



NEHRU COLLEGE OF ENGINEERING AND RESEARCH CENTRE
(NAAC Accredited)
(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS



CS 206 OBJECT ORIENTED DESIGN AND PROGRAMMING

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- ◆ Established in: 2002
- ◆ Course offered : B.Tech in Computer Science and Engineering
M.Tech in Computer Science and Engineering
M.Tech in Cyber Security
- ◆ Approved by AICTE New Delhi and Accredited by NAAC
- ◆ Affiliated to the University of A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.
- 5.

PROGRAMME EDUCATIONAL OBJECTIVES

- PEO1:** Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.
- PEO2:** Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.
- PEO3:** Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.
- PEO4:** Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamwork and leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **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.
3. **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.
4. **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.
5. **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.
6. **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.
7. **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.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **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.
11. **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.
12. **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.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance

optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

CO1	To apply object oriented principles in software design process.
CO2	To develop Java programs for real applications using java constructs and libraries.
CO3	To understand and apply various object oriented features like inheritance, data abstraction, encapsulation and polymorphism to solve various computing problems using Java language
CO4	To implement Exception Handling in java
CO5	Use graphical user interface and Event handling in Java
CO6	To develop and deploy Applet in java

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12
CO1	3	3				3	3				3	3
CO2	3		3			3		3			3	
CO3		3		3	3		3		3	3		3
CO4				3	3				3	3		
CO5			3					3				
CO6			3					3				

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

MAPPING OF COURSE OUTCOMES WITH PROGRAM SPECIFIC OUTCOMES

CO'S	PSO1	PSO2	PSO3
CO1		2	
CO2	2	2	2
CO3	2	2	2
CO4			3
CO5			2
CO6			3

SYLLABUS

Course code	Course Name	L-T-P - Credits	Year of Introduction
CS206	Object Oriented Design and Programming	2-1-0-3	2016
Pre-requisite: CS205 Data structures			
Course Objectives 1. To introduce basic concepts of object oriented design techniques. 2. To give a thorough understanding of Java language. 3. To provide basic exposure to the basics of multithreading, database connectivity etc. 4. To impart the techniques of creating GUI based applications.			
Syllabus Object oriented concepts, Object oriented systems development life cycle, Unified Modeling Language, Java Overview, Classes and objects, Parameter passing, Overloading, Inheritance, Overriding, Packages, Exception Handling, Input/Output, Threads and multithreading, Applets, Event Handling mechanism, Working with frames and graphics, AWT Controls, Swings, Java database connectivity.			
Expected outcome. Students will be able to: 1. apply object oriented principles in software design process. 2. develop Java programs for real applications using java constructs and libraries. 3. understand and apply various object oriented features like inheritance, data abstraction, encapsulation and polymorphism to solve various computing problems using Java language. 4. implement Exception Handling in java. 5. use graphical user interface and Event Handling in java. 6. develop and deploy Applet in java.			
Text Books: 1. Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011. 2. Bahrami A., Object Oriented Systems Development using the Unified Modeling Language, McGraw Hill, 1999.			
References: 1. Y. Daniel Liang, Introduction to Java Programming, 7/e, Pearson, 2013. 2. Nageswararao R., Core Java: An Integrated Approach, Dreamtech Press, 2008. 3. Flanagan D., Java in A Nutshell, 5/e, O'Reilly, 2005. 4. Barclay K., J. Savage, Object Oriented Design with UML and Java, Elsevier, 2004. 5. Sierra K., Head First Java, 2/e, O'Reilly, 2005. 6. Balagurusamy E., Programming JAVA a Primer, 5/e, McGraw Hill, 2014.			

Course Plan			
Module	Contents	Hours (42)	Sem. ExamMarks
I	Object oriented concepts, Object oriented systems development life cycle. Unified Modeling Language, UML class diagram, Use-case diagram. Java Overview: Java virtual machine, <i>data types</i> , <i>operators</i> , <i>control statements</i> , Introduction to Java programming.	08	15%

II	Classes fundamentals, objects, methods, constructors, parameter passing, overloading, access control keywords.	07	15%
FIRST INTERNAL EXAMINATION			
III	Inheritance basics, method overriding, abstract classes, interface. Defining and importing packages. Exception handling fundamentals, <u>multiple catch and nested try statements.</u>	06	15%
IV	Input/Output: files, stream classes, reading console input. Threads: thread model, use of Thread class and Runnable interface, thread synchronization, multithreading.	06	15%
SECOND INTERNAL EXAMINATION			
V	String class - basics. Applet basics and methods. Event Handling: delegation event model, event classes, sources, listeners.	07	20%
VI	Introduction to AWT: working with frames, graphics, color, font. AWT Control fundamentals. Swing overview. Java database connectivity: JDBC overview, creating and executing queries, dynamic queries.	08	20%
END SEMESTER EXAM			

Question Paper Pattern:

1. There will be five parts in the question paper – A, B, C, D, E
2. Part A
 - a. Total marks : 12
 - b. Four questions each having 3 marks, uniformly covering module I and II; All four questions have to be answered.
3. Part B
 - a. Total marks : 18
 - b. Three questions each having 9 marks, uniformly covering module I and II; Two questions have to be answered. Each question can have a maximum of three subparts
4. Part C
 - a. Total marks : 12
 - b. Four questions each having 3 marks, uniformly covering module III and IV; All four questions have to be answered.
5. Part D
 - a. Total marks : 18
 - b. Three questions each having 9 marks, uniformly covering module III and IV; Two questions have to be answered. Each question can have a maximum of three subparts

QUESTION BANK

MODULE I

MODULE I				
Q:NO:	QUESTIONS	CO	KL	PAGE NO:
1	Give the components of use case diagram and its purpose. Draw the Use Case diagram of online railway Ticket reservation system.	CO1	K5	15
2	Write briefly about any five features of Java.	CO1	K1	14
3	Show the use of different type of operators in Java, with the help of suitable examples	CO1	K3	20
4	Write notes on class diagram, Draw a class diagram of an order system	CO1	K2	12
5	Write short notes on any four object oriented concepts	CO1	K2	02
6	Differentiate between while and do-while iterative statements with example	CO1	K4	23
7	Construct the following entities as Java classes with suitable instance variables and methods: i)BankAccount ii)Vehicle iii)Student	CO1	K6	14
8	Write note on Use case Diagram .Draw the use case diagram of an ATM Machine	CO1	K6	08
9	Java is called pure object-oriented language. Give reasons.	CO1	K5	14
10	Write a note on Java Virtual Machine	CO1	K2	16
11	Explain the following statements in Java with examples. i)switch and for ii)break and continue iii)while and do-while	CO1	K6	23

MODULE II

1	Write a java program to check whether a given number is prime or not	CO2	K6	34
2	Illustrate constructor with an example program.	CO2	K6	34
3	Differentiate Default and Parameterized Constructors with an example.	CO2	K4	34
4	Illustrate constructor and method overloading with an example	CO2	K4	35
5	Compare private, protected, and public access modifiers.	CO2	K4	36
6	Write a program to print the sum of n numbers using a constructor . And Point out the characteristics of the constructor	CO2	K6	34
7	Illustrate class with its syntax .Also Point out the parameter passing in java with an example program	CO2	K4	30
8	Explain with an example how access modifiers are used to control the visibility of identifiers.	CO2	K2	36
9	Write a program to print the sum of even numbers in a Given limit using a constructor.	CO2	K6	34
10	Illustrate overloading of methods in java. And also define a Java class having overloaded methods to calculate area of rectangle and circle.	CO2	K4	35

MODULE III

1	Define an exception with an example? Why it needs to be handled	CO3	K1	49
2	Illustrate any two types of inheritance with an example	CO3	K4	56

3	Explain try-catch, Nested-try-catch ,finally Statements with proper examples	CO3	K2	51
4	Illustrate Packages in Java. List any six built-in packages in Java.	CO3	K4	45
5	Describe the concept of interface in Java.	CO3	K2	58
6	Differentiate between abstract class and interface in java with proper example	CO3	K4	58
7	Develop a java package named prime package, with a class Prime containing a static method that check whether a number is prime or not and returns that information. Import this package in another class and use to check a number is prime or not.	CO3	K6	45
8	With the help of examples, explain how inheritance is implemented in Java.	CO3	K6	56
9	Define package. How is a class within a package compiled and executed?	CO3	K1	45
10	List any four exception classes in Java. Briefly explain various exception handling keywords in Java, with examples.	CO3	K1	49
11	Write the steps and java code for creating a package named “AdMath” containing a method to calculate the factorial of a number passed to it. And also mention the Advantage of using a package	CO3	K6	45
12	Define an exception with an example? Why it needs to be handled	CO3	K1	49
MODULE IV				
1	Briefly explain methods used in input and output stream classes	CO4	K2	66
2	Write a program which uses FileReader and	CO4	K6	71

	FileWriter classes to copy an input file (having unicode characters) into an output file:.			
3	Illustrate two ways of creating threads in java with examples	CO4	K4	60
4	With the help of proper examples explain thread synchronization in java	CO4	K6	63
5	With proper examples differentiate byte streams and character streams	CO4	K6	66
6	Briefly explain thread life cycle	CO4	K2	60
7	Write a program to find factorial of an integer given from the keyboard.	CO4	K6	75
8	Using exception handling mechanism, display appropriate message if the input from keyboard is not a validInteger	CO4	K6	75
9	With the help of proper examples How we can read and write data from a console	CO4	K6	71
10	What is thread, write short notes on thread synchronization	CO4	K2	63
11	With the help of examples explain creating multi thread using runnable	CO4	K6	62
MODULE V				
1	Explain life cycle of an applet	CO5	K2	96
2	Explain the working of the Delegation Event Model.	CO5	K2	78
3	With an example explain How graphics is displayed in an applet	CO5	K6	97
4	Explain various string functions with the help of relevant examples	CO5	K2	84
5	Illustrate the different parts of an applet tag.	CO5	K4	94
6	With an example explain creation of applet	CO5	K6	95

7	Explain event handling	CO5	K2	78
8	Sketch and describe architecture of an applet	CO5	K3	95
9	Write an applet that stimulates a desk calculator, Using AWT controls and event handling .it accepts two numbers from user and performs computation (+,-,*,/) based on choice and displays the result	CO5	K6	104
MODULE VI				
1	How to draw graphics using applet	CO6	K4	111
2	With examples explain AWT controls	CO6	K6	108
3	With examples explain JFrame and JTextField	CO6	K6	108
4	Write an applet based program to add two numbers	CO6	K6	104
5	Explain steps using java code for connecting a java program to a database using JDBC API	CO6	K2	116
6	Explain the steps using java code for connecting a Java program to a database using JDBC API	CO6	K2	116
7	List any steps involved in connecting Java to the database	CO6	K1	116
8	Summarize features of swing API	CO6	K1	
9	Identify two event types each generated by a Button, TextField, Checkbox and frame	CO6	K1	108
10	Define swing, What are the advantages of swing over AWT components	CO6	K2	108
11	List out any four container and component classes of swing API	CO6	K1	
12	Explain use of graphics class in java	CO6	K2	

APPENDIX 1

CONTENT BEYOND THE SYLLABUS

S:NO;	TOPIC	PAGE NO:
1	CSS	119

MODULE NOTES

MODULE - I

→ Object Oriented Concepts

* Object Oriented Systems Development life cycle

→ Unified Modelling Language

* UML Class diagram

* Use-case diagram

→ Java Overview

* Java Virtual Machine 3.9 (38)

* Data types 4.4 (46)

* Operators

* Control statements

* Introduction to Java programming

Question Bank:

1. Explain the declaration of array variable in Java. What is dynamic initialization? Give examples.
2. What is meant by polymorphism? Briefly explain the types of polymorphism.
3. What is the o/p of the code? Justify.

```
class Test {
```

```
    public static void main(String a[])
```

```
    { byte b = 50;
```

```
      b = b * 2;
```

```
      System.out.print(b);
```

```
    } }
```

Ans: 100

4. Explain object orientation concepts briefly.
5. List graphical diagrams defined by UML & illustrate with eg.
6. Describe the looping stmts in Java.

Object Oriented Concepts:

Object Oriented Programming (OOP) is an approach to program organization & development. OOP treats data as a critical element in the pgm development & doesn't allow it to flow freely around the slm. OOP allows us to decompose a problem into a no. of entities called Objects & then build data & functions around these entities.

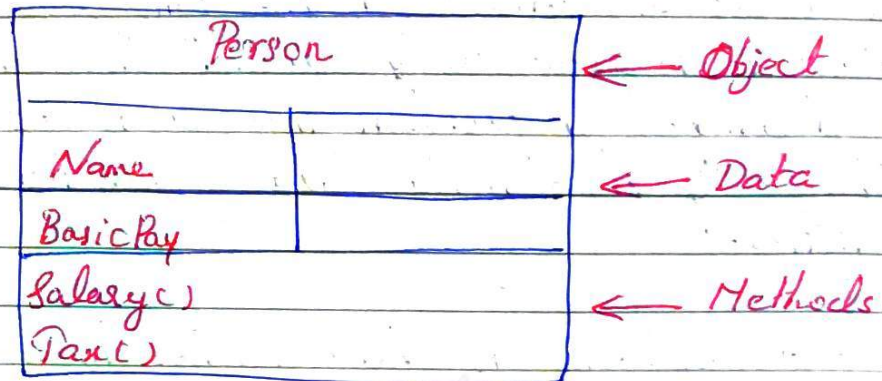
Features of oop:

- Emphasis on data rather than procedure.
- Programs are divided into what are known as Objects.
- Data structures are designed such that they characterise the objects.
- Methods that operate on the data of an object are tied together in the data structure.
- Data is hidden and ~~it~~ cannot be accessed by external fns.
- Objects may communicate with each other through methods.
- New data & methods can be easily added whenever necessary.
- Follows bottom-up approach in pgm design.

Concepts:

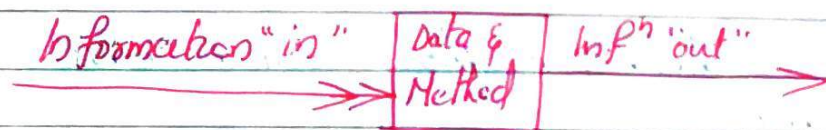
1. **Objects & Classes:** Objects are the basic runtime entities in an object-oriented system. Eg: person, place, Account etc. When a pgm is executed, the objects interact by sending msgs to one another. Each object contains data & code to manipulate the data. Objects can interact without having to know the details of each other's data or objects.

The entire set of data & code of an object can be made a user defined data type using the concept of a class. A class may be data type & an object may be variable of that class.



Data Abstraction & Encapsulation:

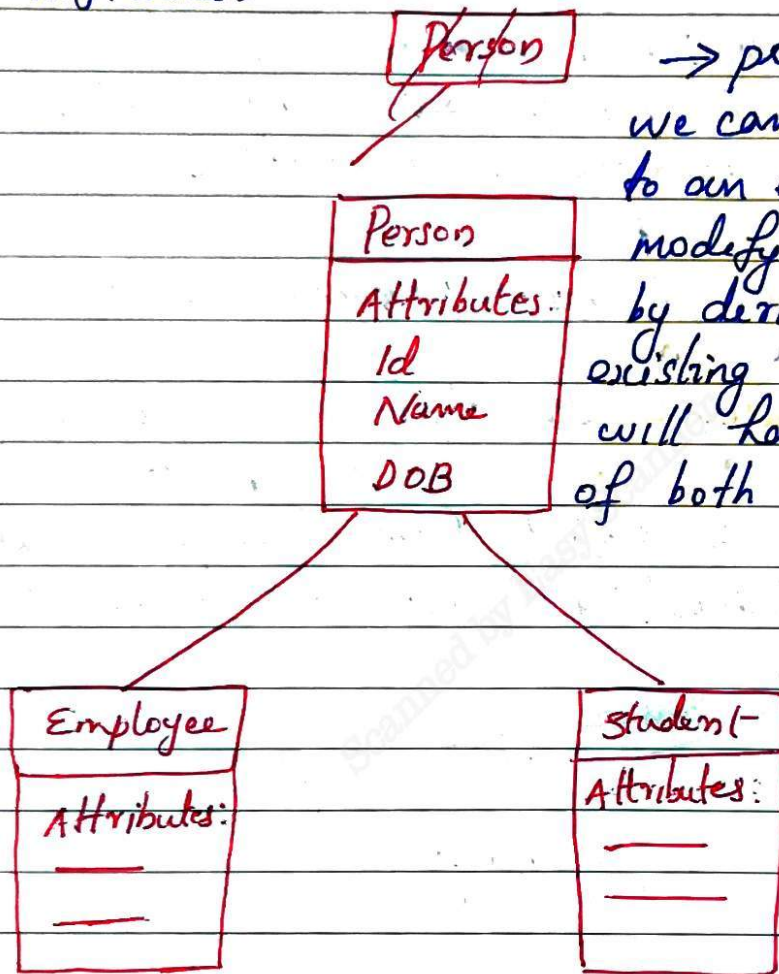
The wrapping up of data & methods into a single unit is known as encapsulation. The data is not accessible to the outside world & only those methods, which are wrapped in the class, can access it. These methods provide interface b/w object's data & the program.



Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction & are defined as a list of abstract attributes such as size, weight & cost & methods that operate on these attributes.

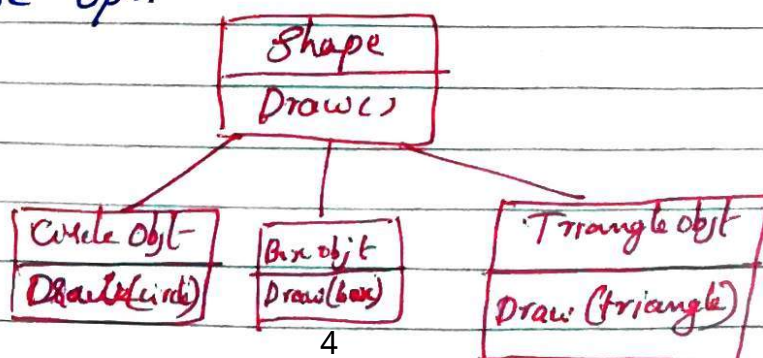
Inheritance:

Inheritance is the process by which objects of one class acquire the properties of objects of another class. Inheritance supports the concepts of hierarchical classification.



→ provides idea of reusability. we can add additional features to an existing class without modifying it. This is possible by deriving a new class from existing one. The new class will have the combined features of both the classes.

Polymorphism: is the ability to take more than one form. An operation may exhibit different behaviours in diff. instances. The behaviour depends upon the types of data used in the opr.



A general class of op's may be accessed in the same manner even though specific actions associated with each opⁿ may differ.

Dynamic Binding: refers to the linking of a procedure call to the code to be executed in response to the call. The code associated with a procedure call is not known until the time of the call at runtime.

Message Communication:

→ set of objects communicate with each other.

Steps:

1. Creating classes that define objects & their behaviour.
2. Creating objects from class definitions.
3. Establishing commⁿ among objects.



Object Oriented Systems Development Life Cycle:

Phases in Object-Oriented Software Development.

The major phases of systems development using object oriented methodology are

1. Object oriented Analysis
2. Object oriented design
3. Object oriented implementation.

1. Object Oriented Analysis: In this phase, the problem is formulated, user requirements are identified, & then a model is built based upon real world objects. The analysis produces models on how the desired system should function & how it must be developed. The models do not include any implementation details so that it can be understood & examined by non-technical application expert.

2. Object-Oriented Design: includes 2 main stages, namely system design & object design.

System design: The complete architecture of the desired system is designed. The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes. System design is done according to both the system analysis model & the proposed system architecture. The emphasis is on the objects comprising the system rather than the processes in the system.

Object Design: In this phase, a design model is developed based on both the models developed in the system analysis phase & architecture is designed in the system design phase. All classes required are identified. The designer decides whether

- new classes are to be created from scratch
- any existing classes can be used in their original form.

— new classes should be inherited from the existing classes.

The associations b/w the identified classes are established & the hierarchies of classes are identified. Developer designs the internal details of the classes & their associations i.e; data structure for each attribute & the algos for ops.

Object Oriented Implementation & Testing: In this phase, the design model developed in the object design is translated into code in an appropriate programming lang. or s/w tool. The databases are created & specific s/w reqmts are ascertained. Once the code is in shape, it is tested using techniques to identify & remove errors.

Unified Modelling Language

UML:

A model is an abstract representation of a s/m constructed to understand the s/m before building or modifying it. It provides the blueprints of the s/m.

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing & documenting the artifacts of a s/w intensive s/m.

— UML is a language: Communication b/w conceptual & physical repⁿ.

— UML visualizes - graphical & textual repⁿ.

— UML specifies - specificⁿ of analysis, design & implⁿ.

— UML documents - exec. code, artifacts about reqmts, Architecture, Design, Source code, project plans etc.

UML contains 3 kinds of building blocks:

* **Things**: abstractions, Relationships tie things together. Diagrams group collⁿ of things.

They are

* **Structural things**: Nouns & static parts of Model. 7 things are Class, Interface, Collaboration, Use Case, Active class, Component & Node.

* **Behavioral Things**: They are verbs & Dynamic parts of UML. They are interaction & State Machine.

* **Grouping Things**: They are organizational parts of UML. Package is a grouping thing.

* **Annotational Things**: They are explanatory parts of models. A Note is an example for it.

* **Relationships**: Relationship is a semantic connection among elements. The diff. kinds of rel'ships in the UML are Dependency, Association (Aggregation), Generalization & Realization.

* **Diagrams**: The nine graphical diagrams of UML are classified into static & dynamic diagrams.

Static/Structural

1. Class Diagram
2. Object Diagrams
3. Implementation Diag.
 - Component
 - Deployment

Dynamic/Behavior

1. Use Case Diag.
2. Interaction Diag.
 - sequence Diag.
 - Collaboration Diag.
3. State chart Diag.
4. Activity Diag.

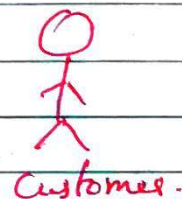
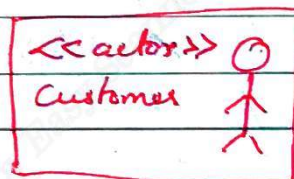
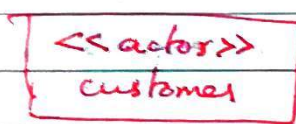
Use Case Diagram:

* **Use Case**: - A use case is an end-to-end process description that includes many steps or transactions; it is not an individual step or activity in a process.

* **Actor**: An actor is a user playing a role w.r. to the s/m. A single actor may perform many usecases. An actor may be an external s/m that needs some infⁿ from the current s/m.

* **Use Case Diagrams**: used to indicate the existence of use cases, actors & their relationships. & courses of actions can be performed. - quickly understand the external actors of a s/m & the best ways in which they can be used.

* **Identifying Actors**: A user may play more than one role. An actor should represent a single user.



Actor notations.

* **Identifying Use Case**:

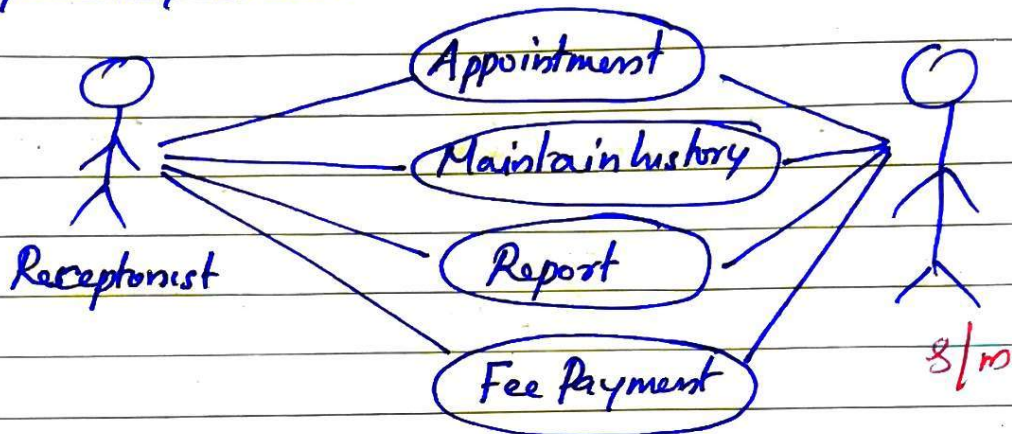
- The processes, tasks, functions initiated or participated, performed by each actors are identified. The usecase should represent a course of events leading to a clear goal.

- The external events that a s/m must respond to are identified.

- Name of the usecases should provide a general description of the use-case function.

Eg: Hospital Information System.

Actors: Doctor, Lab Technician, Patient, Duty Nurse, Receptionist, Visitors.



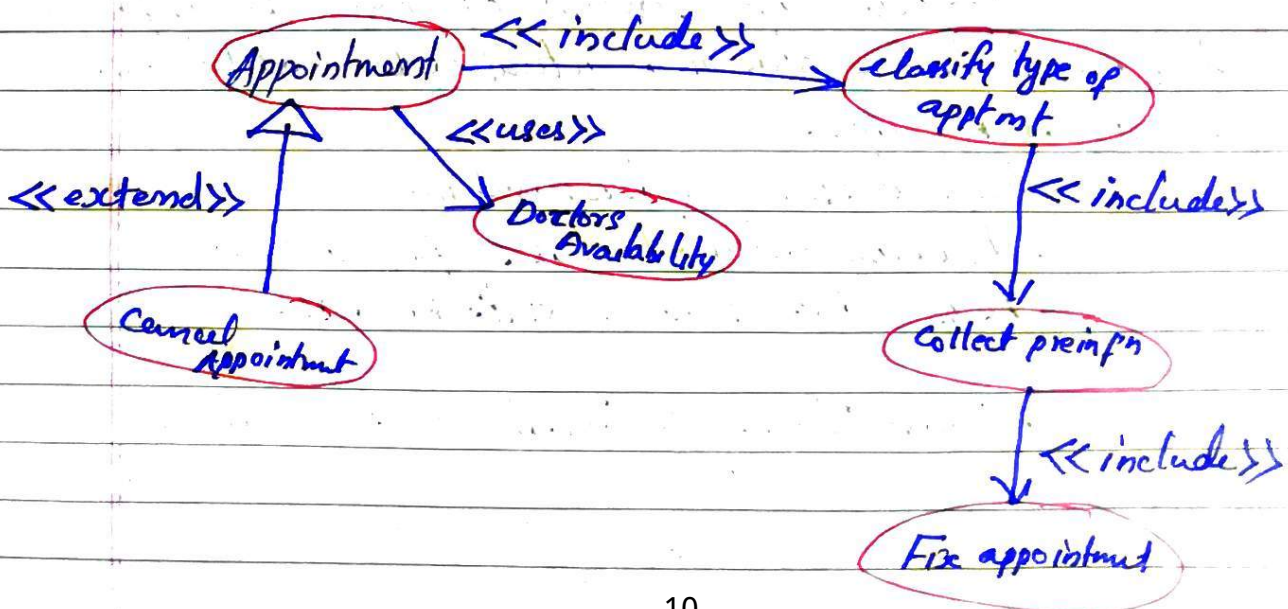
Relationships:

- **Communications:** Connecting the actor symbol to the usecase symbol with a solid path shows the commⁿ relⁿship of an actor in a use case.

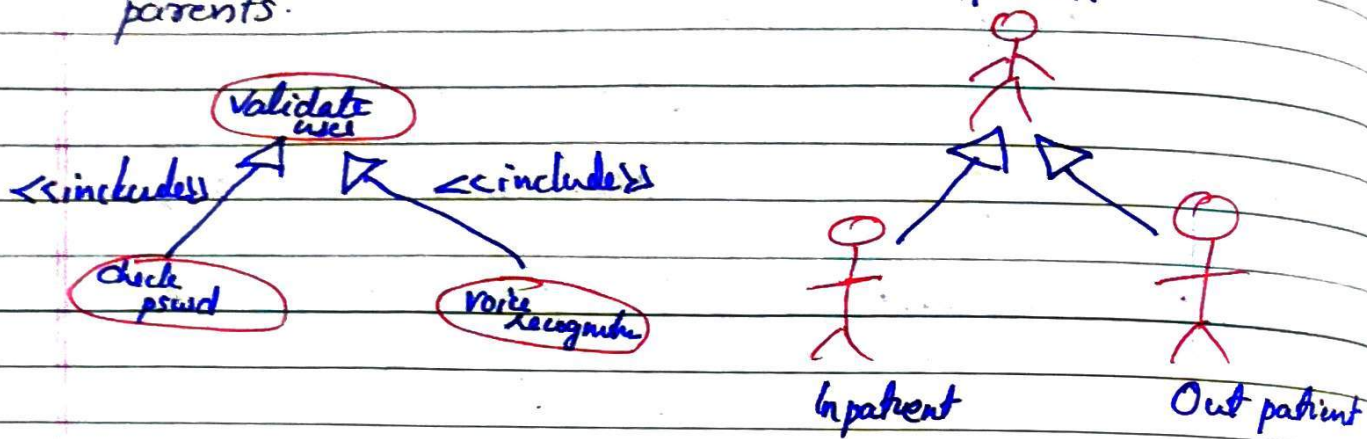
- **Uses:** It occurs when the usecases have some subflows.

- **include:** A usecase continues description of behaviour, by including another usecase.

- **Extend:** when a slm has a sub usecase.
egⁿ: exceptions.



Generalization: Relationship a usecase/actor & the usecase/actor. Sub usecases inherits behaviours of parents.



UML class Diagrams:

The class defines what object can do. A class diagram describes the types of objects in the s/m & the diff. types of relationships that exist among them.

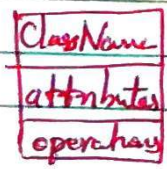
UML class diagram gives an overview of a s/m by displaying classes, attributes, op's & their relationships. Class diagram helps construct the code for the s/m applⁿ development.

Benefits:

- Class diagram illustrates data models for even very complex infⁿ s/ms.
- provides an overview of how the applⁿ is structured before studying actual code. easily reduce maintenance time.
- helps better understanding
- drawing detailed charts for code.
- helps developers & stakeholders.

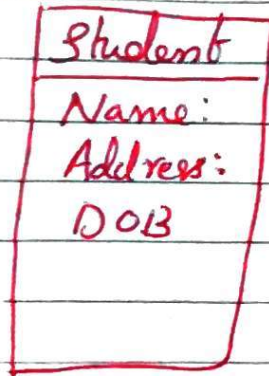
Elements of class Diagram:

* Class Name * Attributes * Operations



Class Name: It needed for graphical repⁿ.

Attributes: is named as property of a class which describes the objt being modeled.



Derived Attribute can be there.
 for eg: Age from DOB.

Relationships:

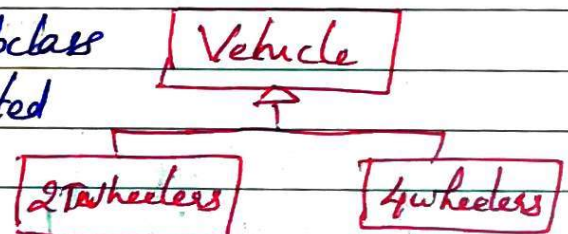
1. Dependancies
2. Generalizations
3. Associations

Dependency: the relation b/w 2 or more classes in which a change in one may force changes in the other.
 - indicates one class depends on another.

Eg:



Generalization: connect a subclass to its superclass. Subclass inherited from its superclass.



Association: ~~Association~~ **Specialization:** Specific features of a set of objects that have not been distinguished by the classes already defined.

Association: static relationship b/w classes A & B. For eg. an employee works for an organization.

Multiplicity: specifies the association is mandatory or not.



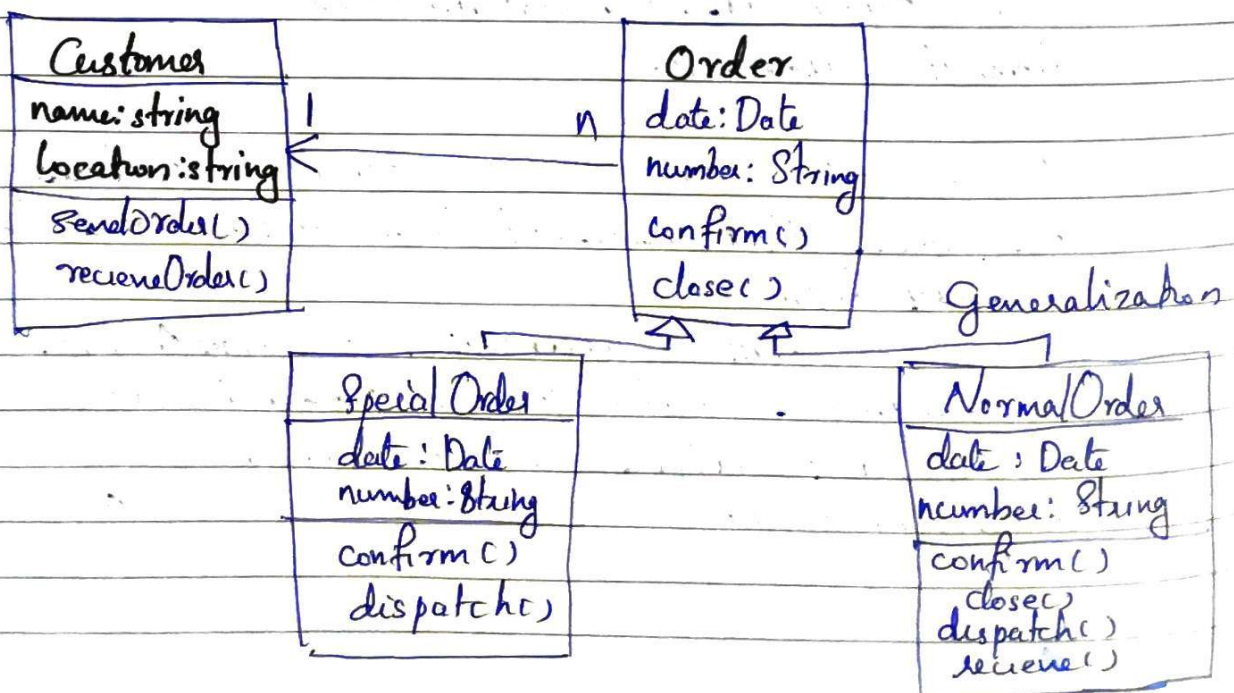
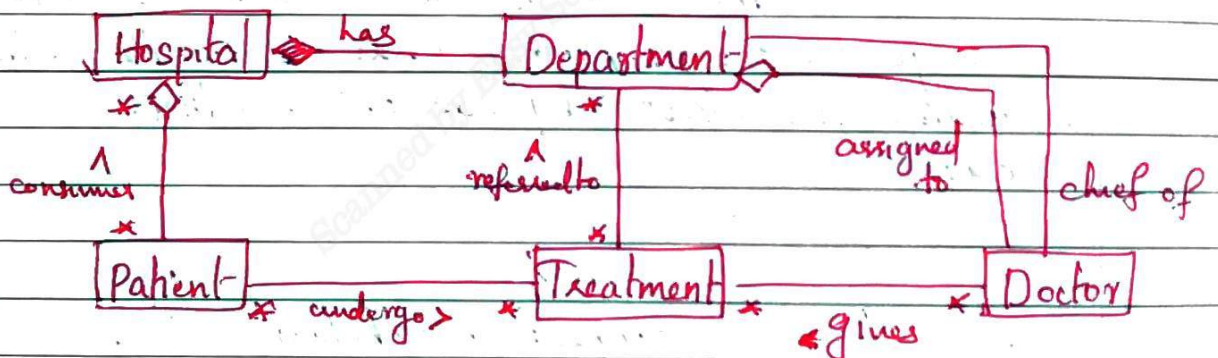
Aggregation: is a special type of association that models a whole-part relationship b/w aggregate & its parts.

College  **Student** The college will remain even if the student is not available.

Composition: special type of aggregation which denotes strong ownership b/w 2 classes when one class is a part of another class.

College  **Student**

College is composed of classes student. The college could contain many students - each student belong to only one college.



JAVA OVERVIEW

Java Features:

1. **Compiled & Interpreted:** First Java compiler translates source code into bytecode instructions (not m/c inst's). Second Java interpreter generates m/c code that can be directly executed by the m/c.
2. **Platform-independent & Portable:** Java pgms can be easily moved from one computer s/m to another, anywhere, any time. Changes & upgrades in O.S, processors & s/m resources will not force any changes in Java programs.
3. **Object Oriented:** All pgm code & data reside within objts & classes. The object model in Java is simple & easy to extend.
4. **Robust & Secure:** It provides safeguards to ensure reliable code. It has strict compile time & runtime checking for data types. Java incorporates the concept of exception handling which captures series errors & eliminates risk of crashing the systems.
5. **Distributed:** creating appl's on networks. It has the ability to share both data & programs.
6. **Simple, Small & Familiar:** Java doesn't use pointers, preprocessor header files, goto statement & many others. It also eliminates operator overloading & multiple inheritance.
7. **Multithreaded & Interactive:** supports multithreaded programs. - No need to wait for the application to finish one task before beginning another.
 - support multiprocess synchronization & construct interactive s/ms.
8. **High performance:** - due to the use of intermediate byte code, reduces overheads during runtime.

9. Dynamic & Extensible: capable of dynamically linking in new class libraries, methods & objects.

Java supports functions written in other lang. such as C, C++ . These fns are called native methods.

10. Ease of Development: reduce work of the programmer by shifting the responsibility of creating the reusable code to the compiler.

Java Program Structure

- Documentation Section
 - Package stmt eg: package student;
 - Import stmts. import student.test;
 - Interface stmts
 - Class Defⁿ.
 - Main method class
- ```
{
 Main method Defn.
}
```

Java Tokens: Reserved Keywords, Identifiers, Literals, Operators, Separators.

### Implementing a Java Program:

1. Creating the Program

Eg: Class Test

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



Program must be saved in a file called Test.java. ensuring that the file name contains the class name properly.

## 2. Compiling the Program

— Run java Compiler javac  
javac Test.java

The javac compiler creates a file called Test.class containing bytecodes of the pgm.

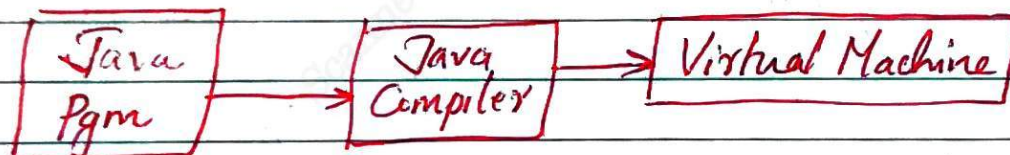
## 3. Running the pgm.

— At the command prompt,  
Java Test

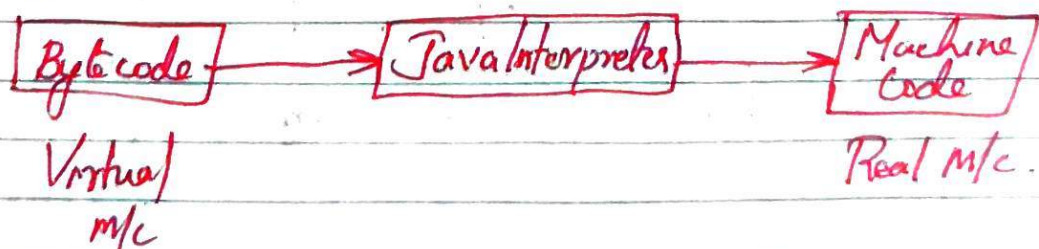
the interpreter looks for the main method in the pgm & begins ex<sup>n</sup> from there.

Hello

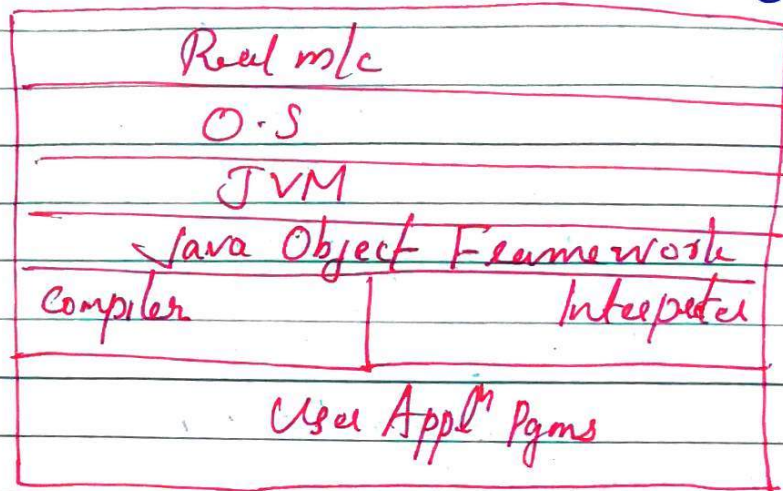
## Java Virtual Machine:



Java Compiler produces an intermediate code known as bytecode for a m/c that doesn't exist. This m/c is called Java Virtual Machine & it exists only inside the computer memory. The virtual machine code is not m/c specific code. The m/c specific code is generated by Java interpreter by acting as an intermediary b/w virtual m/c & real m/c.



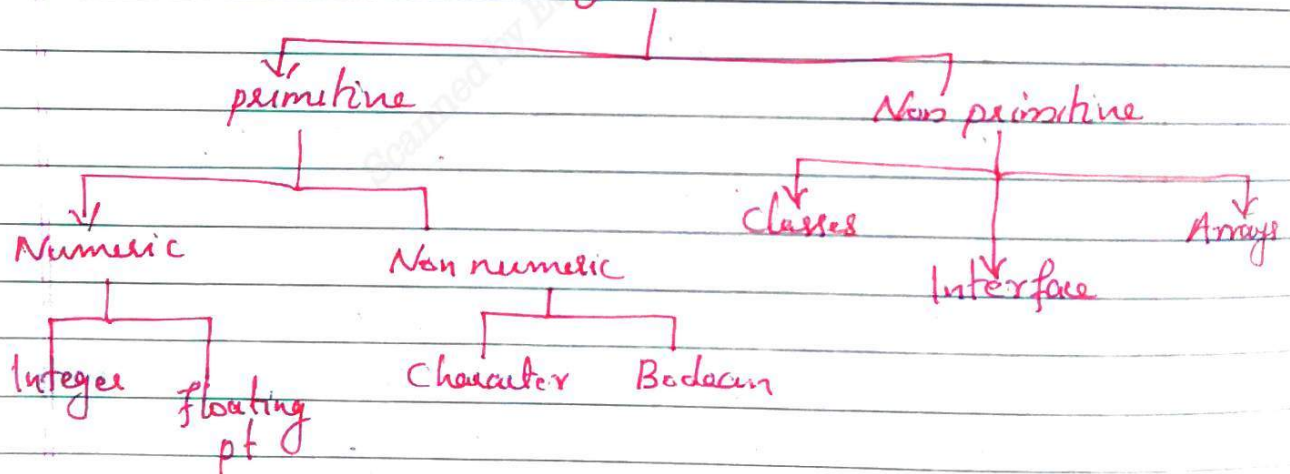
## Layers of interactions for Java programs



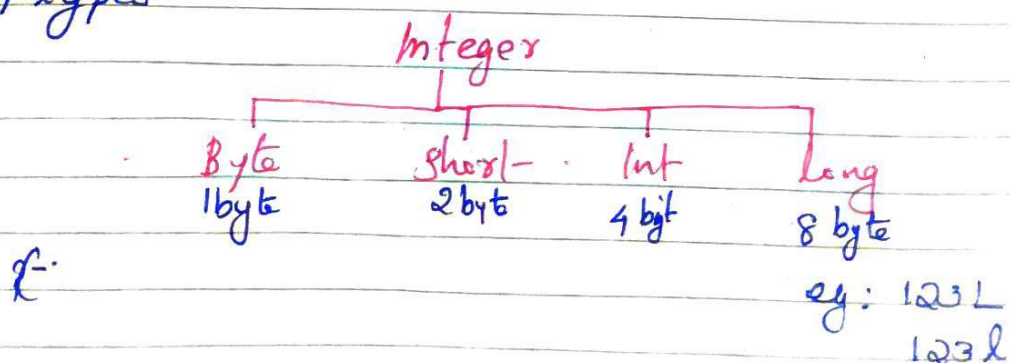
## Data types:

## Module II

### Data types in Java



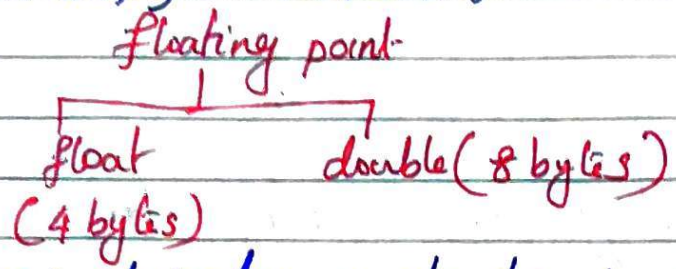
Integer types: can hold whole numbers. Java supports 4 types.





Floating point Types: numbers containing fractional parts.

Eg: 1.23f, 7.56923e5f



Character Type: character constants in memory.

- size of 2 bytes but single character.

Boolean Type: when we want to test a particular condition during the execution of the program. Two values that a boolean type can take: true or false.

Declaration of variables:

type variable 1, ... variable n;

Eg: int c, a;  
 float x, y;  
 byte b;  
 char c1;

Giving values to Variables:

variable name = value;

c = 0;

c1 = 'x'; x = y = z = 0;

type variable name = value;

int finalvalue = 100;

char yes = 'x';

double total = 75.36;

public static void main(String args[]) defines method main() - starting pt. of execution.

public - access specifier that declares the main method accessible to all other classes.

static - declares this method belongs to the entire class not a part of any objects of the class.



void : main method doesn't return any value simply prints to the screen.

## Reading data from keyboard

```
import java.io.DataInputStream;
```

```
class Reading
```

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

```
{ DataInputStream in = new DataInputStream(System.in);
```

```
int intNumber = 0;
```

```
float floatNumber = 0.0f;
```

```
try
```

```
{ System.out.println("Enter an Integer");
```

```
int intNumber = Integer.parseInt(in.readLine());
```

```
System.out.println("Enter float number");
```

```
float floatNumber = Float.valueOf(in.readLine()).
```

```
floatValue();
```

```
}
```

```
catch (Exception e) { }
```

```
System.out.println("int Number = " + intNumber);
```

```
System.out.println("float Number = " + floatNumber);
```

```
}
```

```
}
```

Output

Enter an integer

123

Enter a floatNumber

123.45

intNumber = 123

floatNumber = 123.45



## Operators

Java operators classified into

1. Arithmetic Operators,
2. Relational
3. Logical
4. Assignment
5. Increment and Decrement
6. Conditional
7. Bitwise
8. Special Operators.

### Arithmetic Operators:

$+, -, *, /, \%$

Integer Arithmetic:  $a/b$ , when  $a$  &  $b$  are integer types gives the result of division of  $a$  by  $b$  after truncating the divisor.  $5/2 = 2$ . ( $a=5, b=2$ )

Module division  $\% : a \% b = a - (a/b) * b$

Real Arithmetic:

Modulus operator  $\%$  can be applied to the floating point equivalent of data. It returns the floating pt equivalent of integer division.

float  $a = 20.5F, b = 6.4F;$

$a \% b = 1.3$

Mixed mode Arithmetic: When one of the operands is real & the other is integer, the expression is called a mixed mode arithmetic expression. If either operand is the real type, then the other operand is converted to real & the real arithmetic is performed.

$15/10.0 = 1.5$

$15/10 = 1$



## Relational Operator:

$<, <=, >, >=, ==, !=$

When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions ~~er~~ will be evaluated first. Arithmetic operators have higher priority over relational operators.

## Logical Operators:

$\&\&, ||, ! \Rightarrow$  logical AND, OR, NOT

Eg:  $a > b \&\& x == 10$

## Assignment Operators: (=)

$S = a + b;$  — to assign value of an exp. to a variable.

$v \text{ op} = \text{exp};$  // shorthand assignment operators.

eg:  $x++$

$v \text{ op} = \text{exp}$  is eqt. to  $v = v \text{ op}(\text{exp});$   
 $\uparrow$   $v$  accessed only once.

$x += y + 1;$  //  $x = x + (y + 1);$

$x += 3;$  //  $x = x + 3;$

$a = a * (n + 1); \Rightarrow a *= n + 1;$

$a = a / (n + 1); \Rightarrow a /= n + 1;$

3 advantages:

— what appears on the left hand side need not be repeated

— more concise & easy

— more efficient code.

## Increment & decrement Operators:

$++$  &  $--$



## Class Increment Operator

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

```
{ int m=10, n=20;
```

```
System.out.println("m=" + m);
```

```
System.out.println("n=" + n);
```

```
System.out.println("++m=" + ++m);
```

```
System.out.println("n++=" + n++);
```

```
System.out.println("m=" + m);
```

```
System.out.println("n=" + n);
```

```
}
```

```
}
```

Output: m=10

n=20

++m=11

n++=20

m=11

n=21

## Conditional Operator: ? :

exp1 ? exp2 : exp3

Eg: a=10; b=15;

x=(a>b)? a: b;

ie; if (a>b)

x=a;

else

x=b;

## Bitwise Operators:

& (AND), | (OR), ^ (XOR), ~ (NOT), <<, >>, >>> (Shift right with zero fill)

## Special Operators:

Instance of Operator:

Eg: person instance of student is true if object person belongs to the class student;

## Dot Operator (.)

— to access instance of variables & methods of class objects.

Eg: ~~Person~~ person.age;

- Evaluation of Expression
- Type casting.

## Control Statements:

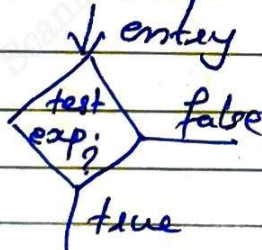
If statement:

Switch statement:

Conditional operator statement:

If statement:

if (test expression)



1. Simple if statement
2. if...else statement
3. Nested if...else stmt
4. else if ladder

Simple if statement:

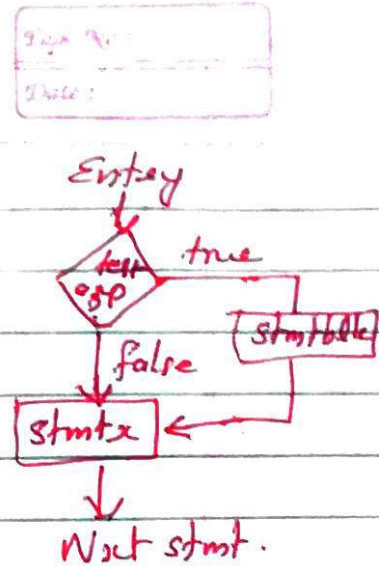
```

if (test exp)
{ stmt blk;
}
stmt x;

```



Eg: `if (category == sports)`  
`{ marks = marks + bonus;`  
`}`  
`System.out.println(marks);`  
`---`  
`---`



Eg: `class IfTest`

`{ public static void main(String arg[])`

`{ int i, count, count1, count2;`

`float[] weight = {45.0F, 55.0F, 47.0F,`  
`51.0F, 54.0F};`

`float[] height = {175.5F, 174.2F, 168.0F,`  
`170.7F, 169.0F};`

`count = 0;`

`count1 = 0;`

`count2 = 0;`

`for (i = 0; i <= 4; i++)`

`{ if (weight[i] < 50.0 & height[i] > 170.0)`

`{ count1 = count1 + 1;`

`}`

`count = count + 1; // Total person`

`}`

`count2 = count - count1;`

`System.out.println("No. of persons with -");`

`System.out.println("weight < 50 & height > 170" + count);`

`System.out.println("Others = " + count2);`

`}`

`}`

Output: No. of persons with  
 weight < 50 and height > 170 = 1  
 Others = 4<sup>24</sup>

## The If...Else Statement

```

if(test exp)
{ True blk stmts
}
else
{ false blk stmts
}
stmt-x

```

Write a pgm to count even & odd no.s in an array.

class IfElseTest

```

{ public static void main(String args[])

```

```

{ int N[] = {50, 65, 56, 71, 81};

```

```

 int even = 0, odd = 0;

```

```

 for(int i=0; i<N.length; i++)

```

```

 { if((N[i]%2)==0)

```

```

 {

```

```

 even+=1;

```

```

 }

```

```

 else

```

```

 {

```

```

 odd+=1;

```

```

 } }

```

```

 System.out.println("Even Nos : " + even + "Odd Nos : "
 + odd);

```

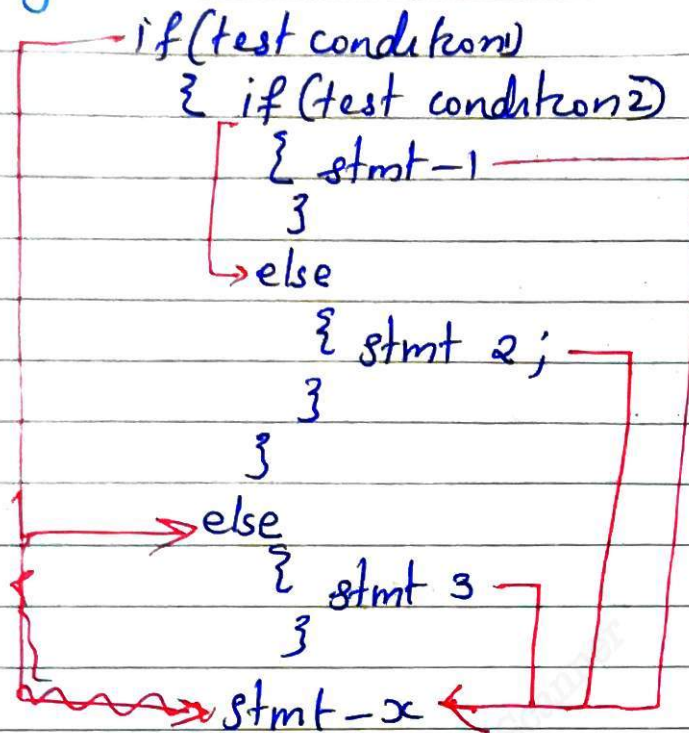
```

}

```



## Nesting of If...Else Statements



class IfElseNesting

```

{ public static void main(String args[])
{ int a = 325, b = 712, c = 478;
 System.out.println("Largest value is");
 if (a > b)
 { if (a > c)
 { System.out.println(a);
 }
 else
 { System.out.println(c);
 }
 }
 else
 { if (c > b)
 { System.out.println(c);
 }
 else
 { System.out.println(b);
 }
 }
}
}

```

Chh  
5/2/19

## The Else If Ladder

```
if (condition 1)
 stmt 1;
else if (condition 2)
 stmt 2;
else if (condn 3)
 stmt 3;
else if (condition n)
 stmt n;
else
 default stmt;
stmt x;
```

Write a pgm to print grade card.

## Switch Statement

```
switch (expression)
{
 case value-1: blk-1
 break;
 case value-2: blk-2
 break;
 - - -
 default:
 default blk
 break;
}
```

stmt-x;



```
class CityGuide
```

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

```
{
```

```
 char choice;
```

```
 System.out.println("Select ur choice");
```

```
 System.out.println(" M → Madras B → Bombay
 C → Calcutta");
```

```
 System.out.println(" choice ");
```

```
 //
```

```
 switch (choice = (char) System.in.read())
```

```
 { case 'M' :
```

```
 case 'm' : System.out.println("Madras");
```

```
 case 'B' : break; System.out.println("Bombay");
```

```
 case 'b' : break; System.out.println("Bombay");
```

```
 case 'C' :
```

```
 case 'c' : break; System.out.println("Calcutta");
```

```
 default : System.out.println("Invalid");
```

```
 }
```

```
}
```

```
}
```

Conditional Operator: ? : Operator.

Conditional exp ? exp1 : exp2



## Module II

- Classes Fundamentals
- Objects
- Methods
- Constructors
- parameter passing
- Overloading
- access control keywords

### Question Bank

1. Write note on access ctrl keywords in Java.
2. Explain overloading in Java with eg.
3. Explain parameter passing in Java with suitable eg.
4. Describe the use of constructor in Java with eg.
5. List out Java features.
6. Define constructor
7. Demonstrate method overloading
8. Explain parameter passing methods in Java.
9. Explain constructor & explain its category with eg.
10. Explain concept of class with eg. method
11. Constructor overloading & operator overloading in Java.
12. Explain Overloading in Java.
13. Illustrate control stmts in Java with eg.

April 2018

1. Explain how access modifiers are used to ctrl the visibility of identifiers.
2. Define a Java class having overloaded methods to calculate area of rectangle & circle.
3. Use of constructors with examples.
4. Is it possible to create an object for class A using, `Aobj = new A();` if the class contains only parameterised constructor? why?



## Classes Fundamentals:

Defining a class:

— class is a user defined data type

```
class class name [extends superclassname]
```

```
{
```

```
 [fields declaration]
```

```
 [methods declaration]
```

```
}
```

### Fields declaration

Data is encapsulated in a class by placing data fields inside the body of the class definition. These variables are called instance variables.

Eg: class Rectangle

```
{
```

```
 int length;
```

```
 int breadth;
```

```
}
```

Instance variables also known as member variables.

### Methods

Methods are declared inside the body of the class but immediately after the declaration of instance variables.

```
type method name (parameter list)
```

```
{
```

```
 method body;
```

```
}
```

4 basic parts: method name, return type, parameter list, body of method.

```

Eg: class Rectangle
 { int length, width;
 void getData(int x, int y)
 {
 length = x;
 width = y;
 }
 int rectArea()
 {
 int area = length * breadwidth;
 return area;
 }
 }

```

class Access

```

{
 int x;
 void method1()
 {
 int y;
 x = 10; // legal
 y = x; // legal
 }
 void method2()
 {
 int z;
 x = 5;
 z = 10;
 y = 1; // illegal
 }
}

```

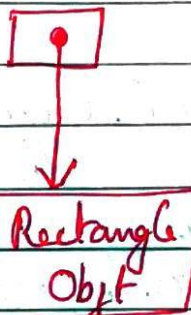
⇒ Instance variables & methods in classes are accessible by all methods in the class but a method cannot access the variables declared in other methods.



## Objects

Objects in Java are created using the new operator. The new operator creates an object of the specified class & returns a reference to that object.

eg: `Rectangle rect1;` // declare object  
`rect1 = new Rectangle();` // instantiate object

| Action             | Statement                             | Result                                                                                                                             |
|--------------------|---------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|
| <u>Declare</u>     | <code>Rectangle rect1;</code>         | <u>null</u> rect1                                                                                                                  |
| <u>Instantiate</u> | <code>rect1 = new Rectangle();</code> |  <p>rect1 is a ref. to<br/>Rectangle objt.</p> |

Both stmts can be combined into one

```
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle();
```

## Accessing class Members:

Since we are outside the class, we cannot access the instance variables & the methods directly. we must use concerned object & the dot operator

objt name . variablename = value;

objt name . methodname (parameter-list);

The instance variables of the Rectangle class may be accessed & assigned values as follows.

`rect1.length = 15; rect1.width = 10;`

`rect2.length = 20; rect2.width = 12;`



rect1.length 15  
rect1.width 10

rect2.length 20  
rect2.width 10

Another & more convenient way of assigning values to the instance variables is to use a method that is declared inside the class.

- the method getData can be used.

Rectangle rect1 = new Rectangle();  
rect1.getData(15, 10); // calling method using the obj.

The code creates rect1 obj & then passes in the values 15 & 10 for the x & y parameters of the method getData

```
void getData(int x, int y)
{
 length = x;
 width = y;
}
```

Eg: Pgm

```
Class Rectangle
{
 int length, width;
 void getData(int x, int y)
 {
 length = x;
 width = y;
 }
 int rectArea()
 {
 int area = length * width;
 return (area);
 }
}
```



```
class RectArea
```

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

```
{ int area1, area2;
```

```
Rectangle rect1 = new Rectangle(); //creating
Rectangle rect2 = new Rectangle(); Objects
```

```
rect1.width = 10;
```

```
rect1.length = 15;
```

```
area1 = rect1.length * rect1.width;
```

```
rect2.getData(20, 12);
```

```
area2 = rect2.rectArea();
```

```
System.out.println("Area1=" + area1);
```

```
System.out.println("Area2=" + area2);
```

```
}
}
```

Output : Area 1 = 150

Area 2 = 240

### Constructors:

All objects that are created must be given initial values. Java supports a special type of method, called a constructor, that enables an object to initialize itself when it is created.

Constructors have the same name as the class itself. They do not specify a return type, not even void. They return the instance of the class itself.

```
class Rectangle
```

```
{ int length, width;
```

```
Rectangle(int x, int y)
```

```
{ length = x;
```

```
width = y;
```

```
}
```

```
int rectArea()
```

```
{ return (length * width);
```

```
}
```



```
class RectangleArea
```

```
{ public static void main(String args[])
{ Rectangle rect1 = new Rectangle(15, 10); // calling Constructor
 int area1 = rect1.rectArea();
 System.out.println("Area1=" + area1);
}
```

## Methods Overloading

It is possible to create methods that have same name, but diff. parameter list & diff. definitions. This is called method overloading. Method overloading is used when objects are required to perform similar tasks but using diff. i/p parameters. When we call a method in an object, Java matches up the method name first & then the no. & type of parameters to decide which one of the def's to execute. This process is known as polymorphism.

Eg: class Room

```
{ float length;
```

```
 float breadth;
```

```
 Room(float x, float y)
```

```
 { length = x;
```

```
 breadth = y;
```

```
 }
```

```
 Room(float x)
```

```
 { length = breadth = x;
```

```
 }
```

```
 int area();
```

```
 { return (length * breadth);
```

```
 }
```

```
 Room ()
```

```
 { length = 1;
```

```
 breadth = 1;
```

```
 }
```



Here we are overloading the constructor method `Room()`. An object representing a rectangular room will be created as

```
Room room1 = new Room(25.0, 15.0); // constructor 1
Room room2 = new Room(20.0); // constructor 2
Room room3 = new Room(); // constructor 3
```

Access Control keywords: / Visibility Control

The visibility modifiers are also known as access modifiers. They are

- public
- private
- protected

**public Access:** Any variable or method is visible to the entire class in which it is defined.

```
Eg: public int number;
 public void sum()
 {
 }
```

**Friendly Access:** When no access modifier is specified, the member defaults to a limited version of public accessibility known as "friendly" level of access.

The diff. b/w "public" & "friendly" access is that public modifier makes fields visible in all classes, while friendly access makes fields visible only in the same package but not in other packages. A package is a group of classes stored separately.

**protected Access:** The visibility of "protected" lies in b/w public & friendly access.

- makes the fields visible not only to all classes & subclasses in the same package but also to subclasses in other packages. Non subclasses in other packages cannot access the protected members.



**private Access:** private fields - highest degree of protection. They are accessible only with their own class. They cannot be inherited by subclasses & therefore not accessible in subclasses. A method declared as private behaves like a method declared as final.

**private protected Access:**

private protected int Number;  
 - visibility level b/w protected & private.  
 - makes the fields visible in all subclasses regardless of what package they are in. These fields are not accessible by other classes in the same package.

| Access Modifier               | public | protected | friendly (default) | private protected | private |
|-------------------------------|--------|-----------|--------------------|-------------------|---------|
| Access location               |        |           |                    |                   |         |
| same class                    | Yes    | Yes       | Yes                | Yes               | Yes     |
| Subclass in same package      | Yes    | Yes       | Yes                | Yes               | No      |
| other class in same package   | Yes    | Yes       | Yes                | No                | No      |
| subclass in other package     | Yes    | Yes       | No                 | (Yes)             | No      |
| Non subclass in other package | Yes    | No        | No                 | No                | No      |



## Arrays: Declaration.

① `type arrayname[];`

② `type[] arrayname;`

eg: `int N[];`

`float avg[];`

`int[] N;`

`float[] marks;`

Creation of Arrays. Java allows us to create arrays using 'new' operator

`arrayname = new type[size];`

Eg: `N = new int[5];`

`Avg = new float[10];`

Start  
`int N[];`  
Result  
`N`   $\rightarrow$  points nowhere.

`number = new int[5];`  
`N`   $\downarrow$  points to obj  


## Initialization of Arrays:

`arrayname[subscript] = value;`

`type arrayname[] = {list of values}`

Eg: `int N[] = {35, 40, 20, 57, 19};`

It is possible to assign an array object to another.

Eg: `int a[] = {1, 2, 3};`

`int b[];`

`b = a;`

Array length: All arrays store the allocated size in a variable length. Eg: `a.length;`

class NumberSorting

{

public static void main(String args[])

{ int N[] = {55, 40, 80, 65, 71};

int n = N.length;

System.out.print("given list");

for (int i = 0; i < n; i++)

{ System.out.print(" " + N[i]);

}

System.out.println("\n");

// sorting

for (int i = 0; i < n; i++)

{ for (j = i + 1; j < n; j++)

{

if (N[i] < N[j])

{ int t = N[i];

N[i] = N[j];

N[j] = t;

}

}

}

System.out.print("Sorted list");

for (int i = 0; i < n; i++)

{ System.out.print(" " + N[i]);

}

System.out.println(" ");

}

}

*[Handwritten signature]*

*[Handwritten mark]*



## Parameter Passing:

→ Call by value: This method copies the value of an arg. into the formal parameter of the subroutine. Therefore changes made to the parameter of the subroutine have no effect on the arg.

→ Call by reference: A ref. to an arg. (not the value of the arg) is passed to the parameter. Inside the subroutine, this ref. is used to access the actual arg. specified in the call. This means that changes made to the parameter will affect the arg. used to call the subroutine.

// simple types are passed by value:

Class Test

```
{ void math(int i, int j)
 { i *= 2;
 j /= 2;
 }
```

→ the op's that occur inside math() have no effect on the values of a & b used in the call. Their values didn't change to 30 & 10.

Class callByValue

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

```
{ Test ob = new Test();
```

```
 int a = 15, b = 20;
```

```
 System.out.println("a & b before call" + a + " " + b);
```

```
 ob.math(a, b);
```

```
 System.out.println("a & b after call" + a + " " + b);
```

```
}
```

O/p: a & b before call : 15 20

a & b after call : 15 20.

Scanned by Easy Scanner



## Parameter Passing:

→ Call by value: This method copies the value of an arg. into the formal parameter of the subroutine. Therefore changes made to the parameter of the subroutine have no effect on the arg.

→ Call by reference: A ref. to an arg. (not the value of the arg) is passed to the parameter. Inside the subroutine, this ref. is used to access the actual arg. specified in the call. This means that changes made to the parameter will affect the arg. used to call the subroutine.

// Simple types are passed by value.

Class Test

```
{ void math(int i, int j)
 { i *= 2;
 j /= 2;
 }
```

→ the ops that occur inside math() have no effect on the values of a & b used in the call - their values didn't change to 20 & 15.

Class callByValue

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

```
{ Test ob = new Test();
```

```
 int a = 15, b = 20;
```

```
 System.out.println("a & b before call " + a + " " + b);
```

```
 ob.math(a, b);
```

```
 System.out.println("a & b after call " + a + " " + b);
}
```

```
}
```

O/p: a & b before call : 15 20

a & b after call : 15 20.



Objects are passed by reference:

```
class Test
{
 int a, b;
 Test(int i, int j)
 {
 a = i; b = j;
 }
 // pass an object
 void math(Test o)
 {
 o.a *= 2;
 o.b /= 2;
 }
}
```

```
class CallByRef
{
 public static void main(String args[])
 {
 Test ob = new Test(15, 20);
 System.out.println("ob.a & ob.b
before call" + ob.a + " " + ob.b);
 ob.math(ob);
 System.out.println("ob.a & ob.b
after call" + ob.a + " " + ob.b);
 }
}
```

O/P ob.a & ob.b before call: 15 20  
ob.a & ob.b after call: 30 10

When you pass this ref. to a method, the parameter that receives it will refer to the same objt as that referred to by the arg. changes to the objt inside the method do affect the objt used as an arg.

Returning an Object:

```
class Test
{
 int a;
 Test(int i)
 {
 a = i;
 }
 Test incByTen()
 {
 Test temp = new Test(a+10);
 return temp;
 }
}
```

```
class RetOb
{
 public static void main(String args[])
 {
 Test ob1 = new Test(2);
 Test ob2;
 ob2 = ob1.incByTen();
 System.out.println("ob1.a: " + ob1.a);
 System.out.println("ob2.a: " + ob2.a);
 ob2 = ob2.incByTen();
 System.out.println("ob2.a after
second incr. " + ob2.a);
 }
}
```

O/P: ob1.a: 2 ob2.a after  
ob2.a: 12 inline 22.



## MODULE III & IV Question Bank

1. Explain the use of packages in Java.
2. Write a note on streams related classes.
3. Define thread synchronization.
4. Write a pgm to create a package named mypack, containing a class RightTriangle in which a static method check whether 3 given sides of a triangle forms a right angled ~~to~~ triangle & returns that info? Import this package in another class.
5. Explain exception handling mechanism, with suitable examples.
6. Write short notes on Inheritance with example. ✓<sup>2</sup>
7. Write a Java pgm that counts no. of words in a text.
8. Explain how multiple threads are created in a pgm, with example.
9. Illustrate the use of Runnable Interface in Java.
10. Describe how we can create a thread. Explain adv. of multithreaded programming.
11. Describe the multiple Inheritance with example.
12. Write notes on method overriding.
13. Define thread in Java.
14. Demonstrate interface in Java.
15. Describe abstract classes in Java.
16. Write notes on try & catch statements in java with eg.
17. Illustrate finally statement & exceptions in Java.
18. Describe the concept of stream classes with eg.
19. Explain the use of thread class in Java.
20. Describe any 6 packages in Java with eg.
21. Write notes on extending interfaces.
22. Discuss about the similarities b/w i/f & classes.
23. Define phases in life cycle of a thread.

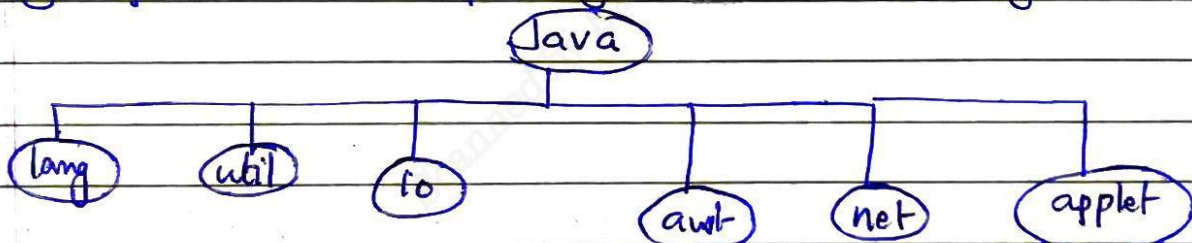


## MODULE III

- > Inheritance Basics
- > Method Overriding
- > Abstract Classes
- > Interface
- > Defining & Importing packages
- > Exception handling fundamentals
- > Multiple catch & nested try statements.

### > Defining & Importing Packages:

JAVA API provides a large no. of classes grouped into diff. packages acc. to finality.



java.lang:— Language support classes. These are classes that Java compiler itself uses & therefore they are automatically imported. They include classes for primitive types, strings, math fns etc.

java.util:— classes such as vectors, hash tables, random numbers, date etc.

java.io:— Input/output classes — facilities for i/p & o/p data.

java.awt:— classes for graphical user Interface. eg: classes for buttons, lists, menus etc.

java.net:— classes for networking; classes for communicating with local computers as well as internet servers.

java.applet:— classes for creating & implementing applets.



## Using 8/10 Packages:

Java includes import stmts to bring certain classes or entire packages into visibility.

`import package.name.classname;`

or

`import package.name.*;`

eg: `import java.awt.Color;` imports the class `Color` & therefore the class name can now be directly used in the pgm.

`import java.awt.*;` will bring all classes of `java.awt` package.

`java` - package contains `awt` package - contains classes.

## Creating Packages:

→ first declare the name of the package using the `package` keyword followed by a package name.

Steps: 1. Declare the package at the beginning of a file

using the form: `package package name;`

2. Define the class that is to be put in the package & declare it public.

3. Create subdirectory under the directory where the main source files are stored.

4. Store the listing as the `classname.java` file in the subdirectory created.

5. Compile the file. This creates `.class` file in the subdirectory.

Using a Package:

```

package package1;
public class ClassA
{
 public void displayA()
 {
 System.out.println("class A");
 }
}

```

This source file should be named classA.java & stored in the subdirectory package1. Compile java file.



ClassA.class file in the same subdirectory.

```

import package1. Class A;
class PackageTest1
{
 public static void main(String args[])
 {
 Class A objectA = new ClassA();
 objectA.displayA();
 }
}

```

Now let us consider another package named package2 containing again a single class

```

package package2;
public class Class B
{
 protected int m=10;
 public void displayBC()
 {
 System.out.println("class B");
 System.out.println("m = " + m);
 }
}

```

Importing classes from other packages.

Eg:

```

import package1. Class A;
import package2. *;
class PackageTest2
{
 public static void main(String args[])

```



```

{ classA objectA = new classA();
 classB objectB = new classB();
 objectA.displayA();
 objectB.displayB();
}
}

```

pgm saved as PackageTests.java - compiled & run

```

class A
class B
m=10

```

### Adding a class to a package:

Eg: package p1;  
 public class A  
 { // body of A }

The package p1 contains one public class by name A.  
 → add another class B to this package.

1. Define the class & make it public.
2. Place the package stmt.

```

package p1;
public class B
{ // body of B }

```

3. Store this as B.java file under the directory p1.
4. Compile B.java file. This will create a B.class file & place it in the directory p1.

Eg: package MyPack;

public class Balance

{ string name;

double bal;

public Balance(string n, double b)

{ name = n;

bal = b;

public void show()

{ if (bal < 0)

System.out.println(name + ":" + bal);

}

import MyPack.\*;

class TestBalance

{ public static void main  
(String args[])

{

Balance Test = new Balance

("ABC", 99.68);

Test.show();

}

}

## Exception Handling Fundamentals:

### Dealing with Errors:

An error may produce an incorrect o/p or may terminate the ex<sup>n</sup> of the pgm abruptly or even may cause the s/m to crash.

Types of Errors:

→ Compile time errors

→ Run time errors.

Compile time errors: All syntax errors will be detected & displayed by the Java Compiler & therefore errors are known as Compile time errors. Whenever the compiler displays an error, it will not create the .class file. It is therefore necessary that we fix all the errors before we can successfully compile & run the pgm.

Eg: class Error1

{ public static void main(String args[])

{ System.out.println("Hello Java") // missing ;

}



Error1.java:7 ';' expected  
System.out.println("Hello Java!")  
^  
/error.

Most of the compile-time errors are due to typing mistakes.

\* Missing semicolons \* missing brackets ..

Run time Errors:

Sometimes a pgm may compile successfully creating the .class file but may not run properly. Such pgms may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow.

Eg: Dividing an integer by zero  
out of bounds of array

Eg: class Error2

```
{ public static void main(String args[])
 { int a=10; int b=5; int c=5;
```

```
 int x = a/(b-c); // Division by zero
```

```
 System.out.println("x=" + x);
```

```
 int y = a/(b+c);
```

```
 } System.out.println("y=" + y);
}
```

while executing, it displays the follo. msg & stops without executing further stmts.

java.lang.ArithmeticException: / by zero  
at Error2.main(Error2.java:10)



## Exceptions:

An exception is a condition that is caused by a run-time error in the pgm. When the java interpreter encounters an error such as dividing an integer by zero, it creates an exception objt & throw it (ie; informs us that an error has occurred).

Java exception handling is managed via five keywords: try, catch, throws & finally

pgm statements that you want to monitor for exceptions are contained within a try block.

If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) & handle it in some rational manner. System generated exceptions are automatically thrown by java run-time s/m.

To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throw clause. Any code that absolutely must be executed before a method returns is put finally block.

try

```
{ // blk of code to be monitored }
catch (ExceptionType1 exOb)
```

```
{ // exception handler for Exception }
```

finally

```
{ // blk of code to be executed before try blk ends. }
}
```

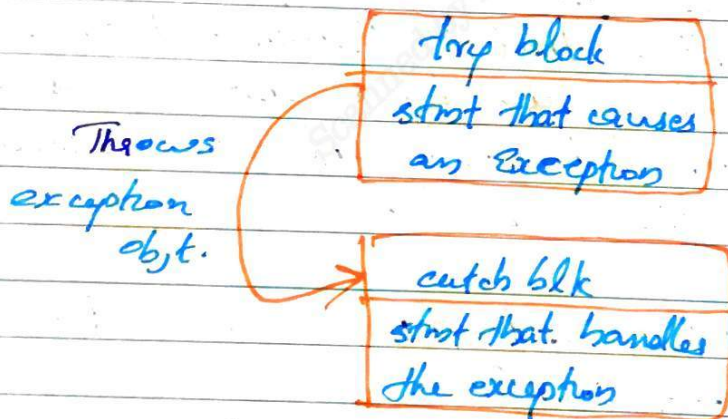
If the exception objt is not caught & handled properly, the interpreter will display an error msg & will terminate the pgm.



If we want the prog to continue with the ex<sup>n</sup> of the remaining code, then we should try to catch the exception obj thrown by the error condition & then display an appropriate msg for taking corrective actions. This task is known as exception Handling.

The purpose of exception handling mechanism is to provide a means to detect & report an exceptional circumstance so that appropriate action can be taken.

1. Find the pblm (Hit the exception)
2. Inform that an error has occurred (Throw the exception)
3. Receive the error inf<sup>n</sup> (catch the exception)
4. Exception Handling Take corrective actions.



→ try keyword: prefixes a blk of code that is likely to cause an error condition & "throw" an exception.

→ catch blk: catches the exceptions "thrown" by the try blk & handles it. The catch blk is added immediately after the try blk.

```
try
{ stmt; } // generate an exception
catch (Exception-type e)
{ statement; } // processes exception
```



The try blk can have one or more stmts that would generate an exception. If any one stmt generates an exception, the remaining stmts in the blk are skipped & ex<sup>n</sup> jumps to the catch blk that is placed next to the try blk.

The catch stmt is passed a single parameter, which is ref. to the exception thrown. If catch parameter matches with the type of exception objt, then the exception is caught & stmts in the catch blk will be executed.

Class Error3

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

```
{ int a=10, b=5, c=5;
```

```
int x, y;
```

```
try { x = a/(b-c); // Exception here
```

```
catch (ArithmeticException e)
```

```
{ System.out.println("Division by zero");
```

```
y = a/(b+c);
```

```
System.out.println("y = " + y);
```

```
} }
```

O/p

Division by zero

y = 1



## Multiple catch statements.

```
try
{
 stmt; // generates an exception
}
catch (Exception-Type 1 e)
{
 statement; // processes exception by type 1
}
catch (Exception type-2 e)
{
 stmt;
}
...
catch (Exception-Type N e)
{
 stmt; // processes exception Type N
}
```

Eg: class Error4

```
{
 public static void main(String args[])
 {
 int a[] = {5, 10};
 int b = 5;
```

```
 try
 {
 int x = a[0] / b - a[1];
 }
```

```
 catch (ArithmeticException e)
```

```
 {
 System.out.println("Division by zero");
 }
```

```
 catch (ArrayIndexOutOfBoundsException e)
```

```
 {
 System.out.println("Array Index Error");
 }
```

catch (ArrayStoreException e)

```
{
 System.out.println("wrong data type");
}
```

```

 int y = a[1]/a[0];
 System.out.println("y=" + y);
}
}
```

O/p:

Array Index error

y=2.

### Using finally statement

Java supports finally stmt that can be used to handle an exception that is not caught by any of the previous catch stmts.

finally blk can be used to handle any exception generated within a try blk. It may be added immediately after the try blk or after the last catch blk shown.

```
try
{
 --
}
finally
{
 --
}
```

```
try
{
 --
}
catch
{
 --
}
:
finally
{
 --
}
```

when a finally blk is defined, this is guaranteed to execute, regardless of whether or not an exception is thrown.

ref. above eg:-

Eg: finally

```
{
 int y = a[1]/a[0];
 System.out.println("y=" + y);
}
```

This will produce same o/p.



## Nested try statements

A try stmt can be inside the blk of another try. Each time a try stmt is entered, the context of that exception is pushed on the stack. If an inner try stmt doesn't have a catch handler for a particular exception, the stack is unwound & the next try stmt

## Inheritance

→ by creating new classes & reusing the properties of existing ones. The mechanism of deriving a new class from an old one is called inheritance.

The old class is known as the base class or super class or parent class & the new one is called subclass or derived class.

### Defining Subclass

```
class subclassname extends superclassname
{
 variable declaration;
 method declaration;
}
```

Eg: Single Inheritance.

```
class Room
{
 int length;
 int b;
 Room(int x, int y)
 {
 l = x; b = y;
 }
 int area()
 {
 return (l * b);
 }
}
```

```
class Bedroom extends Room
{
 int h;
 Bedroom(int x, int y, int z)
 {
 super(x, y); // pass values to super class
 h = z;
 }
 int volume()
 {
 return (l * b * h);
 }
}
```



class InherTest

```
{ public static void main (String args[])
{ Bedroom room1 = new Bedroom(14, 20, 10);
 int area1 = room1.area();
 int volume1 = room1.volume();
 System.out.println ("Area1 = " + area1);
 System.out.println ("Volume1 = " + volume1);
}
}
```

### Subclass Constructor

— is used to construct the instance variables of both the subclass & the superclass. The subclass constructor uses the keyword `super` to invoke the constructor method of the super class. The keyword `super` is used subject to the follo. conditions.

### Method Overriding

— when we want an obj to respond to the same method but have diff. behaviours when that method is called. i.e; we should override the method defined in the superclass. This is possible by defining a method in the subclass that has same name, same args & same return type as method in the super class. when that method is called the method defined in subclass is executed. This is known as overriding

class A

```
{ int i, j;
 A(int a, int b) {
 i = a; j = b;
 }
 void show() {
 System.out.println ("i = " + i + " j = " + j);
 }
}
```

class B extends A

```
{ int k;
 B (int a, int b, int c) {
 super(a, b);
 k = c;
 }
 void show() {
 System.out.println ("k = " + k);
 }
}
```



## Class Override

```
{ public static void main(String args[])
 { B subOb = new(1,2,3);
 subOb.show();
 }
}
```

## Abstract Classes

By making a method final, we ensure that the method is not redefined in a subclass. i.e; the method can never be subclassed. We can indicate that the method must always be redefined in a subclass, thus making overriding compulsory.

Eg: abstract class Shape

```
{

 abstract void draw();

}
```

When a class contains one or more abstract methods, it should also be declared abstract.

## Interfaces: Multiple Inheritance:

Interface defines only abstract methods & final fields.

```
. Interface InterfaceName
{ variables declaration;
 methods declaration;
}
```

Eg: interface Item

```
{ static final int code = 1001;
 static final String name = "Fan";
 void display();
}
```

## Extending Interfaces.

Interfaces can also be extended. i.e; An i/f can be subinterfaced from all other i/fs.

```
interface name2 extends name1
{
 body of name2
}
```

Eg: interface Item Constants

```
{
 int code = 1001;
 String name = "Fan";
}
```

```
interface Item extends ItemConstants
{
 void display();
}
```

## Implementing Interfaces:

```
class classname implements InterfaceName
{
 body of classname
}
```

```
class classname extends superclass implements
 interface1, interface2...
```

```
{
 body of classname
}
```

~~Ch~~  
4/1/19



## MODULE IV

A pgm contains 2 or more parts that can run concurrently. Each part of such pgm is called a thread & each thread defines a separate path of ex<sup>n</sup>. Thread exist in several states:

→ running, ready, suspend/resume, wait, terminated.

### Thread Priorities:

— how that thread should be treated w.r. to others. Thread priorities are integers that specify the relative priority of one thread to another. A thread's priority is used to decide when to switch from running thread to the next. This is called context switch.

To set a thread's priority, use the `setPriority()` method which is a member of `Thread`.

`final void setPriority(int level)`

Level specifies the new priority setting for the calling thread. The value of level must be within the range `MIN-PRIORITY` and `MAX-PRIORITY` (1 to 10). To return a thread to default priority, specify `NORM-PRIORITY` which is 5. Current priority can be obtained by calling `getPriority()` method of `Thread`.

### Creating Thread:

— by instantiating an objt of type `Thread`.

Two ways:

- Implementing `Runnable`
- Extending the `Thread` class.

#### Implementing Runnable:

To implement `Runnable`, a class need only implement a single method called `run()`



public void run()

Inside run(), you will define the code that will constitute the new thread. After create a class that implements Runnable, instantiate an obj. of type Thread from within that class.

class NewThread implements Runnable

{ Thread t;

New Thread();

{ t = new Thread(this, "Demo Thread");

system.out.println("child thread:" + t);

t.start();  
}

public void run()

{ for(int i=5; i>0; i--)

{ system.out.println("Child Thread:" + i);

Thread.sleep(500);

}  
}

system.out.println("Exiting child thread");

}

class ThreadDemo

{ public static void main(String args[])

{ new NewThread();

{ for(int i=5; i>0; i--)

{ system.out.println("Main thread" + i);

Thread.sleep(1000);

}  
}

system.out.println("Main thread exiting");



O/p

child thread: Thread [Demo Thread, 5, main]

Main Thread 5    MainThread 4    MainThread: 3

Child Thread 5    childThread 3    childThread: 1

Child Thread 4    childThread 2    Exiting child Thread

MainThread: 2

MainThread: 1

Main Thread exiting.

t = new Thread(this, "Demo Thread");

### Extending Thread

```
class NewThread extends Thread
```

```
{ NewThread()
```

```
{ super("Demo Thread");
```

```
 system.out.println("Child thread: " + this);
```

```
 start();
```

```
}
```

```
 public void run()
```

```
{
```

```
 for(int i=5; i>0; i--)
```

```
 { system.out.println("Child Thread: " + i);
```

```
 Thread.sleep(500);
```

```
 }
```

```
 } system.out.println("Exiting child thread");
```

```
} }
```

```
class ExtendThread
```

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

```
 { new NewThread();
```

```

 for(int i=5; i>0; i--)
 {
 system.out.println("Main Thread "+i);
 Thread.sleep(1000);
 }
 system.out.println("Main thread exiting");
}

```

## Thread Synchronization

Threads try to use data & methods outside themselves. On such occasions, they may compete for the same resources & may lead to serious problems.

For eg: one thread may try to read a record from a file while another is still writing to the same file.

### the keyword synchronized

For eg: the method that will read info from a file & the method that will update the same file may be declared as synchronized.

```

synchronized void update()
{
 -
}

```

When we declare a method synchronized, Java creates a monitor & hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code. It is also possible to mark a blk of code as synchronized

```

synchronized (lock obj)
{
 -
}

```



## Deadlock

Assume that thread A must access the Method 1 before it can release Method 2 but the thread B cannot release Method 1 until it gets hold of Method 2.

Thread A

```
synchronized method2()
{
 synchronized method1()
 {
 // ...
 }
}
```

Thread B

```
synchronized method1()
{
 synchronized method2()
 {
 // ...
 }
}
```

eg: class Callme

```
{
 void call(String msg)
 {
 System.out.println("[" + msg);
 Thread.sleep(1000);
 System.out.println("]");
 }
}
```

class Caller implements Runnable

```
{ String msg;
 Callme target;
 Thread t;
 public Caller(Callme targ, String s)
 { target = targ;
 msg = s;
 t = new Thread(this);
 t.start();
 }
}
```

```
public void run()
{ target.call(msg);
}
```

```
{
}
```

class Synch

```
{ public static void main (String args []).
{ Callme target = new Callme();
 Caller ob1 = new Caller (target, "Hello");
 Caller ob2 = new Caller (target, "synchronized");
 Caller ob3 = new Caller (target, "world");
 ob1.t.join();
 ob2.t.join();
 ob3.t.join();
}
```

O/P [Hello [synchronized [world]

By calling sleep(), the call() method allows ex to switch to another thread. This results in the mixed-up O/P of the 3 msg strings.

class Callme

```
{ synchronized void call (String msg)
{
}
```

This prevents other threads from entering call() while another thread is using it.

O/P [Hello]  
[synchronized]  
[world].



# Input Output (Module III)

**Files:** Storing data on 2<sup>o</sup> storage devices like disks. The data is stored in these devices using the concept of files. Data stored in files is called persistent data.

A file is a collection of related records placed in a particular area on the disk. A record is composed of several fields & a field is a group of characters.



(a) Reading data into a pgm



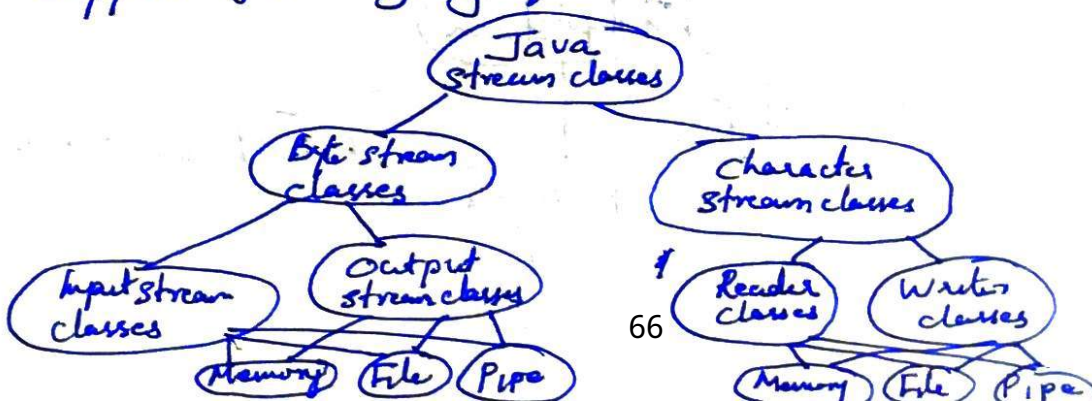
(b) writing data to a dest<sup>n</sup>

## Stream Classes

The java.io package contains a large no. of stream classes that provide capabilities for processing all types of data.

↳ Byte stream classes that provide support for handling I/O op's on bytes.

↳ Character stream classes that provide support for handling I/O op's on characters.



## Byte Stream Classes

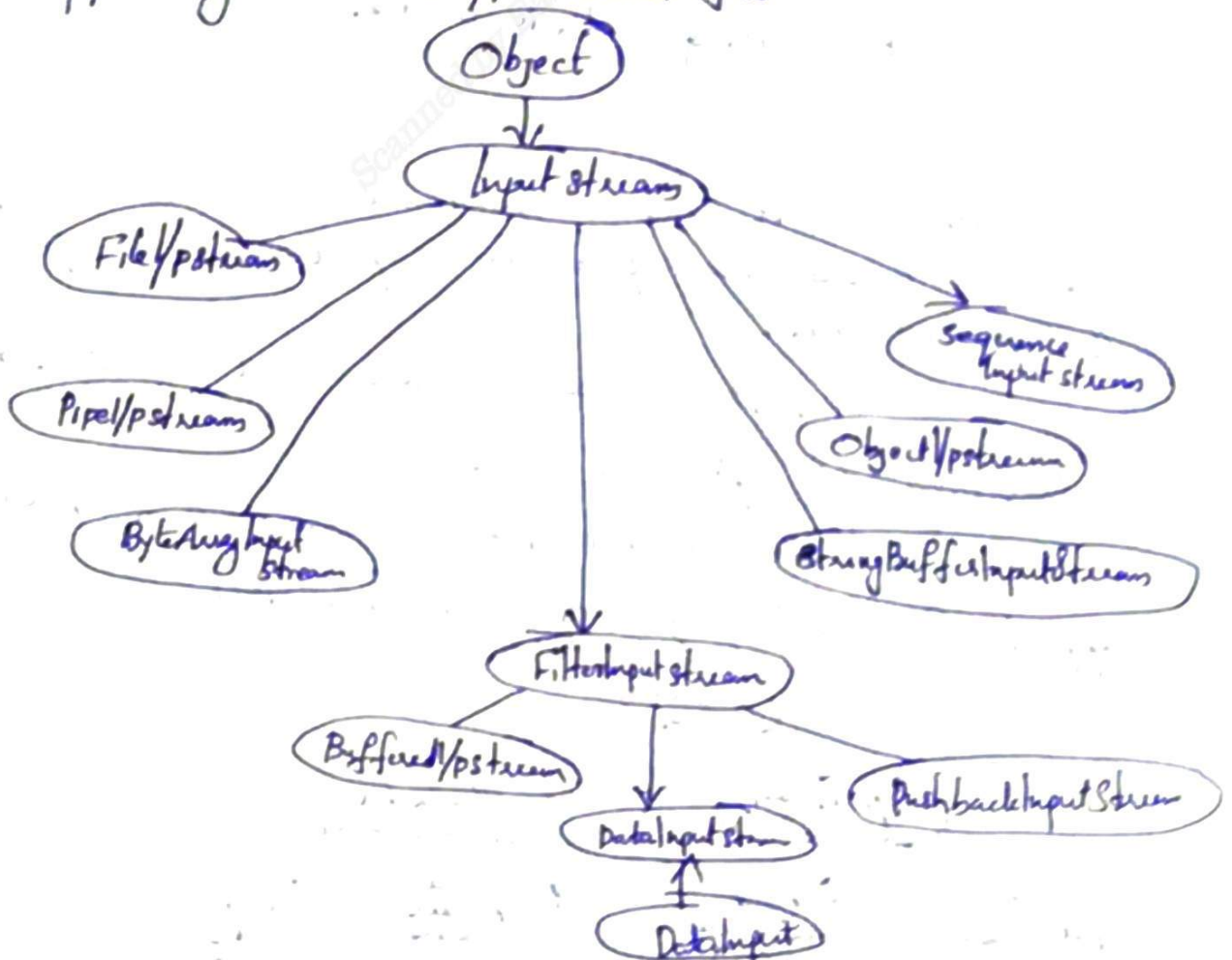
→ to provide final features for creating & manipulating streams & files for reading & writing bytes.

→ Input Stream Classes

→ Output Stream Classes

## Input Stream Classes

→ used to read 8-bit bytes include a superclass known as `InputStream` and a no. of subclasses for supporting various I/O related fns.

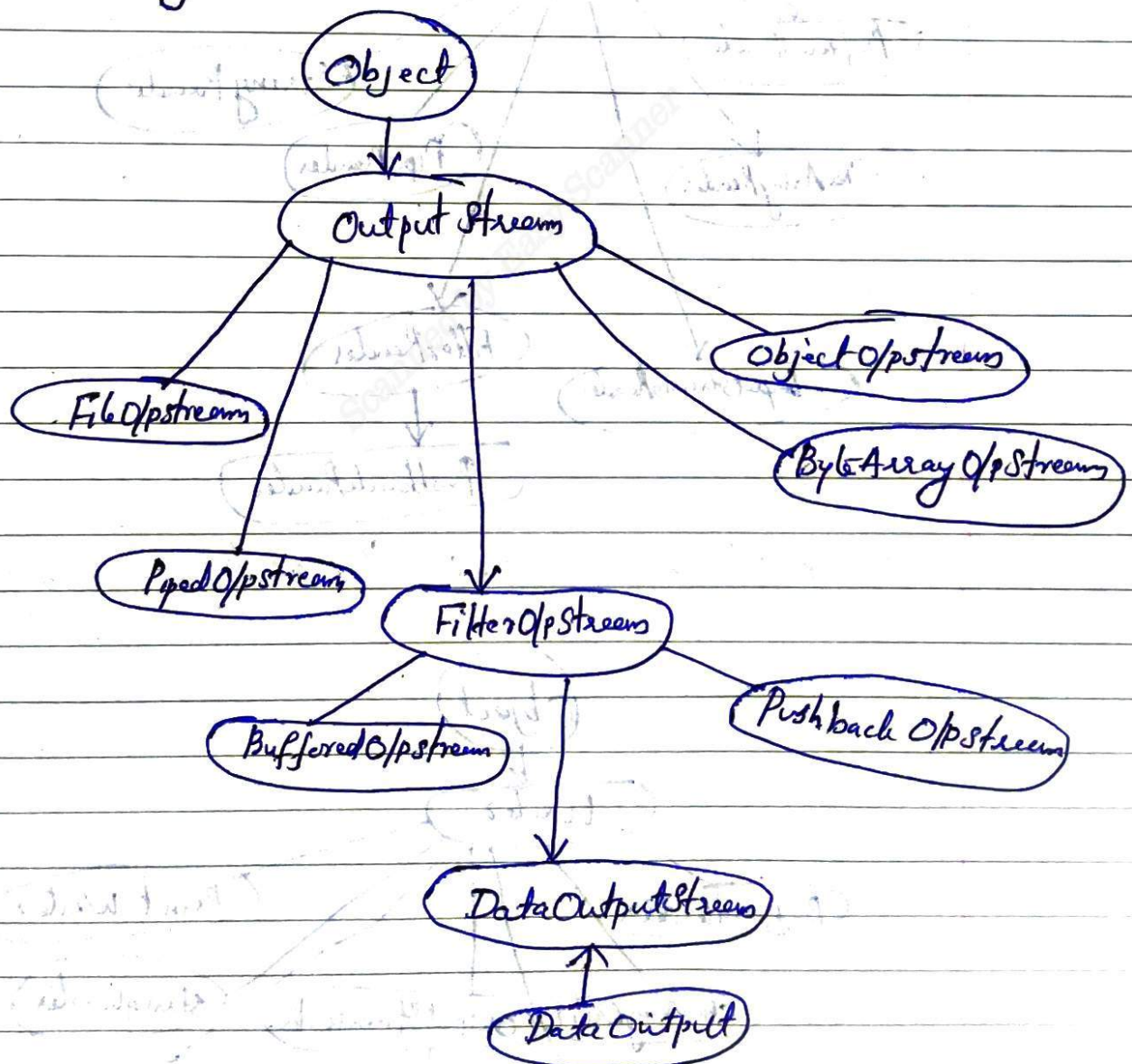




InputStream class defines methods for performing i/p fns such as \* Reading bytes \* Closing streams \* Marking positions in streams \* Stepping ahead in a stream \* Finding the no. of bytes in a stream. Ref: page 292.

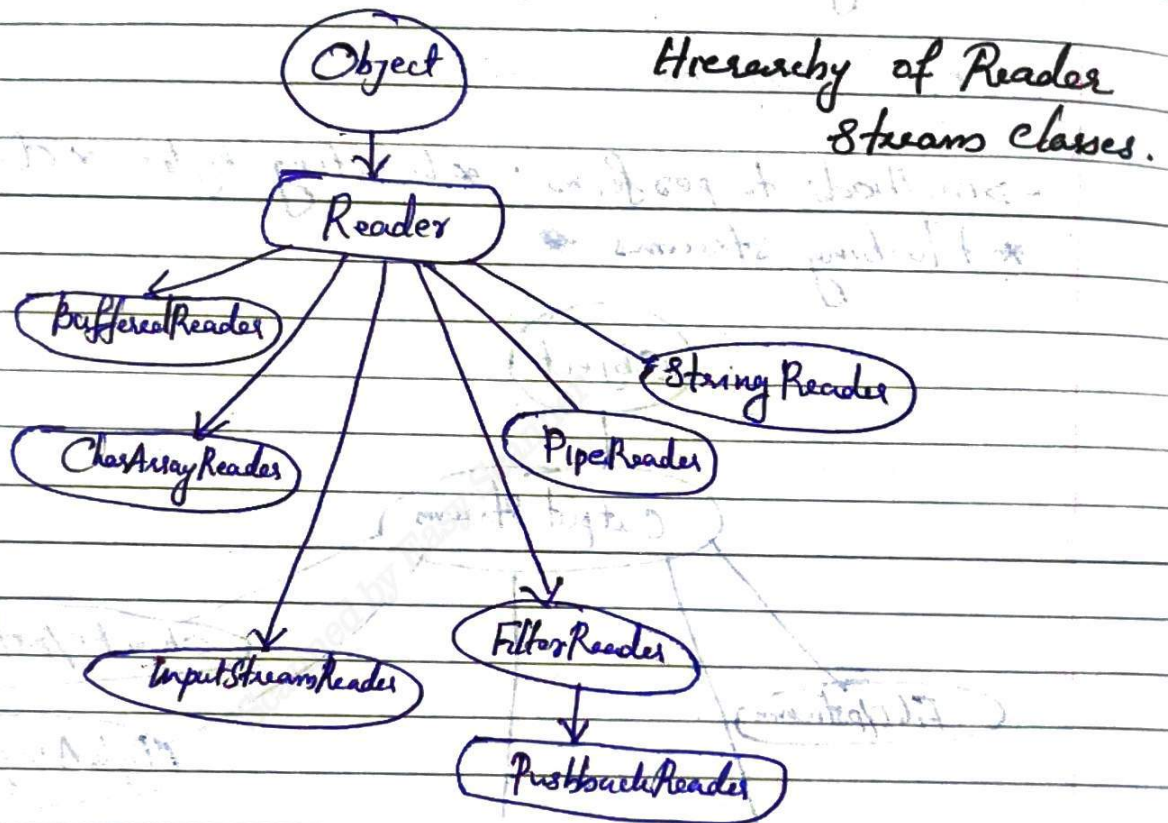
## Output Streams Classes:

→ methods to perform: \* Writing bytes \* Closing streams \* Flushing streams \*

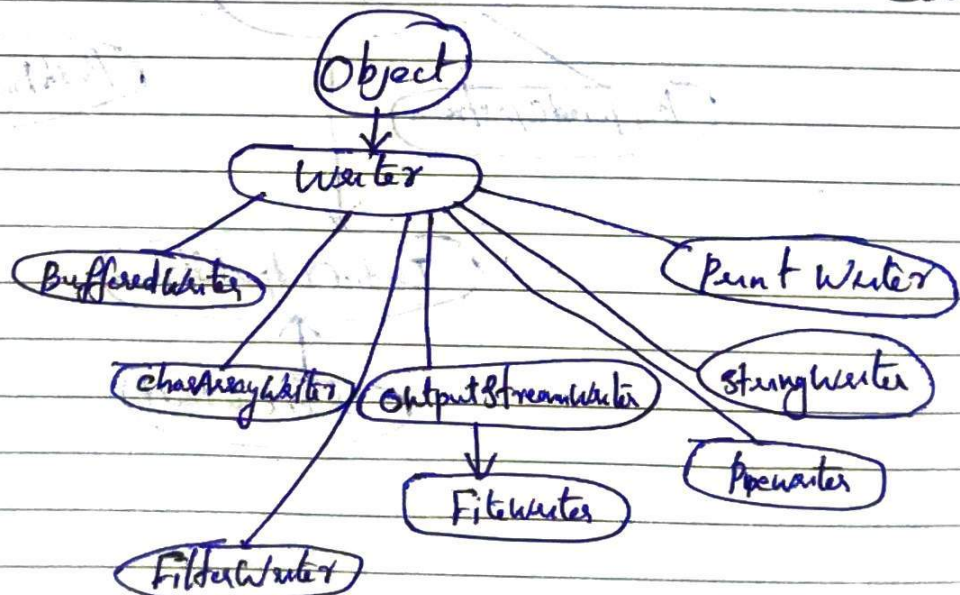


## Character Streams Classes:

- used to read & write 16 bit Unicode character
- ↳ reader streams classes
- ↳ writer streams classes.



## Writer Streams Classes: Hierarchy of writer streams classes:

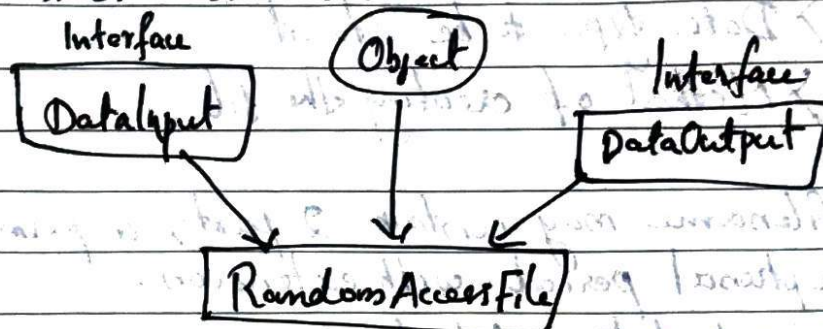




## Specialised fs classes

\* RandomAccessFile

\* StreamTokenizer



## Files

The java.io package includes a class known as the File class that provides support for creating files & directories. The class includes several constructors for instantiating the File objects. Methods for supporting:

- \* Creating a file
- \* Opening a file
- \* Closing a file
- \* Deleting a file
- \* Getting the name of a file
- \* Getting the size of a file
- \* Renaming a file
- \* File is writable / readable.

Each I/O stmt must have an exception handler around it as

— the method must declare that it throws an IOException

try {

    // I/O stmts

} catch (IOException e)

{

    // Msg o/p stmt

}



## Creation of Files

- > Suitable name for the file
- > Purpose (reading, writing)
- > Data type to be stored
- > Method of creating the file.

A filename may contain 2 parts, a primary name & an optional period with extension.

eg: input.data, test.doc

There are 2 ways of initialising the file stream objects. All of the constructors require that we provide the name of the file either directly, or indirectly by giving a file object that has already been assigned filename.

Direct →

```
FileInputStream fis;
try
{
 fis = new FileInputStream("test.dat");
}
catch (IOException e)
```

The indirect approach uses a file obj that has been initialised with the desired filename.

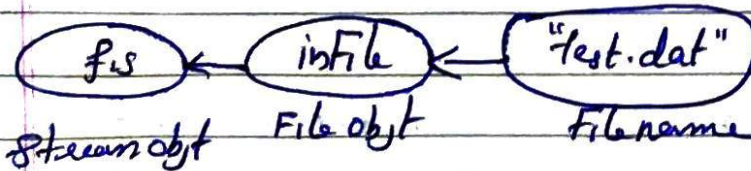
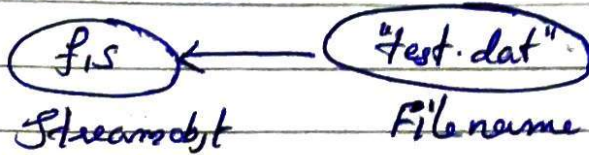
Eg:

```
File inFile;
inFile = new File("test.dat");
FileInputStream fis;
try {
 fis = new FileInputStream(inFile);
}
catch ()
```



The code above includes 5 tasks

\* Select a filename \* Declare a file object



## Reading/Writing Characters:

Eg: // Copying character from one file into another

```

import java.io.*;
class CopyCharacters {
 public static void main(String args[]) {
 // Declare & create i/p & o/p files
 File inFile = new File("input.dat");
 File outFile = new File("output.dat");
 FileReader ins = null;
 FileWriter outs = null;
 ins = new FileReader(inFile);
 outs = new FileWriter(outFile);
 int ch;
 while ((ch = ins.read()) != -1) {
 outs.write(ch);
 }
 ins.close();
 outs.close();
 }
}

```

Page No. \_\_\_\_\_  
Date: \_\_\_\_\_

It creates 2 file objs inFile and outFile & initializes them with "input.dat" and "output.dat" respectively.

The pgs then creates 2 file stream objs ins and outs and initializes them with "null" as follows:

FileReader ins = null; FileWriter outs = null;  
These streams are then connected to the named files using the following

### Reading/Writing Bytes:

```
import java.io.*; //writing bytes to a file.
class WriteBytes
```

```
{
 public static void main(String args[])
 { byte cities[] = {'D', 'E', 'L', 'H', 'I' \n
 'M', 'A', 'D', 'R', 'A', 'S',
 };
```

```
 //create an o/p file stream
```

```
 FileOutputStream outfile = null;
```

```
 outfile = new FileOutputStream("city.txt");
```

```
 //write data into the stream
```

```
 outfile.write(cities);
```

```
 outfile.close();
```

```
 }
}
```

4/4/19



Page No. \_\_\_\_\_  
 Date \_\_\_\_\_

eg: // Reading bytes from a file

```

import java.io.*;
class ReadBytes
{
 public static void main(String args[])
 {
 FileInputStream infile = null;

 // Connect infile stream to the reqd file.
 infile = new FileInputStream(args[0]);

 // Read & Display.
 while (b = infile.read()) != -1)
 {
 System.out.print((char) b);
 }
 infile.close();
 }
}

```

### Reading Console Input:

Java doesn't have a generalised console i/p method that parallels the `scanf()` or other i/p operation. Console input is accomplished by reading from `System.in`. `BufferedReader` supports a buffered i/p streams.

`BufferedReader(Reader inputReader)`

`Reader` is an abstract class.

`BufferedReader br = new BufferedReader(new InputStreamReader(System.in));`

eg: // Reading bytes from a file

```
import java.io.*;
class ReadBytes
{
 public static void main(String args[])
 {
 FileInputStream inFile = null;

 // Connect inFile stream to the reqd file.
 inFile = new FileInputStream(args[0]);

 // Read & Display.
 while ((b = inFile.read()) != -1)
 {
 System.out.print((char) b);
 }
 inFile.close();
 }
}
```

### Reading Console Input:

Java doesn't have a generalised console i/p method that parallels the scanf(). or other i/p operators. Console input is accomplished by reading from System.in. BufferedReader supports a buffered i/p streams.

BufferedReader(Reader inputReader)

Reader is an abstract class.

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));



```

import java.io.*;
class BRRead
{
 public static void main(String args[])
 {
 throws IOException
 {
 char c;
 BufferedReader br = new
 BufferedReader(new InputStreamReader
 (System.in));
 System.out.println("Enter char, q, to quit");
 do
 {
 c = (char) br.read();
 System.out.println(c);
 while(c != 'q');
 }
 }
 }
}

```

### Reading Strings: ReadLine().

```

Eg import java.io.*;
class BRReadlines
{
 public static void main(String args[])
 {
 throws IOException
 {
 BufferedReader br = new BufferedReader(new InputStreamReader
 (System.in));

 String str;
 System.out.println("Enter the lines");
 System.out.println("Enter 'stop' to quit");
 do
 {
 str = br.readLine();
 System.out.println(str);
 } while(!str.equals("stop"));
 }
 }
}

```



## MODULE V

### > Event Handling

→ Delegation Event Model

→ Event classes

→ Sources

→ Listeners

### > Applet Basics & Methods

### > String Classes - Basics.

## Questions

April 2018

1. Explain Event Handling in Java.
2. List any 5 sources & their corresp. event types & listeners.
3. What is applet? Describe the structure of applet.
4. Develop an applet that allows to input velocity in km/hr convert & display in M/sec. Use AWT ctrl's & event handling techniques.
5. Explain the tag used to pass parameters to an applet.
6. Write an applet which displays the character when a key is typed. Use event handling mechanisms.

July 2017

1. Explain the working of the Delegation Event Model
2. Write various event sources & their corresp. listeners.
3. What are the features of an applet.
4. Illustrate the diff parts of an applet tag.
5. Write a java pgm to check whether a given string is palindrome or not without using built in methods.

Feb



## Event Handling Module IV

Events are those generated by the mouse, keyboard & various controls such as a push button. Events are supported by the `java.awt.event` package.

### The Delegation Event Model

— defines standard & consistent mechanisms to generate & process events. Its concept: a source generates an event & sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event & then returns. The advantage is that the appl<sup>n</sup> logic that processes the event & then returns is cleanly separated from the user i/f logic that generate these events. A user i/f elem<sup>t</sup> is able to "delegate" the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification.

— Here notific<sup>n</sup>s are sent only to listeners that want to receive them.

### Events:

An event is an obj<sup>t</sup> that describes a state change in a source. It can be generated as a consequence of a person interacting with the elem<sup>t</sup>s in a GUI. Some of the activities that cause events to be generated are pressing buttons, entering a character via a keyboard, selecting an item in a list & clicking the mouse. Events may also occur that are not directly caused by interactions with a user i/f. For eg: an event may be generated when a timer expires, a counter exceeds a value, a s/w or h/w failure occurs or an op<sup>n</sup> is completed.



## Event Sources

A source is an objt that generates an event. This occurs when the internal state of that objt changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.

The general form:

```
public void addTypeListener(TypeListener el)
```

Here the 'Type' is the name of the event & el is a reference to the event listener. For eg. the method that registers a keyboard event listener is called `addKeyListener()`. The method that registers a mouse motion listener is called `addMouseMotionListener()`. When an event occurs, all registered listeners are notified and receive a copy of the event objt. This is known as multicasting the event.

Some sources may allow only one listener to register.

```
public void addTypeListener(TypeListener el)
throws java.util.TooManyListenersException
```

A source must also provide a method that allows a listener to unregister an interest in a specific type of event.

```
public void removeTypeListener(TypeListener el)
```

Eg: `removeKeyListener()`



## Event Listeners

A listener is an object that is notified when an event occurs. It has 2 major requirements:

→ It must have been registered with one or more sources to receive notifications about specific types of events.

→ It must implement methods to receive & process these notifications.

The methods that receive & process events are defined in a set of *ifcs* found in `java.awt.expt.`. For eg., the `MouseListener` interface defines 2 methods to receive notifications when the mouse is dragged or moved. Any object may receive & process one or both of these events if it provides an *impl* of this interface.

## Event Classes

Classes that represent events are at the core of Java's event handling mechanism.

At the root of the Java event class hierarchy is `EventObject`, which is in `java.util`. It is the superclass for all events.

`EventObject(Object src)`

`src` is the object that generates this event.

`EventObject` contains 2 methods: `getSource()` & `toString()`. The `getSource()` method returns the source of the event.

Its general form is shown here:

`Object getSource()`

`AWTEvent` is a superclass of all AWT events that are handled by the delegation event model.



## Event Classes

An ActionEvent is generated when a button is pressed, a list item is double clicked, or a menu item is selected. The ActionEvent class defines 4 integer constants that can be used to identify any modifiers associated with an action event: ALT-MASK, CTRL-MASK, META-MASK, and SHIFT-MASK.

ActionEvent has these 3 constructors:

ActionEvent(Object src, int type, String cmd)

ActionEvent(Object src, int type, String cmd, int modifiers)

ActionEvent(Object src, int type, String cmd, long when, int modifiers)

The AdjustmentEvent Class:

— AdjustmentEvent is generated by a scrollbar.  
There are 5 types of adjustment events.

BLOCK-DECREMENT - The user clicked inside the scroll bar to decrease its value.

BLOCK-INCREMENT - The user clicked inside the scroll bar to increase its value.

TRACK - The slider was dragged.

UNIT-DECREMENT - The button at the end of the scroll bar was clicked to decrease its value.

UNIT-INCREMENT - The button at the end of the scroll bar was clicked to increase its value.



## Sources of Events:

### Event source

Button - Generates action events when the button is pressed.

Checkbox - when it is selected or deselected.

Choice - when the choice is changed.

List - when an item is double clicked.

MenuItem - when MenuItem is selected.

Scrollbar - when scrollbar is manipulated.

Text components - when the user enters a character.

Window - when a window is activated, closed, deactivated or quit.

### Steps in using Delegation Event Model:

1. Implement the appropriate `yf` in the listener so that it will receive the type of event desired.
2. Implement code to register & unregister the listener as a recipient for the event notifications.

### → Demonstrate Mouse Events:

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

```
/*
```

```
<applet code = "MouseEvents" width=300 ht=100>
```

```
</applet>
```

```
*/
```

```
public class MouseEvents extends Applet
```



Pgm Mouse event Handlers

Key event handlers

Event class

Description

Action Event

— Generated when a button is pressed, a list item is double-clicked or a menu item is selected.

Adjustment Event

— Generated when a scroll bar is manipulated.

Component Event

— Generated when component is hidden, moved, resized or becomes visible.

Container Event

— Generated when component is added or removed from a container.

Focus Event

— Generated when a component gains or loses keyboard focus.

Input Event

— Abstract super class for all component input event classes.

Item Event

— Generated when a check box or list item is clicked.

Key Event

— Generated from when a key is received from keyboard.

Mouse Event

— Generated when the mouse is dragged, moved, clicked, pressed, or released.

Mouse Wheel Event

— Generated when the mouse wheel is moved.

Text Event

— Generated when the value of a text area or text field is changed.

Window Event

— Generated when a window is activated, closed, deactivated or quit.



## String Classes (Module IV)

→ String Class  
→ String Buffer Class } defined in java.lang.

### The String Constructors:

- ~~suppose~~ string class supports several constructors.

To create an empty string

String s = new String(); will create an instance of String with no characters in it.

To create a String initialised by an array of characters use the constructor

String(char chars[])

Eg: char chars[] = {'a', 'b', 'c'};  
String s = new String(chars);

// Construct one string from another.

class MakeString

{ public static void main(String args[])

{ char c[] = {'J', 'a', 'v', 'a'};

String s<sub>1</sub> = new String(c);

String s<sub>2</sub> = new String(s<sub>1</sub>);

System.out.println(s<sub>1</sub>);

System.out.println(s<sub>2</sub>);

}

O/p Java

Java

// Construct string from subset of char array.

```
class SubStringCons
```

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

```
{ byte ascii[] = {65, 66, 67, 68, 69, 70};
```

```
String s1 = new String(ascii);
```

```
System.out.println(s1);
```

```
String s2 = new String(ascii, 2, 3);
```

```
System.out.println(s2);
```

```
}
```

```
}
```

O/P    ABCDEF

      CDE

## String Length

— no. of characters

— length()

```
char chars[] = {'a', 'b', 'c'};
```

```
String s = new String(chars);
```

```
System.out.println(s.length());
```

## String Literals

For each string literal in your pgm, Java automatically constructs a String object. Thus you use a string object.

Eg: 

```
char chars[] = {'a', 'b', 'c'};
```

```
String s1 = new String(chars);
```

```
String s2 = "abc"; // use String Literal
```



## String Concatenation

Java doesn't allow operators to be applied to String objects. The one exception to this rule is the + operator, which concatenates 2 strings, producing a String object as the result.

Eg: `String age = "9";`

`String s = "He is" + age + "years old";`

`System.out.println(s);`

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

Concatenation with Other Data types:

`int age = 9;`

`String s = "He is" + age + "years old";`

`System.out.println(s);`

O/p He is 9 years old.

Eg: `String s = "four" + 2 + 2;`

`System.out.println(s);`

O/p four

four:22

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

O/p four:4

## String Conversion & toString()

When Java converts data into its string representation during concatenation, it does so by calling one of the overloaded versions of the string conversion method `valueOf()` defined by `String`. `valueOf()` is overloaded for all the simple types & for type `Object`. For simple types, `valueOf()` returns a string that contains human-readable equivalent of the value with which it is called.



For objects, `valueOf()` calls the `toString()` method on the object.

The `toString()` method has this general form:

`String toString()`

Eg: demonstrates this by overriding `toString()` for the Box class:

// Override `toString()` for Box class

Class Box

```
{ double wt;
 double ht;
 double dt;
```

```
Box(double w, double h, double d)
```

```
{ wt = w;
 ht = h;
 dt = d;
```

```
public String toString()
```

```
{ return "Dimensions are " + wt + " by " +
 dt + " by " + ht + ".";
```

```
}
```

class ToStringDemo

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

```
{ Box b = new Box(10, 12, 14);
```

```
String s = "Box b: " + b; // concatenate Box
```

```
obj
```

```
System.out.println(b); // convert Box to string
```

```
System.out.println(s);
```

```
}
```

O/P Dimensions are 10.0 by 14.0 by 12.0

Box b: Dimensions are 10.0 by 14.0 by 12.0



Box's `toString()` method is automatically invoked when a Box object is used in a concatenation expr. or in a call to `println()`.

### Character Extractors:

characters can be extracted from a String object.

`charAt()` - To extract a single character from a string

`char charAt(int where)`

where is the index of the character that to obtain  
`char ch;`

`ch = "abc".charAt(1);`

assigns the value to "b" to ch.

`getChars()` - To extract more than one character at a time.

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

Eg: `class getCharsDemo`

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

`{ String s = "This is a demo of the getChars`

`([1] print) some bio state add method";`

`int start = 10; // = 2 print`

`int end = 14; // = 2 print`

`char buf[] = new char[end - start];`

`s.getChars(start, end, buf, 0);`

`System.out.println(buf);`

`{ [1] print`

`"} // = 2 print`

`O/P: demo`

`{ [1] print`

`{ [1] print`



getBytes(): stores the characters in an array of bytes  
byte[] getBytes()

toArray(): to convert all the characters in a String object into a character array. It returns an array of characters for the entire string.  
char[] toArray()

### String Comparison

To compare 2 strings for equality, use equals().  
boolean equals(Object str)

Here str is the String objt being compared with the invoking String object. It returns true if the strings contain the same characters in the same order, & false otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call equalsIgnoreCase().

boolean equalsIgnoreCase(String str)

// Demonstrate equals() & equalsIgnoreCase()  
class equalsDemo

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

```
{ String s1 = "Hello";
```

```
String s2 = "Hello";
```

```
String s3 = "Gbye";
```

```
String s4 = "HELLO";
```

```
System.out.println(s1 + " equals " + s2 + " → " +
s1.equals(s2));
```

```
System.out.println(s1 + " equals " + s3 + " → " +
s1.equals(s3));
```

```
System.out.println(s1 + " equals " + s4 + " → " +
s1.equals(s4));
```



Page No. \_\_\_\_\_  
Date: \_\_\_\_\_

```
System.out.println(s1 + "equals/ignoreCase" + s4 + " → " +
 s1.equalsIgnorecase(s4));
```

```
}
```

Hello equals Hello → true

Hello equals Goodbye → false

Hello equals HELLO → false

Hello equalsIgnorecase HELLO → true

equalsIgnoreCase) versus ==

→ equals() method compares the characters inside a String object. The == operator compares 2 objt references to see whether they refer to the same instance.

// equals() vs ==

```
class EqualsNotEqualTo
```

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

```
{ String s1 = "Hello";
```

```
String s2 = new String(s1);
```

```
System.out.println(s1 + "equals" + s2 + " → " +
 s1.equals(s2));
```

```
System.out.println(s1 + "==" + s2 + " → " +
 (s1 == s2));
```

```
}
```

O/p: Hello equals Hello → true

Hello == Hello → false

Contents of 2 String objects are identical, but they are distinct objts.



## String Buffer:

startsWith() & endsWith()

The `startsWith()` method determines whether a given string begins with a specified string.

`endsWith()` determines whether the string in question ends with a specified string.

`boolean startsWith(String str)`

`boolean endsWith(String str)`

Here `str` is the string being tested. If the string matches, `true` is returned. Else `false` is returned.

`"Foobar".endsWith("bar")`

and

`"Foobar".startsWith("Foo")` are `true`.

`"Foobar".startsWith("bar", 3)` returns `true`.

## CompareTo()

— less than, =, > than the next.

Eg:

```
class SortString
```

```
{ static String arr[] = {"Now", "is", "the", "time",
"for", "all", "good", "men", "to", "come", "to",
"aid", "of", "these", "country"};
```

```
public static void main(String args[])
```

```
{ for (int j=0; j<arr.length; j++)
```

```
{ if (arr[i].compareTo(arr[j])<0)
```

```
{
```

```
 String t = arr[j];
```

```
 arr[j] = arr[i];
```

```
 arr[i] = t;
```

```
}
```

```
}
 System.out.println(arr[j]);
```

```
}
}
}
```



O/p

Now aid all come country for good is men of the the their time to to

Searching Strings: to search for a specified character or substring.

indexOf() searches for the first occurrence of a character or substring.

lastIndexOf() searches the last occurrence of a character or substring.

int indexOf(int ch)

int lastIndexOf(int ch)

Here ch is the character being sought.

int indexOf(String str)

int lastIndexOf(String str) str specifies the substring

// Demonstrate indexOf() & lastIndexOf()

class indexofDemo {

public static void main(String args[]) {

String s = "Now is the time for all good men" +  
"to come the aid of their country";

System.out.println(s);

System.out.println("indexOf('t') = " + s.indexOf('t')  
+ s.lastIndexOf('t'));

}

Concat() - Concatenate 2 strings using concat()

String concat(String str)

eg: String s1 = "one";

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

O/p: onetwo puts the string onetwo into s2.

ie; String s1 = "one";

1, s2 = s1 + "two";



replace(): replaces all occurrences of one character in the invoking string with another character.

String replace(char original, char replacement)

eg:

String s = "Hello".replace('l', 'w');  
puts the string "Hewwo" into s.

trim(): returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

String trim()

eg: String s = "Hello World".trim();

This puts "Hello" eg: String s2 = "Hello World".trim();  
puts the string Hello World into s.



## APPLET BASICS & METHODS

### Applet Programming:

Applets are small appl's that are stored on internet server, transported over the internet, automatically installed & run as part of a web document. An applet like any appl' pgm, it can perform arithmetic op's, display graphics, play sounds, accept user input, create animation & play interactive games.

#### Local & Remote Applets:

- embed applets into web pages in 2 ways.
- \* We can write our own applets & embed them into webpages. (Local)
- \* We can download an applet from a remote computer s/r & then embed into a webpage. (Remote).

### Applets & Applications:

#### → Applets -

- \* do not use the main() method for initiating the ex<sup>n</sup> of the code.
- \* automatically call certain methods of Applet class to start & execute the applet code.
- \* cannot be run independently. They are run from inside a web page using a special feature known as HTML tag.
- \* cannot read from or write to the files in the local computer.
- \* cannot communicate with other servers on the n/w.

- \* cannot run any pgm from the local computer.
- \* Applets are restricted from using libraries from other languages like C, C++.

The steps involved in developing & testing in applet are:

1. Building an applet code (.java file)
2. Creating an executable applet (.class file)
3. Designing a web page using HTML tags
4. Preparing <Applet> tag.
5. Incorporating <APPLET> tag into the web page.
6. Creating the HTML file
7. Testing the applet code.

### Building Applet Code:

Applet Code uses the services of 2 classes,  
 - Applet & Graphics from java class library.  
 The Applet class which is contained in the java.applet package provides life & behaviour to the applet through its methods such as init(), start(), paint().  
 When an applet is loaded, Java automatically calls a series of Applet class methods for starting running & stopping the applet code.

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



Page No. \_\_\_\_\_  
Date. \_\_\_\_\_

This applet begins with 2 import stmts. The first imports the Abstract Window Toolkit (AWT) classes. Applet interact with the user through the AWT, not through console based I/O classes.

The second import stmt imports the applet pkg. which contains the class Applet. Every applet that you create must be a subclass of Applet.

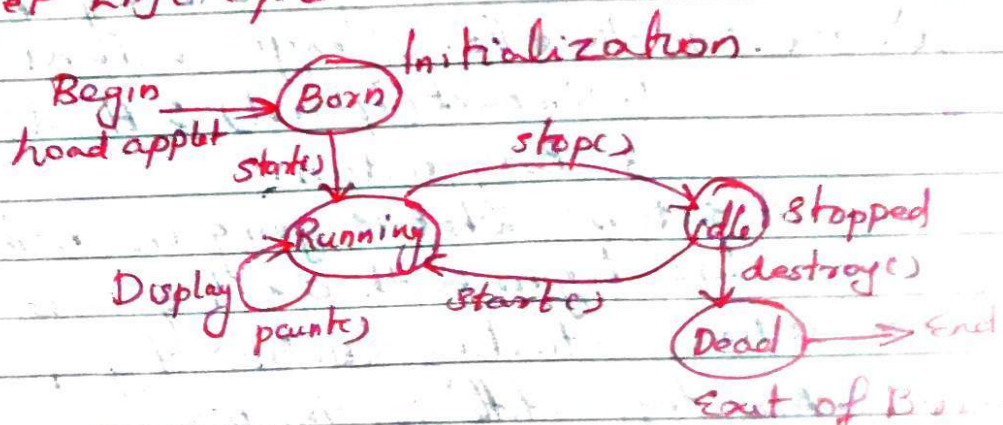
The next line in the pgm declares the class SimpleApplet. This class must be declared as public, b/c it will be accessed by code outside the pgm. Inside SimpleApplet, paint() is declared. This method is defined by the AWT & must be overridden by the Applet. paint() is called each time that the applet must redisplay its o/p.

The paint() method has one parameter of type Graphics. This parameter contains the graphics context, which describes the graphics envt. in which the applet is running.

Inside paint() is a call to drawString(), which is a member of the Graphics class. This method o/p's a string beginning at the specified x, y location.

General form: void drawString(String msg, int x, int y)

### Applet Life Cycle





## Creating an Executable Applet

— .class file of the applet, which is obtained by compiling the source code of the applet. Compiling an applet is exactly the same as compiling an applet.

Steps: 1. Move to the directory containing the source code & type the follo. Cmd.

`javac HelloJava.java`

2. The compiled o/p file called `HelloJava.class` is placed in the same directory as the source.

3. If any error msg is recieved, then we must check for errors, correct them & compile the applet again.

## Designing a Web Page

A webpage is basically made up of text & HTML tags that can be interpreted by a web browser or an applet viewer. Web pages are stored using a file extension .html such as `MyApplet.html`.

Webpages include both text that we want to display & HTML tags to web browsers.

A webpage is marked by an opening HTML tag `<HTML>` and closing HTML tag `</HTML>` & is divided into the follo. 3 major sections.

1. Comment (optional)
2. Head (optional)
3. Body section.

Comment sections:— what is going on in the web page. Comment line begins with a `<!--` & ends with a `-->`.

Head section:— starting `<HEAD>` tag & a closing `</HEAD>` tag. — title for the page.



The text enclosed in the tags <TITLE> and <ITITLE> will appear in the title bar of the web browser when it displays the page.

Body Section: contains the entire infn about the web page & its behaviour.

<BODY>

<CENTER>

<H1> Welcome </H1>

</CENTER>

<BR>

<APPLET...>

</APPLET>

</BODY>

The body shown above display  
welcome

H<sub>1</sub> tag causes the text to be of the largest size.

<H<sub>26</sub>

## Preparing Applet Tag

The <APPLET> tag supplies the name of the applet to be loaded & tells the browser how much space the applet requires. The <APPLET> tag given below specifies the min. sqmnts to place the Hello Java applet on the webpage.

<APPLET>

CODE = helloJava.class

WIDTH = 400

HEIGHT = 200

</APPLET>

HelloJava applet

import java.awt.\*;

import java.applet.\*;

public class HelloJava extends Applet

{  
public void paint(Graphics g)  
{ g.drawString("HelloJava",

10, 100);

}

}



Page No: \_\_\_\_\_  
Date: \_\_\_\_\_

This HTML code tells the browser to load the compiled java applet `helloJava.class`, which is in the same directory as this HTML file. And also specifies the display area for the applet o/p as 400 pixels width & 200 pixels height.

### Incorporating APPLET tag to HTML File.

Insert the `<APPLET>` tag in the page at the place where the o/p of the applet must appear. Eg: `<HTML>`

`<!-- This page includes a welcome title -->`

`<HEAD>`

`<TITLE>`

`welcome to Java Applets`

`</TITLE>`

`</HEAD>`

`<BODY>`

`<CENTER>`

`<H1> Welcome to Applets </H1>`

`</CENTER>`

`<BR>`

`<CENTER>`

`<APPLET`

`CODE = helloJava.class`

`WIDTH = 400`

`HEIGHT = 200>`

`</APPLET>`

`</CENTER>`

`</BODY>`

`</HTML>`

### Creating HTML file

we must name this file as `helloJava.html` & save it in the same directory as the compiled applet.



## Running the Applet

To run an applet, we require one of the follo. tools:

1. Java-enabled web browser
2. Java applet viewer.

If we use Java-enabled web browser, we will be able to see the entire web page containing the applet.

If we use the applet viewer tool, we will only see the applet o/p. It ignores ~~the~~ all of the HTML tags except the part pertaining to the running of the applet.

Applet Viewer: HelloJava.class
Applet
Hello Java   appletloader started

## Passing Parameters to APPLETs

→ user defined parameters to an applet using `<PARAM-->` tags. Each `<PARAM-->` tag has a name attribute such as color, & a value attribute such as red. Inside the applet code, the applet can refer to that parameter by name to find its value.

eg: `<APPLET-->`

`<PARAM = color value = "red">`

`</APPLET>`



To set up & handle parameters, we need to do 2 things

1. Include appropriate `<PARAM...>` tags in the HTML doc.
2. Provide code in the applet to parse these parameters.

We can define the `init()` method in the applet to get hold of the parameters defined in the `<PARAM>` tags. This is done using the `getParameter()` method which takes one string arg. representing the name of the parameter & returns a string containing the value of that parameter.

```

HelloJavaParam.class
import java.awt.*;
import java.applet.*;
public class HelloJavaParam extends Applet
{
 String str;
 public void init()
 {
 str = getParameter("string"); // Receiving Parameter Value
 if (str == null)
 str = "Java";
 str = "Hello" + str; // using the value
 }
 public void paint(Graphics g)
 {
 g.drawString(str, 10, 100);
 }
}

```

HTML File

`<HTML>`

`<!Parameterised HTML file>`

`<HEAD>`

`<TITLE> Welcome to Java Applets </TITLE>`

`</HEAD>`



run using applet viewer

```
appletviewer HelloJavaParam.html
```

```
AppletViewer: HelloJavaParam.class - D x
```

```
Applet
```

```
Hello Applet!
```

```
appletloader: started
```

Remove the `<PARAM>` tag from the HTML file & then run the applet

result  
HelloJava

### The HTML APPLET Tag

The applet tag is used to start an applet both an HTML document & from an applet viewer. An applet viewer will execute each APPLET tag as it finds in a separate window, while web browsers will allow many applets on a single page. The syntax for the standard APPLET tag:

`<APPLET`

`[CODEBASE = Codebase URL]`

`CODE = Applet file`

[ALT = alternateText]

[NAME = appletInstanceName]

WIDTH = pixels HEIGHT = pixels

[ALIGN = alignment]

[VSPACE = pixels] [HSPACE = pixels]

[<PARAM NAME = AttributeName

VALUE = AttributeValue]

[<PARAM NAME = AttributeName2

VALUE = AttributeValue]

</APPLET>

CODEBASE :- an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable file (specified by CODE tag)

CODE :- required attribute that gives the name of the file containing applet's compiled class file. This file is relative to the Code base URL of the applet.

ALT :- specify a shortmsg that should be displayed if the browser understands the APPLET tag but can't currently run java applets.

NAME :- to specify a name for the applet instance.

WIDTH & HEIGHT :- size of the applet display area.

ALIGN :- specifies the alignment of the applet.

eg: LEFT, RIGHT, TOP, BOTTOM, MIDDLE etc.

PARAM NAME & VALUE :- applet specific args in an HTML page. Applets access their attributes with the getParameter() method.



## Displaying Numerical Value:

```
import java.awt.*;
import java.applet.*;
public class NumValues extends Applet
{
 public void paint(Graphics g)
 {
 int value1 = 10;
 int value2 = 20;
 int sum = value1 + value2;
 String s = "sum: " + String.valueOf(sum);
 g.drawString(s, 10, 100);
 }
}
```

<html>

<applet

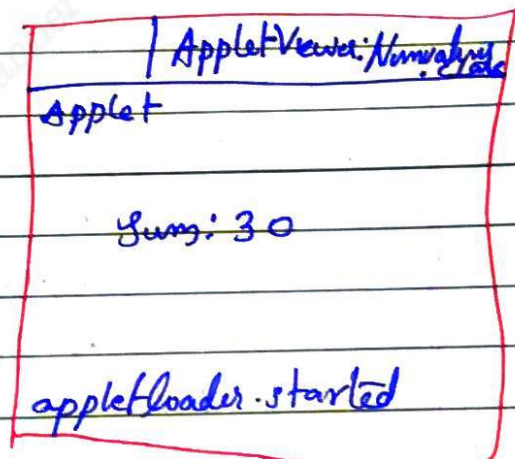
code = NumValues.class

width = 300

height = 300 >

</applet>

</html>



## Getting Input from the User

Applets work in a graphical envt. Applets treat i/p as text strings. First create an area of the screen in which user can type and edit up items. We can do this by using the TextField class of the applet package.

Next to retrieve the items from the fields for the display of calculations, if any.

```

import java.awt.*;
import java.applet.*;
public class UserIn extends Applet
{
 TextField text1, text2;
 public void init()
 {

```

```

 text1 = new TextField(8);
 text2 = new TextField(8);
 add(text1);
 add(text2);
 text1.setText("0");
 text2.setText("0");
 }

```

```

 public void paint(Graphics g)
 {

```

```

 int x=0, y=0, z=0;

```

```

 String s1, s2, s;

```

```

 g.drawString("Input a no in each box", 10, 50);

```

```

 try
 {

```

```

 s1 = text1.getText();

```

```

 x = Integer.parseInt(s1);

```

```

 s2 = text2.getText();

```

```

 y = Integer.parseInt(s2);

```

```

 }

```

```

 catch (Exception e) { }

```

```

 z = x + y;

```

```

 s = String.valueOf(z);

```

```

 g.drawString("The Sum is", 10, 75);

```

```

 g.drawString(s, 100, 75);
 }

```



```
public Boolean action(Event event, Object object)
```

```
{ repaint();
 return true;
}
```

```
}
```

Run the applet UserIn using the follo. steps.

1. Type & save the pgm (.java file)
2. Compile the applet (.class file)
3. Write HTML document (.html file)

```
<html>
```

```
<applet
```

```
 code=UserIn.class
```

```
 width=300
```

```
 height=200>
```

```
</applet>
```

```
</html>
```

↳ Use the appletviewer to display the results.

Applet Viewer: UserIn.class

Applet

123

215

Input a no. in each  
base

The Sum is 338

23/5/18

## MODULE VI

### > Introduction to AWT

- \* Working with Frames, graphics, color, font.
- \* AWT control fundamentals
- \* Swing Overview.

### > JAVA Database Connectivity.

- \* JDBC Overview
- \* Creating & executing queries
- \* dynamic queries.



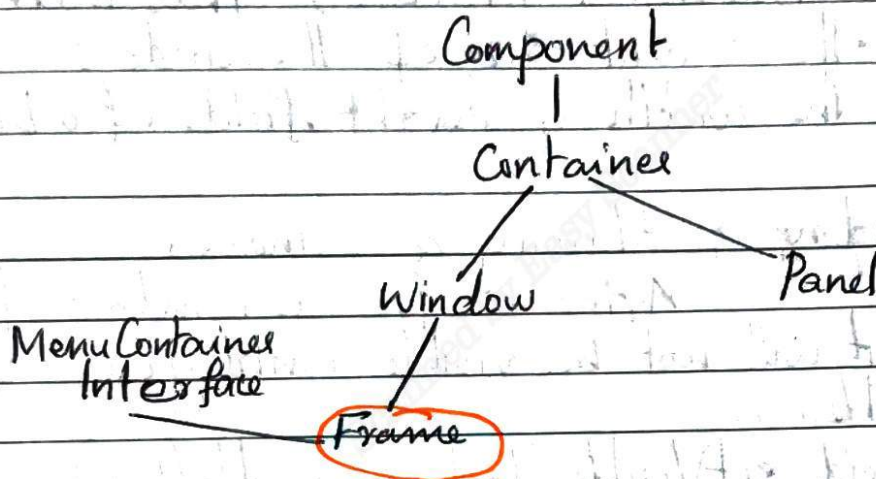
## AWT - Abstract Window Toolkit (AWT)

AWT contains numerous classes & methods that allow you to create & manage windows. The AWT classes are contained in the java.awt package.

Ref: Table 21.1

### \* Working with Frames:

The 2 most common windows are those derived from Panel, which is used by applets, & those derived from Frame, which creates a standard window.



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

If you create a Frame objt from within an applet, it will contain a warning msg, such as "Java Applet Window", to the user that an applet window has been created. This msg warns users that the window they see was started by an applet & not by s/w running on their computers.

Frame() & Frame(string title)

Frame(): creates a std window that doesn't contain a title.



The 2<sup>nd</sup> form creates a window with title specified by title - cannot specify the dimensions of the window.

Setting the Window's Dimensions:

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

The `getSize()` method is used to obtain the current size of a window.

`Dimension getSize()` This method returns the current size of the window contained within the width & height fields of a `Dimension` object.

Hiding & Showing a Window

After a frame window has been created, it will not be visible until you call `setVisible()`.

```
void setVisible(boolean visibleFlag)
```

Setting a Window's Title

```
void setTitle(String newTitle)
```

`newTitle` is the new title for the window.

Closing a Frame Window

When using a frame window, you must remove that window from the screen when it is closed by calling `setVisible(false)`. To intercept a window close event, you must implement the `WindowClosing()` method of the `WindowListener` interface. Inside `windowClosing()`, you must remove the window from the screen.



## Creating a Frame Window in an Applet;

Creating a new frame window from within an applet is actually quite easy. First create a subclass of `Frame`. Next override any of the standard window methods, such as `init()`, `start()`, `stop()` & `paint()`. Finally, implement the `windowClosing()` method of the `WindowListener` interface, calling `setVisible(false)` when the window is closed.

Once you defined a `Frame` subclass, create an object of that class. When created, the window is given a default height & width. It can be reset by calling `setSize` method.

see Pgm. page 695

## Handling Events in a Frame Window:

Since `Frame` is a subclass of `Component`, it inherits all the capabilities defined by `Component`.

This means that we can manage a `Frame` window that you create just like applet's main window.

e.g. For eg: we can override `paint()` to display o/p call `repaint()` when you need to restore the window, & override all event-handlers.

Ref Pgm pg: 697

## Working with Graphics:

All o/p to a window takes place through a graphics context. A graphics context is encapsulated by the `Graphics` class & is obtained in 2 ways.

→ It is passed to an applet when one of its various methods, such as `paint()` or `update()` is called.



## Creating a Frame Window in an Applet:

Creating a new frame window from within an applet is actually quite easy. First create a subclass of `Frame`. Next override any of the standard window methods, such as `init()`, `start()`, `stop()` & `paint()`. Finally, implement the `windowClosing()` method of the `WindowListener` interface, calling `setVisible(false)` when the window is closed.

Once ~~you~~ defined a `Frame` subclass, create an object of that class. When created, the window is given a default height & width. It can be reset by calling `setSize` method.

see Pgm. page 695

## Handling Events in a Frame Window:

Since `Frame` is a subclass of `Component`, it inherits all the capabilities defined by `Component`. This means that we can manage a frame window that you create just like applet's main window.  
e.g. For eg: we can override `paint()` to display o/p call `repaint()` when you need to restore the window, & override all event-handlers.

Ref Pgm pg: 697

## Working with Graphics:

All o/p to a window takes place through a graphics context. A graphics context is encapsulated by the `Graphics` class & is obtained in 2 ways.

→ It is passed to an applet when one of its various methods, such as `paint()` or `update()` is called.



→ It is returned by the `getGraphics()` method of `Component`.

The `Graphics` class defines a no. of drawing fns. Each shape can be drawn edge-only or filled. Objects are drawn & filled in the currently selected graphics color, which is black by default.

**Drawing Lines:** Lines are drawn by means of the `drawLine()` method.

`void drawLine(int startX, int startY, int endX, int endY)`

`drawLine()` displays a line in the current drawing color that begins at `startX, startY` and ends at `endX, endY`.

// Draw lines

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

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

```
/* <applet code="Lines" width=300 height=200>
```

```
</applet>
```

```
*/
```

```
public class Lines extends Applet
```

```
{ public void paint(Graphics g)
```

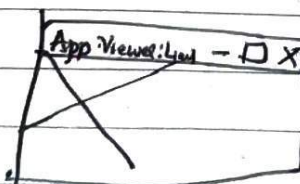
```
{ g.drawLine(0, 0, 100, 100);
```

```
 g.drawLine(0, 100, 100, 0);
```

```
 }
```

```
}
```

Sample O/p





## Drawing Rectangles:

The `drawRect()` and `fillRect()` methods display an outlined & filled rectangle respectively.

`void drawRect(int top, int left, int width, int height)`

`void fillRect(int top, int left, int width, int height)`

The upper-left corner of the rectangle is at `top, left`. The dimensions of the rectangle are specified by `width` & `height`.

To draw a rounded rectangle, use `drawRoundRect()`, `drawRoundRect()` or `fillRoundRect()`

`void drawRoundRect(int top, int left, int width, int height, int xDiam, int yDiam)`

```
// Draw rectangles
import java.awt.*;
import java.applet.*;
/* <applet code = "Rectangles"
 width=300 height=200>
```

```
</applet> */
```

```
public class Rectangles extends Applet
```

```
{ public void paint(Graphics g)
```

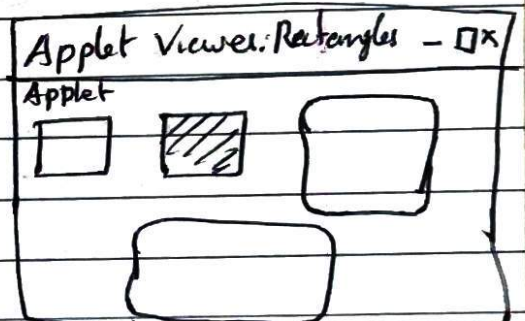
```
{ g.drawRect(10, 10, 60, 50);
```

```
 g.fillRect(100, 10, 60, 50);
```

```
 g.drawRoundRect(140, 10, 60, 50, 15, 15);
```

```
 g.fillRoundRect(70, 90, 140, 100, 30, 40);
```

```
}
}
```





## Drawing Arcs

Arcs can be drawn with `drawArc()` & `fillArc()`, shown here:

```
void drawArc(int top, int left, int width, int height, int
 startAngle, int sweepAngle)
```

```
void fillArc(int top, int left, int width, int height, int
 startAngle, int sweepAngle)
```

The arc is bounded by the rectangle whose upper-left corner is specified by `top`, `left` & whose width & height are specified by `width` & `height`. The arc is drawn from `startAngle` through the angular distance specified by `sweepAngle`. Angles are specified in degrees.

// Draw Arcs

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

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

```
/* <applet code = "Arcs" width=300 height=200>
</applet> */
```

```
public class Arcs extends Applet
```

```
{ public void paint(Graphics g)
```

```
{ g.drawArc(10, 40, 70, 70, 0, 75);
```

```
 g.fillArc(100, 40, 70, 70, 0, 75);
```

```
}
```

## Drawing Ellipses & Circles:

To draw an ellipse, use `drawOval()`. To fill an ellipse, use `fillOval()`.

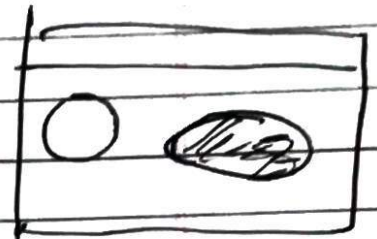
```
void drawOval(int top, int left, int width, int height)
```

```
void fillOval(int top, int left, int width, int height)
```



The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by top, left and whose width & height are specified by width & height. To draw circle, specify a square as the bounding rectangle.

```
// Draw Ellipses
import java.awt.*;
import java.applet.*;
/* <applet code = "Ellipses" width = 300 height = 200>
</applet> */
public class Ellipses extends Applet
{
 public void paint(Graphics g)
 {
 g.drawOval(10, 10, 50, 50);
 g.fillOval(100, 10, 75, 50);
 }
}
```



## Drawing Polygons:

~~drawPolygon()~~ drawPolygon(x) & fillPolygon(x)  
 void drawPolygon(int x[], int y[], int numPoints)  
 void fillPolygon(int x[], int y[], int numPoints)  
 The no. of points defined by x & y is specified by numPoints.

```
// Draw Polygon
import java.awt.*;
import java.applet.*;
/* <applet code = "Hourglass"
width = 230 height = 210>
</applet> */
public class Hourglass extends Applet
{
 int xpoints[] = {30, 200, 30, 200, 30};
 int ypoints[] = {30, 30, 200, 200, 30};
 int num = 5;
 public void paint(Graphics g)
 {
 g.drawPolygon(xpoints, ypoints, num);
 }
}
```



## JDBC

JDBC is Java Database Connectivity.

It's an advancement for Open Db Connectivity. It's an API specification developed in order to move data from front end to backend. This API consists of classes & i/f's written in java. It basically acts as an i/f or channel b/w java pgm & db. It establishes a link b/w the two so that a pgmr could send data from java code & store it in the db for future use.

Steps for connectivity b/w Java & db.

1. Loading the Driver.

or  
- load the driver & register it in the programs. Registration can be done in the pgms in one of 2 ways.

→ `Class.forName()` :- Load the driver's class file into memory at the runtime. No need of using new or creation of object.

eg: `Class.forName("Oracle.jdbc.driver.OracleDriver");`

→ `DriverMngr.registerDriver()` :- `DriverManager` is a Java inbuilt class with static member register.

- call the constructor of the driver class at compile time.

eg: `DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());`

2. Create Connections

After loading driver establish connection.



user - username from which your sql cmd prompt can be accessed.

password - password from which your sql cmd prompt can be accessed.

con :- is a reference to Connection interface.

url :- Uniform Resource Locator

String url = "jdbc:oracle:thin:@localhost:1521:xe"

Where oracle is the db used, thin the driver used, @ local host is the IP address where db is stored, 1521 is the port no. & xe is the service provider. All 4 parameters above are of string type & are to be declared by programmer before calling the fn.

### 3 Create Statement:

Once connection is established you can interact with the db. The JDBC stmt, Callable stmt & PreparedStatement i/f's define the methods that enable you to send SQL cmds & retrieve data from your database.

statement st = con.createStatement();

### 4. Execute the Query:

Queries can be of 2 types:-

- Query for updating/inserting table in a db.
- Query for retrieving data.

executeQuery() method of Statement interface is used to execute queries of retrieving values from the db. This method returns the object of ResultSet that can be used to get all the records of a table. The executeUpdate(sql query) method of Statement interface is used to execute queries of updating/inserting.



```
int m = st.executeUpdate(sql);
if (m == 1)
 System.out.println("inserted successfully : " + sql);
else
 System.out.println("insertion failed");
```

### 5. Close Connections.

Finally data have been sent to the specified location. By closing connection, objects of Statement & ResultSet will be closed automatically. The `close()` method of Connection interface is used to close the connection.

```
con.close();
```

*2/11/21*

## CSS

**cascading Style Sheets (CSS)** is a style sheet language used for describing the presentation of a document written in a markup language such as HTML. CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript.

CSS is designed to enable the separation of presentation and content, including layout, colors, and fonts. This separation can improve content accessibility, provide more flexibility and control in the specification of presentation characteristics, enable multiple web pages to share formatting by specifying the relevant CSS in a separate .css file which reduces complexity and repetition in the structural content as well as enabling the .css file to be cached to improve the page load speed between the pages that share the file and its formatting.

Separation of formatting and content also makes it feasible to present the same markup page in different styles for different rendering methods, such as on-screen, in print, by voice (via speech-based browser or screen reader), and on Braille-based tactile devices. CSS also has rules for alternate formatting if the content is accessed on a mobile device.

The name *cascading* comes from the specified priority scheme to determine which style rule applies if more than one rule matches a particular element. This cascading priority scheme is predictable.

The CSS specifications are maintained by the World Wide Web Consortium (W3C). Internet media type (MIME type) `text/css` is registered for use with CSS by RFC 2318 (March 1998). The W3C operates a free CSS validation service for CSS documents.

CSS has a simple syntax and uses a number of English keywords to specify the names of various style properties.

A style sheet consists of a list of *rules*. Each rule or rule-set consists of one or more *selectors*, and a *declaration block*.

### Selector

In CSS, *selectors* declare which part of the markup a style applies to by matching tags and attributes in the markup itself.

Selectors may apply to the following:

- all elements of a specific type, e.g. the second-level headers h2
- elements specified by attribute, in particular:
  - *id*: an identifier unique within the document, identified with a hash prefix e.g. #id
  - *class*: an identifier that can annotate multiple elements in a document, identified with a period prefix e.g. .classname
- elements depending on how they are placed relative to others in the document tree.



Classes and IDs are case-sensitive, start with letters, and can include alphanumeric characters, hyphens, and underscores. A class may apply to any number of instances of any elements. An ID may only be applied to a single element.

*Pseudo-classes* are used in CSS selectors to permit formatting based on information that is not contained in the document tree. One example of a widely used pseudo-class is `:hover`, which identifies content only when the user "points to" the visible element, usually by holding the mouse cursor over it. It is appended to a selector as in `a:hover` or `#elementid:hover`. A pseudo-class classifies document elements, such as `:link` or `:visited`, whereas a *pseudo-element* makes a selection that may consist of partial elements, such as `::first-line` or `::first-letter`.

Selectors may be combined in many ways to achieve great specificity and flexibility. Multiple selectors may be joined in a spaced list to specify elements by location, element type, id, class, or any combination thereof. The order of the selectors is important. For example, `div .myClass {color: red;}` applies to all elements of class `myClass` that are inside `div` elements, whereas `.myClass div {color: red;}` applies to all `div` elements that are inside elements of class `myClass`. This is not to be confused with concatenated identifiers such as `div.myClass {color: red;}` which applies to `div` elements of class `myClass`.

The following table provides a summary of selector syntax indicating usage and the version of CSS that introduced it

For example, under pre-CSS HTML, a heading element defined with red text would be written as:

```
<h1>Chapter 1.</h1>
```

Using CSS, the same element can be coded using style properties instead of HTML presentational attributes:

```
<h1 style="color: red;">Chapter 1.</h1>
```

The advantages of this may not be immediately clear but the power of CSS becomes more apparent when the style properties are placed in an internal style element or, even better, an external CSS file. For example, suppose the document contains the style element:

```
<style>
 h1 {
 color: red;
 }
</style>
```

All `h1` elements in the document will then automatically become red without requiring any explicit code. If the author later wanted to make `h1` elements blue instead, this could be done by changing the style element to:

```
<style>
 h1 {
 color: blue;
 }
</style>
```

rather than by laboriously going through the document and changing the color for each individual `h1` element.

The styles can also be placed in an external CSS file, as described below, and loaded using syntax similar to:

```
<link href="path/to/file.css" rel="stylesheet" type="text/css">
```

This further decouples the styling from the HTML document and makes it possible to restyle multiple documents by simply editing a shared external CSS file.