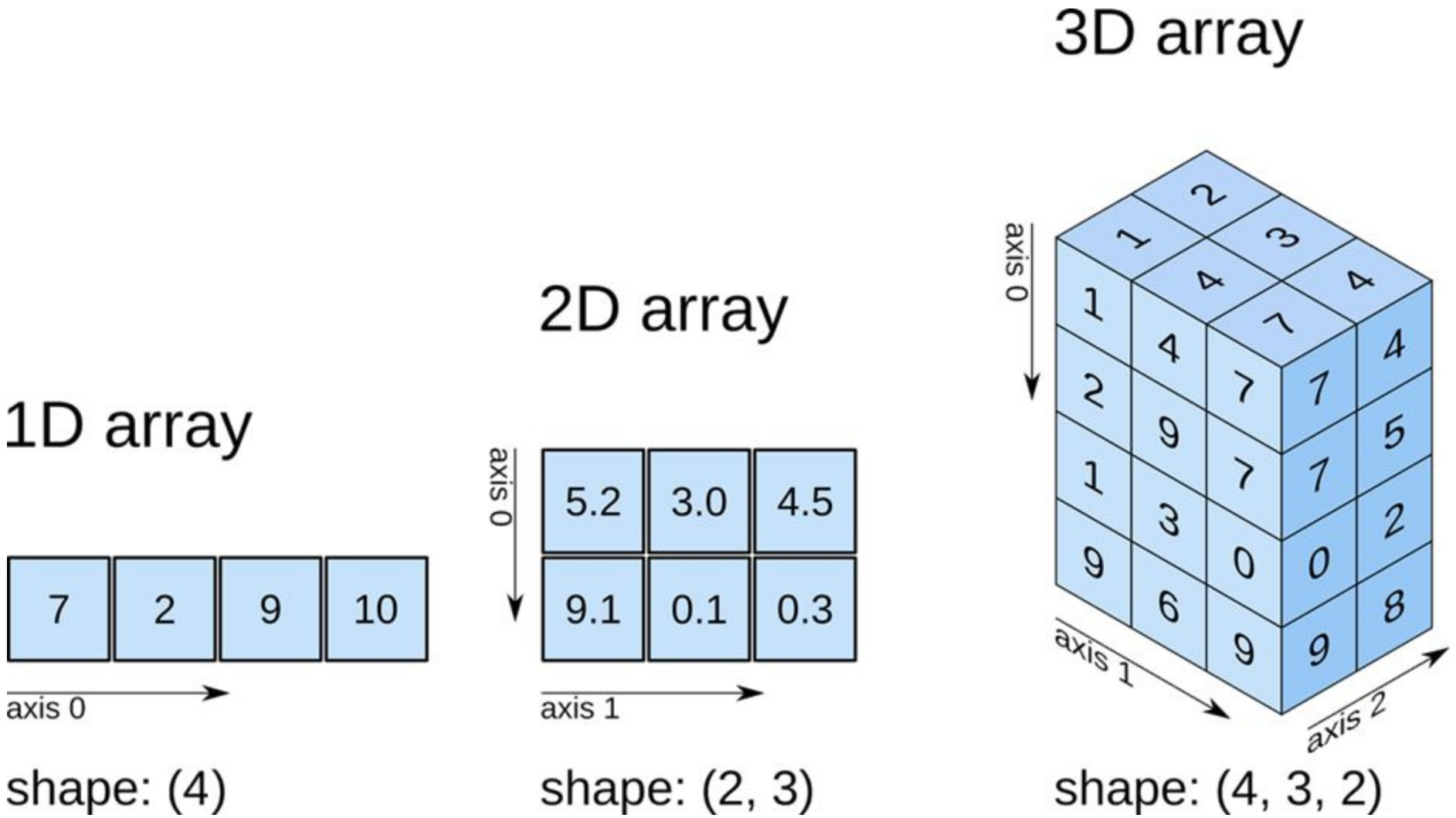
```
enter Array
Length:5
enter a[0]:-3
enter a[1]:5
enter a[2]:0
enter a[3]:-2
enter a[4]:00
Positive no=1
Negative no=2
Zero no=2
```

Exercise: Array

1. WAP to count odd and even elements of an array.
2. WAP to calculate sum and average of n numbers from an array.
3. WAP to find largest and smallest from an array.

Multidimensional Array

Multidimensional Array



WAP to read 3 x 3 elements in 2d array

```
1.  import java.util.*;
2.  class Array2Demo{
3.  public static void main(String[] args) {
4.    int size;
5.    Scanner sc=new Scanner(System.in);
6.    System.out.print("Enter size of an array");
7.    size=sc.nextInt();
8.    int a[][]=new int[size][size];
9.    for(int i=0;i<a.length;i++){
10.     for(int j=0;j<a.length;j++){
11.         a[i][j]=sc.nextInt();
12.     }
13.   }

14.   for(int i=0;i<a.length;i++){
15.     for(int j=0;j<a.length;j++){
16.          System.out.print("a["+i+"]["+j+"]:"+a[i][j]+"
17.     }
18.     System.out.println();
19.   }
20.  }
21. }
```

	Column-0	Column-1	Column-2
Row-0	**11**	**18**	**-7**
Row-1	**25**	**100**	**0**
Row-2	**-4**	**50**	**88**

Output:

```
11
12
13
14
15
16
17
18
19
a[0][0]:11      a[0][1]:12      a[0][2]:13
a[1][0]:14      a[1][1]:15      a[1][2]:16
a[2][0]:17      a[2][1]:18      a[2][2]:19
```

WAP to perform addition of two 3 x 3 matrices

```
1. import java.util.*;
2. class Array2Demo{
3. public static void main(String[] args) {
4. int size;
5. int a[][],b[][],c[][];
6. Scanner sc=new Scanner(System.in);
7.  System.out.print("Enter size of an
          array:");
8.  size=sc.nextInt();
9.  a=new int[size][size];
0.  System.out.println("Enter array
            elements:");
1. for(int i=0;i<a.length;i++){
2.     for(int j=0;j<a.length;j++){
3.        System.out.print("Enter
     a["+i+"]["+j+"]:");
4.        a[i][j]=sc.nextInt();
5.     }
6. }
```

```
1. b=new int[size][size];
2. for(int i=0;i<b.length;i++){
3.  for(int j=0;j<b.length;j++){
4.    System.out.print("Enter
     b["+i+"]["+j+"]:");
5.    b[i][j]=sc.nextInt();
6.  }
7. }
8. c=new int[size][size];
9. for(int i=0;i<c.length;i++){
10.  for(int j=0;j<c.length;j++){
11.  System.out.print("c["+i+"]["+j+"]:“
        +(a[i][j]+b[i][j])+"\t");
12.   }
13.   System.out.println();
14.   }
15.  }//main()
16. }//class
```

WAP to perform addition of two 3 x 3 matrices

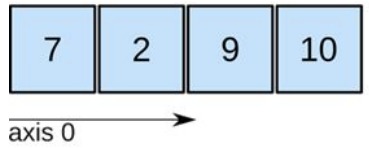
Output:

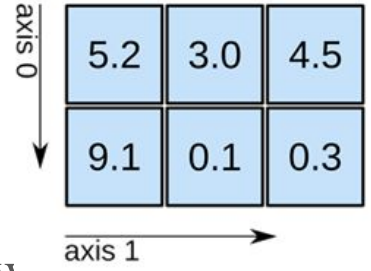
```
Enter size of an array:3
Enter array elements:
Enter a[0][0]:1
Enter a[0][1]:1
Enter a[0][2]:1
Enter a[1][0]:1
Enter a[1][1]:1
Enter a[1][2]:1
Enter a[2][0]:1
Enter a[2][1]:1
Enter a[2][2]:1
Enter b[0][0]:4
Enter b[0][1]:4
Enter b[0][2]:4
Enter b[1][0]:4
Enter b[1][1]:4
Enter b[1][2]:4
Enter b[2][0]:4
Enter b[2][1]:4
Enter b[2][2]:4
c[0][0]:5       c[0][1]:5       c[0][2]:5
c[1][0]:5       c[1][1]:5       c[1][2]:5
c[2][0]:5       c[2][1]:5       c[2][2]:5
```

Initialization of an array elements

1. One dimensional Array

1. int a[5] = { 7, 3, -5, 0, 11 }; // a[0]=7, a[1] = 3, a[2] = -5, a[3] = 0, a[4] = 11
2. int a[5] = { 7, 3 }; // a[0] = 7, a[1] = 3, a[2], a[3] and a[4] are 0
3. int a[5] = { 0 }; // all elements of an array are initialized to 0





2. Two dimensional Array

1. int a[2][4] = { { 7, 3, -5, 10 }, { 11, 13, -15, 2} }; // 1st row is 7, 3, -5, 10 & 2nd row is 11, 13, -15, 2
2. int a[2][4] = { 7, 3, -5, 10, 11, 13, -15, 2 }; // 1st row is 7, 3, -5, 10 & 2nd row is 11, 13, -15, 2
3. int a[2][4] = { { 7, 3 }, { 11} }; // 1st row is 7, 3, 0, 0 & 2nd row is 11, 0, 0, 0
4. int a[2][4] = { 7, 3 }; // 1st row is 7, 3, 0, 0 & 2nd row is 0, 0, 0, 0
5. int a[2][4] = { 0 }; // 1st row is 0, 0, 0, 0 & 2nd row is 0, 0, 0, 0

Multi-Dimensional Array

- In java, multidimensional array is actually **array of arrays.**
- **Example**: int runPerOver[][] = new int[3][6];

First Over (a[0])	**4** a[0][0]	**0** a[0][1]	**1** a[0][2]	**3** a[0][3]	**6** a[0][4]	**1** a[0][5]
Second Over (a[1])	**1** a[1][0]	**1** a[1][1]	**0** a[1][2]	**6** a[1][3]	**0** a[1][4]	**4** a[1][5]
Third Over (a[2])	**2** a[2][0]	**1** a[2][1]	**1** a[2][2]	**0** a[2][3]	**1** a[2][4]	**1** a[2][5]

- **`length` field:**
  - If we use length field with multidimensional array, it will return length of first dimension.
  - Here, if **runPerOver.length** is accessed it will return **3**
  - Also if **runPerOver[0].length** is accessed it will be **6**

Multi-Dimensional Array (Example)

```
Scanner s = new Scanner(System.in);
int runPerOver[][] = new int[3][6];
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 6; j++) {
        System.out.print("Enter Run taken" +
        " in Over numner " + (i + 1) +
        " and Ball number " + (j + 1) + " = ");
        runPerOver[i][j] = s.nextInt();
    }
}
int totalRun = 0;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 6; j++) {
        totalRun += runPerOver[i][j];
    }
}
double average = totalRun / (double) runPerOver.length;
System.out.println("Total Run = " + totalRun);
System.out.println("Average per over = " + average);
```

```
C:\WINDOWS\system32\cmd.exe
Enter Run taken in Over numner 1 and Ball number 1 = 4
Enter Run taken in Over numner 1 and Ball number 2 = 0
Enter Run taken in Over numner 1 and Ball number 3 = 1
Enter Run taken in Over numner 1 and Ball number 4 = 3
Enter Run taken in Over numner 1 and Ball number 5 = 6
Enter Run taken in Over numner 1 and Ball number 6 = 1
Enter Run taken in Over numner 2 and Ball number 1 = 1
Enter Run taken in Over numner 2 and Ball number 2 = 1
Enter Run taken in Over numner 2 and Ball number 3 = 0
Enter Run taken in Over numner 2 and Ball number 4 = 6
Enter Run taken in Over numner 2 and Ball number 5 = 0
Enter Run taken in Over numner 2 and Ball number 6 = 4
Enter Run taken in Over numner 3 and Ball number 1 = 2
Enter Run taken in Over numner 3 and Ball number 2 = 1
Enter Run taken in Over numner 3 and Ball number 3 = 1
Enter Run taken in Over numner 3 and Ball number 4 = 0
Enter Run taken in Over numner 3 and Ball number 5 = 1
Enter Run taken in Over numner 3 and Ball number 6 = 1
Total Run = 33
Average per over = 11.0
```

Multi-Dimensional Array (Cont.)

- **manually** allocate **different** size:

```
int runPerOver[][] = new int[3][];
runPerOver[0] =  new int[6];
runPerOver[1]  =  new int[7];
runPerOver[2]  =  new int[6];
```

- **initialization**:

```
int runPerOver[][] = {
                {0,4,2,1,0,6},
                {1,-1,4,1,2,4,0},
                {6,4,1,0,2,2},
            }
```

Note : here to specify extra runs (Wide, No Ball etc..) negative values are used

Jagged Array

- A jagged array is an array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D array but with a variable number of columns in each row. These types of arrays are also known as Jagged arrays.

- Advantages of Jagged array:
  - **Dynamic allocation:** Jagged arrays allow you to allocate memory dynamically, meaning that you can specify the size of each sub-array at runtime, rather than at compile-time.
  - **Space utilization:** Jagged arrays can save memory when the size of each sub-array is not equal. In a rectangular array, all sub-arrays must have the same size, even if some of them have unused elements. With a jagged array, you can allocate just the amount of memory that you need for each sub-array.
  - **Flexibility:** Jagged arrays can be useful when you need to store arrays of different lengths or when the number of elements in each sub-array is not known in advance.
  - **Improved performance:** Jagged arrays can be faster than rectangular arrays for certain operations, such as accessing elements or iterating over sub-arrays, because the memory layout is more compact.

Jagged Array

```
import java.util.*;
class demo
{
	public static void main(String[] args)
	{
		int a[][] = new int[4][];
		a[0] = new int[4];
		a[1] = new int[2];
		a[2] = new int[1];
		a[3] = new int[3];
		Scanner sc = new Scanner(System.in);

		for(int i=0;i<4;i++)
		{
			for(int j=0;j<a[i].length;j++)
			{
System.out.printf("\nEnter the value for a[%d][%d]",i,j);
a[i][j]=sc.nextInt();
			}
		}
System.out.println("Entered array\n");

			for(int i=0;i<4;i++)
			{
				for(int j=0;j<a[i].length;j++)
				{
					System.out.print("\t"+a[i][j]);
				}
				System.out.println("");
			}

	}
}
```

```
Enter the value for a[0][0]1
Enter the value for a[0][1]2
Enter the value for a[0][2]3
Enter the value for a[0][3]4
Enter the value for a[1][0]5
Enter the value for a[1][1]6
Enter the value for a[2][0]7
Enter the value for a[3][0]8
Enter the value for a[3][1]9
Enter the value for a[3][2]0
Entered array
```

1	2	3	4
5	6		
7			
8	9	0	

length vs length()

Java length Comparison Chart

Java length() method	Java length property
Primarily used by the String class	Used by Java arrays
Must have round brackets at the end	Must not have round brackets at the end
Syntax: stringVariable.length();	Syntax: arrayVariable.length;
Returns the number of characters in a String	Returns the size and capacity of an array
The 'length cannot be resolved' compile error can occur on misuse	The 'cannot invoke length() on array' compile error can occur on misuse

Searching in Array

Searching in Array

- Searching is the process of looking for a specific element in an array. for example, discovering whether a certain element is included in the array.
- Searching is a common task in computer programming. Many algorithms and data structures are devoted to searching.
- We will discuss two commonly used approaches,
  - **Linear Search:** The linear search approach compares the key element key sequentially with each element in the array. It continues to do so until the key matches an element in the array or the array is exhausted without a match being found.
  - **Binary Search:** The binary search first compares the key with the element in the middle of the array. Consider the following three cases:
    - If the key is less than the middle element, you need to continue to search for the key only in the first half of the array.
    - If the key is equal to the middle element, the search ends with a match.
    - If the key is greater than the middle element, you need to continue to search for the key only in the second half of the array.

  Note: Array should be sorted in ascending order if we want to use Binary Search.

Linear Search

```
1. import java.util.*;
2. class LinearSearchDemo{
3. public static void main(String[] args) {
4.     int size;
5.     int a[]={1,2,3,4,5,6,7,8,9};
6.     int search;
7.     boolean flag=false;
8.     Scanner sc=new Scanner(System.in);
9.     System.out.print("Enter element to search");
10.    search=sc.nextInt();
11.       for(int i=0;i<a.length;i++){
12.           if(a[i]==search){
13.           System.out.println("element found at "+i+"th index");
14.           flag=true;
15.           break;
16.           }
17.       }
18.     if(!flag)
19.             System.out.println("element NOT found!");
20.    }
21. }
```

Output:

```
Enter element to search 6
element found at 5th index
Enter element to search 35
element NOT found!
```

Binary Search (Animation)

0	2	5	10	32	51	52	56	57	61	64	76	79	84	87	91

Binary Search

```
1.import java.util.*;
2.class BinaryDemo{
3.public static void main(String[]
            args){
4.int size;
5.int a[]={1,2,3,4,5,6,7,8,9};
6.int search;
7.boolean flag=false;
8.Scanner sc=new Scanner(System.in);
9.System.out.print("Enter element to
                search:");
                extInt();
```

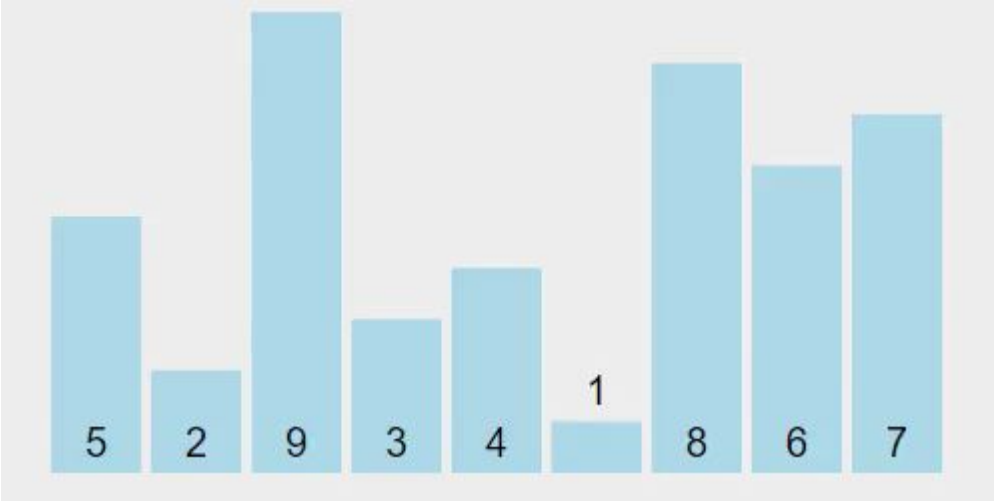
Output:

```
Enter element to search:5
element found at 4 index

Enter element to search:9
element found at 8 index

Enter element to search:56
element not found
```

```
13.while(high>=low){
14.     int mid=(high+low)/2;
15.     if(search==a[mid]){
16.       flag=true;
17.       System.out.println("element found
                  at "+mid+" index ");
18.       break;
19.     }
20.     else if(search<a[mid]){
21.      high=mid-1;
22.     }
23.     else if(search>a[mid]){
24.      low=mid+1;
25.     }
26.}//while
27.if(!flag)
28. System.out.println("element not found");
29.  }
30.}
```

Sorting Array

- Sorting, like searching, is a common task in computer programming. Many different algorithms have been developed for sorting.
- There are many sorting techniques available, we are going to explore selection sort.
- Selection sort
  - finds the smallest number in the list and swaps it with the first element.
  - It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains.



Selection Sort (Example)

```
1.import java.util.*;
2.class SelectionSearchDemo{
3. public static void main(String[] args) {
4. int a[]={ 5, 2, 9, 3, 4, 1, 8, 6, 7 };
```

```
5.for (int i = 0; i < a.length - 1; i++) {
6.// Find the minimum in the list[i..a.length-1]
7.  int min = a[i];
8.  int minIndex = i;
9.  for (int j = i + 1; j < a.length; j++) {
10.   if (min > a[j]) {
11.     min = a[j];
12.     minIndex = j;
13.   }
14.  }//inner for loop j
15.// Swap a[i] with a[minIndex]
16.  if (minIndex != i) {
17.   a[minIndex] = a[i];
18.   a[i] = min;
19.   }
20. }//outer for i
21. for(int temp: a) { // this is foreach loop
22.   System.out.print(temp + ", ");
23.   }
24.  }//main()
25.}//class
```

Output:

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

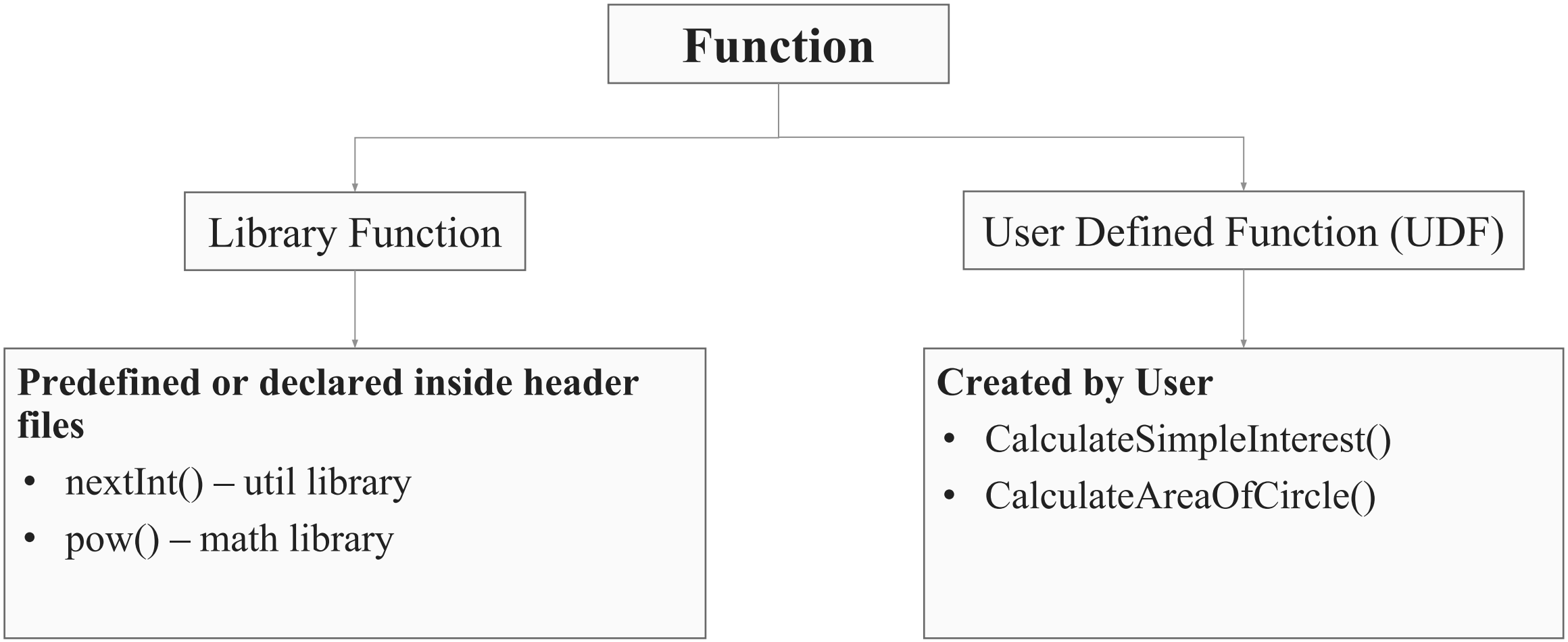
Method

What is Method?

- A **method** is a group of statements that performs a specific task.
- A large program can be divided into the basic building blocks known as **method/function**.
- The function contains the set of programming statements enclosed by { }.
- Program execution in many programming language starts from the **main** function.
- main is also a **method/function**.

```
void main()
{
    // body part
}
```

Types of Function



Program Structure of Function

- User-defined function's program structure is divided into three parts as follows:

C:Function Structure

```
void func1();


void main()
{
   ....
   func1();
}

void func1()
{
   ....
   //function body
   ....
}
```

Function Prototype

Function call

Function definition

Java:Function for addition of 2 numbers

```
class MethodDemo{
 public static void main(String[] ags){
    int a=10,b=20,c;
    c=add(a,b);
    System.out.println("a+b="+c);
 }
 static int add(int i, int j){
    return i+j;
 }
}
```

Method Definition

- A method definition defines the **method header** and **body**.
- A **method body** part defines method logic.
  - Method statements

Syntax

```
return-type method_name(datatyp1 arg1, datatype2 arg2, …)
{
    functions statements
}
```

Example

```
int addition(int a, int b);
{
    return a+b;
}
```

WAP to add two number using add(int, int) Function

```
1.  class MethodDemo{
2.   public static void main(String[] args) {
3.    int a=10,b=20,c;
4.    MethodDemo md=new MethodDemo();
5.    c=md.add(a,b);
6.    System.out.println("a+b="+c);
7.   }//main()
8.   int add(int i, int j){
9.        return i+j;
10.  }
11. }
```

Output:

```
a+b=30
```

Actual parameters v/s Formal parameters

- Values that are passed from the calling functions are known **actual parameters**.
- The variables declared in the function prototype or definition are known as **formal parameters**.
- Name of formal parameters can be same or different from actual parameters.
- Sequence of parameter is **important**, not name.

Actual Parameters

```
int a=10,b=20,c;
MethodDemo md=new MethodDemo();
c=md.add(a,b);
// a and b are the actual parameters.
```

Formal Parameters

```
int add(int i, int j)
{
        return i+j;
}
// i and j are the formal parameters.
```

Return Statement

- The function can **return only one value**.
- Function cannot return more than one value.
- If function is not returning any value then return type should be **`void`**.

Actual Parameters

```
int a=10,b=20,c;
MethodDemo md=new MethodDemo();
c=md.sub(a,b);
// a and b are the actual parameters.
```

Formal Parameters

```
int sub(int i, int j)
{
    return i - j;
}
// i and j are the formal parameters.
```

WAP to calculate the Power of a Number using method

```
1.import java.util.*;
2.public class PowerMethDemo1{
3.public static void main(String[] args){
4.   int num, pow, res;
5.   Scanner sc=new Scanner(System.in);
6.   System.out.print("enter num:");
7.   num=sc.nextInt();
8.   System.out.print("enter pow:");
9.   pow=sc.nextInt();
10.  PowerMethDemo1 pmd=new
 PowerMethDemo1();
11.  res = pmd.power(num, pow);
12.  System.out.print("ans="+res);
13.  } //main()
```

```
14.      int power(int a, int b){
15.       int i, r = 1;
16.       for(i=1; i<=b; i++)
17.       {
18.          r = r * a;
19.       }
20.       return r;
21.     }//power()
22.  }//class
```

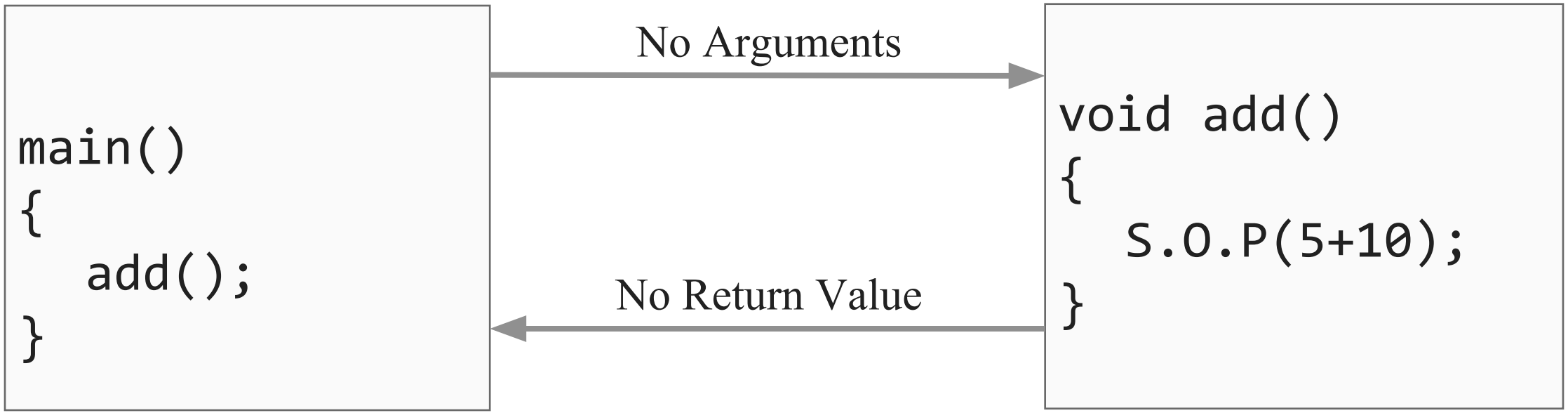
Output:

```
enter num:5
enter pow:3
ans=125
```

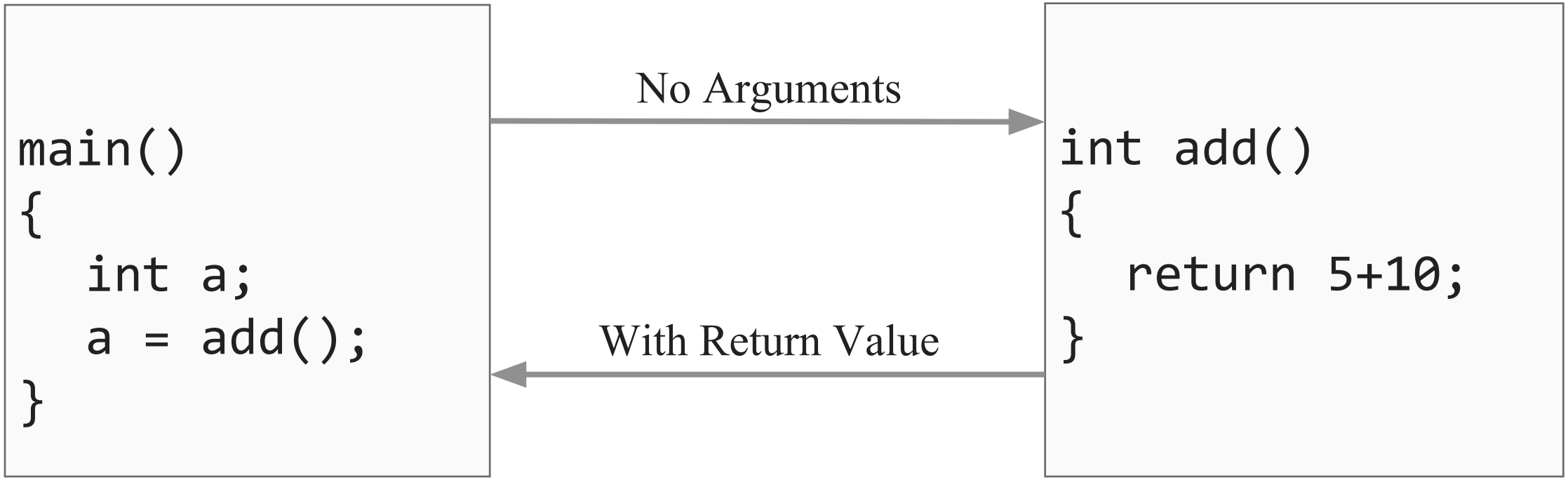
Types of Methods(Method Categories)

- Functions can be divided in 4 categories based on arguments and return value.
  1. Method without arguments and without return value `void add();`
  2. Method without arguments and with return value `int add();`
  3. Method with arguments and without return value `void add(int, int);`
  4. Method with arguments and with return value `int add(int, int);`

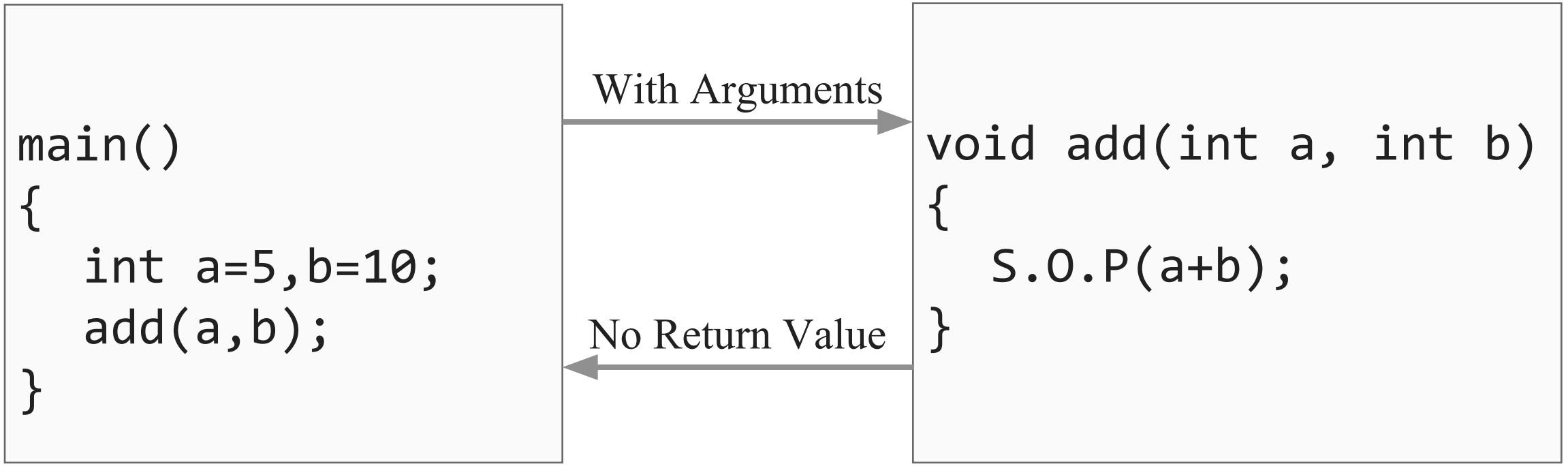
Method without arguments and without return value



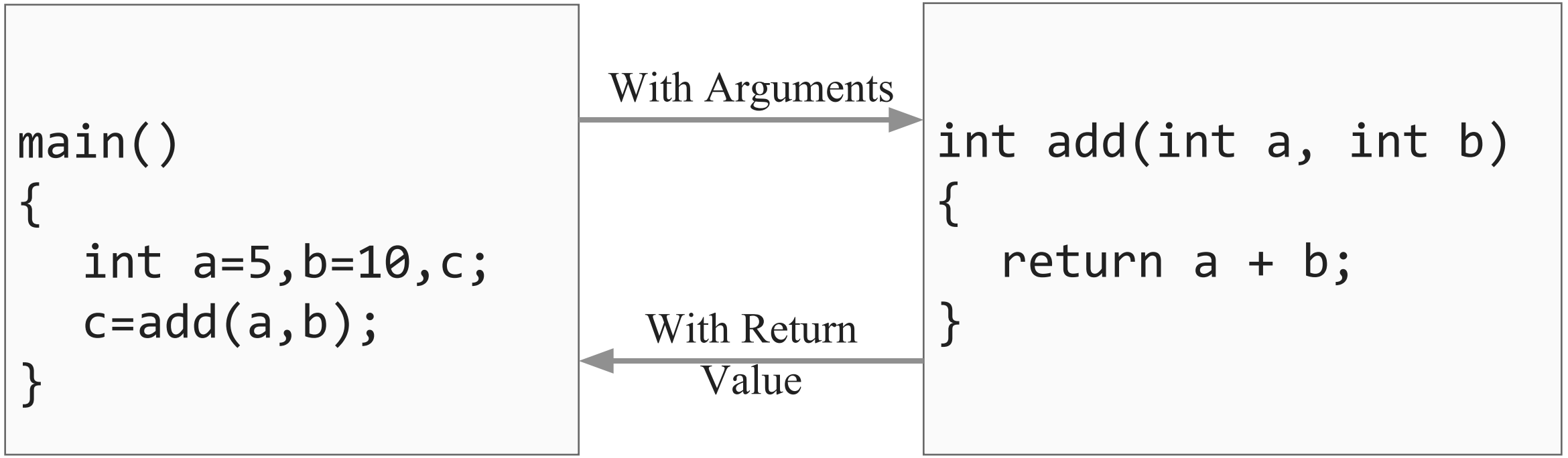
Method without arguments and with return value



Method with arguments and without return value



Method with arguments and with return value



Method Overloading

Method Overloading: Compile-time Polymorphism

- **Definition**: When two or more methods are implemented that share same name but different parameter(s), the methods are said to be ***overloaded***, and the process is referred to as ***method overloading***
- Method overloading is one of the ways that Java implements polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
  - E.g.
```
public void draw()
public void draw(int height, int width)
public void draw(int radius)
```
- Thus, overloaded methods must differ in the type and/or number of their parameters.
- While in overloaded methods with different return types and same name & parameter are not allowed ,as the return type alone is insufficient for the compiler to distinguish two versions of a method.

Method Overloading: Compile-time Polymorphism

```
1.  class Addition{
2.  int i,j,k;
3.    void add(int a){
4.    i=a;
5.    System.out.println("add i="+i);
6.    }
7.    void add(int a,int b){\\overloaded add()
8.    i=a;
9.    j=b;
10.   System.out.println("add i+j="+(i+j));
11.   }
12.   void add(int a,int b,int c){\\overloaded add()
13.   i=a;
14.   j=b;
15.   k=c;
16.   System.out.println("add i+j+k="+(i+j+k));
17.   }
18. }
```

```
19.  class OverloadDemo{
20.  public static void
     main(String[] args){
21.    Addition a1= new Addition();
22.    //call all versions of add()
23.    a1.add(20);
24.    a1.add(30,50);
25.    a1.add(10,30,60);
26.    }
27.  }
```

Output

```
add i=20
add i+j=80
add i+j+k=100
```

Method Overloading: Compile-time Polymorphism

```
1.  class Addition{
2.  int i,j,k;
3.    void add(int a){
4.   i=a;
5.   System.out.println("add i="+i);
6.    }
7.    void add(int a,int b){\\overloaded add()
8.   i=a;
9.   j=b;
10.  System.out.println("add i+j="+(i+j));
11.   }
12.   void add(double a, double b){\\overloaded add()
13.  System.out.println("add a+b="+(a+b));
14.   }
15.   void add(int a,int b,int c){\\overloaded add()
16.  i=a;
17.  j=b;
18.  k=c;
19.  System.out.println("add i+j+k="+(i+j+k));
20.   }
21. }
```

```
22.   class OverloadDemo{
23.  public static void
     main(String[] args){
24.    Addition a1= new Addition();
25.    //call all versions of add()
26.   a1.add(20);
27.   a1.add(30,50);
28.   a1.add(10,30,60);
29.   a1.add(30.5,50.67);
30.   }
31.  }
```

Output

```
add i=20
add i+j=80
add i+j+k=100
add a+b=81.17
```

Method Overloading: Points to remember

- Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm.
- Overloading **increases** the readability of the program.
- There are two ways to overload the method in java
  1. By changing number of arguments
  2. By changing the data type
- In java, method overloading is not possible by changing the return type of the method only because of ambiguity.

Method Overloading: Points to remember

Can we overload java main() method?

- Yes, by method overloading. We can have any number of main methods in a class by method overloading
- But JVM calls main() method which receives string array as arguments only.

Advantages of Method

- Reduced Code Redundancy
  - Rewriting the same logic or code again and again in a program can be avoided.
- Reusability of Code
  - Same function can be call from multiple times without rewriting code.
- Reduction in size of program
  - Instead of writing many lines, just function need to be called.
- Saves Development Time
  - Instead of changing code multiple times, code in a function need to be changed.
- More Traceability of Code
  - Large program can be easily understood or traced when it is divide into functions.
- Easy to Test & Debug
  - Testing and debugging of code for errors can be done easily in individual function.

Scope, Lifetime and Visibility of a Variable

Scope of a Variable

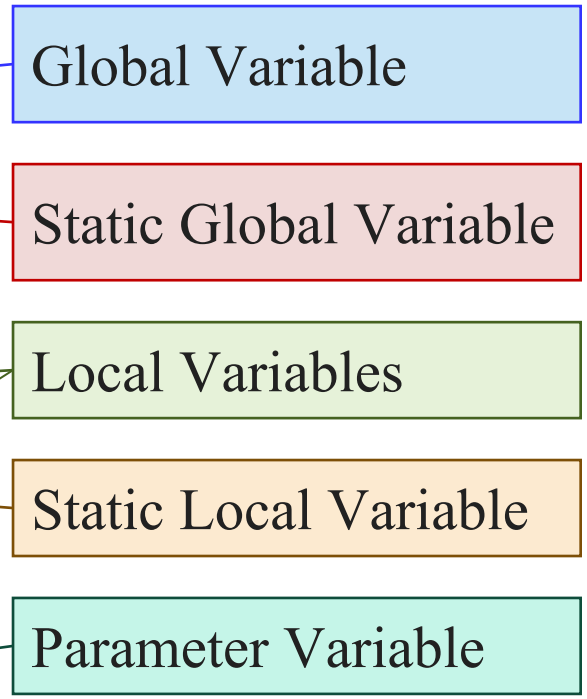
- Whenever we declare a variable, we also determine its scope, lifetime and visibility.

Scope	Scope is defined as the area in which the declared variable is 'accessible'. **There are five scopes**: program, file, function, block, and class. ***Scope*** is the region or section of code where a variable can be accessed. Scoping has to do with **when** a variable is accessible and used.
Lifetime	The lifetime of a variable is the period of time in which the variable is allocated a space (i.e., the period of time for which it "lives"). There are three lifetimes in C: static, automatic and dynamic. ***Lifetime*** **is the time duration where an object/variable is in a valid state.** Lifetime has to do with when a variable is created and destroyed
Visibility	Visibility is the "accessibility" of the variable declared. It is the result of hiding a variable in outer scopes.

Scope of a Variable

Function Structure

```
class FunctionDemo{
float f;
static int a;

public static void main()
{
  int i;
  static int j;
  func1(i);
}

void func1(int value)
{
  int x;
  //function body
  ....
}
}
```



Scope	Description
Local (block/ function)	"visible" within function or statement block from point of declaration until the end of the block.
Class	"seen" by class members.
File (program)	visible within current file.
Global	visible everywhere unless "hidden".

Lifetime of a variable

- The **lifetime** of a variable or object is the time period in which the variable/object has valid memory.
- Lifetime is also called "allocation method" or "storage duration".

	Lifetime	Stored
Static	Entire duration of the program's execution.	data segment
Automatic	Begins when program execution enters the function or statement block and ends when execution leaves the block.	function call stack
Dynamic	Begins when memory is allocated for the object (e.g., by a call to malloc() or using new) and ends when memory is deallocated (e.g., by a call to free() or using delete).	heap

Scope vs Lifetime of a variable

Variable Type	**Scope** of a Variable	**Lifetime** of a Variable
Instance Variable	Throughout the class except in static methods	Until object is available in the memory
Class Variable	Throughout the class	Until end of the Class
Local Variable	Throughout the block/function in which it is declared	Until control leaves the block

Exercise

1. Write a function to check whether given number is prime or not.
2. Write a function to search a given number from an array.

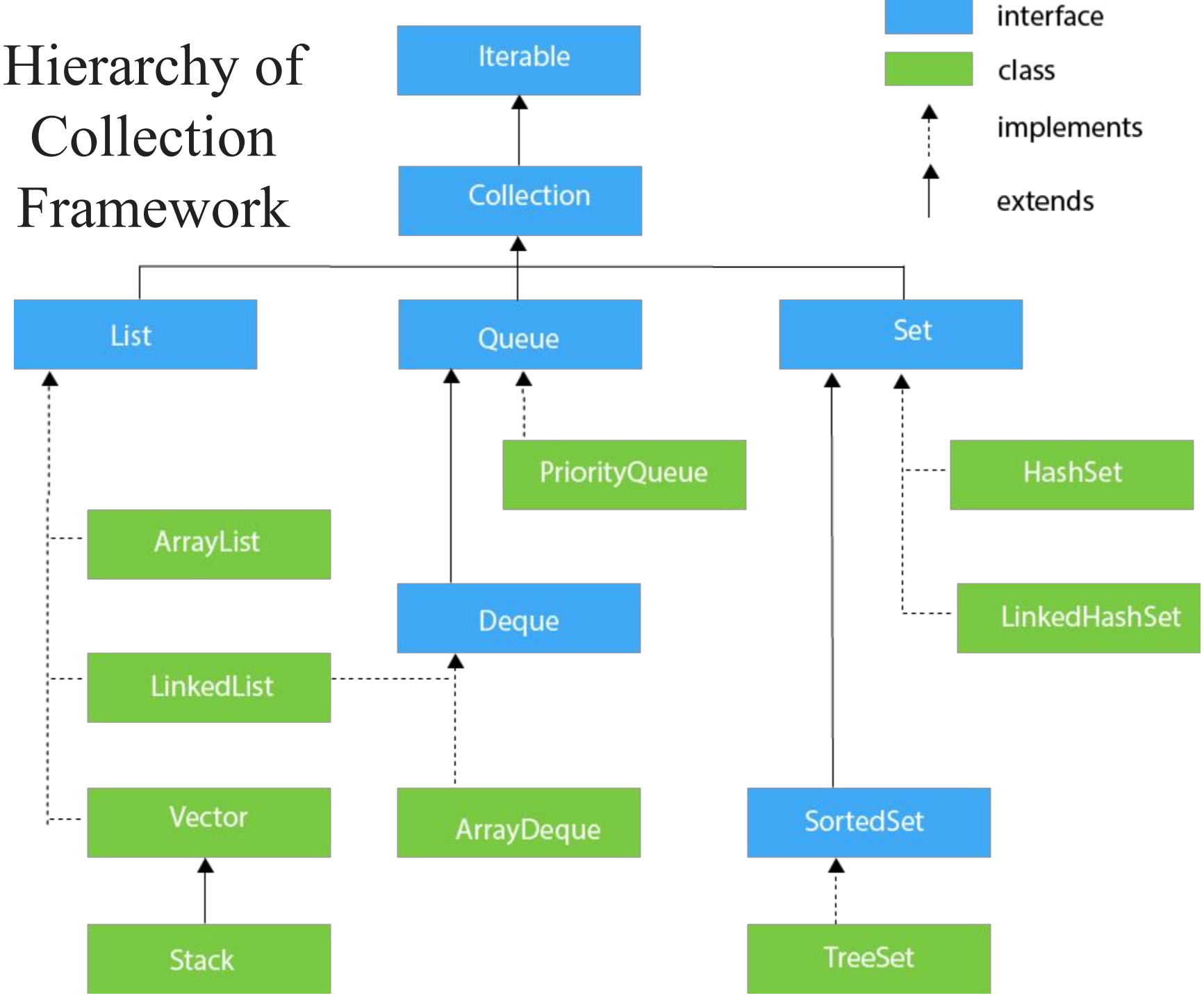
Outline

- ✔ Collection
- ✔ List Interface
- ✔ Iterator
- ✔ Comparator
- ✔ Vector Class
- ✔ Stack
- ✔ Queue
- ✔ List v/s Sets
- ✔ Map Interfaces

Collection

- The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Hierarchy of Collection Framework



Collection Interface - Methods

Sr.	Method & Description
1	boolean add(E e) It is used to insert an element in this collection.
2	boolean addAll(Collection<? extends E> c) It is used to insert the specified collection elements in the invoking collection.
3	void clear() It removes the total number of elements from the collection.
4	booelan contains(Object element) It is used to search an element.
5	boolean containsAll(Collection<?> c) It is used to search the specified collection in the collection.
6	boolean equals(Object obj) Returns true if invoking collection and obj are equal. Otherwise returns false.
7	int hashCode() Returns the hashcode for the invoking collection.

Collection Interface – Methods (Cont.)

Sr.	Method & Description
8	boolean isEmpty() Returns true if the invoking collection is empty. Otherwise returns false.
9	Iterator iterator() It returns an iterator.
10	boolean remove(Object obj) Removes one instance of obj from the invoking collection. Returns true if the element was removed. Otherwise, returns false.
11	boolean removeAll(Collection<?> c) It is used to delete all the elements of the specified collection from the invoking collection.
12	boolean retainAll(Collection<?> c) It is used to delete all the elements of invoking collection except the specified collection.
13	int size() It returns the total number of elements in the collection.

List Interface

- The **`List`** interface extends **`Collection`** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- List is a generic interface with following declaration

```
interface List<E>
```

where `E` specifies the type of object.

List Interface - Methods

Sr.	Method & Description
1	void add(int index, Object obj) Inserts obj into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	boolean addAll(int index, Collection c) Inserts all elements of c into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	Object get(int index) Returns the object stored at the specified index within the invoking collection.
4	int indexOf(Object obj) Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.
5	int lastIndexOf(Object obj) Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, -1 is returned.

List Interface – Methods (Cont.)

Sr.	Method & Description
6	ListIterator listIterator( ) Returns an iterator to the start of the invoking list.
7	ListIterator listIterator(int index) Returns an iterator to the invoking list that begins at the specified index.
8	Object remove(int index) Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one
9	Object set(int index, Object obj) Assigns obj to the location specified by index within the invoking list.
10	List subList(int start, int end) Returns a list that includes elements from start to end-1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

List Interface (example)

```
import java.util.*;
public class CollectionsDemo {
   public static void main(String[] args) {
      List a1 = new ArrayList();
      a1.add("Sachin");
      a1.add("Sourav");
      a1.add("Shami");
      System.out.println("ArrayList Elements");
      System.out.print("\t" + a1);

      List l1 = new LinkedList();
      l1.add("Mumbai");
      l1.add("Kolkata");
      l1.add("Vadodara");
      System.out.println();
      System.out.println("LinkedList Elements");
      System.out.print("\t" + l1);
   }
}
```

Here **ArrayList** & **LinkedList** implements **List** Interface

```
G:\Darshan\Java 2019\PPTs\HAD\Programs>ja
ArrayList Elements
        [Sachin, Sourav, Shami]
LinkedList Elements
        [Mumbai, Kolkata, Vadodara]
G:\Darshan\Java 2019\PPTs\HAD\Programs>
```

Iterator

- `Iterator` interface is used to cycle through elements in a collection, eg. displaying elements.
- `ListIterator` extends `Iterator` to allow bidirectional traversal of a list, and the modification of elements.
- Each of the collection classes provides an `iterator( )` method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.
- To use an iterator to cycle through the contents of a collection, follow these steps:
  1. Obtain an iterator to the start of the collection by calling the collection's `iterator( )` method.
  2. Set up a loop that makes a call to `hasNext( ).` Have the loop iterate as long as `hasNext( )` returns true.
  3. Within the loop, obtain each element by calling `next( ).`

Iterator - Methods

Sr.	Method & Description
1	boolean hasNext() Returns true if there are more elements. Otherwise, returns false.
2	E next() Returns the next element. Throws NoSuchElementException if there is not a next element.
3	void remove() Removes the current element. Throws IllegalStateException if an attempt is made to call remove() that is not preceded by a call to next()

Iterator - Example

```
import java.util.*;
public class IteratorDemo {
    public static void main(String args[]) {
        ArrayList<String> al = new ArrayList<String>();
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
        System.out.print("Contents of list: ");
        Iterator<String> itr = al.iterator();
        while(itr.hasNext()) {
            Object element = itr.next();
            System.out.print(element + " ");
        }
    }
}
```

```
G:\Darshan\Java 2019\PPTs\HAD\Pr
Contents of list: C A E B D F
```

Comparator

- Comparator interface is used to set the sort order of the object to store in the sets and lists.
- The Comparator interface defines two methods: compare( ) and equals( ).
- `int compare(Object obj1, Object obj2)`

  `obj1` and `obj2` are the objects to be compared. This method returns zero if the objects are equal. It returns a positive value if `obj1` is greater than `obj2`. Otherwise, a negative value is returned.

- `boolean equals(Object obj)`

  `obj` is the object to be tested for equality. The method returns true if `obj` and the invoking object are both Comparator objects and use the same ordering. Otherwise, it returns false.

Comparator Example

```
public class ComparatorDemo {
    public static void main(String args[]){
        ArrayList<Student> al=new ArrayList<Student>();
        al.add(new Student("Vijay",23));
        al.add(new Student("Ajay",27));
        al.add(new Student("Jai",21));
        System.out.println("Sorting by age");
        Collections.sort(al,new AgeComparator());
        Iterator<Student> itr2=al.iterator();
        while(itr2.hasNext()){
            Student st=(Student)itr2.next();
            System.out.println(st.name+" "+st.age);
        }
    }
}
```

```
class AgeComparator implements
Comparator<Object>{
    public int compare(Object o1,Object o2){
        Student s1=(Student)o1;
        Student s2=(Student)o2;
        if(s1.age==s2.age) return 0;
        else if(s1.age>s2.age) return 1;
        else return -1;
    }
}
```

```
import java.util.*;
class Student {
    String name;
    int age;
    Student(String name, int
    age){
        this.name = name;
        this.age = age;
    }
}
```

Vector Class

- **`Vector`** implements a dynamic array.
- It is similar to **`ArrayList`**, but with two differences:
  - Vector is synchronized.
  - Vector contains many legacy methods that are not part of the collection framework
- Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.
- Vector is declared as follows:

```
Vector<E> = new Vector<E>;
```

Vector - Constructors

Sr.	Constructor & Description
1	Vector( ) This constructor creates a default vector, which has an initial size of 10
2	Vector(int size) This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size:
3	Vector(int size, int incr) This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward
4	Vector(Collection c) creates a vector that contains the elements of collection c

Vector - Methods

Sr.	Method & Description
7	boolean containsAll(Collection c) Returns true if this Vector contains all of the elements in the specified Collection.
8	Enumeration elements() Returns an enumeration of the components of this vector.
9	Object firstElement() Returns the first component (the item at index 0) of this vector.
10	Object get(int index) Returns the element at the specified position in this Vector.
11	int indexOf(Object elem) Searches for the first occurence of the given argument, testing for equality using the equals method.
12	boolean isEmpty() Tests if this vector has no components.

Vector – Method (Cont.)

Sr.	Method & Description
13	Object lastElement() Returns the last component of the vector.
14	int lastIndexOf(Object elem) Returns the index of the last occurrence of the specified object in this vector.
15	Object remove(int index) Removes the element at the specified position in this Vector.
16	boolean removeAll(Collection c) Removes from this Vector all of its elements that are contained in the specified Collection.
17	Object set(int index, Object element) Replaces the element at the specified position in this Vector with the specified element.
18	int size() Returns the number of components in this vector.

Stack

- `Stack` is a subclass of `Vector` that implements a standard last-in, first-out stack.
- `Stack` only defines the default constructor, which creates an empty stack.
- `Stack` includes all the methods defined by `Vector` and adds several of its own.
- `Stack` is declared as follows:

```
Stack<E> st = new Stack<E>();
```

where `E` specifies the type of object.

Stack - Methods

- Stack includes all the methods defined by Vector and adds several methods of its

Sr.	Method & Description
1	boolean empty() Returns true if the stack is empty, and returns false if the stack contains elements.
2	E peek() Returns the element on the top of the stack, but does not remove it.
3	E pop() Returns the element on the top of the stack, removing it in the process.
4	E push(E element) Pushes element onto the stack. Element is also returned.
5	int search(Object element) Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

Queue

- `Queue` interface extends `Collection` and declares the behaviour of a queue, which is often a first-in, first-out list.
- `LinkedList` and `PriorityQueue` are the two classes which implements Queue interface
- `Queue` is declared as follows:

```
Queue<E> q = new LinkedList<E>();
Queue<E> q = new PriorityQueue<E>();
```

where `E` specifies the type of object.

Queue - Methods

Sr.	Method & Description
1	E element() Returns the element at the head of the queue. The element is not removed. It throws NoSuchElementException if the queue is empty.
2	boolean offer(E obj) Attempts to add obj to the queue. Returns true if obj was added and false otherwise.
3	E peek() Returns the element at the head of the queue. It returns null if the queue is empty. The element is not removed.
4	E poll() Returns the element at the head of the queue, removing the element in the process. It returns null if the queue is empty.
5	E remove() Returns the element at the head of the queue, returning the element in the process. It throws NoSuchElementException if the queue is empty.

Queue Example

```
import java.util.*;
public class QueueDemo {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<String>();
        q.add("Tom");
        q.add("Jerry");
        q.add("Mike");
        q.add("Steve");
        q.add("Harry");
        System.out.prin
        System.out.prin
        System.out.prin
        System.out.println("poll(): "+q.poll());
        System.out.println("peek(): "+q.peek());
        System.out.println("Elements in Queue:"+q);
    }
}
```

```
G:\Darshan\Java 2019\PPTs\HAD\Programs>java QueueDemo
Elements in Queue:[Tom, Jerry, Mike, Steve, Harry]
Removed element: Tom
Head: Jerry
poll(): Jerry
peek(): Mike
Elements in Queue:[Mike, Steve, Harry]
```

PriorityQueue

- `PriorityQueue` extends `AbstractQueue` and implements the `Queue` interface.
- It creates a queue that is prioritized based on the queue's comparator.
- `PriorityQueue` is declared as follows:

```
PriorityQueue<E> = new PriorityQueue<E>;
```

- It builds an empty queue with starting capacity as 11.

PriorityQueue - Example

```
import java.util.*;
public class PriorityQueueExample {
    public static void main(String[] args) {
        PriorityQueue<Integer> numbers = new
PriorityQueue<>();
        numbers.add(750);
        numbers.add(500);
        numbers.add(900);
        numbers.add(100);
        while (!numbers.isEmpty()) {
            System.out.println(numbers.remove());
        }
    }
}
```

```
G:\Darshan\Java
100
500
750
900
```

List v/s Sets

List	Set
Lists allow duplicates.	Sets allow only unique elements.
List is an ordered collection.	Sets is an unordered collection.
Popular implementation of List interface includes ArrayList, Vector and LinkedList.	Popular implementation of Set interface includes HashSet, TreeSet and LinkedHashSet.

When to use List and Set?

Lists - If insertion order is maintained during insertion and allows duplicates.

Sets – If unique collection without any duplicates without maintaining order.

Maps

- A map is an object that stores associations between keys and values, or key/value pairs.
- Given a key, you can find its value. Both keys and values are objects.
- The keys must be unique, but the values may be duplicated. Some maps can accept a null key and null values, others cannot.
- Maps don't implement the Iterable interface. This means that you cannot cycle through a map using a for-each style for loop. Furthermore, you can’t obtain an iterator to a map.

Map Interfaces

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of Map.
NavigableMap	Extends SortedMap to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends Map so that the keys are maintained in ascending order.

Map Classes

Class	Description
AbstractMap	Implements most of the Map interface.
EnumMap	Extends AbstractMap for use with enum keys.
HashMap	Extends AbstractMap to use a hash table.
TreeMap	Extends AbstractMap to use a tree.
WeakHashMap	Extends AbstractMap to use a hash table with weak keys.
LinkedHashMap	Extends HashMap to allow insertion-order iterators.
IdentityHashMap	Extends AbstractMap and uses reference equality when comparing documents.

HashMap Class

- The HashMap class extends AbstractMap and implements the Map interface.
- It uses a hash table to store the map. This allows the execution time of get() and put() to remain constant even for large sets.
- HashMap is a generic class that has declaration:

```
class HashMap<K,V>
```

HashMap - Constructors

Sr.	Constructor & Description
1	HashMap() Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
2	HashMap(int initialCapacity) Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
3	HashMap(int initialCapacity, float loadFactor) Constructs an empty HashMap with the specified initial capacity and load factor.
4	HashMap(Map<? extends K,? extends V> m) Constructs a new HashMap with the same mappings as the specified Map.

```java
import java.util.*;
class HashMapDemo {
    public static void main(String args[]) {
        // Create a hash map.
        HashMap<String, Double> hm = new HashMap<String, Double>();
        // Put elements to the map
        hm.put("John Doe", new Double(3434.34));
        hm.put("Tom Smith", new Double(123.22));
        hm.put("Jane Baker", new Double(1378.00));
        hm.put("Tod Hall", new Double(99.22));
        hm.put("Ralph Smith", new Double(-19.08));
        // Get a set of the entries.
        Set<Map.Entry<String, Double>> set = hm.entrySet();
        // Display the set.
        for(Map.Entry<String, Double> me : set) {
            System.out.print(me.getKey() + ": ");
            System.out.println(me.getValue());
        }
        System.out.println();
        //Deposit 1000 into John Doe's account.
        double balance = hm.get("John Doe");
        hm.put("John Doe", balance + 1000);
        System.out.println("John Doe's new balance: " +
        hm.get("John Doe"));
    }
}
```

```
C:\Users\hardi\Desktop>java HashMapDemo
Tod Hall: 99.22
John Doe: 3434.34
Ralph Smith: -19.08
Tom Smith: 123.22
Jane Baker: 1378.0

John Doe's new balance: 4434.34
```

Exceptions

- An **exception** is an object that describes an **unusual** or **erroneous situation**.
- Exceptions are thrown by a program, and may be caught and handled by another part of the program.
- A program can be separated into a **normal execution flow** and an **exception execution flow**.
- An error is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught.
- Java has a predefined set of exceptions and errors that can occur during execution.
- A program can deal with an exception in one of three ways:
  - **ignore** it
  - **handle** it where it occurs
  - handle it at **another place** in the program
- The manner in which an exception is processed is an important design consideration.

Using try and catch

- Example:

```
try{
    // code that may cause exception
}
catch(Exception e){
    // code when exception occurred
}
```

Exception is the superclass of all the exception that may occur in Java

- Multiple catch:

```
try{
    // code that may cause exception
}
catch(ArithmeticException ae){
    // code when arithmetic exception occurred
}
catch(ArrayIndexOutOfBoundsException aiobe){
    // when array index out of bound exception occurred
}
```

Nested try statements

```
try
{
    try
    {
        // code that may cause array index out of bound exception
    }
    catch(ArrayIndexOutOfBoundsException aiobe)
    {
        // code when array index out of bound exception occured
    }
    // other code that may cause arithmetic exception
}
catch(ArithmeticException ae)
{
    // code when arithmetic exception occurred
}
```

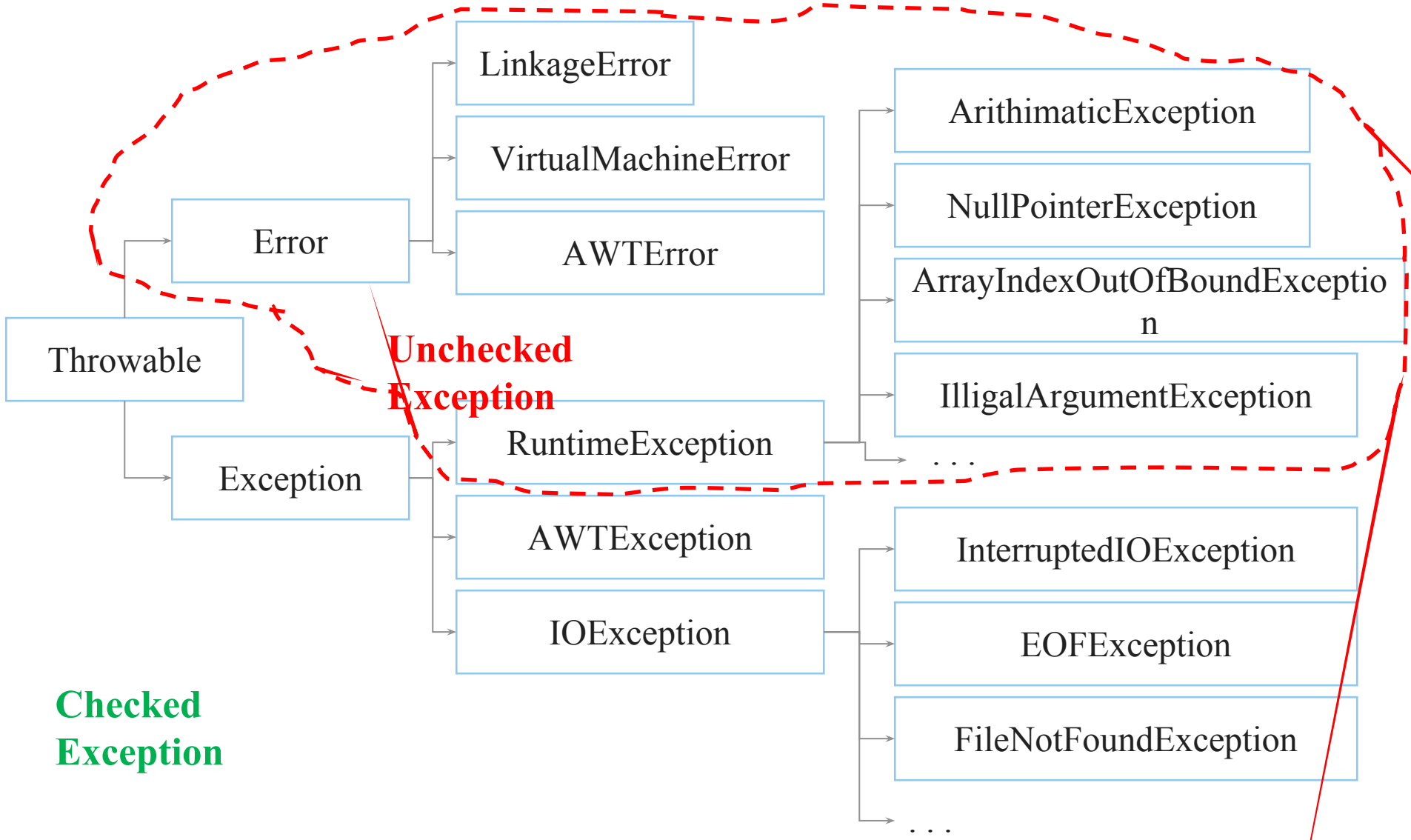
Types of Exceptions

- An exception is either checked or unchecked.

- **Checked** Exceptions
  - A checked exception either must be caught by a method, or must be listed in the throws clause of any method that may throw or propagate it.
  - The compiler will issue an error if a checked exception is not caught or asserted in a throws clause
  - Example: IOException, SQLException etc…
- **Unchecked** Exceptions
  - An unchecked exception does not require explicit handling, though it could be processed using try catch.
  - The only unchecked exceptions in Java are objects of type RuntimeException or any of its descendants.
  - Example: ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException etc..

The Exception Class Hierarchy

- Classes that define exceptions are related by inheritance, forming an exception class hierarchy.
- All error and exception classes are descendents of the Throwable class
- The custom exception can be created by extending the Exception class or one of its descendants.



Java's Inbuilt Unchecked Exceptions

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.

Java's Inbuilt Checked Exceptions

Exception	Meaning
ClassNotFoundException	Class not found.
IOException	Input Output Exceptions
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

throw statement

- it is possible for your program to throw an exception **explicitly**, using the **throw** statement.
- The general form of throw is shown here:

  throw *ThrowableInstance;*
- Here, ***ThrowableInstance*** *must be an object of type* ***Throwable*** *or a* ***subclass*** *of Throwable.*
- Primitive types, such as int or char, as well as non-throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways you can obtain a Throwable object:
  - using a parameter in a catch clause,
  - or creating one with the new operator.

Throw (Example)

```
public class DemoException {
    public static void main(String[] args) {
        int balance = 5000;

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Amount to withdraw");
        int withdraw = sc.nextInt();
        try {
            if(balance - withdraw < 1000) {
                throw new Exception("Balance must be grater than 1000");
            }
            else {
                balance = balance - withdraw;
            }
        }catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

The finally statement

- The purpose of the **finally** statement will allow the execution of a segment of code regardless if the try statement throws an exception or executes successfully
- The advantage of the **finally** statement is the ability to clean up and release resources that are utilized in the **try** segment of code that might not be released in cases where an exception has occurred.

```
public class MainCall {
    public static void main(String args[]) {
        int balance = 5000;
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter Amount to withdraw");
        int withdraw = sc.nextInt();
        try {
            if(balance - withdraw < 1000) {
                throw new Exception("Balance < 1000 error");
            }
            else {
                balance = balance - withdraw;
            }
        }catch(Exception e) {
            e.printStackTrace();
        }
        finally {
            sc.close();
        }
    }
}
```

throws statement

- A throws statement lists the types of exceptions that a **method** might throw.
- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.
- This is the general form of a method declaration that includes a **throws clause:**

type method-name(parameter-list) ***throws*** *exception-list* {
    // body of method
}

- Here, *exception-list is a comma-separated list of the exceptions that a method can throw.*
- Example :

```
void myMethod() throws ArithmeticException, NullPointerException
{
    // code that may cause exception
}
```

Create Your Own Exception

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- This is quite easy to do: just define a subclass of Exception (which is, of course, a subclass of Throwable).
- The Exception class does not define any methods of its own. It does inherit those methods provided by Throwable.
- Thus, all exceptions have methods that you create and defined by Throwable.

Methods of Exception class

Method	Description
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement.
Throwable initCause(Throwable causeExc)	Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement elements[ ])	Sets the stack trace to the elements passed in elements.
String toString( )	Returns a String object containing a description of the exception.

Custom Exception (Example)

```
// A Class that represents
user-defined exception
class MyException
    extends Exception
{
  public MyException(String s)
  {
     // Call constructor of
    parent (Exception)
        super(s);
  }
}
```

```
class MainCall {
    static int currentBal = 5000;
    public static void main(String args[]) {
        try {
        int amount = Integer.parseInt(args[0]);
            withdraw(amount);
        } catch (Exception ex) {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
    public static void withdraw(int amount) throws Exception
    {
        int newBalance = currentBal - amount;
        if(newBalance<1000) {
            throw new MyException("demo Exception");
        }
    }
}
```

Outline

- ✔ File class
- ✔ Stream
- ✔ Byte Stream
- ✔ Character Stream

File class

- Java **`File`** class represents the files and directory pathnames in an **abstract manner**. This class is used for **creation of files** and **directories**, **file searching**, **file deletion** etc.
- The File object represents the actual file/directory on the disk. Below given is the list of constructors to create a File object.
- (

Sr.	Constructor
1	File(String pathname) Creates a new File instance by converting the given pathname string into an abstract pathname.
2	File(String parent, String child) Creates a new File instance from a parent pathname string and a child pathname string.
3	File(URI uri) Creates a new File instance by converting the given file: URI into an abstract pathname.

Methods of File Class

Sr.	Method
1	public boolean isAbsolute() Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise
2	public String getAbsolutePath() Returns the absolute pathname string of this abstract pathname.
3	public boolean canRead() Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
4	public boolean canWrite() Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
5	public boolean exists() Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise

Methods of File Class (Cont.)

Sr.	Method
6	public boolean isDirectory() Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
7	public boolean isFile() Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria
8	public long lastModified() Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970).
9	public long length() Returns the length of the file denoted by this abstract pathname.
10	public boolean delete() Deletes the file or directory.
11	public String[] list() Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

File Class Example

```
import java.io.File;
class FileDemo {
    public static void main(String args[]) {
        File f1 = new File("FileDemo.java");
        System.out.println("File Name: " + f1.getName());
        System.out.println("Path: " + f1.getPath());
        System.out.println("Abs Path: " + f1.getAbsolutePath());
        System.out.println("Parent: " + f1.getParent());
        System.out.println(f1.exists() ? "exists" : "does not exist");
        System.out.println(f1.canWrite() ? "is writeable" : "is not writeable");
        System.out.println (f1.canRead () ? "is readable" : "is not readable");
        System.out.println ("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
        System.out.println(f1.isFile() ? "is normal file" : "might be a named pipe");
        System.out.println(f1.isAbsolute() ? "is absolute" : "is not absolute");
        System.out.println("File last modified: " + f1.lastModified());
        System.out.println("File size: " + f1.length() + " Bytes");
    }
}
```

Stream

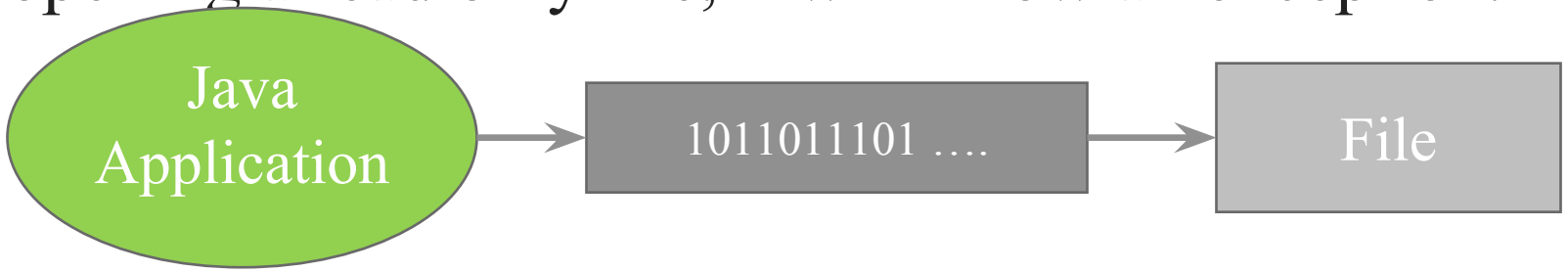
- A stream can be defined as a sequence of data.
- All streams represent an input source and an output destination.
- There are two kinds of Streams
  - **Byte Stream**
  - **Character Stream**
- The **java.io** package contains all the classes required for input-output operations.
- The stream in the java.io package supports all the datatype including primitive.

Byte Streams

- Byte streams provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.

FileOutputStream

- Java `FileOutputStream` is an output stream for writing data to a file.
- `FileOutputStream` will create the file before opening it for output.
- On opening a read only file, it will throw an exception.



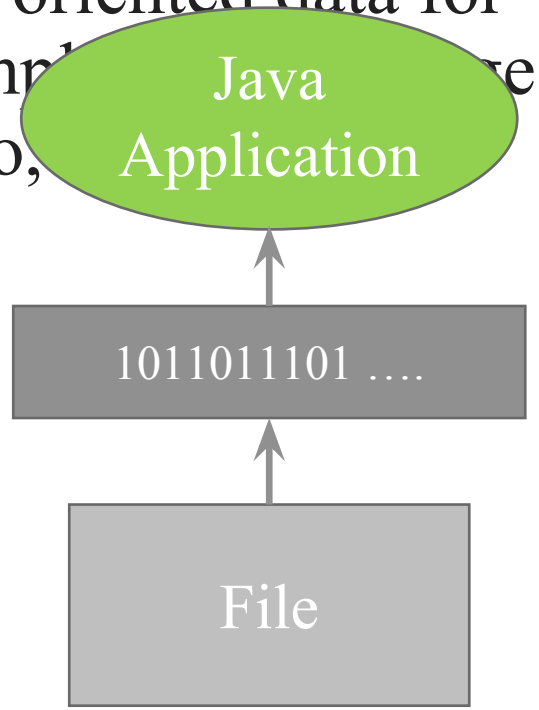
Sr.	Method
1	**void write(byte[] b)** This method writes b.length bytes from the specified byte array to this file output stream.
2	**void write(byte[] b, int off, int len)** This method writes len bytes from the specified byte array starting at offset off to this file output stream.
3	**void write(int b)** This method writes the specified byte to this file output stream.
4	**void close()** This method closes this file output stream and releases any system resources associated with this stream.

FileOutputStream Example

```
class FileOutDemo {
    public static void main(String args[]) {
        try {
            FileOutputStream fout = new FileOutputStream("abc.txt");
            String s = "Sourav Ganguly is my favorite player";
            byte b[] = s.getBytes();
            fout.write(b);
            fout.close();
            System.out.println("Success...");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

FileInputStream

- `FileInputStream` class is used to read bytes from a file.
- It should be used to read byte-oriented data for examp[illegible]e, audio,



Sr.	Method
1	**public int read()**
2	**public int read(byte[] b)** b - the buffer into which the data is read. Returns: the total number of bytes read into the buffer, or -1.
3	**public int read(byte[] b, int off, int len)** b - the buffer into which the data is read. off - the start offset in the destination array b len - the maximum number of bytes read. Returns: the total number of bytes read into the buffer, or -1
4	**public long skip(long n)** n - the number of bytes to be skipped. Returns: the actual number of bytes skipped.
5	**public int available()** an estimate of the number of remaining bytes that can be read
6	**public void close()** Closes this file input stream and releases any system resources associated.

FileInputStream Example

```
class SimpleRead {
    public static void main(String args[]) {
        try {
            FileInputStream fin = new FileInputStream("abc.txt");
            int i = 0;
            while ((i = fin.read()) != -1) {
                System.out.println((char) i);
            }
            fin.close();
        } catch (Exception e) {
             System.out.println(e);
        }
    }
}
```

Example of Byte Streams

```
import java.io.*;
public class CopyFile {
    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;
        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");
            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    } }
```

Character Streams

- Character Streams provide a convenient means for handling input and output of characters.
- Internationalization is possible as it uses Unicode.
- For character streams we have two base classes
  - Reader
  - Writer

Reader

- The Java `Reader` class is the base class of all Reader's in the IO API.
- Subclasses include a `FileReader, BufferedReader, InputStreamReader, StringReader` and several others.
- Here is a simple Java IO Reader example:

```
Reader reader = new FileReader("c:\\data\\myfile.txt");
int data = reader.read();
while (data != -1) {
    char dataChar = (char) data;
    data = reader.read();
}
```

- Combining Readers with InputStream

```
Reader reader = new InputStreamReader("c:\\data\\myfile.txt");
```

Writer

- The Java `Writer` class is the base class of all Writers in the I-O API.
- Subclasses include `BufferedWriter, PrintWriter, StringWriter` and several others.
- Here is a simple Java IO Writer example:

```
Writer writer = new FileWriter("c:\\data\\file-output.txt");
writer.write("Hello World Writer");
writer.close();
```

- Combining Readers With OutputStreams

```
Writer writer = new OutputStreamWriter("c:\\data\\file-output.txt");
```

BufferedReader

- The `java.io.BufferedReader` class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- Following are the important points about `BufferedReader`:
  - The buffer size may be specified, or the default size may be used.
  - Each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream.
- (

Sr.	Constructor
1	BufferedReader(Reader in) This creates a buffering character-input stream that uses a default-sized input buffer.
2	BufferedReader(Reader in, int sz) This creates a buffering character-input stream that uses an input buffer of the specified size.

BufferedReader (Methods)

Sr.	Methods
1	void close() This method closes the stream and releases any system resources associated with it.
2	int read() This method reads a single character.
3	int read(char[] cbuf, int off, int len) This method reads characters into a portion of an array.
4	String readLine() This method reads a line of text.
5	void reset() This method resets the stream.
6	long skip(long n) This method skips characters.

BufferedReader – Example

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

class BufferedReaderDemo {
    public static void main(String[] args) throws IOException {
        FileReader fr = new FileReader("input.txt");
        BufferedReader br = new BufferedReader(fr);
        char c[] = new char[20];
        br.skip(8);
        if (br.ready()) {
            System.out.println(br.readLine());
            br.read(c);
            for (int i = 0; i < 20; i++) {
                System.out.print(c[i]);
            }
        }
    }
}
```

Generics

Content

- **Generics Fundamentals,**
- **Bounded Types,**
- **Using wildcard arguments & bounded wirldcards,**
- **Generic methods, constructors, class hierarchies & Interfaces.**

Generics in Java

- The **Java Generics** programming is introduced in J2SE 5 to deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.
- Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

Advantage of Java Generics

1) Type-safety: We can hold only a single type of objects in generics. It doesn't allow to store other objects.

- Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
```

With Generics, it is required to specify the type of object we need to store.

```
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

2) Type casting is not required: There is no need to typecast the object.

- Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

- **3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32);//Compile Time Error
```

- **Syntax** to use generic collection
  - ClassOrInterface<Type>
- **Example** to use Generics in java
  - ArrayList<String>

```
import java.util.*;
class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("rahul");
list.add("jai");
//list.add(32);//compile time error

String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);

Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

- Example of Java Generics using Map

```
import java.util.*;
class TestGenerics2{
public static void main(String args[]){
Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"vijay");
map.put(4,"umesh");
map.put(2,"ankit");

//Now use Map.Entry for Set and Iterator
Set<Map.Entry<Integer,String>> set=map.entrySet();

Iterator<Map.Entry<Integer,String>> itr=set.iterator();
while(itr.hasNext()){
Map.Entry e=itr.next();//no need to typecast
System.out.println(e.getKey()+" "+e.getValue());
}

}}
```

Generic class

- A class that can refer to any type is known as a generic class. Here, we are using the T type parameter to create the generic class of specific type.
- Let's see a simple example to create and use the generic class.
- Creating a generic class:

```
class MyGen<T>{
T obj;
void add(T obj){this.obj=obj;}
T get(){return obj;}
}
```

- The T type indicates that it can refer to any type (like String, Integer, and Employee). The type you specify for the class will be used to store and retrieve the data.

- Using generic class:
- Let's see the code to use the generic class.

```
class TestGenerics3{
public static void main(String args[]){
MyGen<Integer> m=new MyGen<Integer>();
m.add(2);
//m.add("vivek");//Compile time error
System.out.println(m.get());
}}
```

Type Parameters

- The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:
  - T - Type
  - E - Element
  - K - Key
  - N - Number
  - V - Value
- Generic Method
  - Like the generic class, we can create a generic method that can accept any type of arguments. Here, the scope of arguments is limited to the method where it is declared. It allows static as well as non-static methods.

```
public class TestGenerics4{

   public static < E > void printArray(E[] elements) {
       for ( E element : elements){
          System.out.println(element );
        }
        System.out.println();
    }
    public static void main( String args[] ) {
       Integer[] intArray = { 10, 20, 30, 40, 50 };
       Character[] charArray = { 'J', 'A', 'V', 'A', 'T','P','O','I','N','T' };

       System.out.println( "Printing Integer Array" );
       printArray( intArray  );

      System.out.println( "Printing Character Array" );
       printArray( charArray );
    }
}
```

Wildcard in Java Generics

- The ? (question mark) symbol represents the wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number, e.g., Integer, Float, and double. Now we can call the method of Number class through any child class object.
- We can use a wildcard as a **type of a parameter, field, return type, or local variable. However, it is not allowed to use a wildcard as a type argument for a generic method invocation, a generic class instance creation, or a supertype**.

```
import java.util.*;
abstract class Shape{
abstract void draw();
}
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle extends Shape{
void draw(){System.out.println("drawing circle");}
}
class GenericTest{
//creating a method that accepts only child class of Shape
public static void drawShapes(List<? extends Shape> lists){
for(Shape s:lists){
s.draw();//calling method of Shape class by child class instance
}
}
```

```
public static void main(String args[]){
List<Rectangle> list1=new ArrayList<Rectangle>();
list1.add(new Rectangle());

List<Circle> list2=new ArrayList<Circle>();
list2.add(new Circle());
list2.add(new Circle());

drawShapes(list1);
drawShapes(list2);
}}
```

Upper Bounded Wildcards

- The purpose of upper bounded wildcards is to decrease the restrictions on a variable. It restricts the unknown type to be a specific type or a subtype of that type. It is used by declaring wildcard character ("?") followed by the extends (in case of, class) or implements (in case of, interface) keyword, followed by its upper bound.
- Syntax
- List<? **extends** Number>
- Here,
- **?** is a wildcard character.
- **extends**, is a keyword.
- **Number**, is a class present in java.lang package
- Suppose, we want to write the method for the list of Number and its subtypes (like Integer, Double). Using **List<? extends Number>** is suitable for a list of type Number or any of its subclasses
  whereas **List<Number>** works with the list of type Number only. So, **List<? extends Number>** is less restrictive than **List<Number>**.

```
import java.util.ArrayList;

public class UpperBoundWildcard {


   private static Double add(ArrayList<? extends
     Number> num) {

      double sum=0.0;

      for(Number n:num)
      {
         sum = sum+n.doubleValue();
      }

      return sum;
   }

   public static void main(String[] args) {

         ArrayList<Integer> l1=new ArrayList<
Integer>();
         l1.add(10);
         l1.add(20);
         System.out.println("displaying the su
m= "+add(l1));

         ArrayList<Double> l2=new ArrayList<
Double>();
         l2.add(30.0);
         l2.add(40.0);
         System.out.println("displaying the su
m= "+add(l2));


      }

}
```

Unbounded Wildcards

- The unbounded wildcard type represents the list of an unknown type such as List<?>. This approach can be useful in the following scenarios: -
  - When the given method is implemented by using the functionality provided in the Object class.
  - When the generic class contains the methods that don't depend on the type parameter.

```
import java.util.Arrays;
import java.util.List;

public class UnboundedWildcard {

   public static void display(List<?> list)
   {

      for(Object o:list)
      {
         System.out.println(o);
      }

   }

 public static void main(String[] args) {

   List<Integer> l1=Arrays.asList(1,2,3);
   System.out.println("displaying the Integer va
lues");
   display(l1);
   List<String> l2=Arrays.asList("One","Two","T
hree");
     System.out.println("displaying the String val
ues");
       display(l2);
   }

}
```

Lower Bounded Wildcards

- The purpose of lower bounded wildcards is to restrict the unknown type to be a specific type or a supertype of that type. It is used by declaring wildcard character ("?") followed by the super keyword, followed by its lower bound.
- Syntax
- List<? **super** Integer>
- Here,
- **?** is a wildcard character.
- **super**, is a keyword.
- **Integer**, is a wrapper class.
- Suppose, we want to write the method for the list of Integer and its supertype (like Number, Object). Using **List<? super Integer>** is suitable for a list of type Integer or any of its superclasses
  whereas **List<Integer>** works with the list of type Integer only. So, **List<? super Integer>** is less restrictive than **List<Integer>**.

```
import java.util.Arrays;
import java.util.List;

public class LowerBoundWildcard {

   public static void addNumbers(List<? super Integer> list) {

      for(Object n:list)
      {
          System.out.println(n);
      }



   }
public static void main(String[] args) {

   List<Integer> l1=Arrays.asList(1,2,3);
     System.out.println("displaying the Integer values");
   addNumbers(l1);

   List<Number> l2=Arrays.asList(1.0,2.0,3.0);
     System.out.println("displaying the Number values");
   addNumbers(l2);
}

}
```

JAVA APPLETS

1.1 Applet Programming

- Java can be used to create two types of programs:

1. Applications
2. Applets.

1.1 Applet Programming

1. Java Application :

- An application is a program that runs on your computer, under the operating system of that computer.
- That is, an application created by Java is more or less like one created using C or C++.

2. Java Applet :

- An applet is a small Internet-based program written in Java, which is designed to be transmitted over the Internet can be downloaded by any computer across the network Internet and executed by a Java-compatible Web browser.
- The applet is usually embedded in an HTML page on a Web site and can be executed within a browser.

1.1 Applet Programming

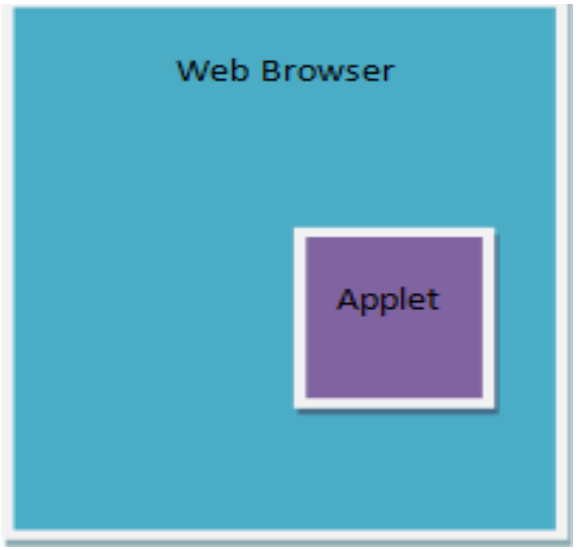
1.1.1 Local and Remote Applets

- There are two different types of applet.
- The applet types vary based on how the applet is embedded into web page.
- Applet Types are:
    1. Local Applet
    2. Remote Applet

1.1 Applet Programming

1.1.1 Local and Remote Applets

- **Local Applet**
  - Local applets are stored in local system.
  - The web browser will search the local system directories, find the local applet and execute it.
  - Execution of local applet does not require internet connection.



1.1 Applet Programming

1.1.1 Local and Remote Applets

- **Specifying Local Applet**

```
<applet      codebase="MyGame"   code=" MyGame.class"
          width=130   height=130    >
</applet>
```

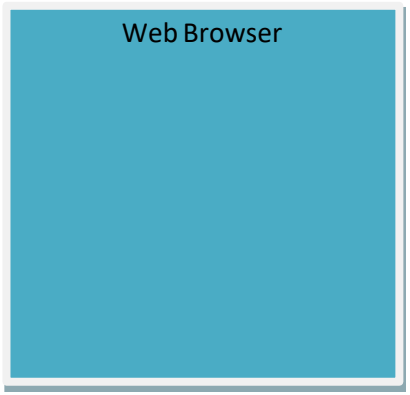
- The codebase attribute specifies a path on your system for the local applet, whereas the code attribute specifies the name of the byte-code file that contains the applet's code.

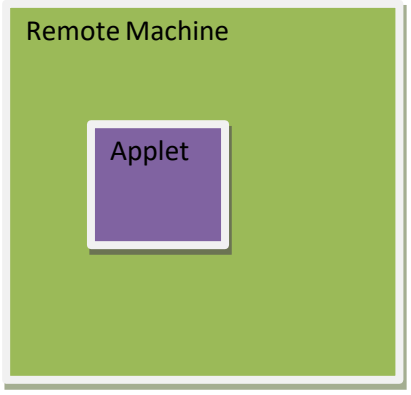
- The path specified in the codebase attribute is relative to the folder containing the HTML document that references the applet.

1.1 Applet Programming

1.1.1 Local and Remote Applets

- **Remote Applet**
  - The remote applets are stored in remote computer.
  - The web browser requires internet connection to locate and load the remote applet from the remote computer.
  - To access a remote applet in your Web page, you must know the applet's URL in order to display the applet correctly.





1.1 Applet Programming

- **Local and Remote Applets**

- **Specifying Remote Applet**

```
<applet    codebase="http://www.website.com/applets/"
        code=“ MyGame.class"    width=130   height=130    >
</applet>
```

- The codebase attribute specifies a URL of remote system for the remote applet, whereas the code attribute specifies the name of the byte-code file that contains the applet's code.

- The only difference between accessing local and remote applet is the value of the codebase attribute. In the local applet, codebase specifies a local folder, and in the remote applet, it specifies the URL at which the applet is located.

1.1 Applet Programming

1.1.2 Difference between Applet and Application

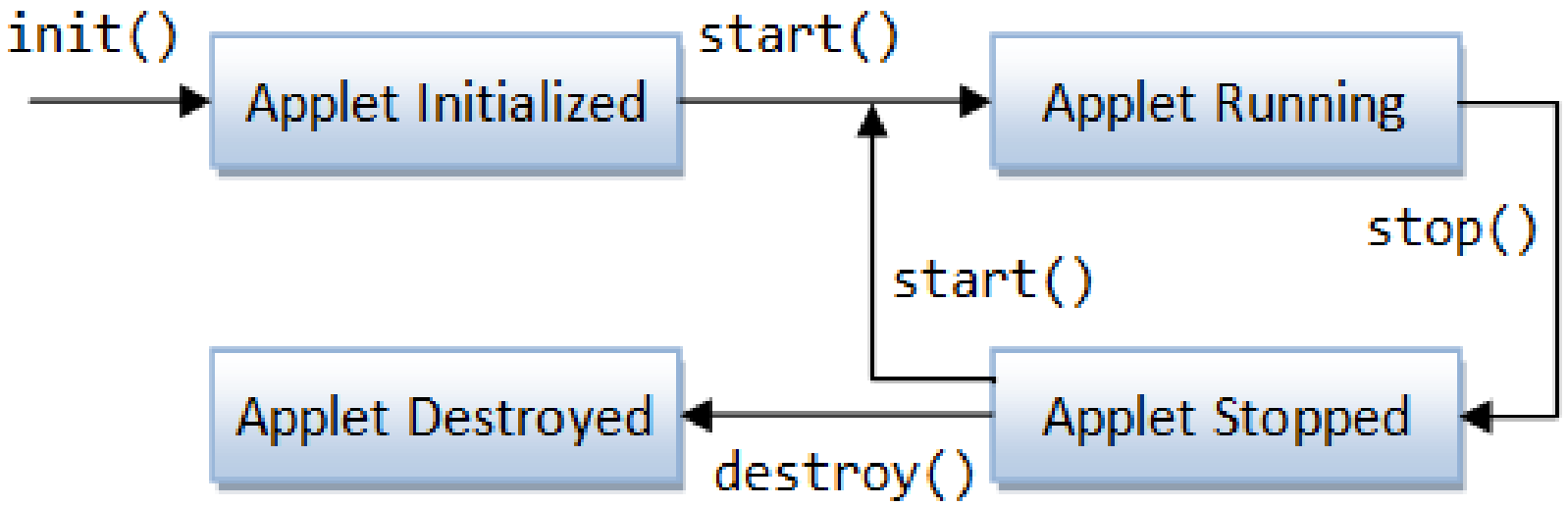
Following are the difference between Applet and Application program:

APPLET	APPLICATION
Applet does not require main() method to start its execution.	Application program requires main() method to start its execution.
Applet is embedded within webpage.	Application can not be embedded within webpage.
Applet does not run independently.	Application runs independently.
Applet accepts input through PARAM tag.	Application accepts input through command line argument.
Applet executes under strict security restriction. It can not access the local file system.	Applet does not have strict security restriction. It can access the local file system.

1.1 Applet Programming

3. Applet Life Cycle

- An applet pass through following states during life cycle as shown in following figure:



1.1 Applet Programming

1.1.3 Applet Life Cycle

- The Applet class defines following methods for managing the life-cycle of the applets.

Method	Description
init()	This method is invoked when the applet is first loaded. It is called (only once) when the applet is first loaded to initialize variables, resize the applet, setting up GUI components, and etc
start()	Invoked every time the browser displays the web page containing the applet. The start method is automatically called after init method.

1.1 Applet Programming

1.1.3 Applet Life Cycle

Method	Description
stop()	Invoked when the browser is closed, stops or suspends anything the applet is doing e.g. if the start method started an animation, it should be stopped in this method.
destroy()	Invoked when the browser determines that it no longer needs the applet – this is when the applet is removed from the browser's cache. Therefore the invocation of this method is controlled by the browser itself.
paint()	Called immediately after start, the paint method is what displays your applet on a webpage. Paint is also called any time the applet needs to repaint itself.

1.1 Applet Programming

1.1.3 Applet Life Cycle

- **Applet Life Sequence :**

- User visits page containing an applet
  - Browser calls init method on that applet, once and then
  - Browser calls start method on that applet
  - Browser calls paint method on that applet
- User goes away from that page
  - Browser calls stop on that applet
- User comes back to that page
  - Browser calls start again on that applet
  - Browser calls paint method on that applet
- User shuts down the browser
  - Browser calls destroy on the applet

1.1 Applet Programming

1.1.3 Applet Life Cycle

```
import java.applet.*;
import java.awt.*;

public class LifeCycle extends Applet
{
        String msg="";
        public void init()
        {
                msg +=  " init() ==>";
        }
```

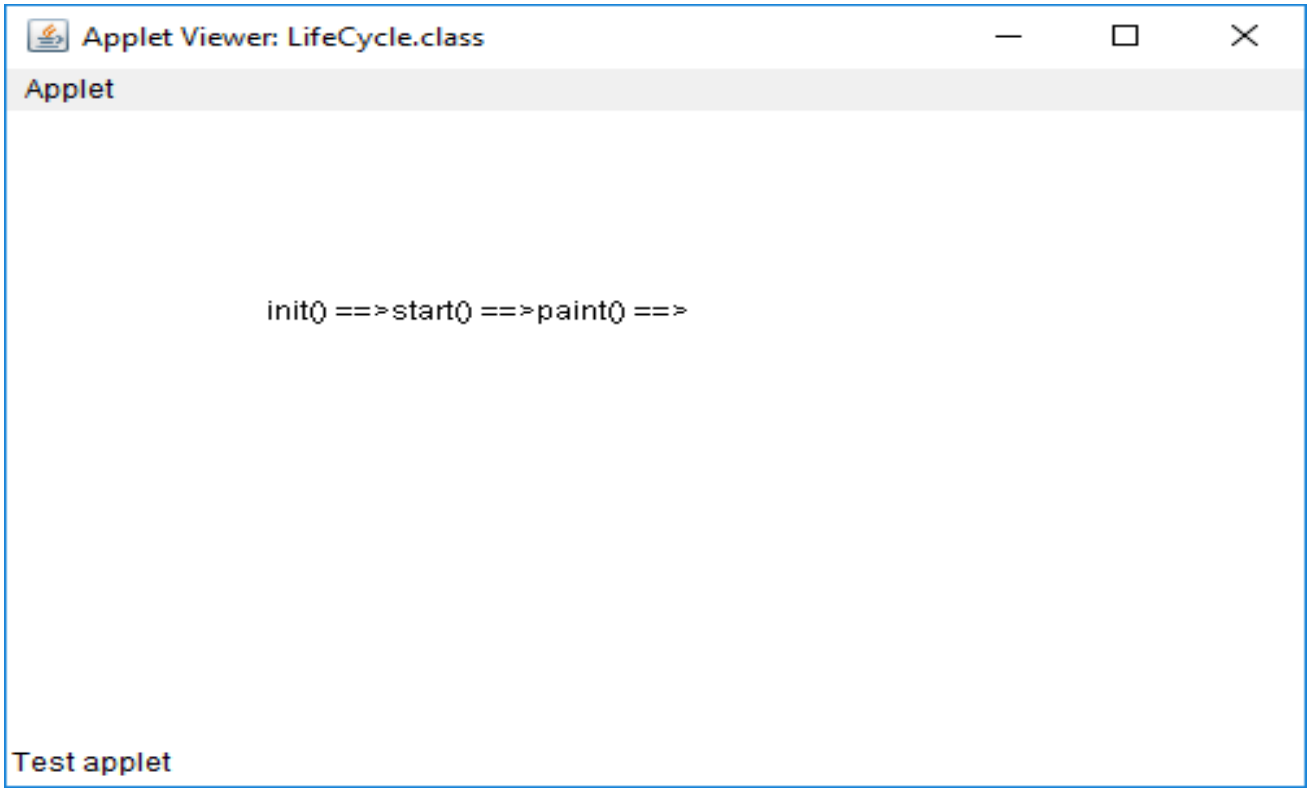
1.1 Applet Programming

1.1.3 Applet Life Cycle

```
    public void start()
    {
      msg += "start()  ==>";
    }

    public void stop()
    {
       msg += "stop() ==>";
    }

    public void paint(Graphics g)
    {
       msg += "paint() ==>";

      g.drawString(msg,100,100);
    }
}
```

1.1 Applet Programming

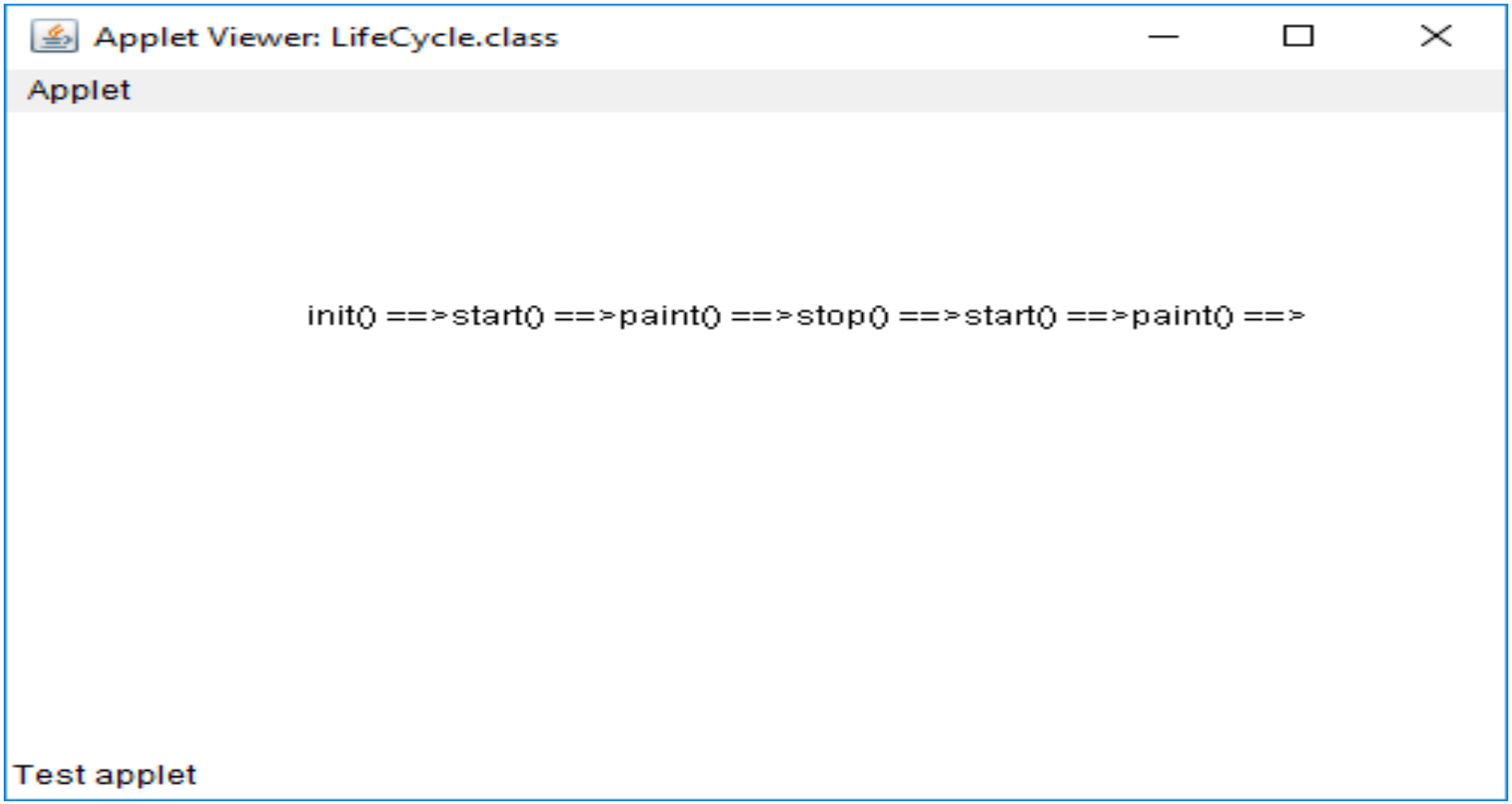
1.1.3 Applet Life Cycle



Now, Minimize the window and then maximize the applet window.

1.1 Applet Programming

1.1.3 Applet Life Cycle



1.1 Applet Programming

4. Developing Executable Applet Code

- Let's begin with the simple applet shown here:

```
import java.awt.*;
import java.applet.*;

public class SimpleApplet extends Applet
{
        public void paint( Graphics g)
        {
                g.drawString( "A Simple Applet", 20, 20);
        }
}
```

1.1 Applet Programming

1.1.4 Developing Executable Applet Code

- This applet begins with two import statements.
- The first import statement imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical interface.
- The second import statement imports the applet package, which contains the class Applet.
- Every applet that you create must be a subclass of Applet.
- The next line in the program declares the class SimpleApplet. This class must be declared as public, because it will be accessed by code that is outside the program.

1.1 Applet Programming

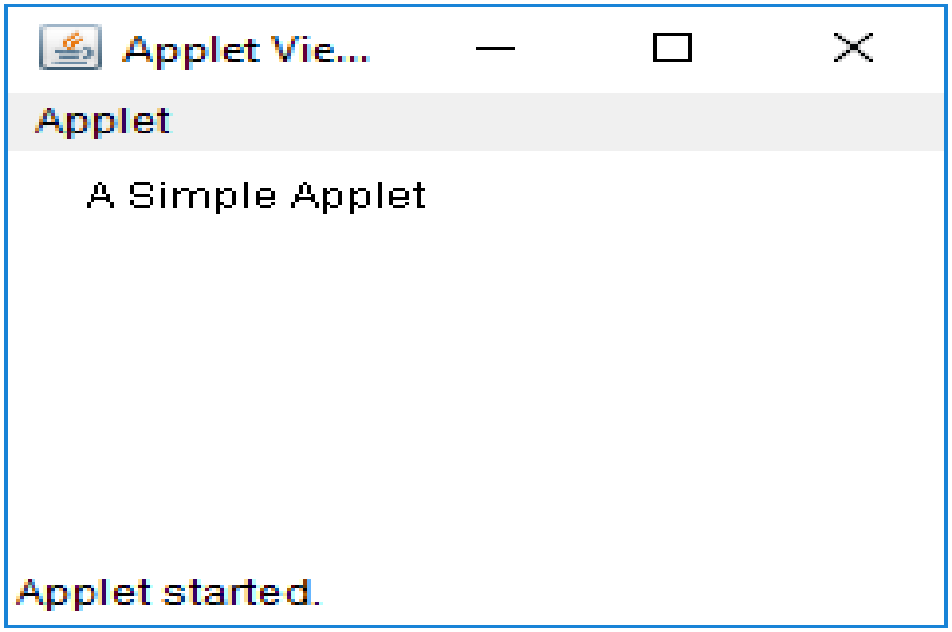
1.1.4 Developing Executable Applet Code

- Inside SimpleApplet, only paint( ) method is defined.
- This method is defined by the AWT and must be overridden by the applet. paint( ) is called each time that the applet must redisplay its output.
- paint( ) is also called when the applet begins execution.
- The paint( ) method has one parameter of type Graphics. This parameter contains the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.
- Inside paint( ) is a call to drawString( ), which is a member of the Graphics class. This method outputs a string beginning at the specified X,Y location.

1.1 Applet Programming

1.1.4 Developing Executable Applet Code

- After you enter the source code for SimpleApplet, compile in the same way that you have been compiling other java programs till now.



1.2 Web page Design

- A static webpage can be designed using HTML.
- HTML file consist of Tag and Text. Tags are written between < and > symbol, in order to distinguish it from the text.
- The basic structure of the HTML file is given below:

```
<html>
    <head>
        <title> Webpage title </title>
    </head>

    <body>
        Here Tags and Text are used to design webpage.
    </body>
</html>
```

1.2 Web page Design

1. Applet tag

- Web documents are written in Hypertext Markup Language (HTML). HTML uses tags to describe the structure of Web documents.
- Tags are used to identify headings, paragraphs, and lists, as well as other elements of Web pages such as links, images, forms, and applets.
- Applets are displayed as a part of a Web page by using the HTML tag <APPLET>.
- Once an applet has been compiled, it is included in an HTML file using the APPLET tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file.

1.2 Web page Design

1. Applet tag

- Applet tag is used to add the applet into HTML page.
- The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

- < APPLET
  [CODEBASE = *codebaseURL]*
  CODE = *appletFile*
  [ALT = *alternateText]*
  [NAME = *appletInstanceName]*
  WIDTH = *pixels HEIGHT = pixels*
  [ALIGN = *alignment]*
  [VSPACE = *pixels] [HSPACE = pixels]* >

- </APPLET>

1.2 Web page Design

1.2.1 Applet tag

Attribute	Explanation	Example
Code	Name of class file	Code="applet0.class"
Width	Width of applet	Width=300
height	Height of applet	Height=60
Codebase	Applet's Directory	Codebase="/applets"
alt	Alternate text if applet not available	Alt="menu applet"
name	Name of the applet	Name="appletExam"
Align(top,left,right,bottom)	Justify the applet with text	Align="right"

1.2 Web page Design

1.2.1 Applet tag

- **Example**

```
<applet   code = “SimpleApplet.class"   width = “130”   height = “130”
   >    </applet>
```

1.2 Web page Design

2. Adding Applet to HTML file

- An applet is like a child application of the browser. The browser launches the applet in a predefined environment inside the browser.

- An applet can be added to HTML file using <APPLET> tag.

- The applet tag provides all the information needed to launch the applet.

1.2 Web page Design

1.2.2 Adding Applet to HTML file

```
<HTML>


    <HEAD>
        <TITLE>Applet HTML Page</TITLE>
    </HEAD>

    <BODY>
        <APPLET code="SimpleApplet.class"  width=350  height=200>  </APPLET>
    </BODY>


</HTML>
```

1.2 Web page Design

3. Running the Applet

- There are two ways in which you can run an applet:

1. Using Java-compatible Web browser.
2. Using Applet viewer

1.2 Web page Design

1.2.3 Running the Applet

1. Using Java-compatible Web browser :

- To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate APPLET tag.
- Here is the HTML file that executes **SimpleApplet:**

```
<HTML>
    <HEAD>
        <TITLE>Applet HTML Page</TITLE>
    </HEAD>

    <BODY>
        <APPLET code="SimpleApplet.class" width=350 height=200>    </APPLET>
    </BODY>
</HTML>
```

1.2 Web page Design

1.2.3 Running the Applet

2. Using Applet Viewer :

- JDK provides a utility called "appletviewer.exe" for testing applet.
- To execute SimpleApplet with an applet viewer, you need to execute the HTML file using applet viewer.
- For example, if the preceding HTML file is called RunApp.html, then the following command line will run SimpleApplet:
- C:\>appletviewer RunApp.html
- Applet Viewer processes only the <applet> tag, and ignores all other tags in the HTML file.

1.2 Web page Design

4. Passing Parameter to Applet

- It is possible to pass parameter to an Applet program, form HTML file.
- In order to pass parameter to Applet program from HTML page, we need to use <PARAM> tag.
- The general syntax of PARAM tag is given below:
- <PARAM NAME = parameter_name VALUE = parameter_value>

1.2 Web page Design

1.2.4 Passing Parameter to Applet

- Complete syntax for the APPLET tag including Param Tag :

```
<APPLET
        CODEBASE = codebaseURL
        CODE = appletFile
        ALT = alternateText
        NAME = appletInstanceName
        WIDTH = pixels HEIGHT = pixels
        ALIGN = alignment
>
<PARAM NAME = appletAttribute1 VALUE = value>
<PARAM NAME = appletAttribute2 VALUE = value>
. . .

</APPLET>
```

1.2 Web page Design

1.2.4 Passing Parameter to Applet

- Example of Applet with parameters:

```
public class AppletWithPara extends java.applet.Applet
{
public void paint (Graphics gp)
 {
            String au = getParameter("author");
            String ag = getParameter("age");
            String desg = getParameter("designation");
             String inst = getParameter("institute");

            gp.drawString("Author:“ + au , 20 , 40);
            gp.drawString("Age:“ + ag, 20, 70);
            gp.drawString("Designation:“+ desg,  20,  100);
            gp.drawString("Institute:“ + inst, 20, 130);
}
}
```

1.2 Web page Design

1.2.4 Passing Parameter to Applet

- Example of Applet with parameters:

```
<APPLET code=AppletWithPara width=200 height =150>

      <PARAM   NAME=“author”          VALUE="Demo">
      <PARAM   NAME =“age”            VALUE ="20">
      <PARAM   NAME =“designation”    VALUE ="demo mb">
      <PARAM   NAME =“institute”      VALUE ="MB demo">

</APPLET>
```

1.2 Web page Design

1.2.5 Various Methods to Develop basic Applet

Method	Description
void setBackground (Color colorname)	To set the background of an applet window.
void setForeground (Color colorname)	To set the foreground color of an applet window.
Color getBackground ( )	To obtain the current settings for the background color
Color getForeground ( )	To obtain the current settings for the foreground color
Applet getApplet (String name)	To obtain the applet specified by given name from the current applet context.
Void showStatus(String status)	To display the status message in the status bar of applet window
URL getDocumentBase ( )	To obtain the directory of the current browser page.
URL getCodeBase( )	To obtain the directory from which the applet's class file was loaded