



JAWAHARLAL COLLEGE OF ENGINEERING AND TECHNOLOGY

(Approved by AICTE, Affiliated to APJ Abdul Kalam Technological University, Kerala)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(NBA Accredited)



COURSE MATERIAL

CST 205 OBJECT ORIENTED PROGRAMMING USING JAVA

VISION OF THE INSTITUTION

Emerge as a centre of excellence for professional education to produce high quality engineers and entrepreneurs for the development of the region and the Nation.

MISSION OF THE INSTITUTION

- To become an ultimate destination for acquiring latest and advanced knowledge in the multidisciplinary domains.
- To provide high quality education in engineering and technology through innovative teaching-learning practices, research and consultancy, embedded with professional ethics.
- To promote intellectual curiosity and thirst for acquiring knowledge through outcome based education.

- To have partnership with industry and reputed institutions to enhance the employability skills of the students and pedagogical pursuits.
- To leverage technologies to solve the real life societal problems through community services.

ABOUT THE DEPARTMENT

- Established in: 2008
- Courses offered: B.Tech in Computer Science and Engineering
- Affiliated to the A P J Abdul Kalam Technological University.

DEPARTMENT VISION

To produce competent professionals with research and innovative skills, by providing them with the most conducive environment for quality academic and research oriented undergraduate education along with moral values committed to build a vibrant nation.

DEPARTMENT MISSION

- Provide a learning environment to develop creativity and problem solving skills in a professional manner.
- Expose to latest technologies and tools used in the field of computer science.
- Provide a platform to explore the industries to understand the work culture and expectation of an organization.
- Enhance Industry Institute Interaction program to develop the entrepreneurship skills.
- Develop research interest among students which will impart a better life for the society and the nation.

PROGRAMME EDUCATIONAL OBJECTIVES

Graduates will be able to

- Provide high-quality knowledge in computer science and engineering required for a computer professional to identify and solve problems in various application domains.
- Persist with the ability in innovative ideas in computer support systems and transmit the knowledge and skills for research and advanced learning.
- Manifest the motivational capabilities, and turn on a social and economic commitment to community services.

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 modelling 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.

COURSE OUTCOMES

SUBJECT CODE: C204	
COURSE OUTCOMES	
C204.1	Write Java programs using the object oriented concepts - classes, objects, constructors, data hiding, inheritance and polymorphism.
C204.2	Utilize datatypes, operators, control statements, built in packages & interfaces, Input/Output Streams and Files in Java to develop programs.
C204.3	Illustrate how robust programs can be written in Java using exception handling mechanism.
C204.4	Write application programs in Java using multithreading and database connectivity
C204.5	Write Graphical User Interface based application programs by utilizing event handling features and Swing in Java.

PROGRAM SPECIFIC OUTCOMES (PSO)

The students will be able to

- Use fundamental knowledge of mathematics to solve problems using suitable analysis methods, data structure and algorithms.
- Interpret the basic concepts and methods of computer systems and technical specifications to provide accurate solutions.
- Apply theoretical and practical proficiency with a wide area of programming knowledge, design new ideas and innovations towards research.

CO PO MAPPING

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

CO'S	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
C204.1	3	3	3	2	3	2	2	-	-	-	-	2
C204.2	3	3	3	2	3	2	2	-	-	-	-	2
C204.3	3	3	3	2	3	2	2	-	-	-	-	2
C204.4	3	3	3	2	3	2	2	-	-	-	-	2
C204.5	3	3	3	2	3	2	2	-	-	-	-	2
C204	3	3	3	2	3	2	2	-	-	-	-	2

CO PSO MAPPING

CO'S	PSO1	PSO2	PSO3
C204.1	3	3	3
C204.2	3	3	3
C204.3	3	3	3
C204.4	3	3	3
C204.5	2	3	3
C204	2.8	3	3

Reference Materials

MODULE-1 INTRODUCTION

CHAPTER – 1

Approaches to Software Design

Software & Software Design

❓Software

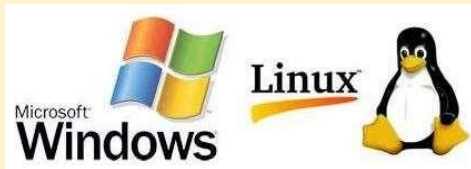
- Software is a collection of instructions that enable the user to interact with a computer , its hardware or perform tasks
- Without software, most computers would be useless. For example, without your Internet [browser](#) software, you could not surf the Internet. Without an [operating system](#), the browser could not run on your computer.

There are two types of software

- 1 . System Software
- 2 . Application Software

☐ Examples of system software are Operating System, Compilers, Interpreter, Assemblers, etc.

☐ Examples of Application software are Railways Reservation Software, Microsoft Office Suite Software, Microsoft Word, Microsoft PowerPoint , etc.



System Software's



Application software's

☐Software Design

☐Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.

☐The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system on the basis of SRS (Software Requirement Specification) and how the modules should be interconnected.

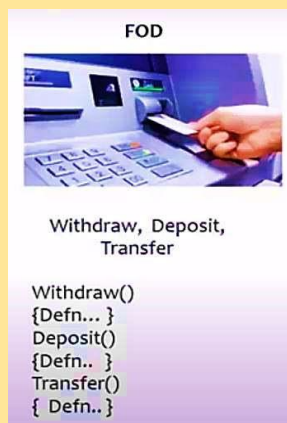
☐Software design is the first step in SDLC (Software Design Life Cycle)

☐It tries to specify how to fulfil the requirements mentioned in SRS document.

Functional Oriented Design (FOD)

- ❑ In function-oriented design, the system is comprised of many smaller sub-systems known as functions.
- ❑ These functions are capable of performing significant task in the system
- ❑ Function oriented design inherits some properties of structured design where divide and conquer methodology is used.
- ❑ This design mechanism divides the whole system into smaller functions

- ❑ These functional modules can share information among themselves by means of information passing and using information available globally.



Eg: Banking process

Here withdraw, Deposit, Transfer are functions and that can be divided in to subfunctions again.

So, in FOD, the entire problem is divided in to number of functions and those functions are broken down in to smaller functions and these smaller functions are converted in to software modules.

Object Oriented Design (OOD)

❑ OOD is based on **Objects** and interaction between the objects ❑ Interaction between objects is called **message communication**.

❑ It involves the designing of **Objects**, **Classes** and the relationship between the classes



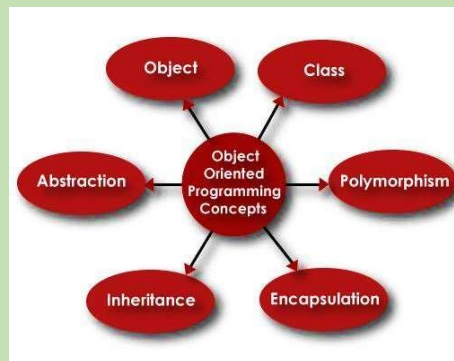
Consider the previous example of Banking process.

Here, customer, money and account are objects

❑ In OOD, implementation of a software based on the concepts of objects.

❑ This approach is very close to the real-world applications

Basic Object Oriented concepts



❑ OBJECT

❑ Objects are real-world entities that has their own properties and behavior.

❑ It has physical existence

Eg: person, banks, company, customers etc

❑ CLASS

❑ A class is a blueprint or prototype from which objects are created

❑ A class is a generalized description of an object.

❑ An object is an instance of a class

❑ Relationship between Object & Class

❑ Let's take Human Being as a class. My name is John, and I am an instance/object of the class Human Being

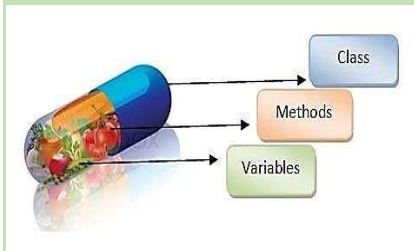
❑ Object has a **physical existence** while a **class** is just a **logical definition**.

❓Encapsulation

❓The wrapping up of data(variables) and function (methods) into a single unit (called class) is known as encapsulation.

❓It is also called "information hiding."

10



Key Points of Encapsulation

- Protection of data from accidental
- Flexibility and extensibility of the code and reduction in
- Encapsulation of a class can hide the internal details of how an does
- Encapsulation protects

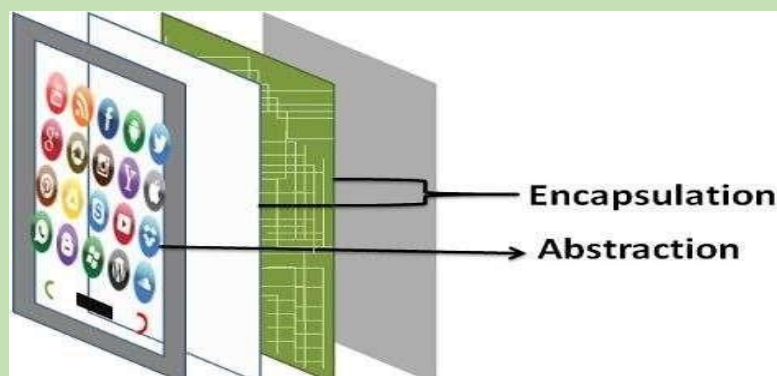
❓ABSTRACTION

❓Abstraction means **displaying only essential information** and hiding the details.

❓Data abstraction refers to providing only essential information about the data to the outside world, **hiding the background details or implementation.**

❓Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

Abstraction & Encapsulation



1

❓POLYMORPHISM

❓The word polymorphism means having many forms

❓ In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Eg: A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

❓ An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.



❓ Inheritance

❓ The capability of a class to derive properties and characteristics from another class is called Inheritance.

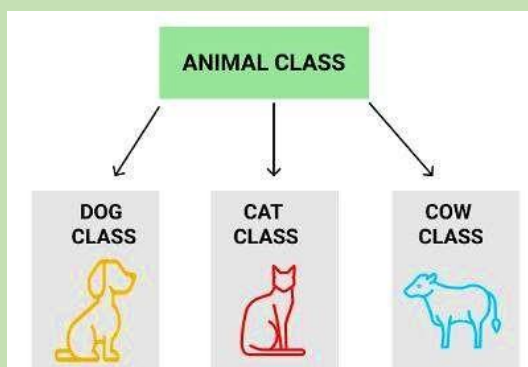
OR

Inheritance is the process by which objects of one class acquired the properties of objects of another classes

❓ **Sub Class** : The class that inherits properties from another class is called Sub class or Derived Class.

❓ **Super Class** : The class whose properties are inherited by sub class is called Base Class or Super class.

❓ **Reusability**: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.



Eg: Dog, Cat, Cow can be Derived Class of Animal Base Class.

Unified Modeling Language (UML)

❓ UML (Unified Modeling Language) is a general-purpose, **graphical modeling language** in the field of Software Engineering

❓ UML is used to specify, visualize, construct, and document the artifacts (major elements) of the software system

❓ UML is a **visual language** for **developing software blue prints** (designs). A blue print or design represents the model.

❓ For example, while constructing buildings, a designer or architect develops the building blueprints. Similarly, we can also develop blue prints for a software system.

UML is the most commonly and frequently used language for building software system blueprints

UML is **not a programming language**, it is rather a **visual language**.

The UML has the following features:

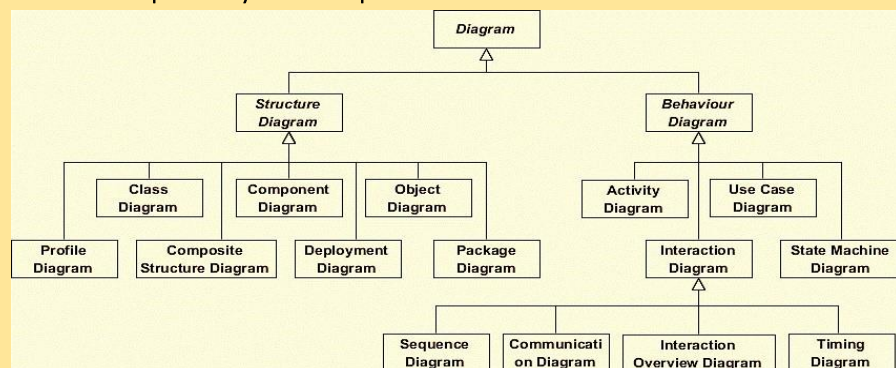
- It is a generalized modeling language.
- It is distinct from other programming languages like C++, Python, etc.
- It is interrelated to object-oriented analysis and design.
- It is used to visualize the workflow of the system.
- It is a **pictorial language**, used to generate powerful modeling artifacts

UML is linked with **object oriented design** and analysis

Diagrams in UML can be broadly classified as:

Structure Diagrams : Capture static aspects or structure of a system

Behavior Diagrams: Capture dynamic aspects or behavior of the system



❓CLASS DIAGRAM

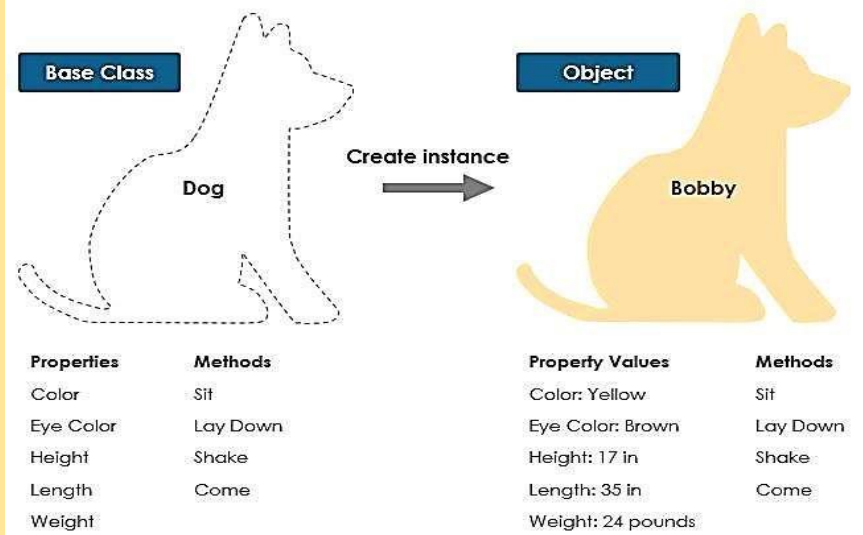
❓The **most widely use** UML diagram is the class diagram. It is the building block of all object oriented software systems.

❓Using class diagrams we can create the **static structure** of a system by showing system's classes, their methods and attributes.

❓Class diagrams also help us identify relationship between different classes or objects.

❓There are several software available which can be used online and offline to draw these diagrams Like **Edraw max**, **lucid chart** etc.

Class & Object



Class Notation

A class notation consists of three parts:

Class Name:

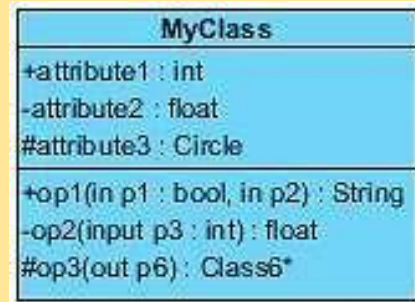
- The name of the class appears in the first partition.

Class Attributes:

- Attributes** are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

Class Operations (Methods):

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters are shown after the colon following the parameter name. Operations map onto class methods in code

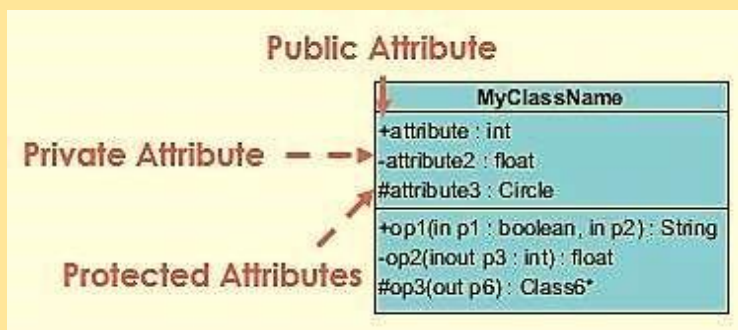


The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation

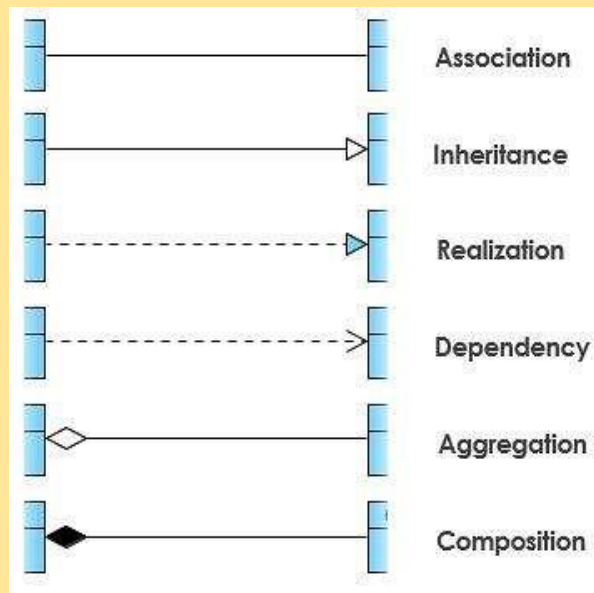
+ denotes public attributes or operations

- denotes private attributes or operations

denotes protected attributes or operations

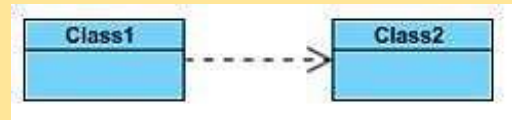


Relationships between classes



1. Dependency

- A dependency means the **relation between two or more classes** in which a change in one may force changes in the other.
- Dependency indicates that one class depends on another.
- A dashed line with an open arrow



2. Inheritance (or Generalization)

- A generalization helps to connect a subclass to its superclass.
- A sub-class is inherited from its superclass.
- A solid line with a hollow arrowhead that point from the child to the parent class

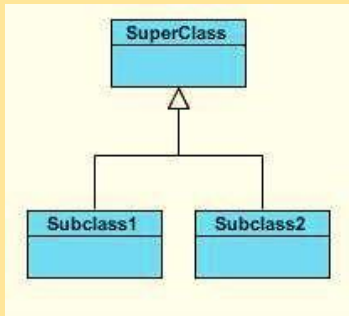


Fig: Inheritance (or Generalization)

3 . Association

- This kind of relationship represents static relationships between classes A and B.
- There is an association between Class1 and Class2
- A solid line connecting two classes

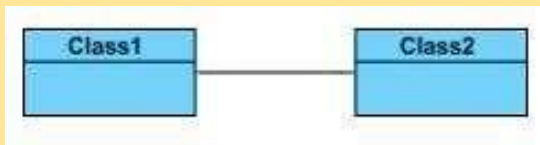
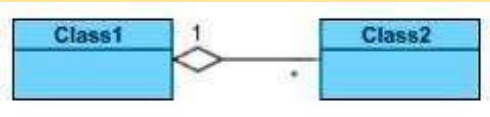


Fig: Association

4. Aggregation

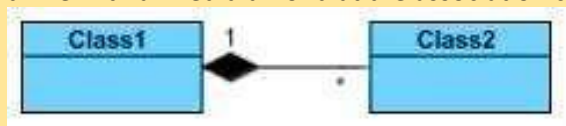
- A special type of association. It represents a "part of" relationship
- Class2 is part of Class1.



- Many instances (denoted by the *) of Class2 can be associated with Class1.
- A solid line with an unfilled diamond at the association end connected to the class of composite

5. Composition

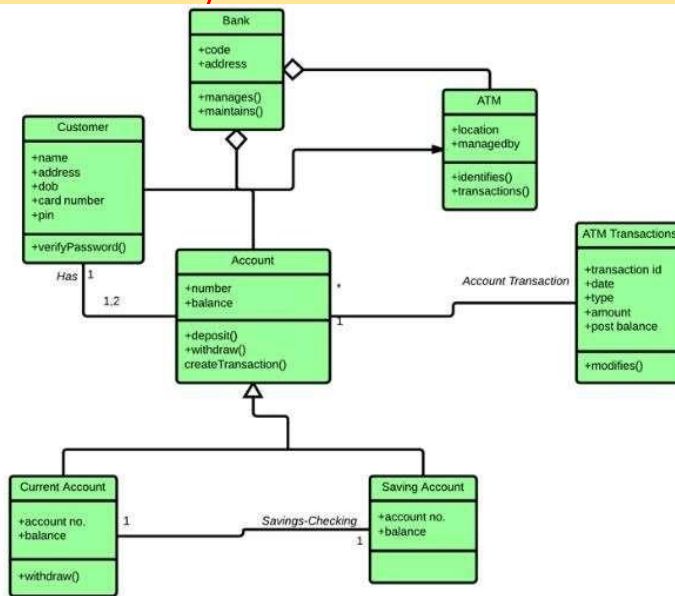
- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.
- Class2 cannot stand by itself.
- A solid line with a filled diamond at the association connected to the class of composite



Multiplicity

- It means, how many objects of each class take part in the relationships
- Exactly one - 1
- Zero or one - 0..1
- Many - 0..* or *
- One or more - 1..*
- Exact Number - e.g. 3..4 or 6
- Or a complex relationship - e.g. 0..1, 3..4, 6.* would mean any number of objects other than 2 or 5

Eg: Class diagram for an ATM system



USE CASE MODEL / USE CASE DIAGRAM

The purpose of a use case diagram in UML is to demonstrate the different ways that a **user** might interact with a system.

It captures the **dynamic behavior** of a live system.

a use case diagram can summarize the details of your system's users (also known as actors) and their interactions with the system.

To build a use case diagram, we will use a set of specialized symbols and connectors

A use case diagram doesn't go into a lot of detail, but it depicts a high-level overview of the relationship between use cases, actors, and systems.

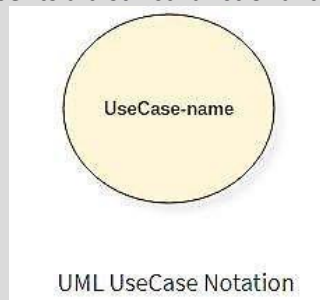
❓A use-case model is a model of how different types of users interact with the system to solve a problem **Use case diagram components**

- **Actors:** The **users** that interact with a system. An actor can be a person, an organization, or an outside system that interacts with your application or system. They must be external objects that produce or consume data.
- **System:** A specific sequence of actions and interactions between actors and the system. A system may also be referred to as a **scenario**
- **Goals:** The **end result** of most use cases. A successful diagram should describe the activities and variants used to reach the goal.

Use case diagram symbols and notation

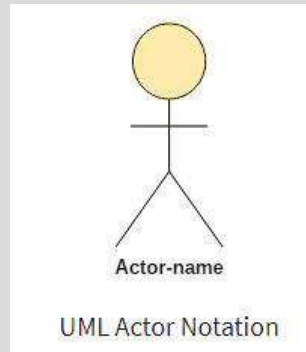
1. Use cases

- Horizontally shaped ovals that represent the **different uses** that a user might have
- A use case represents a distinct functionality of a system, a component, a package, or a class



2 . Actors

- Stick figures that **represent the people** actually employing the use cases.
- A user is the best example of an actor
- One actor can be associated with multiple use cases in the system

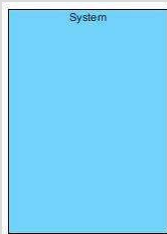


3. Associations

- A line between actors and use cases
- In complex diagrams, it is important to know which actors are associated with which use cases.

4. System boundary boxes

- A box that sets a system scope to use cases
- All use cases outside the box would be considered outside the scope of that system.



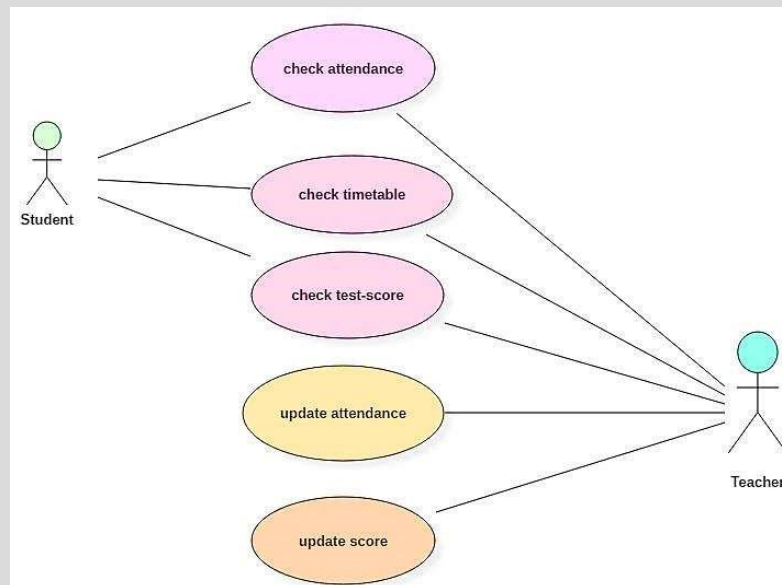
5. Packages

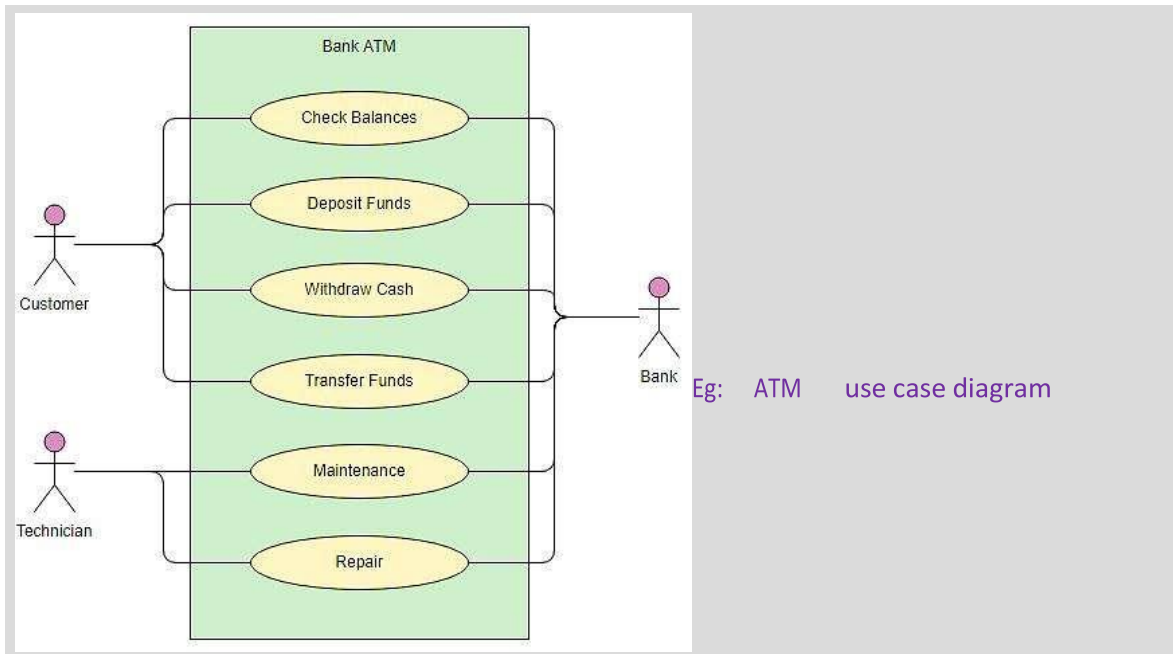
- A UML shape that allows you to put different elements into groups
- Just as with component diagrams, these groupings are represented as file folders.

purposes of use case diagram

- ☑ Used to gather the requirements of a system.
- ☑ Used to get an outside view of a system.
- ☑ Identify the external and internal factors influencing the system. ☑ Show the interaction among the requirements and actors

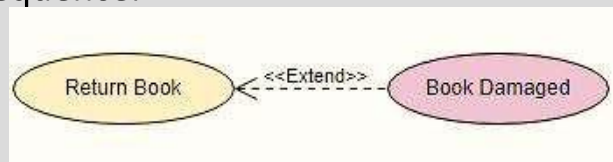
Eg: Use case diagram of a student management system





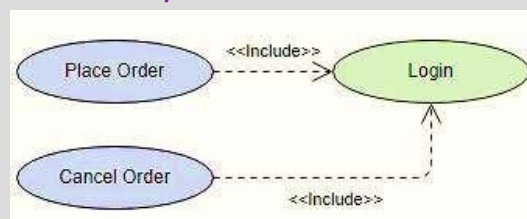
<<extend>> Use Case

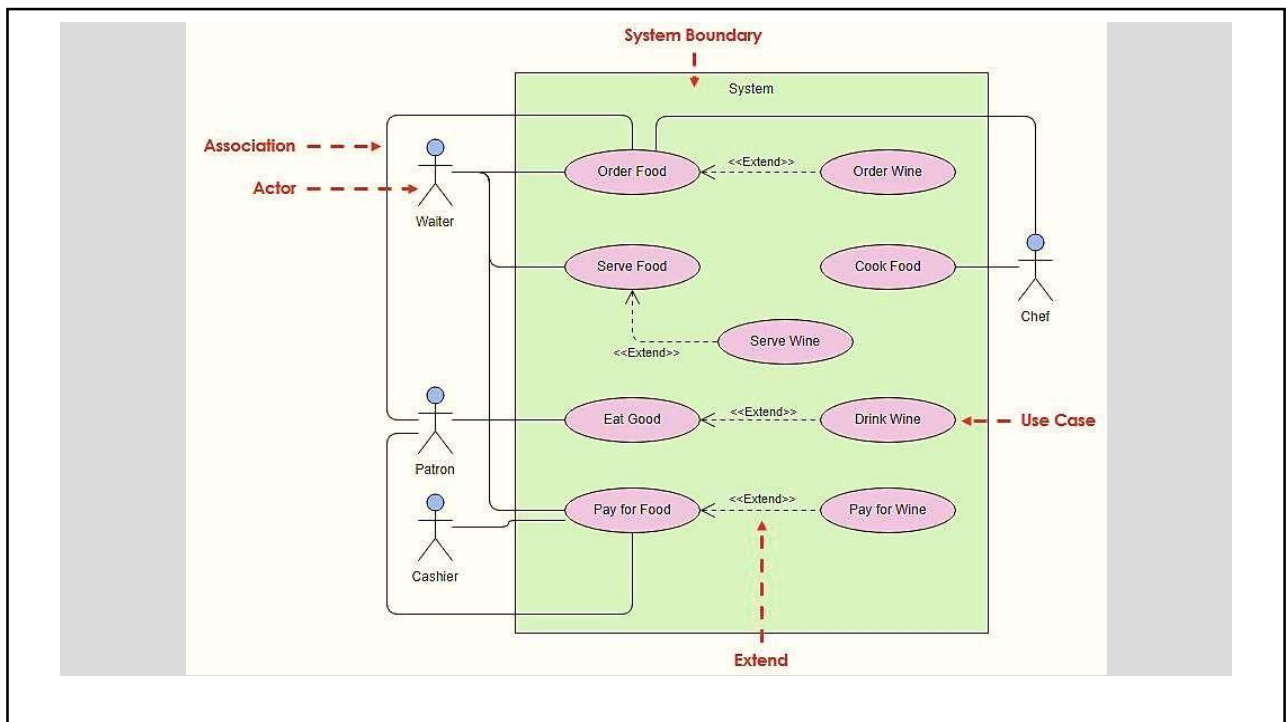
The <<extend>> use case inserting additional action sequences into the base use-case sequence.



<<include>> Use Case

The time to use the <<include>> relationship is after you have completed the first cut description of all your main Use Cases.





❓INTERACTION DIAGRAM

❓INTERACTION DIAGRAMS are used in UML to establish communication between objects

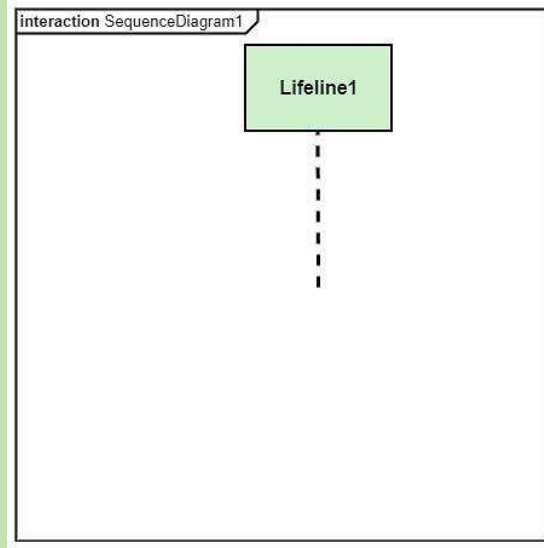
❓Interaction diagrams mostly focus on message passing and how these messages make up one functionality of a system

❓The critical component in an interaction diagram is lifeline and messages.

❓ Interaction diagrams capture the dynamic behavior of any system

❓The details of interaction can be shown using several notations such as sequence diagram, timing diagram, collaboration diagram.

Notation of an Interaction Diagram



Purpose of an Interaction Diagram

- To capture the **dynamic behavior** of a system.
- To describe the **message flow** in the system.
- To describe the structural organization of the objects.
- To describe the **interaction among objects**.
- Interaction diagram visualizes the communication and sequence of message passing in the system.
- Interaction diagram represents the ordered sequence of interactions within a system.
- Interaction diagrams can be used to explain the architecture of an object-oriented system.

Different types of Interaction Diagrams

1. Sequence diagram

- Purpose - To visualize the sequence of a **message flow** in the system
- Shows the **interaction between two lifelines**

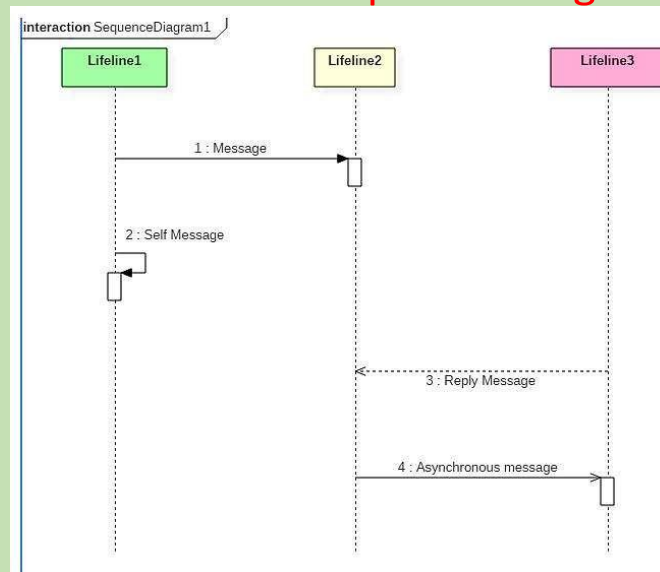
2. Collaboration diagram

- Also called as a **communication diagram**
- Shows how various lifelines in the system connects.

3. Timing diagram

- Focus on the instance at which a message is sent from one object to another object.

How to draw a Sequence Diagram



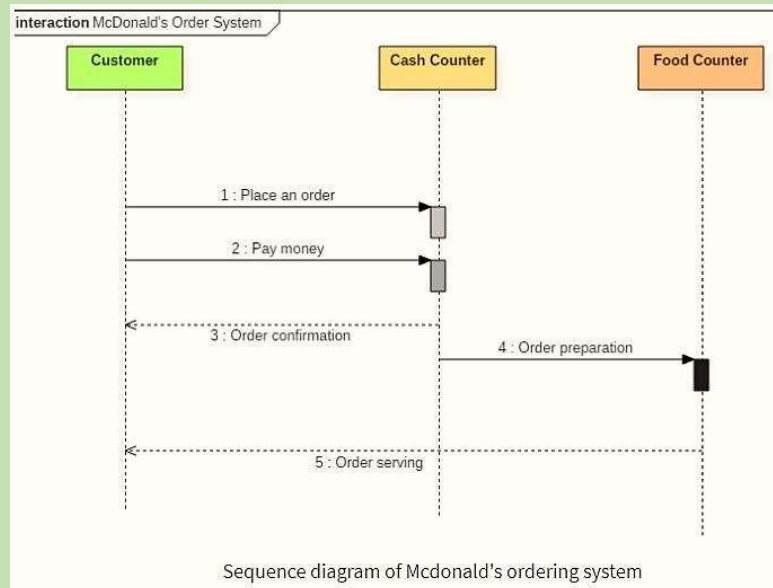
- In a sequence diagram, a lifeline is represented by a vertical bar.
- A **lifeline** represents an **individual participant** in a sequence diagram
- A lifeline will usually have a **rectangle containing its object name**

- A message flow between two or more objects is represented using a vertical dotted line which extends across the bottom of the page.
- In a sequence diagram, different types of messages and operators are used
- In a sequence diagram, iteration and branching are also used. 47

Messages used

Message Name	Meaning
Synchronous message	The sender of a message keeps waiting for the receiver to return control from the message execution.
Asynchronous message	The sender does not wait for a return from the receiver; instead, it continues the execution of a next message.
Return message	The receiver of an earlier message returns the focus of control to the sender.
Object creation	The sender creates an instance of a classifier.
Object destruction	The sender destroys the created instance.
Found message	The sender of the message is outside the scope of interaction.
Lost message	The message never reaches the destination, and it is lost in the interaction.

Sequence diagram example



Benefits of a Sequence Diagram

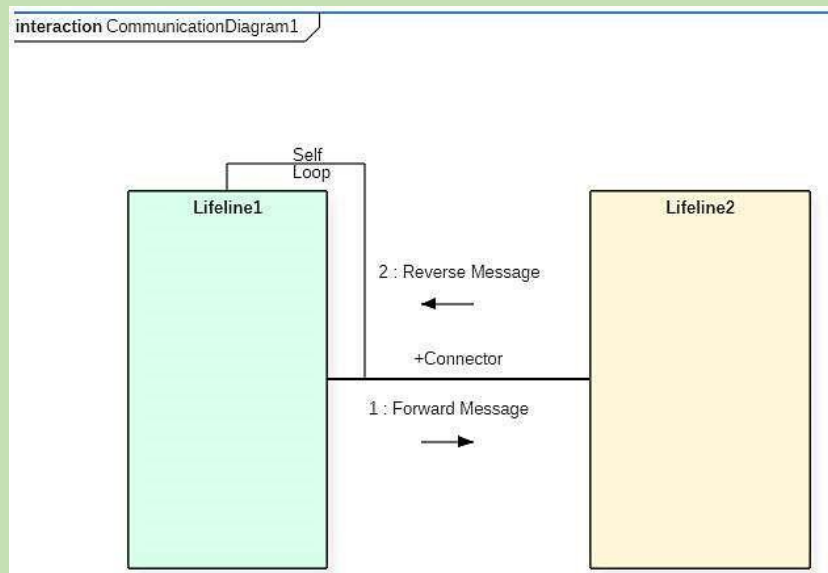
- Sequence diagrams are used to **explore** any **real application** or a system.
- Sequence diagrams are used to **represent message flow** from one object to another **object**.
- Sequence diagrams are **easier to maintain**.
- Sequence diagrams are **easier to generate**.
- Sequence diagrams can be **easily updated** according to the changes within a system.
- Sequence diagram **allows reverse as well as forward engineering**.

Drawbacks of a sequence diagram

- Sequence diagrams can **become complex** when **too many lifelines** are involved in the system.
- If the order of message sequence is changed, then incorrect results are produced.
- Each sequence needs to be represented using different message notation, which can be a little complex.
- The type of message decides the type of sequence inside the diagram

51

How to draw a Collaboration /Communication Diagram



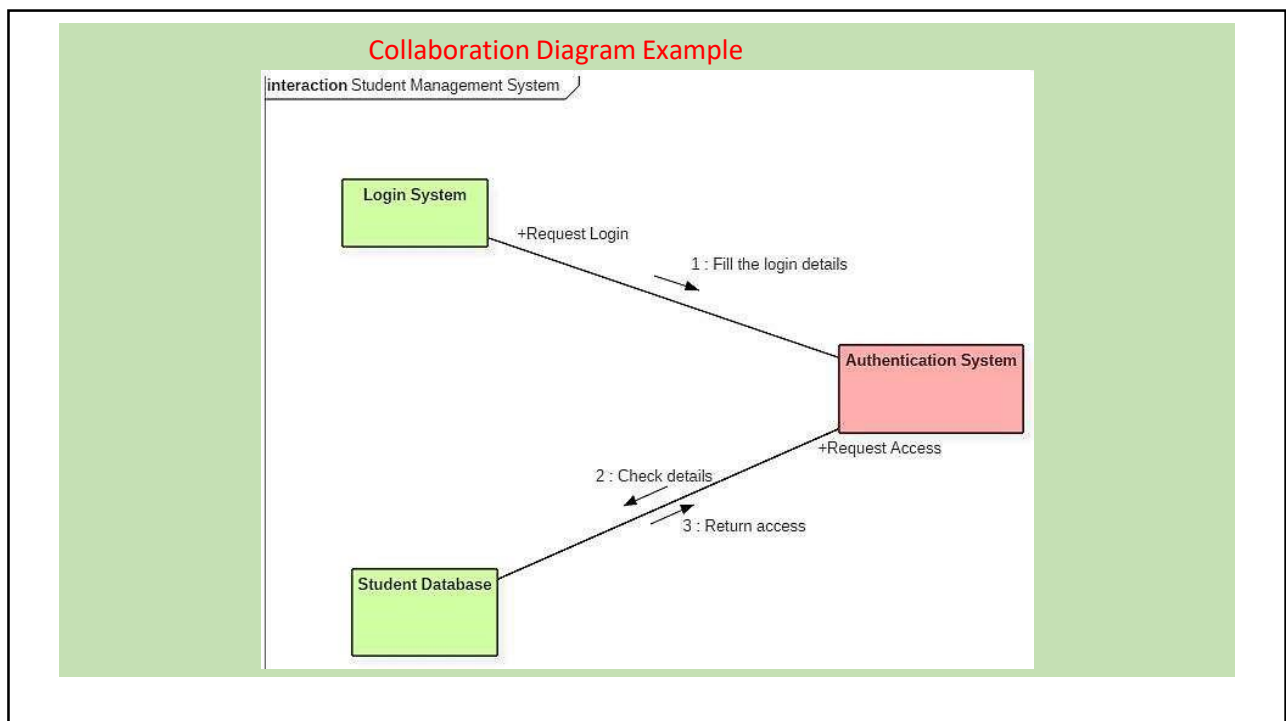
As per Object-Oriented Programming (OOPs), an object entity has various attributes associated with it.

Usually, there are **multiple objects** present inside an objectoriented system where each object can be associated with any other object inside the system

Collaboration Diagrams **are used to explore the architecture of objects inside the system.**

The **message flow** between the objects can be represented using a collaboration diagram.

53



The above collaboration diagram represents a student information management system. The flow of communication in the above diagram is given by,

- A student requests a login through the login system.
- An authentication mechanism of software checks the request.
- If a student entry exists in the database, then the access is allowed; otherwise, an error is returned.

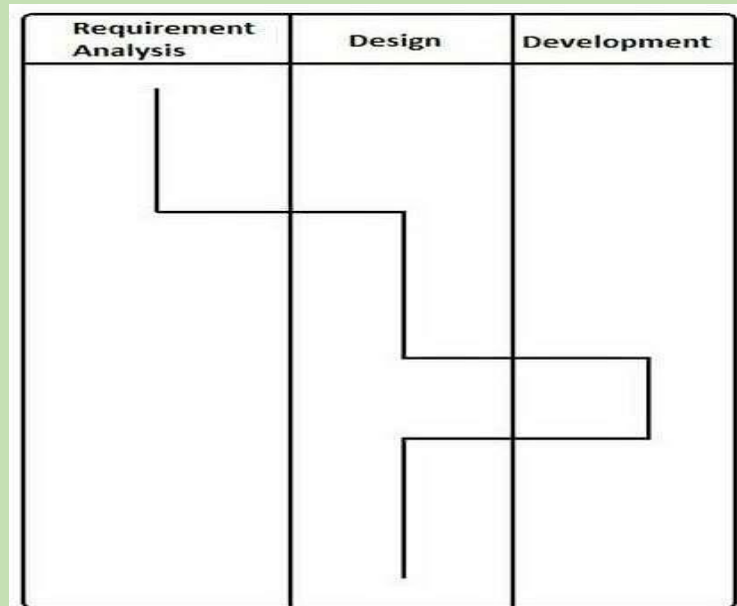
Benefits of Collaboration Diagram

- It is also called as a communication diagram.
- It emphasizes the structural aspects of an interaction diagram how lifeline connects.
- Its syntax is similar to that of sequence diagram except that **lifeline don't have tails**.
- Messages passed over sequencing is indicated by numbering each message hierarchically.
- It allows you to focus on the elements rather than focusing on the message flow as described in the sequence diagram.
- Sequence diagrams can be easily converted into a collaboration diagram as collaboration diagrams are not very expressive.

Drawbacks of a Collaboration Diagram

- Collaboration diagrams can become **complex** when **too many objects** are present within the system.
- It is hard to explore each object inside the system.
- Collaboration diagrams are time consuming.
- The object is destroyed after the termination of a program.
- The state of an object changes momentarily, which makes it difficult to keep track of every single change the occurs within an object of a system.

How to draw a Timing Diagram



❑ In the above diagram, first, the software passes through the requirements phase then the design and later the development phase.

❑ The output of the previous phase at that given instance of time is given to the second phase as an input

❑ Thus, the timing diagram can be used to describe SDLC (Software Development Life Cycle) in UML.

Benefits of a Timing Diagram

- Timing diagrams are used to represent the state of an object at a particular instance of time.
- Timing diagram allows reverse as well as forward engineering.
- Timing diagram can be used to keep track of every change inside the system.

Drawbacks of a Timing Diagram

- Timing diagrams are difficult to understand.
- Timing diagrams are difficult to maintain.

60

? ACTIVITY DIAGRAM

? ACTIVITY DIAGRAM is basically a flowchart to represent the flow from one activity to another activity.

? The activity can be described as an operation of the system

? The basic purpose of activity diagrams is to capture the dynamic behavior of the system

? It is also called object-oriented flowchart

? Activity diagrams are not only used for visualizing the dynamic nature of a system, but they are also used to construct the executable system by using forward and reverse engineering techniques.

61

Basic components of an activity diagram

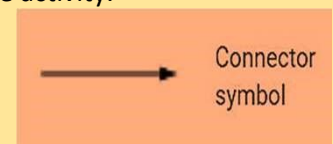
- **Action:** A step in the activity wherein the users or software perform a given task.

- **Decision node:** A **conditional branch** in the flow that is represented by a diamond. It includes a single input and two or more outputs.
- **Control flows:** Another name for the **connectors** that show the flow between steps in the diagram.
- **Start node:** Symbolizes the **beginning** of the activity. The start node is represented by a black circle.
- **End node:** Represents the **final step** in the activity. The end node is represented by an outlined black circle.

62

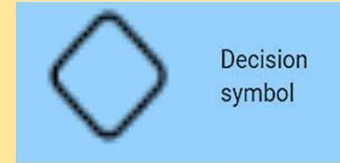
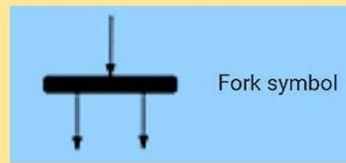
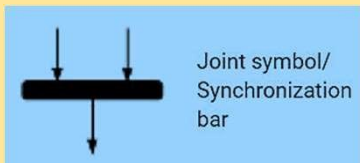
Activity diagram symbols

- **Start symbol** - Represents the beginning of a process or workflow in an activity diagram.
- **Activity symbol** - Indicates the activities that make up a modeled process. These symbols, which include short descriptions within the shape, are the **main building blocks of an activity diagram**.
- **Connector symbol** - Shows the directional flow, or control flow, of the activity.



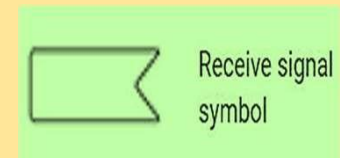
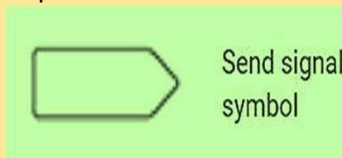
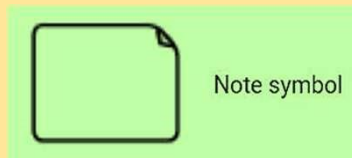
63

- **Joint symbol / Synchronization bar** - Combines two concurrent activities and re-introduces them to a flow where only one activity occurs at a time. Represented with a thick vertical or horizontal line.
- **Fork symbol** - Splits a single activity flow into two concurrent activities. Symbolized with multiple arrowed lines from a join.
- **Decision symbol** - Represents a decision and always has at least two paths branching out with condition text.



64

- **Note symbol** - Allows the diagram creators or collaborators to communicate additional messages that don't fit within the diagram itself. Leave notes for added clarity and specification.
- **Send signal symbol** - Indicates that a signal is being sent to a receiving activity
- **Receive signal symbol** - Demonstrates the acceptance of an event. After the event is received, the flow that comes from this action is completed.

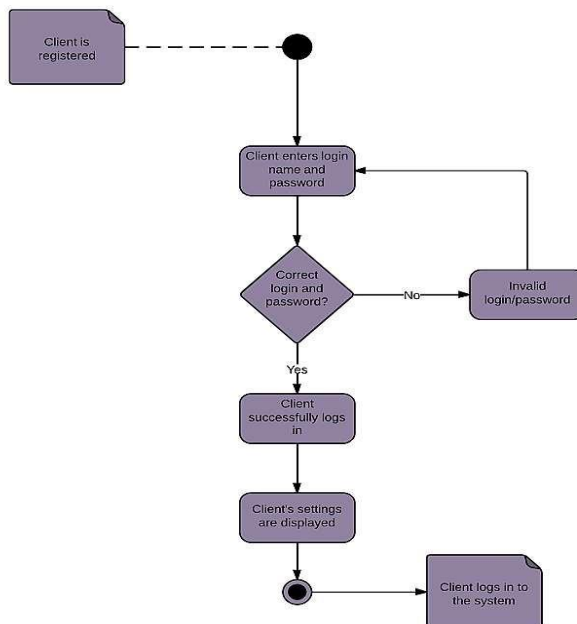


65

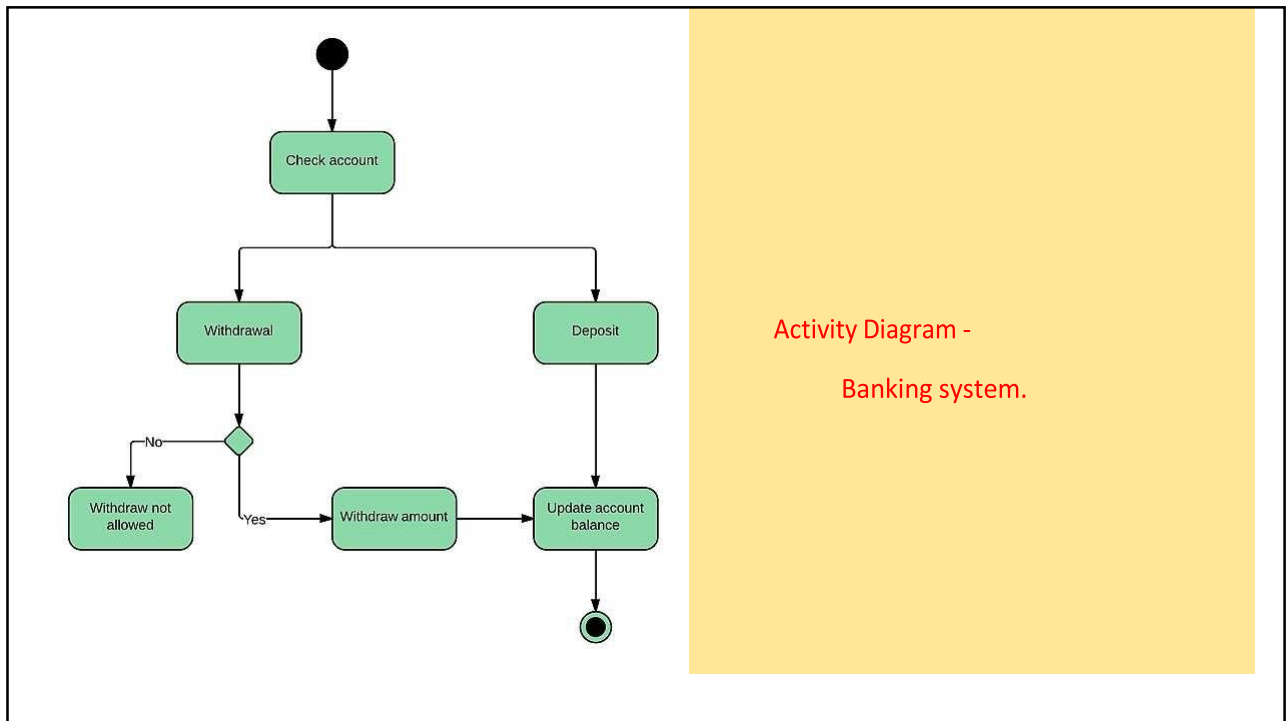
- **Flow final symbol** - Represents the **end of a specific process flow**. This symbol shouldn't represent the end of all flows in an activity. The flow final symbol should be placed at the end of a single activity flow.
- **Condition text** - Placed **next to a decision marker** to let you know under what condition an activity flow should split off in that direction
- **End symbol** - Marks the **end state of an activity** and represents the completion of all flows of a process.



66



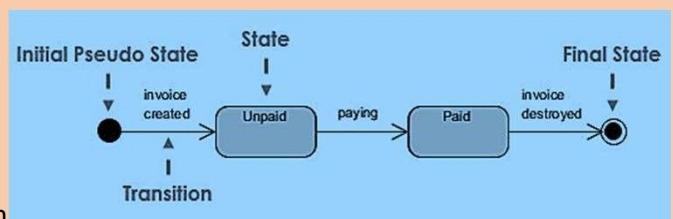
Activity diagram - a login page



STATE CHART DIAGRAM

- State chart diagram is used to capture the **dynamic aspect** of a system
- An **object goes through various states during its lifespan**. The lifespan of an object remains until the program is terminated. **The object goes from multiple states** depending upon the event that occurs within the object.
- Each state represents some unique information about the object.
- State chart diagram visualizes the flow of execution from one state to another state of an object.
- It **represents the state of an object** from the creation of an object until the object is destroyed or terminated.

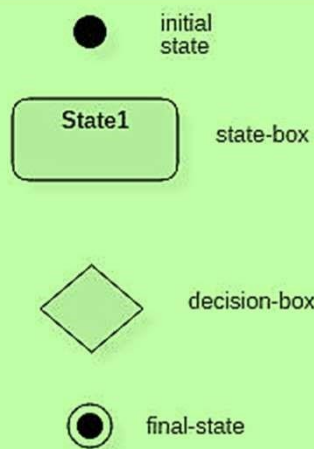
- The primary purpose of a state chart diagram is to model interactive systems and define each and every state of an object.
- **State chart diagrams** are also referred to as **State machines** and **state diagrams**.
- A state machine consists of states, linked by transitions. A state is a condition of an object in which it performs some activity or waits for an event



Simple State Machine Diagram Notation

70

Notation and Symbol for State Machine / State Chart Diagram



UML state diagram notations

- **Initial state** - The initial state symbol is used to indicate the beginning of a state machine diagram.
- **Final state** - This symbol is used to indicate the end of a state machine diagram.
- **Decision box** - It contains a condition. Depending upon the result of an evaluated guard condition, a new path is taken for program execution.
- **Transition** - A transition is a change in one state into another state which is occurred because of some event. A transition causes a change in the state of an object.

72

- **State box**

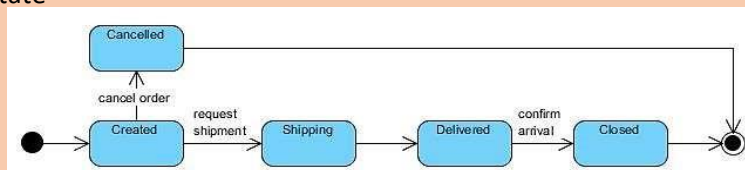
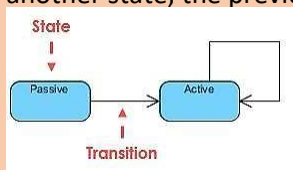
☐ States represent situations during the life of an object.

☐ It is denoted using a **rectangle with round corners**.

☐ The name of a state is written inside the rounded rectangle.

☐ A state can be either **active or inactive**.

☐ When a state is in the working mode, it is active, as soon as it stops executing and transits into another state, the previous state becomes inactive, and the current state becomes active.



73

Types of State

Simple state

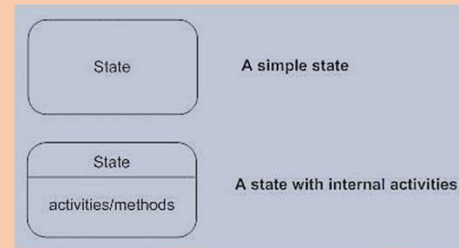
- They do not have any sub state.

Composite state

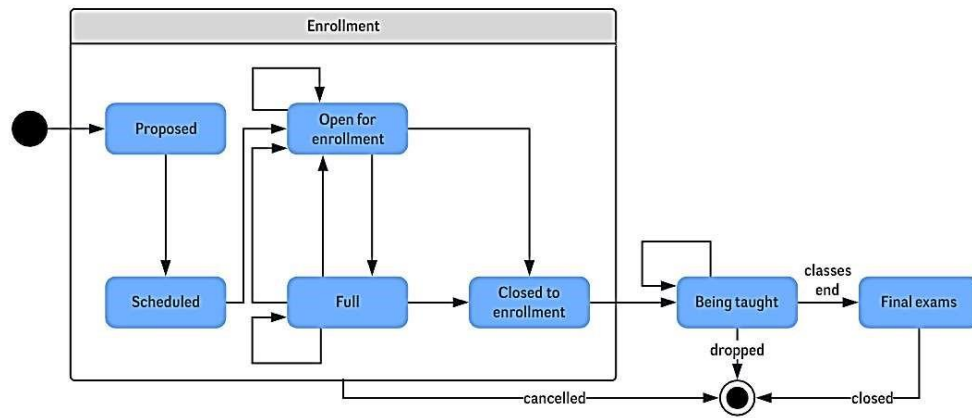
- These types of states can have one or more than one sub state.
- A composite state with two or more sub states is called an orthogonal state.

Submachine state

- These states are semantically equal to the composite states • Unlike the composite state, we can reuse the submachine states.



74



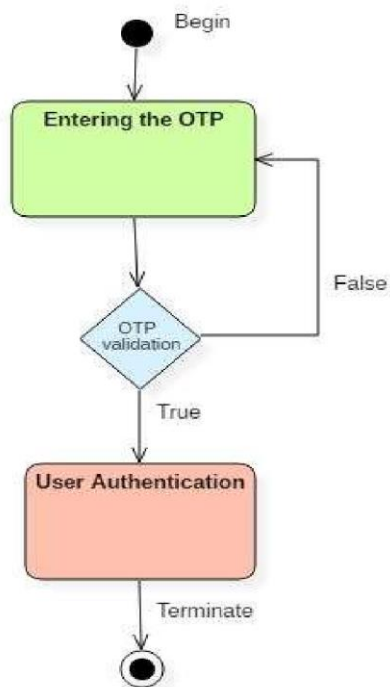
University
state
Diagram

- The composite state

“Enrollment” is made up of various sub states that will lead students through the enrollment process.

- Once the student has enrolled, they will proceed to “Being taught” and finally to “Final exams.”

75



Eg: state chart diagram
user authentication process.

State machine vs. Flowchart

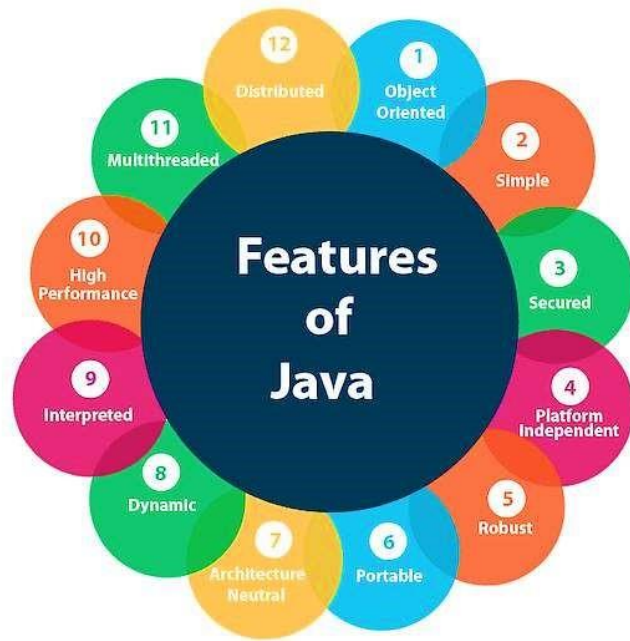
Statemachine	FlowChart
It represents various states of a system.	The Flowchart illustrates the program execution flow.
The state machine has a WAIT concept, i.e., wait for an action or an event.	The Flowchart does not deal with waiting for a concept.
State machines are used for a live running system.	Flowchart visualizes branching sequences of a system.
The state machine is a modeling diagram.	A flowchart is a sequence flow or a DFD diagram.
The state machine can explore various states of a system.	Flowchart deal with paths and control flow.

MODULE 1

CHAPTER 2 INTRODUCTION TO JAVA

JAVA

- Java is a powerful general-purpose , Object Oriented programming language developed by Sun Micro System of USA in 1991.
- Development team members are James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan
- First name of Java is “Oak,” but was renamed “Java” in 1995.
- Java derives much of its character from C and C++.
- Java Changed the Internet by simplifying web programming
- Java innovated a new type of networked program called the **applet**



FEATURES OF JAVA

(Java Buzzwords)

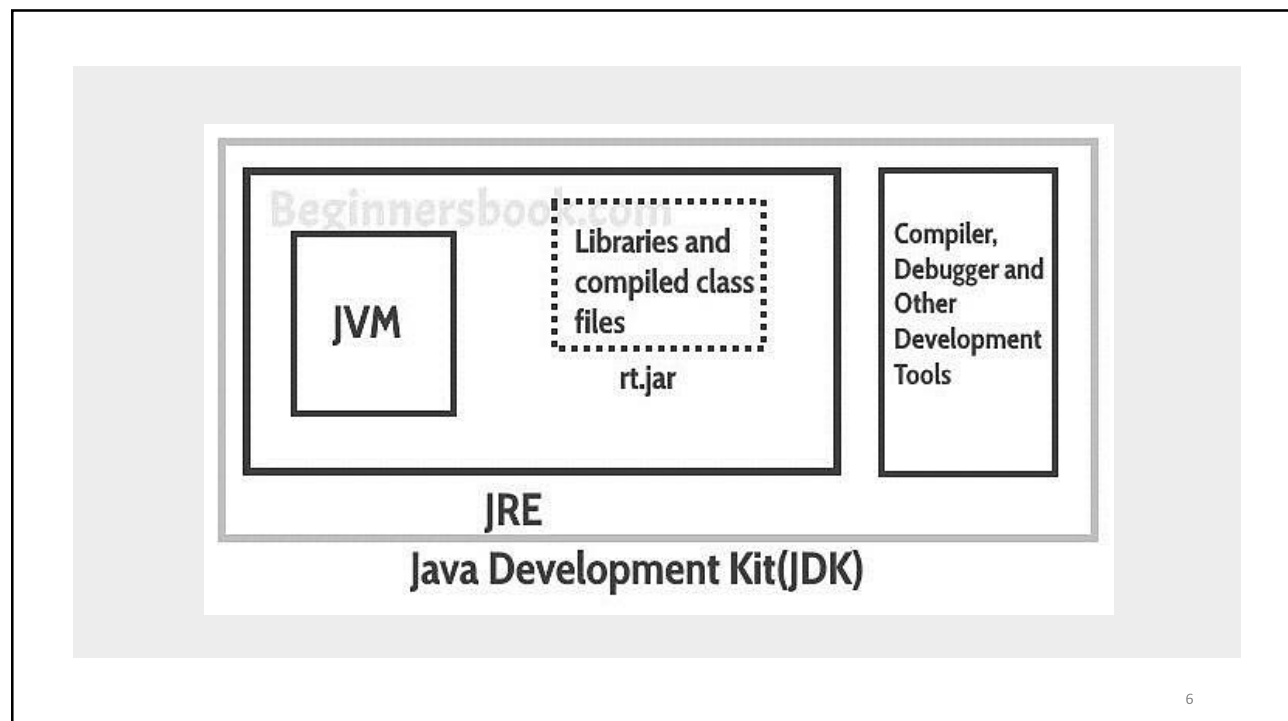
❓ JAVA RUNTIME ENVIRONMENT (JRE)

- A software program needs an environment to run .
- The runtime environment loads class files and ensures there is access to memory and other **system resources** to run them.
- Java Runtime Environment provides the minimum requirements for executing a Java application programs.
- JRE is an **installation package** which provides environment to **only run(not develop)** the java program(or application)onto your machine.
- JRE is only used by them who only wants to run the Java Programs
i.e. end users of your system. JRE can be view as a **subset of JDK**.

? JAVA DEVELOPMENT KIT (JDK)

- The Java Development Kit (JDK) is a software development environment used for developing and executing Java applications and applets
- It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.
- **JDK is only used by Java Developers.**

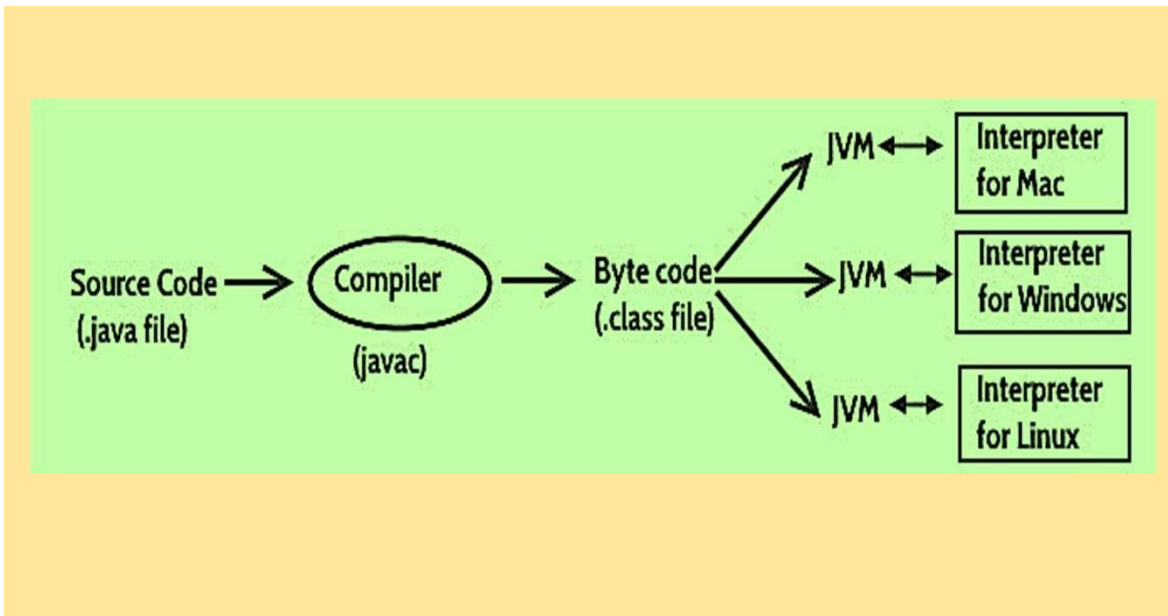
5



6

❓ JAVA VIRTUAL MACHINE (JVM)

- JVM is a program which provides the runtime environment to execute Java programs. Java programs cannot run if a supporting JVM is not available.
 - JVM is a virtual machine that resides in the real machine (your computer) and the **machine language for JVM is byte code**.
 - The Java compiler generate byte code for JVM rather than different machine code for each type of machine.
 - **JVM executes the byte code generated by compiler** and produce output.
 - JVM is the one that makes java platform independent.
-
- The **primary function** of JVM is to **execute the byte code** produced by compiler
 - The JVM doesn't understand Java source code, that's why we need to have **javac** compiler
 - Java compiler (javac) compiles *.java files to obtain *.class files that contain the byte codes understood by the JVM.
 - JVM makes java portable (write once, run anywhere).
 - Each operating system has different JVM, however the output they produce after execution of byte code is same across all operating systems.

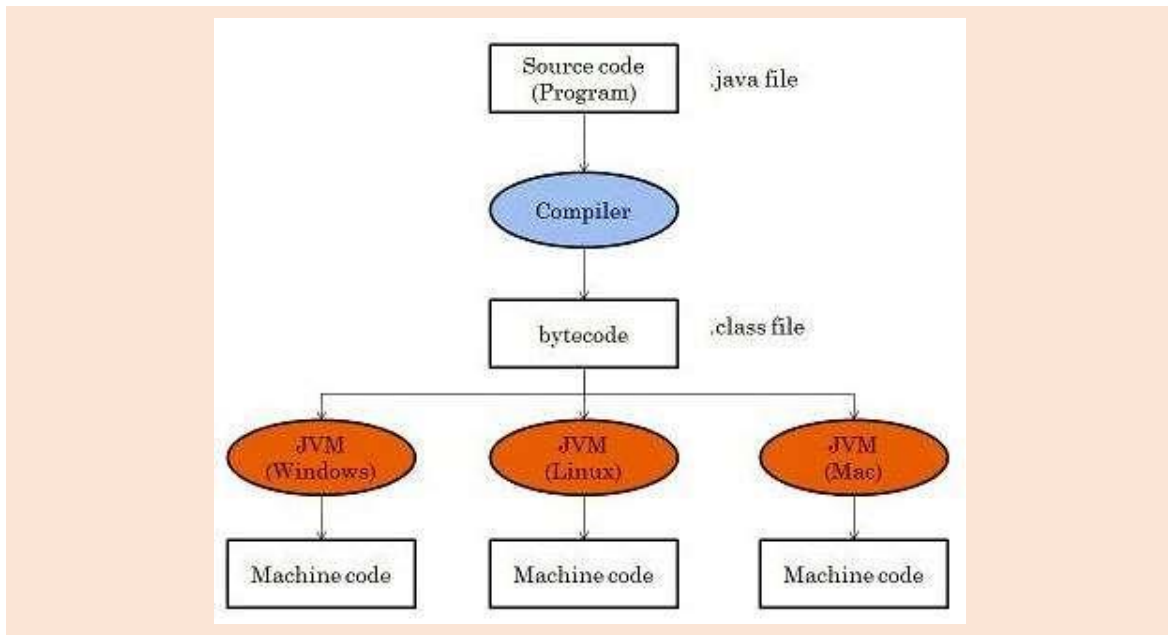


9

❓ BYTE CODE

- Java byte code is the instruction set for the Java Virtual Machine
- It is the machine code in the form of a .class file.
- Byte code is a machine independent code
- It is not completely a compiled code but it is an intermediate code somewhere in the middle which is later interpreted and executed by JVM.
- Byte code is a machine code for JVM.
- Byte code implementation makes Java a platform- Independent language.

10



11

🔗 JAVA COMPILER

- Java is compiled language. But it is very different from traditional compiling in the way that after compilation source code is converted to byte code.

• **Javac** is the most popular Java compiler

- Java has a virtual machine called JVM which then converts byte code to target code of machine on which it is run.
- JVM performs like an interpreter. It doesn't do it alone, though. It has its own compiler to convert the byte code to machine code. This compiler is called **Just In Time** or **JIT compiler**.

12

❏ JAVA APPLET

- An applet is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser
- It runs inside the web browser and works at client side
- Applets are used to make the web site more dynamic and entertaining
- Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. JDK provides a standard applet viewer tool called applet viewer.
- In general, execution of an applet does not begin at main() method.

13



Lifecycle of Java Applet

14

Java Applet vs Java Application

Java Application	Java Applet
Java Applications are the stand-alone programs which can be executed independently	Java Applets are small Java programs which are designed to exist within HTML web document
Java Applications must have main() method for them to execute	Java Applets do not need main() for execution
Java Applications just needs the JRE	Java Applets cannot run independently and require API's
Java Applications do not need to extend any class unless required	Java Applets must extend java.applet.Applet class
Java Applications can execute codes from the local system	Java Applets Applications cannot do so
Java Applications has access to all the resources available in your system	Java Applets has access only to the browser-specific services

15

JAVA BUZZWORDS

Simple

- It's simple and easy to learn if you already know the basic concepts of Object Oriented Programming.
- C++ programmer can move to JAVA with very little effort to learn.
- Java syntax is based on C++
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.

16

Object oriented

- Java is true object oriented language. Everything in Java is an object.
- All program code and data reside within objects and classes.
- Java comes with an extensive set of classes, arranged in packages that can be used in our programs through inheritance.

Distributed

- Java is designed for distributed environment of the Internet. Its used for creating applications on networks

- Java enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

17

□ Compiled and Interpreted

- Usually a computer language is either compiled or Interpreted. Java combines both this approach and makes it a two-stage system.
- Compiled : Java enables creation of a cross platform programs by compiling into an intermediate representation called Java Byte code.
- Interpreted : Byte code is then interpreted, which generates machine code that can be directly executed by the machine that provides a Java Virtual machine.

18

□ Robust

- It provides many features that make the **program execute reliably** in variety of environments.
- Java is a **strictly typed language**. It checks code both at compile time and runtime.
- Java takes care of all memory management problems with **garbage-collection**.
- Java, with the help of **exception handling** captures all types of serious errors and eliminates any risk of crashing the system.

19

□ Secure

- Java provides a “**firewall**” between a networked application and your computer.
- When a Java Compatible Web browser is used, **downloading can be done safely** without fear of viral infection or malicious intent.
- Java achieves this protection by confining a Java program to the java execution environment and not allowing it to access other parts of the computer.

□ Architecture Neutral

- Java language and Java Virtual Machine helped in achieving the goal of “**write once; run anywhere, any time, forever.**”
- Changes and upgrades in operating systems, processors and system resources will not force any changes in Java Programs.

20

□ Portable

- Java is portable because it facilitates you to **carry the Java byte code to any platform**. It doesn't require any implementation.
- Java Provides a way to download programs dynamically to all the various types of platforms connected to the Internet.

□ High Performance

- Java performance is high because of the use of byte code.
- The byte code can be easily translated into native machine code.

21

□ Multithreaded

- Multithreaded Programs **handled multiple tasks simultaneously**, which was helpful in creating interactive, networked programs.

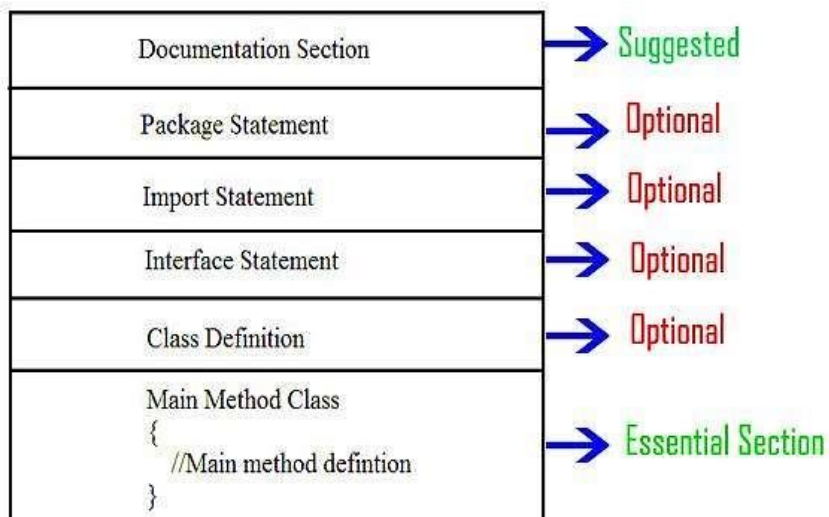
- Java run-time system comes with tools that support multiprocess synchronization used to construct smoothly interactive systems

□ Dynamic

- Java is capable of linking in new class libraries, methods, and objects.
- It supports functions from native languages (the functions written in other languages such as C and C++).
- It supports dynamic loading of classes. It means classes are loaded on demand

22

JAVA PROGRAM STRUCTURE



23

❓ Documentation Section

- You can write a comment in this section. It helps to understand the code. These are optional
- It is used to improve the readability of the program.
- The compiler ignores these comments during the time of execution
- There are three types of comments that Java supports

❓ Single line Comment `//This is single line comment` ❓ Multi-line Comment `/*`

`this is multiline comment. and support multiple lines*/`

❓ Documentation Comment `/** this is documentation cmnt*/`

24

❓ Package Statement

- We can create a package with any name. A package is a **group of classes** that are defined by a name.
- That is, if you want to declare many classes within one element, then you can declare it within a package
- It is an optional part of the program, i.e., if you do not want to declare any package, then there will be no problem with it, and you will not get any errors.
- Package is declared as: `package package_name;`

Eg: `package mypackage;`

25

❓ Import Statement

- If you want to use a class of another package, then you can do this by importing it directly into your program.
- Many predefined classes are stored in packages in Java
- We can import a specific class or classes in an import statement.

Examples: `import java.util.Date;` //imports the date class

```
import java.applet.*;           /*imports all the classes from the java
                                applet package*/
```

26

❓ Interface Statement

- This section is used to specify an interface in Java
- Interfaces are like a class that includes a group of method declarations
- It's an optional section and can be used when programmers want to implement multiple inheritances within a program.

❓ Class Definition

- A Java program may contain several class definitions.
- Classes are the main and essential elements of any Java program.
- A class is a collection of variables and methods

27

📌 Main Method Class

- The main method is from where the **execution actually starts** and follows the order specified for the following statements
- Every Java stand-alone program requires the main method as the starting point of the program.
- This is an essential part of a Java program.
- There may be many classes in a Java program, and only one class defines the main method
- Methods contain data type declaration and executable statements.

28

A simple java program to print hello world

```
public class Hello
{
    //main method declaration public static void main(String[] args)
    {
        System.out.println("hello world");
    }
}
```

29

❏ `public class Hello` - This creates a class called Hello. We should make sure that the **class name starts with a capital letter**, and the public word means it is accessible from any other classes.

❏ **Braces** - The curly brackets are used to group all the commands together

❏ `public static void main`

- When the main method is declared **public**, it means that it can be used outside of this class as well.
- The word **static** means that we want to access a method without making its objects
- The word **void** indicates that it does not return any value. The main is declared as void because it does not return any value.
- main is a method; this is a starting point of a Java program.

30

❏ `String[] args`

It is an array where each element is a string, which is named as args. If you run the Java code through a console, you can pass the input parameter. The main() takes it as an input.

❏ `System.out.println();`

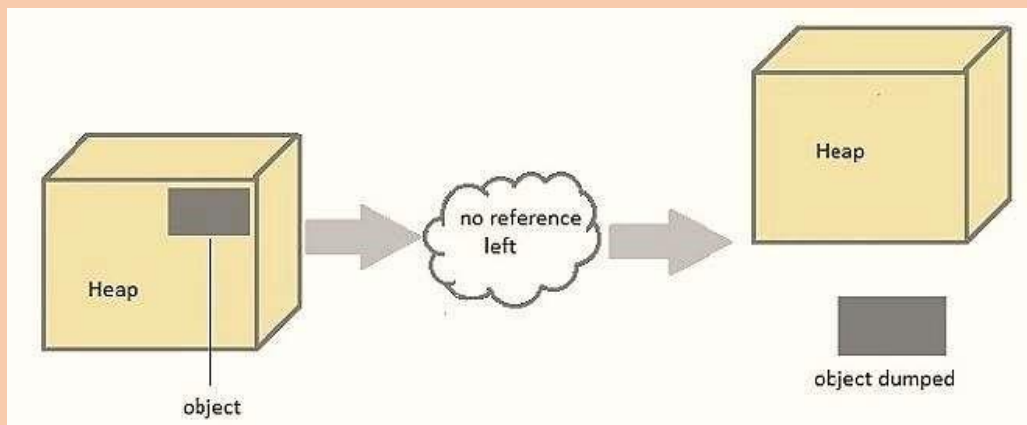
- This statement is used to print text on the screen as output
- system is a predefined class, and out is an object of the PrintWriter class defined in the system
- The method println prints the text on the screen with a new line.
- We can also use print() method instead of println() method. All Java statement ends with a semicolon.

31

Garbage Collection in Java (A process of releasing unused memory)

- When JVM starts up, it creates a heap area which is known as runtime data area. This is where all the objects (instances of class) are stored
- Since this area is limited, it is required to manage this area efficiently by removing the objects that are no longer in use.
- The process of removing unused objects from heap memory is known as Garbage collection and this is a part of memory management in Java.
- Languages like C/C++ don't support automatic garbage collection, however in java, the garbage collection is automatic.

32



33

- In java, **garbage** means **unreferenced objects**.
- Main objective of Garbage Collector is to free heap memory by destroying unreachable objects.

- Unreachable objects : An object is said to be unreachable iff it doesn't contain any reference to it.
- Eligibility for garbage collection : An object is said to be eligible for GC(garbage collection) iff it is unreachable.
- **finalize() method** – This method is invoked each time before the object is garbage collected and it perform cleanup processing.
- The Garbage collector of JVM collects only those objects that are created by new keyword. So if we have created any object without new, we can use finalize method to perform cleanup processing

34

Request for Garbage Collection

- We can request to JVM for garbage collection however, it is upto the JVM when to start the garbage collector.
- Java **gc()** method is used to **call garbage collector explicitly**.
- However gc() method does not guarantee that JVM will perform the garbage collection.
- It only request the JVM for garbage collection. This method is present in System and Runtime class.

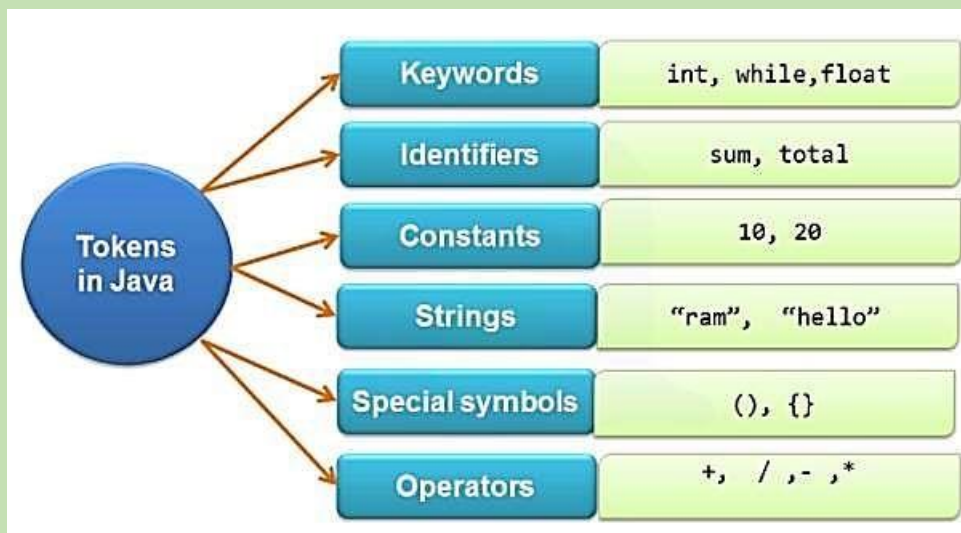
35

Java Lexical Issues (Java Tokens)

TOKENS

- Java Tokens are the smallest individual building block or smallest unit of a Java program
- Java program is a collection of different types of tokens, comments, and white spaces.

36



37

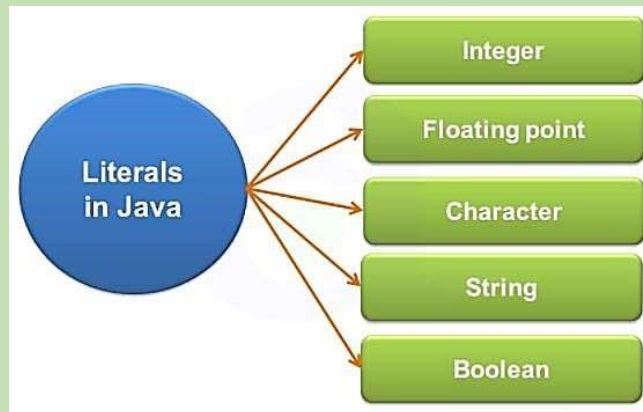
Keywords

- A keyword is a **reserved word**. You cannot use it as a variable name, constant name etc.

- The meaning of the keywords has already been described to the java compiler. These meaning cannot be changed.

❓ Constants or Literals

- Constants are fixed values of a particular type of data, which cannot be modified in a program.
- Java language specifies five major type of literals.



Prepared By EBIN PM, AP, IESCE

39

- Thus, the keywords cannot be used as variable names because that would try to change the existing meaning of the keyword, which is not allowed.

- Java language has reserved **50 words** as keywords

38

❑ Identifiers

- Identifiers are the names of variables, methods, classes, packages and interfaces
- Identifier must follow some rules.

❓ All identifiers must start with either a letter(a to z or A to Z) or currency character(\$) or an underscore.

❓ They must not begin with a digit

❓ After the first character, an identifier can have any combination of characters.

❓ A Java keywords cannot be used as an identifier.

❓ Identifiers in Java are case sensitive, foo and Foo are two different identifiers.

❓ They can be any length

Eg: int a; char name;

40

? String

- In java, string is basically an object that represents sequence of char values.
- An array of characters works same as java string.
Eg: `char[] ch = {'a','t','n','y','l','a'};`
`String s = "atnyla";`
- Java String class provides a lot of methods to perform operations on string such as `compare()`, `concat()`, `equals()`, `split()`, `length()`, `replace()`, `compareTo()`, `intern()`, `substring()` etc.

43

Eg: Integer literal : 100
Floating-point literal : 98.6
Character literal : 's'
String literal : "sample"

? Comments

Comment type	Meaning
// comment	Single-line comments
/* comment */	Multi-line comments
/** documentation */	Documentation comments

42

shift Operator **Brackets[]** : Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts. **Parentheses()** : These special symbols are used to indicate function calls and function parameters.

Braces{ } : These opening and ending curly braces mark the start and end of a block of code containing more than one executable statement.

semicolon ; : It is used to separate more than one statements like in for loop is separates initialization, condition, and increment.

comma , : It is an operator that essentially invokes something called an initialization list. **asterisk *** : It is used for multiplication. **assignment operator =** : It is used to assign values.

Period . : Used to separate package names from subpackages and classes

45

Special symbol

,	<	>	.	_
()	;	\$:
%	[]	#	?
'	&	{	}	"
^	!	*	/	
-	\	~	+	

44

Operators

- An operator is a symbol that takes one or more arguments and operates on them to produce a result.
- Unary Operator
- Arithmetic Operator

- Relational Operator
- Bitwise Operator

❓Whitespace

- Java is a free-form language. This means that you do not need to follow any special indentation rules
- White space in Java is used to separate tokens in the source file. It is also used to improve readability of the source code. **Eg:** `int i = 0;`
- White spaces are required in some places. For example between the `int` keyword and the variable name.
- In java whitespace is a space, tab, or newline

47

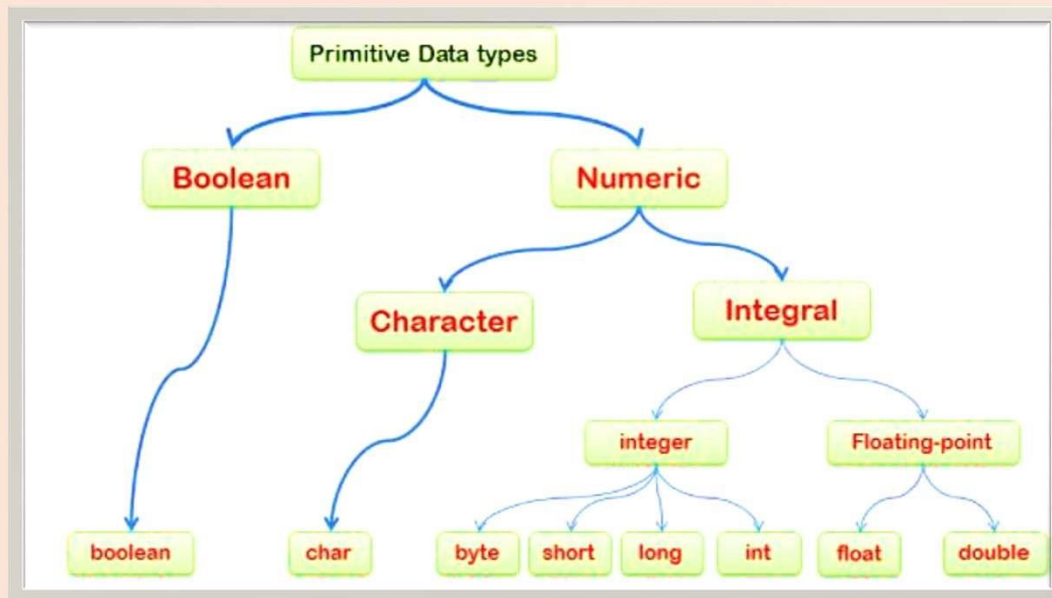
MODULE 2 CORE JAVA FUNDAMENTALS

CHAPTER 1 DATA TYPES, OPERATORS & CONTROL STATEMENTS

1

- Logical Operator
- Ternary Operator
- Assignment Operator

46



3

DATA TYPES

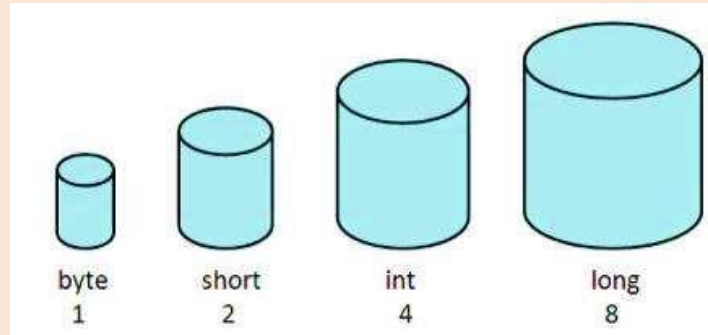
- ❑ Data type defines the values that a variable can take, for example if a variable has int data type, it can only take integer values.
- ❑ Data types specify the different sizes and values that can be stored in the variable.
- ❑ There are two types of data types in Java:

Primitive data types

Non-primitive data types

2

- byte , short , int and long data types are used for storing whole numbers.
- float and double are used for fractional numbers.
- char is used for storing characters(letters).
- boolean data type is used for variables that holds either true or false.



5

Primitive Data Types (Fundamental Data Types)

- Primitive Data Types are predefined and available within the Java language. There are 8 types of primitive data types:

Data Type	Default Value	Default size
byte	0	1 byte
short	0	2 bytes
int	0	4 bytes
long	0L	8 bytes
float	0.0f	4 bytes
double	0.0d	8 bytes
boolean	false	1 bit
char	'\u0000'	2 bytes

4

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

        boolean b = false;
        System.out.println(b);
    }
}
```

Output false

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

        char ch = 'Z';
        System.out.println(ch);
    }
}
```

Output Z

7

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

        byte num;

        num = 113;
        System.out.println(num);
    }
}
```

Output 113

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

        short num;

        num = 150;
        System.out.println(num);
    }
}
```

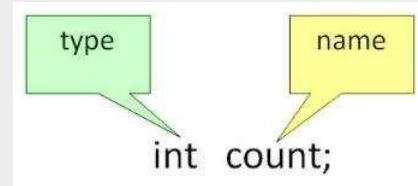
Output 150

6

1 . Variable Declaration

Syntax: data_type variable_name ;

Eg: int a,b,c;
float pi;
double d;



2 . Variable Initialization

Syntax : data_type variable_name = value;

Eg: int a=2,b=4,c=6; int num = 45.66;
float pi = 3.14f;
double val = 20.22d;
char a = 'v';

9

Variables In JAVA

- Variable in Java is a data **container** that stores the data values during Java program execution.
- Variable is a memory location name of the data.
- variable="vary + able" that means its value can be changed.
- In order **to use a variable** in a program we need to perform 2 steps

1. Variable Declaration

2. Variable Initialization

8

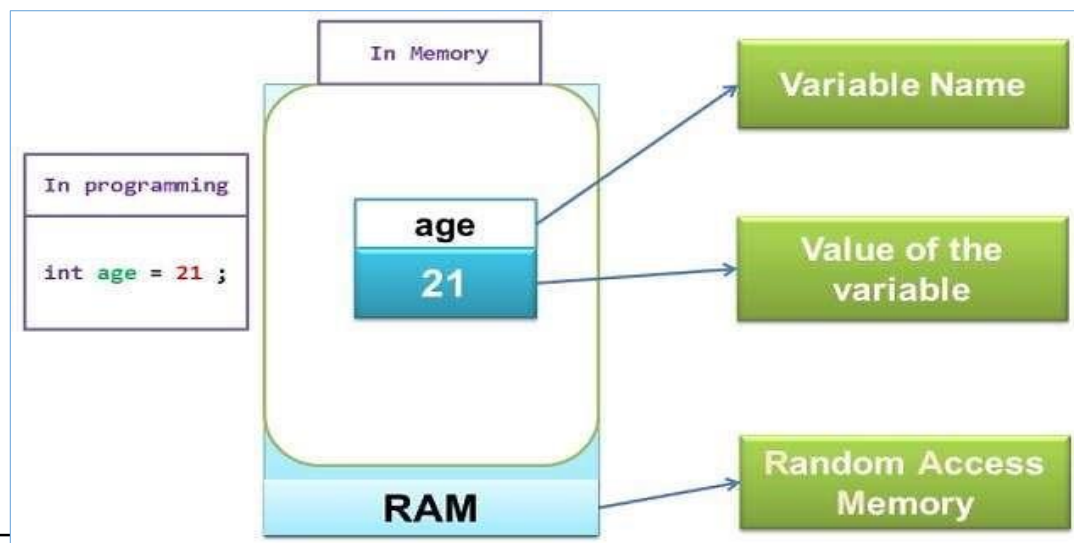
Types of variables

1. Local variables - declared inside the method.
2. Instance Variable - declared inside the class but outside the method.
3. Static variable - declared as with static keyword.

Example:

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

11



10

Java Type Casting or Type Conversion

☐ Type casting is when you assign a value of one primitive data type to another type.

☐ In Java, there are two types of casting:

1. **Widening Casting (automatically)** – converting a smaller type to a larger type size (called **Type Conversion**) `byte -> short -> char -> int -> long -> float -> double`

2. Narrowing Casting (manually) – converting a larger type to a smaller size type (called Type Casting) double -> float -> long -> int -> char -> short -> byte

12

□ Truncation

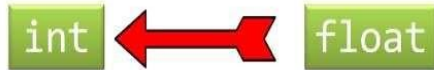
☐ When a floating-point value is assigned to an integer type: truncation takes place, As you know, integers do not have fractional components

☐ Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

☐ For example, if the value 45.12 is assigned to an integer, the resulting value will simply be 45. The 0.12 will have been truncated.

☐ No automatic conversions from the numeric types to char or boolean. Also, char and boolean are not compatible with each other.

14



```
int number;  
float fval= 32.33f;  
number= (int)fval;
```

Type in which you
want to convert

Variable name
Which you want to convert

```
class Casting{  
public static void main(String[] args){  
    int number;  
    float fval= 32.33f;  
    number= (int)fval;  
    System.out.println(number);  
}  
}
```

Output:

```
32  
Press any key to continue . . .
```

15

OPERATORS

❑ An operator is a symbol that tells the computer to perform certain mathematical or logical manipulation.

❑ Java operators can be divided into following categories:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- conditional operator (Ternary)

16

Arithmetic Operators

Operator	Description	Example
+ (Addition)	Adds two operands	5 + 10 = 15
- (Subtraction)	Subtract second operands from first. Also used to Concatenate two strings	10 - 5 = 5
* (Multiplication)	Multiplies values on either side of the operator.	10 * 5 = 50
/ (Division)	Divides left-hand operand by right-hand operand.	10 / 5 = 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	5 % 2 = 1
++ (Increment)	Increases the value of operand by 1.	2++ gives 3
-- (Decrement)	Decreases the value of operand by 1.	3-- gives 2

17

```
class ArithmeticOperations {  
    public static void main (String[] args){  
        int answer = 2 + 2;  
        System.out.println(answer);  
  
        answer = answer - 1;  
        System.out.println(answer);  
  
        answer = answer * 2;  
        System.out.println(answer);  
  
        answer = answer / 2;  
        System.out.println(answer);  
  
        answer = answer + 8;  
        System.out.println(answer);  
  
        answer = answer % 7;  
        System.out.println(answer);  
    }  
}
```

Output

4

3

6

3

11

4

18

```

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

        int x= 5;
        System.out.println(x++);
        System.out.println(++x);
        System.out.println(x--);
        System.out.println(--x);
    }
}

```

Output

```

5
7
7
5
Press any key to continue . . .

```

X++ is **Use –Then - Change**

```

class IncrementDecrementExample{

    public static void main(String args[]){
        int p=10;
        int q=10;
        System.out.println(p++ + ++p);//10+12=22
        System.out.println(q++ + q++);//10+11=21
    }
}

```

Output

```

22
21
Press any key to continue . . .

```

++ x is **Change – Then - Use**

19

Use of Modulus Operator

```

class ModulusOperator {
    public static void main(String args[]) {
        int    R = 42;
        double S = 62.25;

        System.out.println("R mod 10 = " + R % 10);
        System.out.println("S mod 10 = " + S % 10);
    }
}

```

Output

```

R mod 10 = 2
S mod 10 = 2.25
Press any key to continue . . .

```

Joining or Concatenate two strings

```

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

        String firstName = "Rahim";
        String lastName  = "Ramboo";

        String fullName  = firstName + lastName;
        System.out.println(fullName);
    }
}

```

Output

```

RahimRamboo
Press any key to continue . . .

```

20

Relational Operators

Operators	Descriptions	Examples
== (equal to)	This operator checks the value of two operands, if both are equal , then it returns true otherwise false.	(2 == 3) is not true.
!= (not equal to)	This operator checks the value of two operands, if both are not equal , then it returns true otherwise false.	(4 != 5) is true.
> (greater than)	This operator checks the value of two operands, if the left side of the operator is greater , then it returns true otherwise false.	(5 > 56) is not true.
< (less than)	This operator checks the value of two operands if the left side of the operator is less , then it returns true otherwise false.	(2 < 5) is true.
>= (greater than or equal to)	This operator checks the value of two operands if the left side of the operator is greater or equal , then it returns true otherwise false.	(12 >= 45) is not true.
<= (less than or equal to)	This operator checks the value of two operands if the left side of the operator is less or equal , then it returns true otherwise false.	(43 <= 43) is true.

21

```
public class RelationalOperator {

    public static void main(String args[]) {
        int p = 5;
        int q = 10;

        System.out.println("p == q = " + (p == q) );
        System.out.println("p != q = " + (p != q) );
        System.out.println("p > q = " + (p > q) );
        System.out.println("p < q = " + (p < q) );
        System.out.println("q >= p = " + (q >= p) );
        System.out.println("q <= p = " + (q <= p) );
    }
}
```

Output

```
p == q = false
p != q = true
p > q = false
p < q = true
q >= p = true
q <= p = false
Press any key to continue
```

22

Bitwise Operators

Operator	Description
& (bitwise and)	Bitwise AND operator give true result if both operands are true. otherwise, it gives a false result.
(bitwise or)	Bitwise OR operator give true result if any of the operands is true.
^ (bitwise XOR)	Bitwise Exclusive-OR Operator returns a true result if both the operands are different. otherwise, it returns a false result.
~ (bitwise compliment)	Bitwise One's Complement Operator is unary Operator and it gives the result as an opposite bit.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.

23

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

        int A = 10;
        int B = 3;
        int Y;
        Y = A & B;
        System.out.println(Y);

    }
}
```

Output

```
2
Press any key to continue . . .
```

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

        int A = 10;
        int B = 3;
        int Y;
        Y = A | B;
        System.out.println(Y);

    }
}
```

Output

```
11
Press any key to continue . . .
```

24

📌 Logical Operators

Operator	Description	Example
&& (logical and)	If both the operands are non-zero, then the condition becomes true.	(0 && 1) is false
 (logical or)	If any of the two operands are non-zero, then the condition becomes true.	(0 1) is true
! (logical not)	Logical NOT Operator Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(0 && 1) is true

25

```
public class LogicalOperatorDemo {  
    public static void main(String args[]) {  
        boolean b1 = true;  
        boolean b2 = false;  
  
        System.out.println("b1 && b2: " + (b1&&b2));  
        System.out.println("b1 || b2: " + (b1||b2));  
        System.out.println("!(b1 && b2): " + !(b1&&b2));  
    }  
}
```

Output:

```
b1 && b2: false  
b1 || b2: true  
!(b1 && b2): true
```

26

? Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
=	x = 3	x = x 3

27

? conditional Operator / Ternary Operator

(? :)

Expression1 ? Expression2 : Expression3

Expression ? value if true : value if false

```
public class ConditionalOperator {  
    public static void main(String args[]) {  
        int a, b;  
        a = 20;  
        b = (a == 1) ? 10 : 25;  
        System.out.println( "Value of b is : " + b );  
        b = (a == 20) ? 20 : 30;  
        System.out.println( "Value of b is : " + b );  
    }  
}
```

Output

```
Value of b is : 25  
Value of b is : 20  
Press any key to continue . . .
```

28

```

public class TernaryOperatorDemo {

    public static void main(String args[]) {
        int num1, num2;
        num1 = 25;
        /* num1 is not equal to 10 that's why
         * the second value after colon is assigned
         * to the variable num2
         */
        num2 = (num1 == 10) ? 100: 200;
        System.out.println( "num2: "+num2);

        /* num1 is equal to 25 that's why
         * the first value is assigned
         * to the variable num2
         */
        num2 = (num1 == 25) ? 100: 200;
        System.out.println( "num2: "+num2);
    }
}

```

Output:

```

num2: 200
num2: 100

```

29

Operator Precedence

- Evaluate $2 * x - 3 * y$?
 $(2 * x) - (3 * y)$ or $2 * (x - 3 * y)$ which one is correct?????
 - Evaluate $A / B * C$
 $A / (B * C)$ or $(A / B) * C$ Which one is correct?????
- ☐ To answer these questions satisfactorily one has to understand the priority or precedence of operations.

30

Priority	Operators	Description
1st	* / %	multiplication, division, modular division
2nd	+ -	addition, subtraction
3rd	=	assignment

- **Precedence order** - When two operators share an operand the operator with the higher precedence goes first.
- **Associativity** - When an expression has two operators with the same precedence, the expression is evaluated according to its associativity.

31

2 Larger number means higher precedence


Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (type)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right

32

Larger number means higher precedence

10	<<	Bitwise left shift	Left to right
	>>	Bitwise right shift with sign extension	
	>>>	Bitwise right shift with zero extension	
9	<	Relational less than	Left to right
	<=	Relational less than or equal	
	>	Relational greater than	
	>=	Relational greater than or equal	
8	instance of	Type comparison (objects only)	Left to right
	==	Relational is equal to	
7	!=	Relational is not equal to	Left to right
	&	Bitwise AND	
6	^	Bitwise exclusive OR	Left to right
5		Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3		Logical OR	Left to right
2	?:	Ternary conditional	Right to left
1	=	Assignment	Right to left
	+=	Addition assignment	
	-=	Subtraction assignment	
	*=	Multiplication assignment	
	/=	Division assignment	
	%=	Modulus assignment	

33

 Evaluate $i = 2 * 3 / 4 + 4 / 4 + 8 - 2 + 5 / 8$

$i = 6 / 4 + 4 / 4 + 8 - 2 + 5 / 8$ operation: *

$i = 1 + 4 / 4 + 8 - 2 + 5 / 8$ operation: /

$i = 1 + 1 + 8 - 2 + 5 / 8$ operation: /

$i = 1 + 1 + 8 - 2 + 0$ operation: /

$i = 2 + 8 - 2 + 0$ operation: +

$i = 10 - 2 + 0$ operation: +

$i = 8 + 0$ operation: -

$i = 8$ operation: +

34

SELECTION STATEMENTS

☐ Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.

☐ Also called decision making statements

☐ Java supports various selection statements, like **if**, **if-else** and **switch**

☐ There are various **types of if statement** in java.

☐ **if statement**

☐ **if-else statement**

☐ **nested if statement**

☐ **if-else-if ladder**

35

☐ If statement

☐ Use the if statement to specify a block of Java code to be executed if a condition is true.

Syntax

```
if (condition)
{
    // block of code to be executed if the condition is true
}
```

36

Example


```
class SampleIf
{
    public static void main(String args[])
    {
        int a=10;
        if (a > 0) {
            System.out.println("a is greater than 0");
        }
    }
}
```

Output:

a is greater than 0

37

if-else Statement

-  If-else statement also tests the condition. It executes the if block if condition is true otherwise else block is executed.

Syntax

```
if (condition)
{
    // block of code to be executed if the condition is true
}
else
{
    // block of code to be executed if the condition is false
}
```

38

```

class SampleNestedIfElse
{
    public static void main(String args[])
    {
        int a=10,b=20,c=30;
        if (a>b)
        {
            if (a>c)
            {
                System.out.println("a is greatest.");
            }
            else
            {
                System.out.println("c is greatest.");
            }
        }
        else
        {
            if (b>c)
            {
                System.out.println("b is greatest.");
            }
        }
    }
}

```

```

        else
        {
            System.out.println("c is greatest.");
        }
    }
}

```

Output:

c is greatest.

41

```

if (condition) {
    if (condition)
    {
        // block of code to be executed if the condition is true
    }
    else
    {
        // block of code to be executed if the condition is false
    }
}
else
{
    if (condition) {
        // block of code to be executed if the condition is true
    }
    else
    {
        // block of code to be executed if the condition is false
    }
}
}

```

Nested if else Statement
Syntax

40

```

class IfElseIfLadder {
    public static void main(String[] args){
        double score = 55;

        if (score >= 90.0)
            System.out.println('A');
        else if (score >= 80.0)
            System.out.println('B');
        else if (score >= 70.0)
            System.out.println('C');
        else if (score >= 60.0)
            System.out.println('D');
        else
            System.out.println('F');
    }
}

```

Output

```

F
Press any key to continue : ...

```

```

class SampleLadderIfElse
{
    public static void main(String args[])
    {
        int a=10;
        if (a > 0) {
            System.out.println("a is +ve");
        }
        else if (a < 0) {
            System.out.println("a is -ve");
        }
        else {
            System.out.println("a is zero");
        }
    }
}

```

Output:

```

a is +ve

```

if else if ladder

Syntax

```

if (condition) {
    // block of code to be executed if the condition is true
} else if
(condition) {
    // block of code to be executed if the condition is true
}
else {
    // block of code to be executed if the condition is true
}

```

❓ switch case

❓The if statement in java, makes selections based on a single true or false condition. But switch case have multiple choice for selection of the statements

❓It is like if-else-if ladder statement ❓How to Java switch works:

- Matching each expression with case
- Once it match, execute all case from where it matched.
- Use break to exit from switch
- Use default when expression does not match with any case

45

❓ If...Else & Ternary Operator – A comparison

```
int time = 20;
if (time < 18) {
    System.out.println("Good day.");
} else {
    System.out.println("Good evening.");
}
```

```
int time = 20;
String result = (time < 18) ? "Good day." : "Good evening.";
System.out.println(result);
```

44

Syntax

```
switch (expression) {  
  case value1:  
    // statement sequence  
    break;  
  case value2:  
    // statement sequence  
    break;  
  .  
  .  
  .  
  case valueN:  
    // statement sequence  
    break;  
  default:  
    // default statement sequence  
}
```

46

Why **break** is necessary in switch statement ?

- The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement


```

class SampleSwitch
{
    public static void main(String args[])
    {
        int day = 4;
        switch (day) {
            case 1:
                System.out.println("The day is Monday");
                break;
            case 2:
                System.out.println("The day is Tuesday");
                break;
            case 3:
                System.out.println("The day is Wednesday");
                break;
            case 4:
                System.out.println("The day is Thursday");
                break;
            case 5:
                System.out.println("The day is Friday");
                break;

```

```

        case 6:
            System.out.println("The day is Saturday");
            break;
        case 7:
            System.out.println("The day is Sunday");
            break;
        default:
            System.out.println("Please enter between 1 to 7.");
    }
}

```

Output

The day is Thursday

47

- This has the effect of jumping out of the switch.
- The break statement is optional. If you omit the break, execution will continue on into the next case.

48

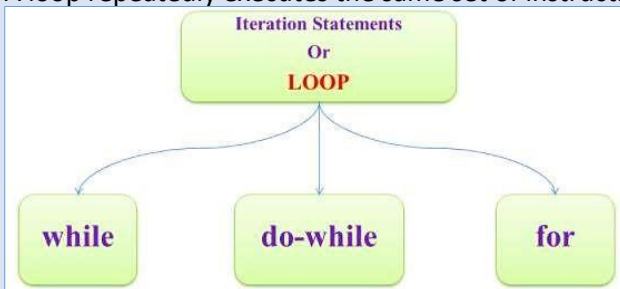
Nested Switch

```
class NestedSwitchCase {  
    public static void main(String args[]) {  
        int count = 1;  
        int target = 1;  
        switch(count) {  
            case 1:  
                switch(target) { // nested switch  
                    case 0:  
                        System.out.println("target is zero inner switch");  
                        break;  
                    case 1: // no conflicts with outer switch  
                        System.out.println("target is one inner switch");  
                        break;  
                }  
                break;  
            case 2:  
                System.out.println("case 2 outer switch");  
                }  
        }  
    }  
}
```

```
target is one inner switch  
Press any key to continue
```

Iteration Statements (Loop)

- A loop can be used to tell a program to execute statements repeatedly
- A loop repeatedly executes the same set of instructions until a termination condition is met.



50

While Loop

❑ In while loop first checks the condition if the condition is true then control goes inside the loop body otherwise goes outside of the body.

Syntax while (condition)

```
{  
    // code block to be executed  
}
```

51

Example - 1

Output

```
class WhileLoopExample
{
    public static void main(String args[])
    {
        int count = 0;
        while(count < 100){
            System.out.println("Welcome to atnyla!");
            count++;
        }
    }
}
```

Welcome to atnyla!
Welcome to atnyla!
Welcome to atnyla!

.....

.....

.....

Welcome to atnyla!
Welcome to atnyla!
Welcome to atnyla!
Press any key to continue . . .

52

Example - 2

Output

```
public class WhileLoopExample {
    public static void main(String[] args) {
        int n=1;
        while(n<=10){
            System.out.println(n);
            n++;
        }
    }
}
```

1
2
3
4
5
6
7
8
9
10

Press any key to continue . . .

53

Example - 3

Output

```
class WhileLoopSingleStatement {  
    public static void main(String[] args){  
        int count = 1;  
        while (count <= 11)  
            System.out.println("Number Count : " + count++);  
    }  
}
```

```
Number Count : 1
Number Count : 2
Number Count : 3
Number Count : 4
Number Count : 5
Number Count : 6
Number Count : 7
Number Count : 8
Number Count : 9
Number Count : 10
Number Count : 11
Press any key to continue
```

54

Example - 4

Output

```
public class WhileInfiniteLoop {
    public static void main(String[] args) {
        while(true){
            System.out.println("infinite while loop");
        }
    }
}
```

[illegible]

55

Example - 5

(Boolean Condition inside while loop)

Output

```
class WhileLoopBoolean {
    public static void main(String[] args){
        boolean a = true;
        int count = 0 ;
        while (a)
        {
            System.out.println("Number Count : " + count);
            count++;
            if(count==5)
                a = false;
        }
    }
}
```

```
Number Count : 0
Number Count : 1
Number Count : 2
Number Count : 3
Number Count : 4
Press any key to continue ...
```

56

? do...while loop

? A do while loop is a control flow statement that executes a block of code at **least once**, and then repeatedly executes the block, or not, depending on a given condition at the end of the block (in while).

Syntax do {

 // code block to be executed

} while (condition);

57

Example -1

Output

```
class DoWhile {  
    public static void main(String args[]) {  
        int n = 0;  
        do {  
            System.out.println("Number " + n);  
            n++;  
        } while(n < 10);  
    }  
}
```

```
Number 0  
Number 1  
Number 2  
Number 3  
Number 4  
Number 5  
Number 6  
Number 7  
Number 8  
Number 9  
Press any key to continue . .
```

58

Example -2 (Infinite do-while Loop)

Output

```
public class InfiniteDoWhileLoop {  
    public static void main(String[] args) {  
        do{  
            System.out.println("infinite do while loop");  
        }while(true);  
    }  
}
```

```
infinite do while loop  
infinite do while loop  
infinite do while loop  
infinite do while loop  
infinite do while loop  
infinite do while loop  
infinite do while loop  
infinite do while loop  
infinite do while loop  
.....  
.....  
.....  
.....  
infinite time it will print like this
```

59

Difference Between while and do-while Loop

BASIS FOR COMPARISON	WHILE	DO-WHILE
General Form	<pre>while (condition) { statements; //body of loop }</pre>	<pre>do{ . statements; // body of loop. . }while(Condition);</pre>
Controlling Condition	In 'while' loop the controlling condition appears at the start of the loop.	In 'do-while' loop the controlling condition appears at the end of the loop.
Iterations	The iterations do not occur if, the condition at the first iteration, appears false.	The iteration occurs at least once even if the condition is false at the first iteration.

60

❓ for loop

❓ For Loop is used to execute set of statements repeatedly until the condition is true.

Syntax **for** (initialization; condition; increment/decrement)

```
{
    // code block to be executed
}
```

Initialization : It executes at once.

Condition : This check until get true.

Increment/Decrement: This is for increment or decrement.

61

Example 1

Output

```
class ForLoopExample {  
    public static void main(String[] args) {  
        for(int i=1;i<=10;i++){  
            System.out.println(i);  
        }  
    }  
}
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
Press any key to continue . . .
```

62

Example 2

Output

```
/*  
Demonstrate the for loop.  
Call this file "ForLoopExample.java".  
*/  
  
class ForLoopExample {  
    public static void main(String[] args) {  
  
        for(int x = 15; x < 25; x = x + 1) {  
            System.out.print("value of x : " + x );  
            System.out.print("\n");  
        }  
    }  
}
```

```
value of x : 15  
value of x : 16  
value of x : 17  
value of x : 18  
value of x : 19  
value of x : 20  
value of x : 21  
value of x : 22  
value of x : 23  
value of x : 24  
Press any key to continue . . .
```

63

For-each or Enhanced For Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

Syntax **for (type variableName :
arrayName)**
{
// code block to be executed
}

64

Example

Output

```
/*  
Demonstrate the for each loop.  
save file "ForEachExample.java".  
*/  
  
public class ForEachExample {  
    public static void main(String[] args) {  
        int array[]={10,11,12,13,14};  
        for(int i:array){  
            System.out.println(i);  
        }  
    }  
}
```

```
10  
11  
12  
13  
14  
Press any key to continue . .
```

65

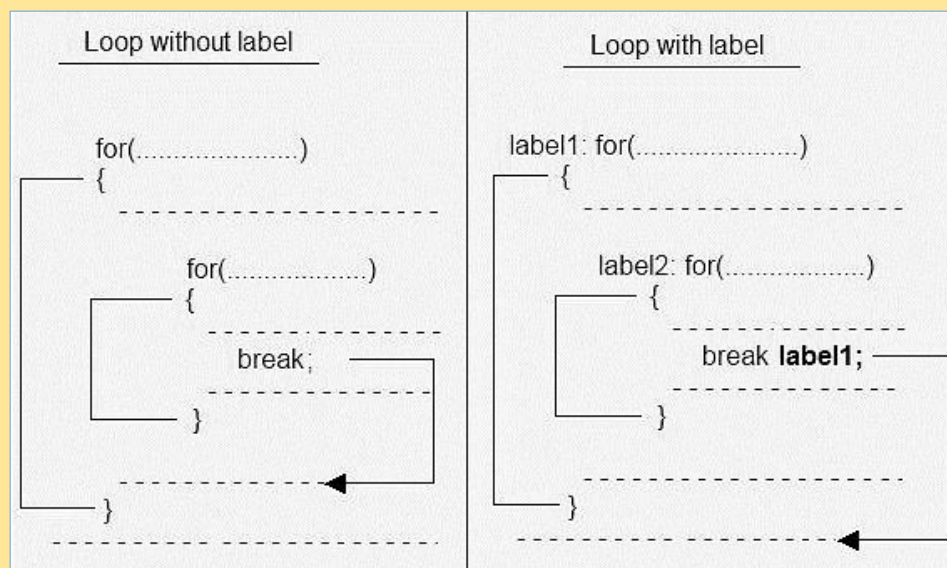
❓ Labeled For Loop

❓ According to nested loop, if we put break statement in inner loop, compiler will jump out from inner loop and continue the outer loop again.

❓ What if we need to jump out from the outer loop using break statement given inside inner loop? The answer is, we should define label along with colon(:) sign before loop. **Syntax** **labelname:**
for(initialization; condition; increment/decrement)

```
{  
    //code to be executed  
}
```

66



67

Example without labelled loop

Output

```
class WithoutLabelledLoop
{
    public static void main(String args[])
    {
        int i,j;
        for(i=1;i<=10;i++)
        {
            System.out.println();
            for(j=1;j<=10;j++)
            {
                System.out.print(j + " ");
                if(j==5)
                    break;           //Statement 1
            }
        }
    }
}
```

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5 Press any key to continue ...
```

68

Example with labelled loop

Output

```
class WithLabelledLoop
{
    public static void main(String args[])
    {
        int i,j;
        loop1: for(i=1;i<=10;i++)
        {
            System.out.println();
            loop2: for(j=1;j<=10;j++)
            {
                System.out.print(j + " ");
                if(j==5)
                    break loop1;    //Statement 1
            }
        }
    }
}
```

```
1 2 3 4 5 Press any key to continue ...
```

69

Jump Statements

❑ Java Break Statement

- ❑ The Java break statement is used to break loop or switch statement
- ❑ It breaks the current flow of the program at specified condition
 - ❑ When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- ❑ In case of inner loop, it breaks only inner loop.

70

Example 1

```
class SampleBreak
{
    public static void main(String args[])
    {
        int num= 1;
        while (num <= 10) {
            System.out.println(num);
            if(num==5)
            {
                break;
            }
            num++;
        }
    }
}
```

Output

```
1
2
3
4
5
```

71

Example 2

```
//Java Program to demonstrate the use of break statement
//inside the for loop.
public class BreakExample {
    public static void main(String[] args) {
        //using for loop
        for(int i=1;i<=10;i++){
            if(i==5){
                //breaking the loop
                break;
            }
            System.out.println(i);
        }
    }
}
```

Output:

```
1
2
3
4
```

72

Example 3

```
//Java Program to illustrate the use of break statement.
//inside an inner loop
public class BreakExample2 {
    public static void main(String[] args) {
        //outer loop
        for(int i=1;i<=3;i++){
            //inner loop
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    //using break statement inside the inner loop
                    break;
                }
                System.out.println(i+" "+j);
            }
        }
    }
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

73

Example 4

```
//Java Program to demonstrate the use of break statement
//inside the Java do-while loop.
public class BreakDoWhileExample {
    public static void main(String[] args) {
        //declaring variable
        int i=1;
        //do-while loop
        do{
            if(i==5){
                //using break statement
                i++;
                break;//it will break the loop
            }
            System.out.println(i);
            i++;
        }while(i<=10);
    }
}
```

Output:

1
2
3
4

74

□ Java Continue Statement

- ☑ The Java continue statement is used to continue the loop
- ☑ The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately
- ☑ It continues the current flow of the program and skips the remaining code at the specified condition.
- ☑ In case of an inner loop, it continues the inner loop only.

75

Example 1

```
//Java Program to demonstrate the use of continue statement
//inside the for loop.
public class ContinueExample {
    public static void main(String[] args) {
        //for loop
        for(int i=1;i<=10;i++){
            if(i==5){
                //using continue statement
                continue;//it will skip the rest statement
            }
            System.out.println(i);
        }
    }
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

76

Example 2

```
//Java Program to illustrate the use of continue statement
//inside an inner loop
public class ContinueExample2 {
    public static void main(String[] args) {
        //outer loop
        for(int i=1;i<=3;i++){
            //inner loop
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    //using continue statement inside inner loop
                    continue;
                }
                System.out.println(i+" "+j);
            }
        }
    }
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

77

Example 3

```
//Java Program to demonstrate the use of continue statement
//Inside the while loop.
public class ContinueWhileExample {
    public static void main(String[] args) {
        //while loop
        int i=1;
        while(i<=10){
            if(i==5){
                //using continue statement
                i++;
                continue;//it will skip the rest statement
            }
            System.out.println(i);
            i++;
        }
    }
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

78

Example 4

```
class myClass {
    public static void main( String args[] ) {
        label:
        for (int i=0;i<6;i++)
        {
            if (i==3)
            {
                continue label; //skips 3
            }
            System.out.println(i);
        }
    }
}
```

Output

```
0
1
2
4
5
```

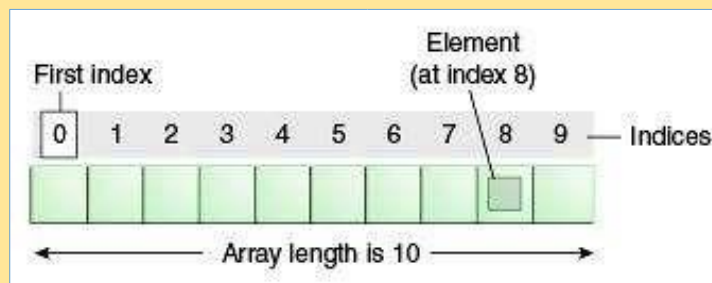
79

ARRAY

- An array is a collection of similar data types.
- Java array is an object which contains elements of a similar data type.
- The elements of an array are stored in a contiguous memory location
- the size of an array is fixed and cannot increase to accommodate more elements
- It is also known as **static data structure** because size of an array must be specified at the time of its declaration.
- Array in Java is index-based, the **first element** of the array is stored at the **0th index**

80

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



Advantage of Java Array

- Code Optimization: It makes the code optimized, we can retrieve or sort the data easily.
- Random access: We can get any data located at any index position.

81

Disadvantage of Java Array

- Size Limit: We can store the only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java. **Features of Array**
- It is always indexed. The index begins from 0.

- It is a collection of similar data types.
- It occupies a contiguous memory location. [Types of Java Array](#)
- [Single Dimensional Array](#)
- [Multidimensional Array](#)

82

Single Dimensional Array in java

Array Declaration

Syntax: `datatype[] arrayname;`

Eg : `int[] arr;`

`char[] name;`

`short[] arr;`

`long[] arr;`

`int[][] arr; //two dimensional array`

[In C program](#) `datatype arrayname[];`

83

? Initialization of Array

`new` operator is used to initializing an array.

Eg 1 : `int[] arr = new int[10];`

or

`int[] arr = {10,20,30,40,50};`

Eg 2: `String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};`

Eg 3: `double[] myList = new double[10];`

84

? Accessing array element

Example: To access 4th element of a given array

```
int[ ] arr = {10,24,30,50};  
System.out.println("Element at 4th place" + arr[3]);
```

? To find the length of an array, we can use the following syntax:
`array_name.length`

Example: `public class MyClass`

```
{  
    public static void main(String[] args)  
    {  
        String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
        System.out.println(cars.length);  
    }  
}
```

Output 4

85

❓ Loop Through an Array

```
public class MyClass
{
    public static void main(String[] args)
    {
        String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
        for (int i = 0; i < cars.length; i++)
        {
            System.out.println(cars[i]);
        }
    }
}
```



Volvo
BMW
Ford
Mazda

86

❓ Loop Through an Array with For-Each

```
public class MyClass
{
    public static void main(String[] args)
    {
        String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
        for (String i : cars)
        {
            System.out.println(i);
        }
    }
}
```



Volvo
BMW
Ford
Mazda

87

```

class ArrayDemo{
    public static void main(String args[]){
        int array[] = new int[7];
        for (int count=0;count<7;count++){
            array[count]=count+1;
        }

        for (int count=0;count<7;count++){
            System.out.println("array["+count+"] = "+array[count]);
        }
    }
}

```

Output

```

array[0] = 1
array[1] = 2
array[2] = 3
array[3] = 4
array[4] = 5
array[5] = 6
array[6] = 7

```

88

```

public class ArrayExample {

    public static void main(String[] args) {
        double[] myList = {3.9, 5.9, 22.4, 31.5};

        // Print all the array elements
        for (int i = 0; i < myList.length; i++) {
            System.out.println(myList[i] + " ");
        }

        // Summing all elements
        double total = 0;
        for (int i = 0; i < myList.length; i++) {
            total += myList[i];
        }
        System.out.println("Total is " + total);

        // Finding the largest element
        double max = myList[0];
        for (int i = 1; i < myList.length; i++) {
            if (myList[i] > max) max = myList[i];
        }
        System.out.println("Max is " + max);
    }
}

```

Output

```

3.9
5.9
22.4
31.5
Total is 63.7
Max is 31.5

```

89

? Two Dimensional array

? Array Declaration

Syntax : datatype[][] arrayname;

Eg: int[][] myNumbers ;

? Array Initialization

```
int[ ][ ] arrName = new int[10][10];
```

Or

```
int[ ][ ] arrName = {{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15}};
```

// 3 by

5 is the size of the array.

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

90

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[ ][ ]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
System.out.print(arr[i][j]+" ");
}
System.out.println();
}
}}
```

Output

```
1 2 3
2 4 5
4 4 5
```

91

```
//Java Program to demonstrate the addition of two matrices in Java
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};

//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}
}}
```

Output

```
2 6 8
6 8 10
```

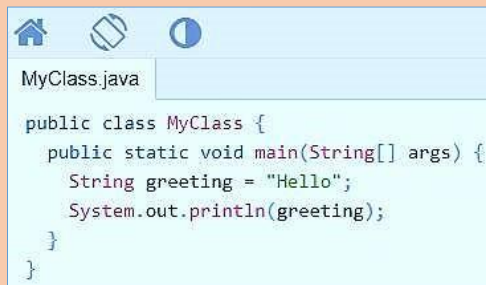
92

STRING

- Strings are used for storing text
- A **String** variable contains a collection of characters surrounded by double quotes

Eg: Create a variable of type String and assign it a value

```
String greeting = "Hello";
```



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

Output

Hello

93

- In Java, **string** is basically an **object** that represents sequence of char values
- An array of characters works same as Java string. For example:

```
char[] ch={'j','o','s','e','p','h'};
```

```
String s= new String(ch); //converting char array to string
```

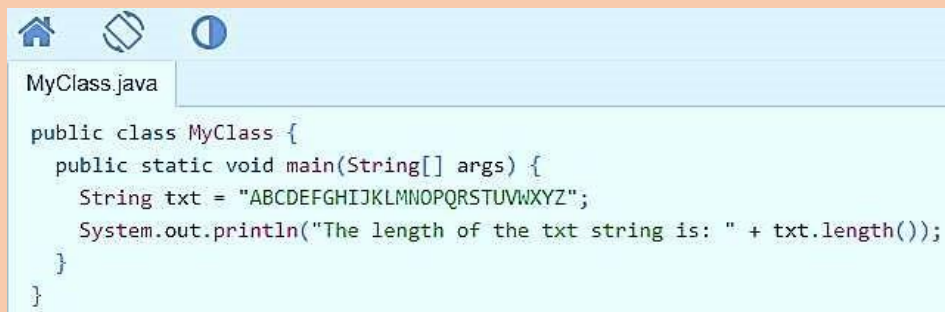
is same as

```
String s="joseph"; //creating string by java string literal
```

94

? String Length

- The length of a string can be found with the **length()** method



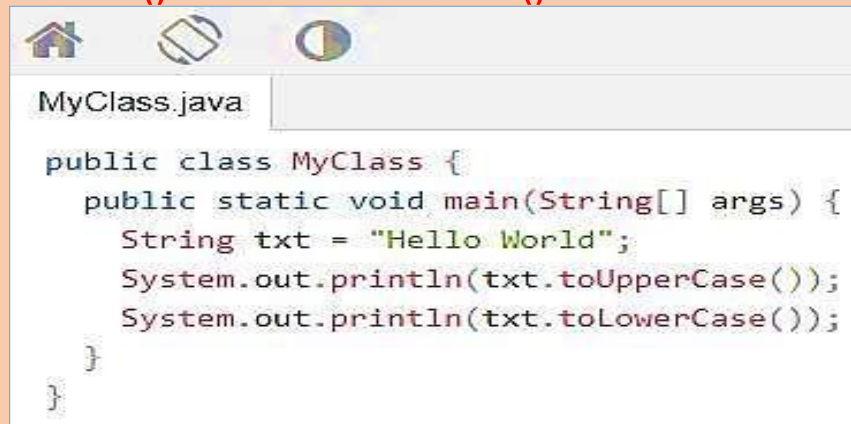
```
MyClass.java  
public class MyClass {  
    public static void main(String[] args) {  
        String txt = "ABCDEFGHJKLMNOPQRSTUVWXYZ";  
        System.out.println("The length of the txt string is: " + txt.length());  
    }  
}
```

Output

The length of the txt string is: 26

95

? toUpperCase() and toLowerCase()



```
MyClass.java  
  
public class MyClass {  
    public static void main(String[] args) {  
        String txt = "Hello World";  
        System.out.println(txt.toUpperCase());  
        System.out.println(txt.toLowerCase());  
    }  
}
```

Output

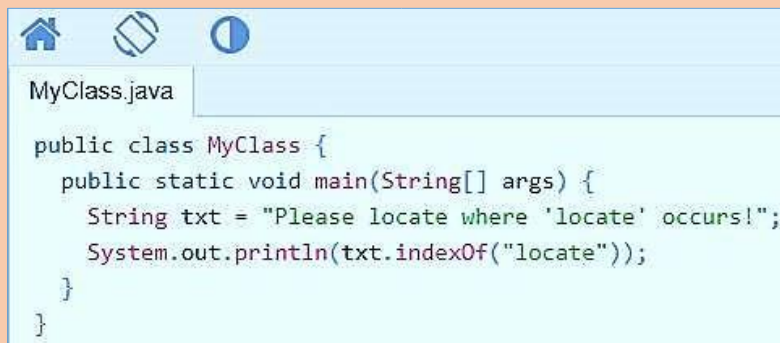
HELLO WORLD

hello world

96

? Finding a Character in a String

? The `indexOf()` method returns the index (the position) of the first occurrence of a specified text in a string (including whitespace)



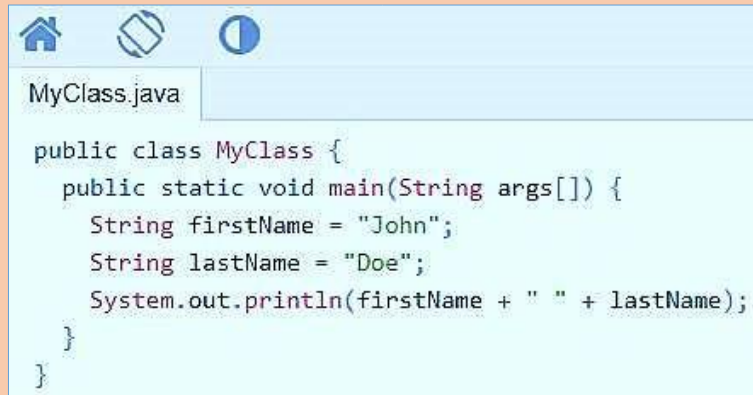
```
MyClass.java  
  
public class MyClass {  
    public static void main(String[] args) {  
        String txt = "Please locate where 'locate' occurs!";  
        System.out.println(txt.indexOf("locate"));  
    }  
}
```

Output 7

97

? String Concatenation

- The `+` operator can be used between strings to combine them. This is called concatenation



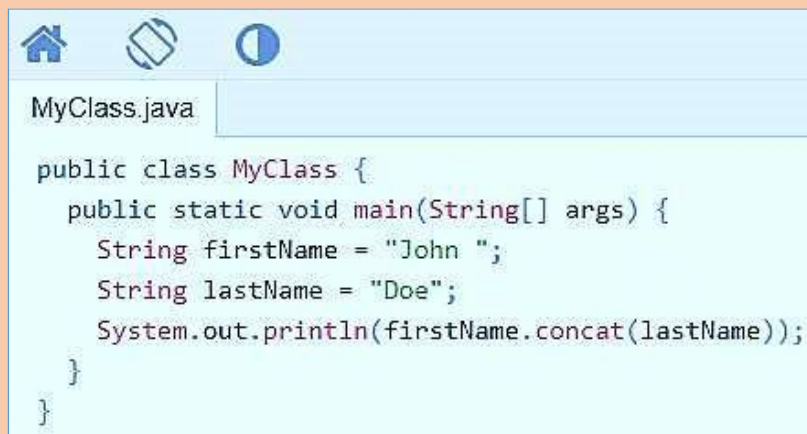
```
MyClass.java  
  
public class MyClass {  
    public static void main(String args[]) {  
        String firstName = "John";  
        String lastName = "Doe";  
        System.out.println(firstName + " " + lastName);  
    }  
}
```

Output John Doe

98

? concat() method

- We can also use the `concat()` method to concatenate two strings:



```
MyClass.java  
  
public class MyClass {  
    public static void main(String[] args) {  
        String firstName = "John ";  
        String lastName = "Doe";  
        System.out.println(firstName.concat(lastName));  
    }  
}
```

Output John Doe

99

❓ Special Characters

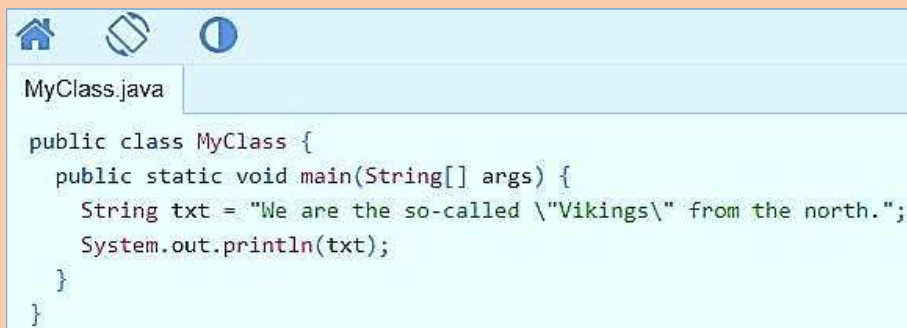
Consider the following example

```
String txt = "We are the so-called "Vikings" from the north.";
```

- Because strings must be written within quotes, Java will misunderstand this string
- The solution to avoid this problem, is to use the backslash escape character

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash

100



```
MyClass.java
public class MyClass {
    public static void main(String[] args) {
        String txt = "We are the so-called \"Vikings\" from the north.";
        System.out.println(txt);
    }
}
```

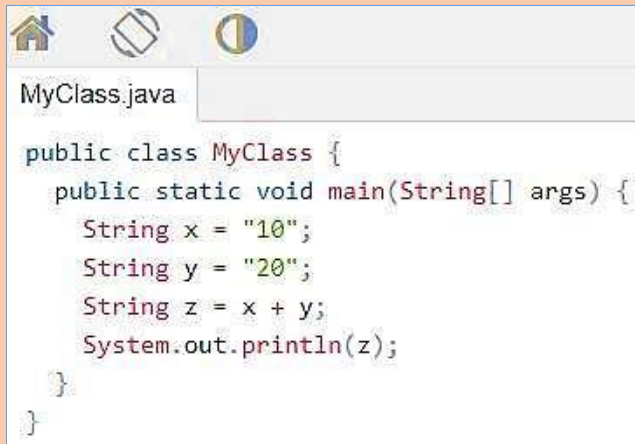
Output We are the so-called "Vikings" from the north.

- ❓ The sequence \" inserts a double quote in a string
- ❓ The sequence \' inserts a single quote in a string
- ❓ The sequence \\ inserts a single backslash in a string

101

❓ Adding Numbers and Strings

- Java uses the + operator for both addition and concatenation.
- If we add two strings, the result will be a string concatenation



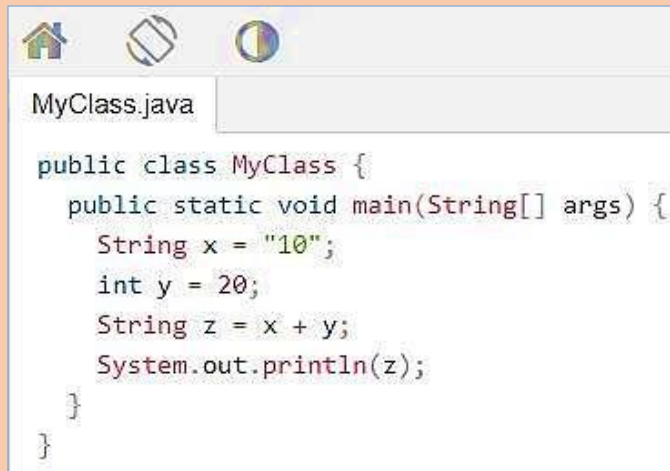
```
MyClass.java
public class MyClass {
    public static void main(String[] args) {
        String x = "10";
        String y = "20";
        String z = x + y;
        System.out.println(z);
    }
}
```

Output

1020

102

- If we add a number and a string, the result will be a string concatenation



```
MyClass.java
public class MyClass {
    public static void main(String[] args) {
        String x = "10";
        int y = 20;
        String z = x + y;
        System.out.println(z);
    }
}
```

Output

1020

103

MODULE 2

1

CHAPTER 2

OBJECT ORIENTED PROGRAMMING IN JAVA

Class Fundamentals

Classes and Objects

- Classes and objects are the two main aspects of object-oriented programming.



- So, a class is a template for objects, and an object is an instance of a class.
- A Class is a "blueprint" for creating objects.

EDULINE

? Create a Class

- To create a class, use the keyword `class`

? Create an Object

- To create an object of MyClass, specify the class name, followed by the object name, and use the keyword `new`

```
public class MyClass {  
    int x = 5;  
}
```

```
public class MyClass {  
    int x = 5;  
  
    public static void main(String[] args) {  
        MyClass myObj = new MyClass();  
        System.out.println(myObj.x);  
    }  
}
```

? Multiple Objects

- You can create multiple objects of one class

```
public class MyClass {  
    int x = 5;  
  
    public static void main(String[] args) {  
        MyClass myObj1 = new MyClass(); // Object 1  
        MyClass myObj2 = new MyClass(); // Object 2  
        System.out.println(myObj1.x);  
        System.out.println(myObj2.x);  
    }  
}
```

? Initialize the object through a reference variable

```
class Student{
    int id;
    String name;
}
class TestStudent2{
    public static void main(String args[]){
        Student s1=new Student();
        s1.id=101;
        s1.name="Sonoo";
        System.out.println(s1.id+" "+s1.name);//printing members with a white space
    }
}
```

Output

101 sonoo

?Using Multiple Classes

- We can also create an object of a class and access it in another class.
- This is often used for better organization of classes
- One class has all the attributes and methods, while the other class holds the main() method (code to be executed).
- Remember that the name of the java file should match the class name.
- In the following example, we have created two files in the same directory/folder:

MyClass.java

OtherClass.java

MyClass.java

```
public class MyClass {  
    int x = 5;  
}
```

OtherClass.java

```
class OtherClass {  
    public static void main(String[] args) {  
        MyClass myObj = new MyClass();  
        System.out.println(myObj.x);  
    }  
}
```

JAVA CLASS ATTRIBUTES

- Class attributes are **variables** within a class

Example

Create a class called "MyClass" with two attributes x and y

```
public class MyClass {  
    int x = 5;  
    int y = 3;  
}
```

- Another term for class attributes is fields / Data members

? Accessing Attributes

- We can access attributes by creating an object of the class, and by using the **dot syntax** (`.`)

Example

Create an object called "myObj" and print the value of x

```
public class MyClass {  
    int x = 5;  
  
    public static void main(String[] args) {  
        MyClass myObj = new MyClass();  
        System.out.println(myObj.x);  
    }  
}
```

Output

5

? Modify Attributes

- We can also modify attribute values

Example

Set the value of x to 40

```
public class MyClass {  
    int x;  
  
    public static void main(String[] args) {  
        MyClass myObj = new MyClass();  
        myObj.x = 40;  
        System.out.println(myObj.x);  
    }  
}
```

Output

40

? Override existing values

Example

Change the value of x to 25

```
public class MyClass {  
    int x = 10;  
  
    public static void main(String[] args) {  
        MyClass myObj = new MyClass();  
        myObj.x = 25; // x is now 25  
        System.out.println(myObj.x);  
    }  
}
```

Output

25

? If you don't want the ability to override existing values, declare the attribute as **final**

```
public class MyClass {  
    final int x = 10;  
  
    public static void main(String[] args) {  
        MyClass myObj = new MyClass();  
        myObj.x = 25; // will generate an error: cannot assign a value to a final variable  
        System.out.println(myObj.x);  
    }  
}
```

? The final keyword is useful when we want a variable to always store the same value, like PI (3.14159...)

? The final keyword is called a "modifier".

? Multiple Objects

? If we create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other

Example - Change the value of x to 25 in myObj2, and leave x in myObj1 unchanged

```
public class MyClass {  
    int x = 5;  
  
    public static void main(String[] args) {  
        MyClass myObj1 = new MyClass(); // Object 1  
        MyClass myObj2 = new MyClass(); // Object 2  
        myObj2.x = 25;  
        System.out.println(myObj1.x); // Outputs 5  
        System.out.println(myObj2.x); // Outputs 25  
    }  
}
```

Multiple Attributes

? We can specify as many attributes as you want

```
public class Person {  
    String fname = "John";  
    String lname = "Doe";  
    int age = 24;  
  
    public static void main(String[] args) {  
        Person myObj = new Person();  
        System.out.println("Name: " + myObj.fname + " " + myObj.lname);  
        System.out.println("Age: " + myObj.age);  
    }  
}
```

JAVA CLASS METHODS

☐ Methods are declared within a class, and that they are used to perform certain actions

Example - Create a method named myMethod() in MyClass

```
public class MyClass {  
    static void myMethod() {  
        System.out.println("Hello World!");  
    }  
}
```

☐ myMethod() prints a text (the action), when it is called.

☐ To call a method, write the method's name followed by two parentheses () and a semicolon;

Example

Inside main, call myMethod()

```
public class MyClass {  
    static void myMethod() {  
        System.out.println("Hello World!");  
    }  
  
    public static void main(String[] args) {  
        myMethod();  
    }  
}  
  
// Outputs "Hello World!"
```

❓ Static vs. Non-Static Methods

❓ Java programs have either static or public attributes and methods.

❓ Static method can be accessed without creating an object of the class

❓ Public methods can only be accessed by objects

Example

The differences between static and public methods

```

public class MyClass {
    // Static method
    static void myStaticMethod() {
        System.out.println("Static methods can be called without creating objects");
    }

    // Public method
    public void myPublicMethod() {
        System.out.println("Public methods must be called by creating objects");
    }

    // Main method
    public static void main(String[] args) {
        myStaticMethod(); // Call the static method
        // myPublicMethod(); This would compile an error

        MyClass myObj = new MyClass(); // Create an object of MyClass
        myObj.myPublicMethod(); // Call the public method on the object
    }
}

```

Access Methods With an Object

```

public class Car {

    // Create a fullThrottle() method
    public void fullThrottle() {
        System.out.println("The car is going as fast as it can!");
    }

    // Create a speed() method and add a parameter
    public void speed(int maxSpeed) {
        System.out.println("Max speed is: " + maxSpeed);
    }

    // Inside main, call the methods on the myCar object
    public static void main(String[] args) {
        Car myCar = new Car(); // Create a myCar object
        myCar.fullThrottle(); // Call the fullThrottle() method
        myCar.speed(200); // Call the speed() method
    }
}

// The car is going as fast as it can!
// Max speed is: 200

```

Remember that..

- The dot (.) is used to access the object's attributes and methods.
- To call a method in Java, write the method name followed by a set of parentheses (), followed by a semicolon (;)

Using Multiple Classes

- It is a good practice to create an object of a class and access it in another class.
- Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory:

Car.java

OtherClass.java

Car.java

```
public class Car {  
    public void fullThrottle() {  
        System.out.println("The car is going as fast as it can!");  
    }  
  
    public void speed(int maxSpeed) {  
        System.out.println("Max speed is: " + maxSpeed);  
    }  
}
```

OtherClass.java

```
class OtherClass {  
    public static void main(String[] args) {  
        Car myCar = new Car();    // Create a myCar object  
        myCar.fullThrottle();      // Call the fullThrottle() method  
        myCar.speed(200);          // Call the speed() method  
    }  
}
```


CONSTRUCTORS

❓ A constructor in Java is a special method that is used to initialize objects.

❓ The constructor is called when an object of a class is created.

❓ It can be used to set initial values for object attributes

Types of Java constructors

Default constructor (no-argument constructor)

Parameterized constructor

Syntax of default constructor:

`<class_name>(){}`

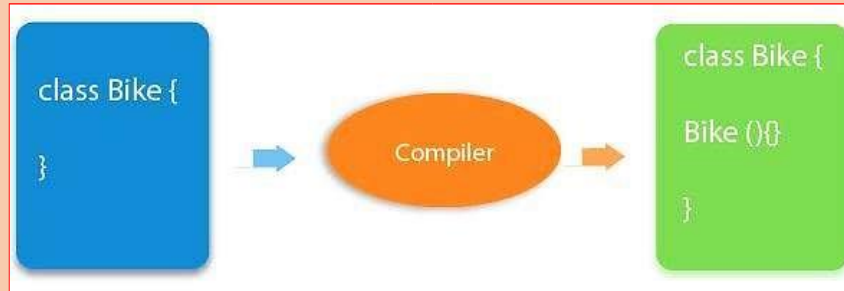
❓ In the following example, we are creating the no-argument constructor in the Bike class. It will be invoked at the time of object creation.

Output

Bike is created

```
//Java Program to create and call a default constructor
class Bike1{
    //creating a default constructor
    Bike1(){System.out.println("Bike is created");}
    //main method
    public static void main(String args[]){
        //calling a default constructor
        Bike1 b=new Bike1();
    }
}
```

- ❓ If there is no constructor in a class, compiler automatically creates a default constructor.



The purpose of a default constructor

- The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

```
// Create a MyClass class
public class MyClass {
    int x; // Create a class attribute

    // Create a class constructor for the MyClass class
    public MyClass() {
        x = 5; // Set the initial value for the class attribute x
    }

    public static void main(String[] args) {
        MyClass myObj = new MyClass(); // Create an object of class MyClass (This will call the constructor)
        System.out.println(myObj.x); // Print the value of x
    }
}

// Outputs 5
```

- ❑ The constructor name must match the class name, and it cannot have a return type (like void).
- ❑ The constructor is called when the object is created.
- ❑ All classes have constructors by default
- ❑ If you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Constructor Parameters (Parameterized constructor)

- ❑ Constructors can also take parameters, which is used to initialize attributes

```
public class MyClass {  
    int x;  
  
    public MyClass(int y) {  
        x = y;  
    }  
  
    public static void main(String[] args) {  
        MyClass myObj = new MyClass(5);  
        System.out.println(myObj.x);  
    }  
}  
  
// Outputs 5
```

Constructor Parameters - We can have as many parameters as you want

```
public class Car {  
    int modelYear;  
    String modelName;  
  
    public Car(int year, String name) {  
        modelYear = year;  
        modelName = name;  
    }  
  
    public static void main(String[] args) {  
        Car myCar = new Car(1969, "Mustang");  
        System.out.println(myCar.modelYear + " " + myCar.modelName);  
    }  
}  
  
// Outputs 1969 Mustang
```

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Method Overloading

- With method overloading, multiple methods can have the same name with different parameters
- Method overloading is one of the ways that Java supports polymorphism.

There are two ways to overload the method in java

By changing number of arguments

By changing the data type

Example

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```

? Advantage of method overloading

- The main advantage of this is cleanliness of code.
- Method overloading increases the readability of the program.
- Flexibility

Example - Consider the following example, which have two methods that add numbers of different type

```
static int plusMethodInt(int x, int y) {  
    return x + y;  
}  
  
static double plusMethodDouble(double x, double y) {  
    return x + y;  
}  
  
public static void main(String[] args) {  
    int myNum1 = plusMethodInt(8, 5);  
    double myNum2 = plusMethodDouble(4.3, 6.26);  
    System.out.println("int: " + myNum1);  
    System.out.println("double: " + myNum2);  
}
```

Instead of defining two methods that should do the same thing, it is better to overload one.

```
static int plusMethod(int x, int y) {  
    return x + y;  
}  
  
static double plusMethod(double x, double y) {  
    return x + y;  
}  
  
public static void main(String[] args) {  
    int myNum1 = plusMethod(8, 5);  
    double myNum2 = plusMethod(4.3, 6.26);  
    System.out.println("int: " + myNum1);  
    System.out.println("double: " + myNum2);  
}
```

RECURSION

- Recursion is the technique of making a **method call itself**.
- This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Syntax

```
returntype methodname(){  
    //code to be executed  
    methodname();//calling same method  
}
```

```
public class MyClass {  
    public static void main(String[] args) {  
        int result = sum(10);  
        System.out.println(result);  
    }  
    public static int sum(int k) {  
        if (k > 0) {  
            return k + sum(k - 1);  
        } else {  
            return 0;  
        }  
    }  
}
```

Working

```
10 + sum(9)  
10 + ( 9 + sum(8) )  
10 + ( 9 + ( 8 + sum(7) ) )  
...  
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)  
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Example

Use recursion to add all of the numbers up to 10.

```

public class RecursionExample3 {
    static int factorial(int n){
        if (n == 1)
            return 1;
        else
            return(n * factorial(n-1));
    }

    public static void main(String[] args) {
        System.out.println("Factorial of 5 is: "+factorial(5));
    }
}

```

Working

```

factorial(5)
  factorial(4)
    factorial(3)
      factorial(2)
        factorial(1)
          return 1
        return 2*1 = 2
      return 3*2 = 6
    return 4*6 = 24
  return 5*24 = 120

```

Example

Factorial of a number

USING OBJECT AS A PARAMETER / ARGUMENT

```

class Operation2{
    int data=50;

    void change(Operation2 op){
        op.data=op.data+100;//changes will be in the instance variable
    }

    public static void main(String args[]){
        Operation2 op=new Operation2();

        System.out.println("before change "+op.data);
        op.change(op);//passing object
        System.out.println("after change "+op.data);
    }
}

```

```

Output: before change 50
       after change 150

```


THIS KEYWORD

There can be a lot of usage of java this keyword. In java, **this** is a **reference variable** that refers to the current object. **Usage of java this keyword**

- this can be used to **refer current class instance variable**.
- this can be used to **invoke current class method** (implicitly)
- **this()** can be used to **invoke current class constructor**.
- this can be passed as an argument in the method call.
- this can be passed as argument in the constructor call.

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        rollno=rollno;
        name=name;
        fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}
class TestThis1{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }}
}
```

Output

```
0 null 0.0
0 null 0.0
```

Understanding the problem

without this keyword

```

class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        this.rollno=rollno;
        this.name=name;
        this.fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
    public static void main(String args[]){
        Student s1=new Student(111,"ankit",5000f);
        Student s2=new Student(112,"sumit",6000f);
        s1.display();
        s2.display();
    }}

```

Output

```

111 ankit 5000
112 sumit 6000

```

Solution of the problem

with this keyword

- It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

🔍 this: to invoke current class method

- You may invoke the method of the current class by using the this keyword.
- If you don't use the this keyword, compiler automatically adds this keyword while invoking the method

```
class A{
    void m(){System.out.println("hello m");}
    void n(){
        System.out.println("hello n");
        //m();//same as this.m()
        this.m();
    }
}

class TestThis4{
    public static void main(String args[]){
        A a=new A();
        a.n();
    }}

```

Output

```
hello n
hello m

```

JAVA INNER CLASS

- ❑ In Java, it is also possible to **nest classes** (a class within a class).
- ❑ The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable.
- ❑ To access the inner class, create an object of the outer class, and then create an object of the inner class

```

class OuterClass {
    int x = 10;

    class InnerClass {
        int y = 5;
    }
}

public class MyMainClass {
    public static void main(String[] args) {
        OuterClass myOuter = new OuterClass();
        OuterClass.InnerClass myInner = myOuter.new InnerClass();
        System.out.println(myInner.y + myOuter.x);
    }
}

// Outputs 15 (5 + 10)

```

Access Outer Class From Inner Class

```

class OuterClass {
    int x = 10;

    class InnerClass {
        public int myInnerMethod() {
            return x;
        }
    }
}

public class MyMainClass {
    public static void main(String[] args) {
        OuterClass myOuter = new OuterClass();
        OuterClass.InnerClass myInner = myOuter.new InnerClass();
        System.out.println(myInner.myInnerMethod());
    }
}

// Outputs 10

```

One advantage of inner classes, is that they can access attributes and methods of the outer class

Command-Line Arguments

- Sometimes we want to pass information into a program when we run it. This is accomplished by passing command-line arguments to `main()`.
- The `main` method can receive string arguments from the command line
- To access the command-line arguments inside a Java program is quite easy— they are stored as strings in a `String` array passed to the `args` parameter of `main()`.
- The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on.

```
// Display all command-line arguments.
class CmdLine {
public static void main(String args[]) {
    for(int i=0; i<args.length; i++)
        System.out.println("args[" + i + "]: " +args[i]);
    }
}
```

```
C:\Users\Hello World\Desktop\JAVA>javac CmdLine.java
```

```
C:\Users\Hello World\Desktop\JAVA>java CmdLine This is Commend-Line Argument Example
```

```
args[0]: This
```

```
args[1]: is
```

```
args[2]: Commend-Line
```

```
args[3]: Argument
```

```
args[4]: Example
```

```
C:\Users\Hello World\Desktop\JAVA>
```

MODULE - 2

CHAPTER – 3 INHERITANCE

INHERITANCE IN JAVA

- Inheritance in Java is a mechanism in which **one object acquires all the properties and behaviors of a parent object.**
- The **idea** behind inheritance in Java is that **you can create new classes that are built upon existing classes.**
- When you inherit from an existing class, **you can reuse methods and attributes of the parent class.** Moreover, you can add new methods and attributes in your current class also
- Inheritance represents the **IS-A relationship** which is also known as a **parent-child** relationship.

📖Terms used in Inheritance

Class: A class is a template or blueprint from which objects are created.

Sub Class/Child Class: Subclass is a class which inherits the other class. It is also called a **derived class**, **extended class**, or **child class**.

Super Class/Parent Class: Superclass is the class from where a subclass inherits the features. It is also called a **base class** or a **parent class**.

Reusability: As the name specifies, reusability is a mechanism which facilitates you to **reuse the attributes and methods of the existing class when you create a new class**. We can use the same attributes and methods already defined in the previous class.

3

📖Access Modifiers - There are four types of Java access modifiers:

Private: The access level of a private modifier is **only within the class**. It cannot be accessed from outside the class.

Default: The access level of a default modifier is **only within the package**. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Protected: The access level of a protected modifier is **within the package and outside the package through child class**. If you do not make the child class, it cannot be accessed from outside the package.

Public: The access level of a public modifier is **everywhere**. It can be accessed from within the class, outside the class, within the package and outside the package.

4

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

5

❏The syntax of Java Inheritance

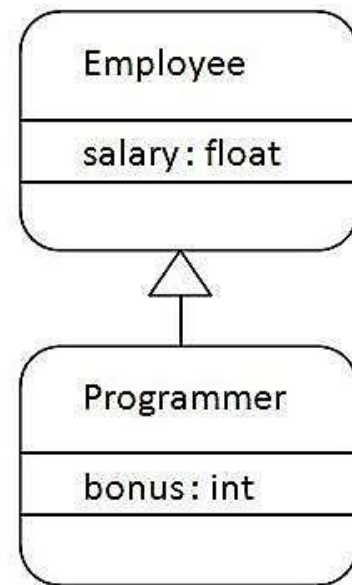
```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

❑ The extends keyword indicates that you are making a new class that derives from an existing class.

❑ The meaning of "extends" is to increase the functionality.

❑ In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

- Programmer is the subclass (child class)
- Employee is the superclass (Parent class)
- The relationship between the two classes is Programmer IS-A Employee
- It means that Programmer is a type of Employee.




```

class Employee{
    float salary=40000;
}
class Programmer extends Employee{
    int bonus=10000;
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}

```

Output

```

Programmer salary is:40000.0
Bonus of programmer is:10000

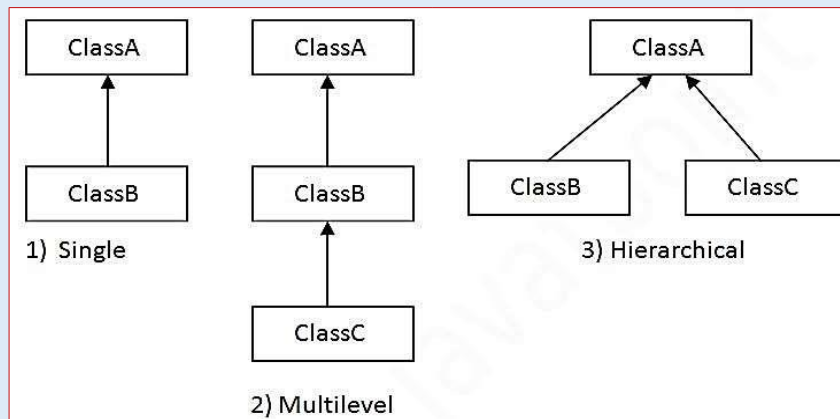
```

- Programmer object can access the attribute of its own class as well as of Employee class i.e. code reusability.

8

Types of inheritance in java

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



9

```

class Vehicle {
    protected String brand = "Ford";           // Vehicle attribute
    public void honk() {                       // Vehicle method
        System.out.println("Tuut, tuut!");
    }
}

class Car extends Vehicle {
    private String modelName = "Mustang";      // Car attribute
    public static void main(String[] args) {

        // Create a myCar object
        Car myCar = new Car();

        // Call the honk() method (from the Vehicle class) on the myCar object
        myCar.honk();

        // Display the value of the brand attribute (from the Vehicle class) and the value of the modelName
        System.out.println(myCar.brand + " " + myCar.modelName);
    }
}

```

SUPER KEYWORD

☐ The super keyword in Java is a **reference variable** which is used to **refer immediate parent class object**. **Usage of Java super Keyword**

- super can be used to refer immediate parent class instance variable.
- super can be used to invoke immediate parent class method.
- super() can be used to invoke immediate parent class constructor.

☐ We can use super keyword to access the data member (attribute) of parent class. It is used if parent class and child class have same attribute.

```

class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}

```

output

```

black
white

```

Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

12

```

class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();
}}

```

Output

```

eating...
barking...

```

The super keyword can also be used to invoke(call) parent class method.

- ❑ In the above example Animal and Dog both classes have eat() method
- ❑ If we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.
- ❑ To call the parent class method, we need to use **super** keyword.

14

```
class Animal{
    Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
    Dog(){
        super();
        System.out.println("dog is created");
    }
}
class TestSuper3{
    public static void main(String args[]){
        Dog d=new Dog();
    }
}
```

The super keyword can also be used to invoke the parent class constructor .

Output

```
animal is created
dog is created
```

15

Calling order of constructors in inheritance

- ❑ Order of execution of constructors in inheritance relationship is from base (parent) class to derived (child) class.
- ❑ We know that when we create an object of a class then the constructors get called automatically.
- ❑ In inheritance relationship, when we create an object of a child class, then first base class constructor and then derived class constructor get called implicitly.
- ❑ In simple word, we can say that the parent class constructor get called first, then of the child class constructor.

16

```
class A {  
    A() {  
        System.out.println("Inside A's constructor.");  
    }  
}  
// Create a subclass by extending class A.  
class B extends A {  
    B() {  
        System.out.println("Inside B's constructor.");  
    }  
}  
// Create another subclass by extending B.  
class C extends B {  
    C() {  
        System.out.println("Inside C's constructor.");  
    }  
}  
public class Main {  
    public static void main(String args[])  
    {  
        C c = new C();  
    }  
}
```

Output

**Inside A's constructor.
Inside B's constructor.
Inside C's constructor.**

17

METHOD OVERRIDING

☐ If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.

☐ In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for **runtime polymorphism**

Rules for Java Method Overriding

- The method must have the same name as in the parent class
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance). **Remember.....**
- **A static method cannot be overridden.** It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.
- **Can we override java main method?** - No, because the main is a static method.

Example - method overriding

```
//Java Program to illustrate the use of Java Method Overriding
//Creating a parent class.
class Vehicle{
    //defining a method
    void run(){System.out.println("Vehicle is running");}
}
//Creating a child class
class Bike2 extends Vehicle{
    //defining the same method as in the parent class
    void run(){System.out.println("Bike is running safely");}

    public static void main(String args[]){
        Bike2 obj = new Bike2();//creating object
        obj.run();//calling method
    }
}
```

Output

```
Bike is running safely
```

we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

20

method overloading Vs. method overriding

No.	Method Overloading	Method Overriding
1)	Method overloading is used to <i>increase the readability</i> of the program.	Method overriding is used to <i>provide the specific implementation</i> of the method that is already provided by its super class.
2)	Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
3)	In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
4)	Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
5)	In java, method overloading can't be performed by changing return type of the method only. <i>Return type can be same or different</i> in method overloading. But you must have to change the parameter.	<i>Return type must be same or covariant</i> in method overriding.

FINAL KEYWORD

❏ The final keyword in java is used to **restrict** the user. The java final keyword can be used in many context. Final can be:

- 1 . **variable**
- 2 . **method**
- 3 . **class**

❏ Java final variable

- If you make any variable as final, you **cannot change the value of final variable(It will be constant)**

22

```
class Bike9{  
    final int speedlimit=90;//final variable  
    void run(){  
        speedlimit=400;  
    }  
    public static void main(String args[]){  
        Bike9 obj=new Bike9();  
        obj.run();  
    }  
} //end of class
```

Output:Compile Time Error

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

❓ Java final method

❓ If we make any method as final, we cannot override it

```
class Bike{  
    final void run(){System.out.println("running");}  
}  
  
class Honda extends Bike{  
    void run(){System.out.println("running safely with 100kmph");}  
  
    public static void main(String args[]){  
        Honda honda= new Honda();  
        honda.run();  
    }  
}
```

Output:Compile Time Error

24

❓ Java final class

❓ If we make any class as final, we cannot extend it.

```
final class Bike{  
  
}  
  
class Honda1 extends Bike{  
    void run(){System.out.println("running safely with 100kmph");}  
  
    public static void main(String args[]){  
        Honda1 honda= new Honda1();  
        honda.run();  
    }  
}
```

Output:Compile Time Error

Is final method inherited?

☐ Yes, final method is inherited but you cannot override it. For

Example:

```
class Bike{  
    final void run(){System.out.println("running...");}  
}  
class Honda2 extends Bike{  
    public static void main(String args[]){  
        new Honda2().run();  
    }  
}
```

Output:running...

26

☐ Points to Remember

- 1) A constructor cannot be declared as final.
- 2) Local final variable must be initializing during declaration.
- 3) We cannot change the value of a final variable.
- 4) A final method cannot be overridden.
- 5) A final class not be inherited.
- 6) If method parameters are declared final then the value of these parameters cannot be changed.
- 7) **final**, **finally** and **finalize** are three different terms. **finally** is used in exception handling and **finalize** is a method that is called by JVM during garbage collection.

7

ABSTRACT CLASSES AND METHODS

- Data abstraction is the process of **hiding certain details** and **showing only essential information** to the user.
- **Abstract class**: is a restricted class **that cannot be used to create objects** (to access it, it must be inherited from another class).
- **Abstract method**: can **only be used in an abstract class**, and it **does not have a body**. The body is provided by the subclass (inherited from).
- An abstract class can have both abstract and regular methods:

8

Abstract class

Rules for Java Abstract class



```

abstract class Animal {
    public abstract void animalSound();
    public void sleep() {
        System.out.println("Zzz");
    }
}

```

- From the example above, it is not possible to create an object of the Animal class

`Animal myObj = new Animal();` // will generate an error

- To access the abstract class, it must be inherited from another class

30

Example

```

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

// Subclass (inherit from Animal)
class Pig extends Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
}

class MyMainClass {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}

```

Example - Here Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

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

Output

```
running safely
```

32

THE OBJECT CLASS

- The Object class is the **parent class of all the classes in java** by default. In other words, **it is the topmost class of java**.
- The Object class **provides some common behaviors to all the objects** such as object can be compared, object can be cloned, object can be notified etc.
- Object class is present in **java.lang package**
- Every class in Java is directly or indirectly derived from the Object class

33

Methods of Object class

Method	Description
<code>public final Class getClass()</code>	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
<code>public int hashCode()</code>	returns the hashcode number for this object.
<code>public boolean equals(Object obj)</code>	compares the given object to this object.
<code>protected Object clone() throws CloneNotSupportedException</code>	creates and returns the exact copy (clone) of this object.
<code>public String toString()</code>	returns the string representation of this object.
<code>public final void notify()</code>	wakes up single thread, waiting on this object's monitor.
<code>public final void notifyAll()</code>	wakes up all the threads, waiting on this object's monitor.
<code>public final void wait(long timeout) throws InterruptedException</code>	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes <code>notify()</code> or <code>notifyAll()</code> method).

34

MODULE 3

CHAPTER 1

PACKAGES INTERFACES & EXCEPTION HANDLING

1

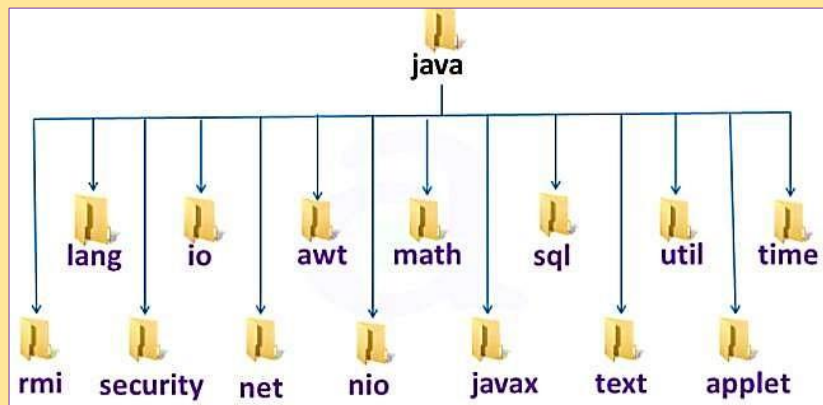
PACKAGES

- A package in Java is used to **group related classes and interfaces**
- Think of it as a folder in a file directory.
- We use packages to **avoid name conflicts**, and to write a better maintainable code
- Packages in Java is a mechanism to **encapsulate** a group of classes, interfaces and sub packages which is used to **providing access protection**
- Package in Java can be categorized in two form, **built-in package user-defined package**

2

❑ **Built-in Package:-** Existing Java package. for example, java.io.*, java.lang , java.util etc.

❑ **User-defined-package:-** Java package created by user to categorized classes and interface

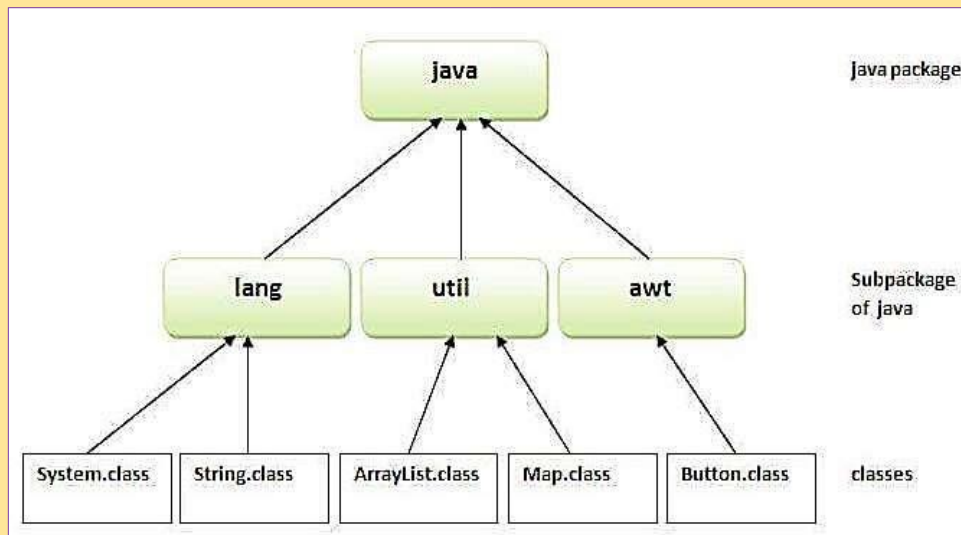


3

❑ **Advantage of Java Package**

- 1) Java package is used to **categorize the classes and interfaces** so that they can be easily maintained.
- 2) Java package **provides access protection**.
- 3) In real life situation there may arise scenarios where we need to define files of the same name. This may lead to name-space collisions. Java package **removes naming collision**.
- 4) **Reusability**: Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.
- 5) Easy to locate the files.

4



❓ To use a class or a package from the library, we need to use the

import keyword:

Syntax:

```
import package.name.Class;    // Import a single class
import package.name.*;       // Import the whole package
```

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

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

6

Access Packages from another package

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

```
import package.*; import
package.classname; fully qualified
name
```

1. Using packagename.*

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

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

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

8

2 . Using packagename.classname

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

- Example



Output:Hello

```
//save by A.java
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.A;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

3. Using fully qualified name

- If we use fully qualified name then only declared class of this package will be accessible.
- Now there is **no need to import**. But you need to use fully qualified name every time when you are accessing the class or interface.
- It is generally used when **two packages have same class name**

e.g. `java.util` and `java.sql` packages contain `Date` class.

- Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}

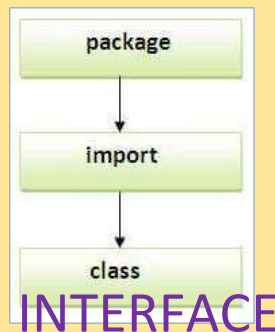
//save by B.java
package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A();//using fully qualified name
        obj.msg();
    }
}
```

Output:Hello

☐ If we import a package, subpackages will not be imported.

☒ If we import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages.

☐ Hence, you need to import the subpackage as well **Note:** Sequence of the program must be **package** then **import** then **class**.



- An interface in Java is a **blueprint of a class**. It has static constants and abstract methods.

- The interface in Java is a mechanism to **achieve abstraction**. There can be only abstract methods in the Java interface, not method body. It is used to **achieve abstraction and multiple inheritance** in Java.
- In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

- Like abstract classes, interfaces cannot be used to create objects
- Interface methods do not have a body - the body is provided by the "**implement**" class
- On implementation of an interface, you must override all of its methods
- **Interface methods** are by default **abstract and public**
- **Interface attributes** are by default **public, static and final**
- An interface cannot contain a constructor (as it cannot be used to create objects)

□ Declare an interface

❓ An interface is declared by using the **interface** keyword.

❓ It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.

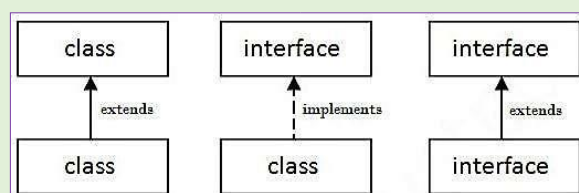
❓ A class that implements an interface must implement all the methods declared in the interface.

```
// interface
interface Animal {
    public void animalSound(); // interface method (does not have a body)
```

❓ To access the interface methods, the interface must be "implemented" by another class with the **implements** keyword (instead of extends).

❓ The body of the interface method is provided by the "implement" class

❓ The relationship between classes and interfaces



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

16

```
// Interface
interface Animal {
    public void animalSound(); // interface method (does not have a body)
    public void sleep(); // interface method (does not have a body)
}

// Pig "implements" the Animal interface
class Pig implements Animal {
    public void animalSound() {
        // The body of animalSound() is provided here
        System.out.println("The pig says: wee wee");
    }
    public void sleep() {
        // The body of sleep() is provided here
        System.out.println("Zzz");
    }
}

class MyMainClass {
    public static void main(String[] args) {
        Pig myPig = new Pig(); // Create a Pig object
        myPig.animalSound();
        myPig.sleep();
    }
}
```

Output

The pig says: wee wee
Zzz

Why And When To Use Interfaces

1) To achieve security - hide certain details and only show the important details of an object (interface).

2) Java does not support "multiple inheritance". However, it can be achieved with interfaces, because the class can implement multiple interfaces.

Note: To implement multiple interfaces, separate them with a comma (see example below).

18

```

interface FirstInterface {
    public void myMethod(); // interface method
}

interface SecondInterface {
    public void myOtherMethod(); // interface method
}

class DemoClass implements FirstInterface, SecondInterface {
    public void myMethod() {
        System.out.println("Some text..");
    }
    public void myOtherMethod() {
        System.out.println("Some other text...");
    }
}

class MyMainClass {
    public static void main(String[] args) {
        DemoClass myObj = new DemoClass();
        myObj.myMethod();
        myObj.myOtherMethod();
    }
}

```

Output

```

Some text...
Some other text...

```

19

EXCEPTION HANDLING

- Exception is an **abnormal condition**.
- In Java, an exception is an event that **disrupts the normal flow** of the program. It is an object which is thrown at runtime.
- Exception Handling is a mechanism to **handle runtime errors** such as **ClassNotFoundException**, **IOException**, **SQLException**, **RemoteException**, etc.
- The core advantage of exception handling is to maintain the normal flow of the application.
- An exception normally disrupts the normal flow of the application that is why we use exception handling.

20

Let's take a scenario:

- Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed.
- If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

```
statement 1;  
statement 2;  
statement 3;  
statement 4;  
statement 5;//exception occurs  
statement 6;  
statement 7;  
statement 8;  
statement 9;  
statement 10;
```

21

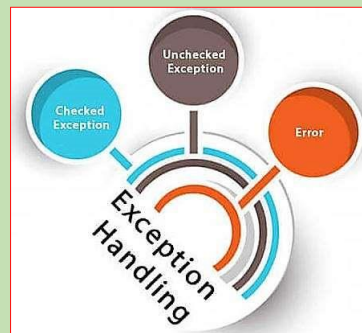
? Types of Java Exceptions

- There are mainly two types of exceptions: checked and unchecked.
- Here, an **error** is considered as the **unchecked exception**.
- According to Oracle, there are three types of exceptions:

Checked Exception

Unchecked Exception

Error



22

?Checked Exception

- The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions
- e.g. IOException, SQLException etc.
- Checked exceptions are **checked at compile-time**.

❓ Unchecked Exception

- The classes which inherit RuntimeException are known as unchecked exceptions
- e.g. ArithmeticException, NullPointerException,
- Unchecked exceptions are not checked at compile-time, but they are **checked at runtime**

23

❓ Error

- Error is **irrecoverable**
- e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

❓ Java Exception Keywords

Keyword	Description
try	The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

24

❓ Common Scenarios of Java Exceptions

- ❓ A scenario where ArithmeticException occurs

- If we divide any number by zero, there occurs an ArithmeticException. `int a=50/0;`
`//ArithmeticException`

❗ A scenario where NullPointerException occurs

- If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;  
System.out.println(s.length()); //NullPointerException
```

25

❗ A scenario where NumberFormatException occurs

- The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException. `String s="abc"; int i=Integer.parseInt(s);`
`//NumberFormatException`

❗ A scenario where ArrayIndexOutOfBoundsException occurs

- If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below: `int a[]=new int[5];`

```
a[10]=50; //ArrayIndexOutOfBoundsException
```

26

TRY & CATCH

- ❑ The try statement allows you to define a block of code to be tested for errors while it is being executed.
- ❑ The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.
- ❑ The try and catch keywords come in pairs

Syntax

```
try {  
    // Block of code to try  
}  
catch(Exception e) {  
    // Block of code to handle errors  
}
```

27

Consider the following example

```
public class MyClass {  
    public static void main(String[] args) {  
        int[] myNumbers = {1, 2, 3};  
        System.out.println(myNumbers[10]); // error!  
    }  
}
```

This will generate an error, because myNumbers[10] does not exist.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10  
at MyClass.main(MyClass.java:4)
```

- ❑ If an error occurs, we can use **try...catch** to catch the error and execute some code to handle it

28

```

public class MyClass {
    public static void main(String[] args) {
        try {
            int[] myNumbers = {1, 2, 3};
            System.out.println(myNumbers[10]);
        } catch (Exception e) {
            System.out.println("Something went wrong.");
        }
    }
}

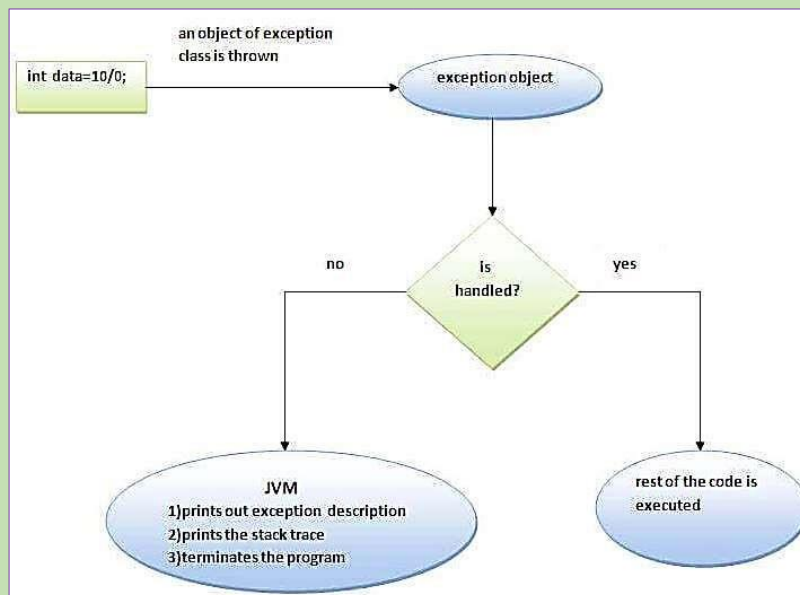
```

Output

Something went wrong.

29

Internal working of java try-catch block



30

❓The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

❓But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

31

Multi-catch block

❓A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

❓At a time only one exception occurs and at a time only one catch block is executed.

❓All catch blocks must be ordered from most specific to most general, i.e. catch for `ArithmeticException` must come before catch for `Exception`.

32

```

public class MultipleCatchBlock1 {

    public static void main(String[] args) {

        try{
            int a[]=new int[5];
            a[5]=30/0;
        }
        catch(ArithmeticException e)
        {
            System.out.println("Arithmetic Exception occurs");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("ArrayIndexOutOfBoundsException Exception occurs");
        }
        catch(Exception e)
        {
            System.out.println("Parent Exception occurs");
        }
        System.out.println("rest of the code");
    }
}

```

Output

Arithmetic Exception occurs
rest of the code

33

Nested try block

- ❑ The try block within a try block is known as nested try block in java.
- ❑ Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

```

....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....

```

34

```

class Excep6{
public static void main(String args[]){
try{
    try{
        System.out.println("going to divide");
        int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
        int a[]=new int[5];
        a[5]=4;
    }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

    System.out.println("other statement");
}catch(Exception e){System.out.println("handeled");}

System.out.println("normal flow..");
}
}

```

35

finally block

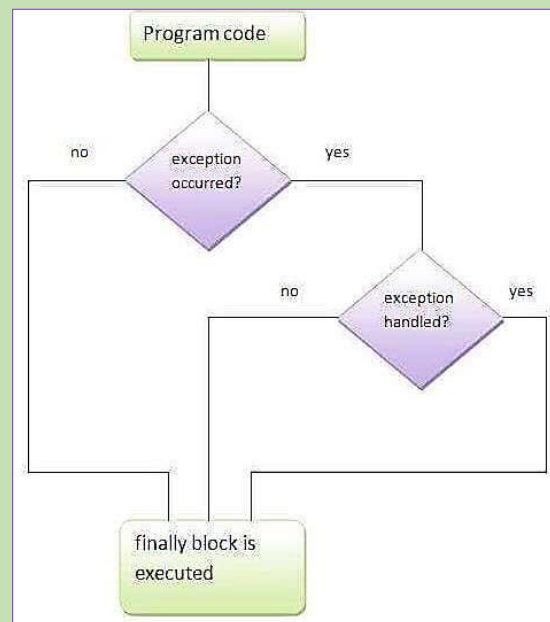
❑ Java finally block is a block that is used to execute important code such as **closing connection, stream** etc.

❑ Java finally block is always executed whether exception is handled or not.

❑ Java finally block follows try or catch block.

Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

36



37

? Usage of Java finally

- Case 1 - Let's see the java finally example where exception doesn't occur .

exception doesn't

```
class TestFinallyBlock{  
    public static void main(String args[]){  
        try{  
            int data=25/5;  
            System.out.println(data);  
        }  
        catch(NullPointerException e){System.out.println(e);}  
        finally{System.out.println("finally block is always executed");}  
        System.out.println("rest of the code...");  
    }  
}
```

Output:5
finally block is always executed
rest of the code...

38

Case 2 - Let's see the java finally example where
and not handled.

exception occurs

```
class TestFinallyBlock1{  
    public static void main(String args[]){  
        try{  
            int data=25/0;  
            System.out.println(data);  
        }  
        catch(NullPointerException e){System.out.println(e);}  
        finally{System.out.println("finally block is always executed");}  
        System.out.println("rest of the code...");  
    }  
}
```

Output:finally block is always executed

Exception in thread main java.lang.ArithmeticException:/ by zero

39

Case 3 - Let's see the java finally example where
and handled

exception occurs

```
public class TestFinallyBlock2{  
    public static void main(String args[]){  
        try{  
            int data=25/0;  
            System.out.println(data);  
        }  
        catch(ArithmeticException e){System.out.println(e);}  
        finally{System.out.println("finally block is always executed");}  
        System.out.println("rest of the code...");  
    }  
}
```

Output:Exception in thread main java.lang.ArithmeticException:/ by zero
finally block is always executed
rest of the code...

40

throw keyword

❑ The Java throw keyword is used to **explicitly throw an exception.**

❑ We can throw either checked or unchecked exception in java by throw keyword

Output

Exception in thread main java.lang.ArithmeticException: not valid

```
public class TestThrow1{  
    static void validate(int age){  
        if(age<18)  
            throw new ArithmeticException("not valid");  
        else  
            System.out.println("welcome to vote");  
    }  
    public static void main(String args[]){  
        validate(13);  
        System.out.println("rest of the code...");  
    }  
}
```

41

throws keyword

❑ The Java throws keyword is used to **declare an exception.**

- ❑ It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained

❑ Exception Handling is mainly used to handle the checked exceptions.

- ❑ If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

42

Syntax of java throws

```
return_type method_name() throws exception_class_name{  
    //method code  
}
```

Which exception should be declared

- checked exception only, because:
- unchecked Exception: under your control so correct your code.
- error: beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

43

Eg:

```
import java.io.*;  
class ThrowExample {  
    void myMethod(int num)throws IOException, ClassNotFoundException{  
        if(num==1)  
            throw new IOException("IOException Occurred");  
        else  
            throw new ClassNotFoundException("ClassNotFoundException");  
    }  
}  
  
public class Example1{  
    public static void main(String args[]){  
        try{  
            ThrowExample obj=new ThrowExample();  
            obj.myMethod(1);  
        }catch(Exception ex){  
            System.out.println(ex);  
        }  
    }  
}
```

Output

Output:

```
java.io.IOException: IOException Occurred
```

44

MODULE 3

CHAPTER 2 JAVA INPUT OUTPUT (I/O) & FILES

1

STREAM

- Java I/O (Input and Output) is used to process the input and produce the output.
- Java uses the concept of a stream to make I/O operation fast. The `java.io` package contains all the classes required for input and output operations.
- We can perform file handling in Java by `Java I/O API`.

STREAM

- A stream is a `sequence of data`. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

2

☐ In Java, `3 streams` are created for us automatically. All these streams are attached with the console.

1) `System.out` : standard output stream

2) `System.in` : standard input stream

3) `System.err` : standard error stream

❏ The code to print output and an error message to the console.

```
System.out.println("simple message");
```

```
System.err.println("error message");
```

❏ The code to get input from console.

```
int i=System.in.read(); //returns ASCII code of 1st character
```

3

❏ OutputStream vs InputStream

❏ OutputStream

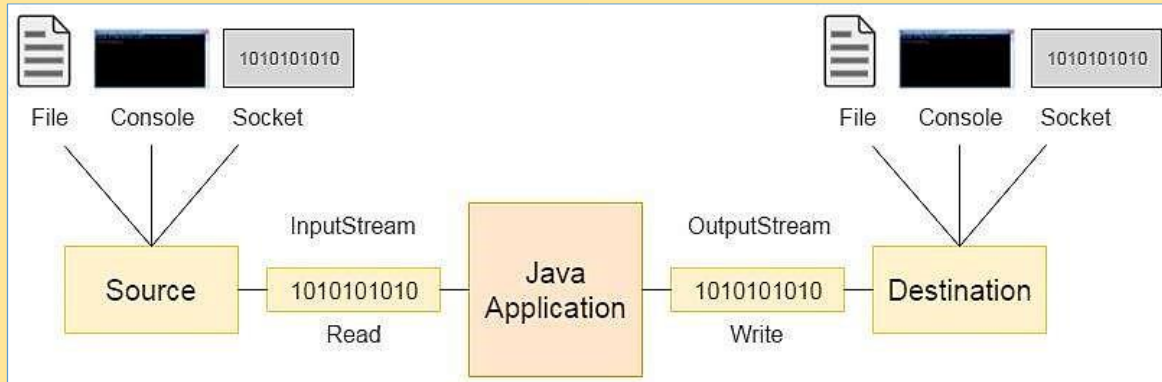
Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

❏ InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

4

? The working of Java OutputStream and InputStream



5

? OutputStream class

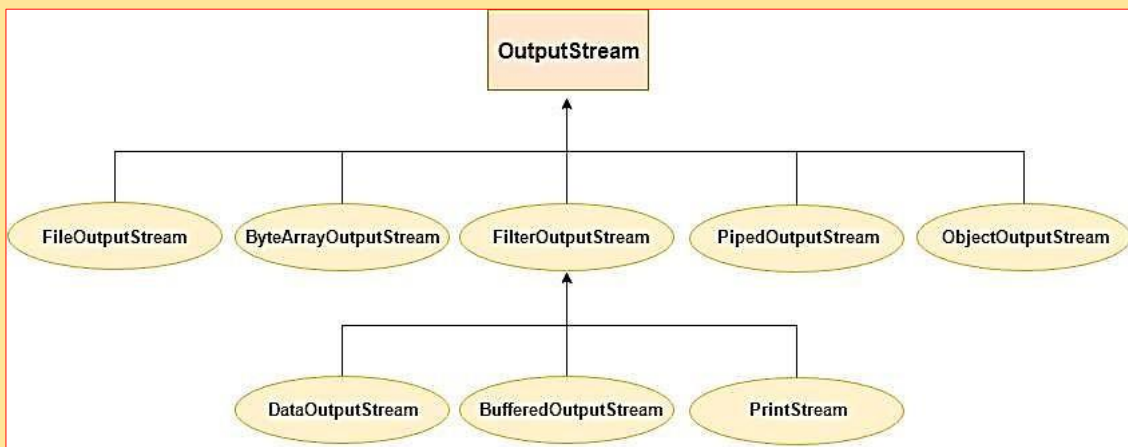
- OutputStream class is an **abstract class**.
- It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

Method	Description
1) public void write(int)throws IOException	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException	flushes the current output stream.
4) public void close()throws IOException	is used to close the current output stream.

6

? OutputStream Hierarchy



7

? InputStream class

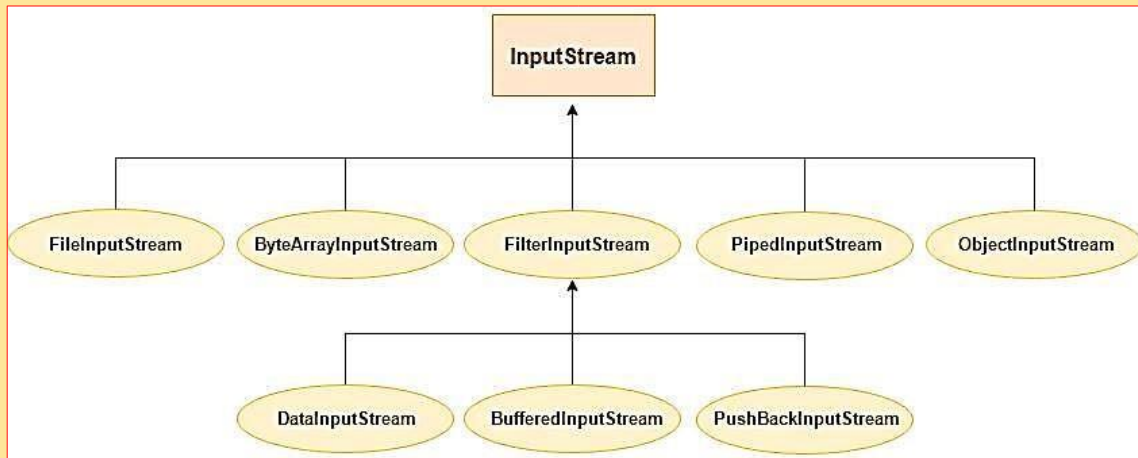
- InputStream class is an **abstract class**.
- It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

Method	Description
1) public abstract int read()throws IOException	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()throws IOException	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException	is used to close the current input stream.

8

? InputStream Hierarchy



9

READING CONSOLE INPUT

? In Java, there are **three different ways** for reading input from the user in the command line environment(console).

1. Using Buffered Reader Class

- This is the Java classical method to take input, Introduced in JDK1.0.
- This method is used by wrapping the `System.in` (standard input stream) in an `InputStreamReader` which is wrapped in a `BufferedReader`, we can read input from the user in the command line.
- **Advantage** - The input is buffered for efficient reading • **Drawback** - The wrapping code is hard to remember.

10

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

        // Reading data using readLine
        String name = reader.readLine();
        // Printing the read line
        System.out.println(name);
    }
}
```

1

2. Using Scanner Class

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

Advantages: Convenient methods for parsing primitives (nextInt(), nextFloat(), ...) from the tokenized input.

- Regular expressions can be used to find tokens. **Drawback:**
- The reading methods are not synchronized

12

// Java program to demonstrate working of Scanner in Java

```
import java.util.Scanner;
```

```
class GetInputFromUser
```

```
{
```

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

```
    {
```

```
        // Using Scanner for Getting Input from User
```

```
        Scanner in = new Scanner(System.in);
```

```
        String s = in.nextLine();
```

```
        System.out.println("You entered string "+s);
```

```
        int a = in.nextInt();
```

```
        System.out.println("You entered integer "+a);
```

```
        float b = in.nextFloat();
```

```
        System.out.println("You entered float "+b);
```

```
    }
```

```
}
```

Input:

HelloStudents

12

3.4

Output:

You entered string

HelloStudents

You entered integer 12

You entered float 3.4

13

3. Using Console Class

☑ It has been becoming a preferred way for reading user's input from the command line.

☑ In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like System.out.printf()).

Advantages:

☑ Reading password without echoing the entered characters.

☑ Reading methods are synchronized.

☑ Format string syntax can be used.

Drawback: Does not work in non-interactive environment (such as in an IDE).

14

```
// Java program to demonstrate working of System.console()
// Note that this program does not work on IDEs as
// System.console() may require console
public class Sample
{
    public static void main(String[] args)
    {
        // Using Console to input data from user
        String name = System.console().readLine();

        System.out.println(name);
    }
}
```

WRITING CONSOLE OUTPUT

- Console output is most easily accomplished with `print()` and `println()` methods.
- These methods are defined by the class `PrintStream` which is the type of object referenced by `System.in`.
- Because the `PrintStream` is an output stream derived from the `OutputStream`, it also implements the low-level method `write()`.
- Thus, `write()` can be used to write to the console. The simplest form of `write()` defined by the `PrintStream` is shown below :

```
void write(int byteval)
```

- Following is a short example that uses write() to output the character 'X' followed by a newline to the screen:

```
/* Java Program Example - Java Write Console Output
 * This program writes the character X followed by newline
 * This program demonstrates System.out.write() */

class WriteConsoleOutput
{
    public static void main(String args[])
    {
        int y;

        y = 'X';

        System.out.write(y);
        System.out.write('\n');

    }
}
```

17

PrintWriter CLASS

- Java PrintWriter class is the implementation of Writer class.
- It is used to print the formatted representation of objects to the text-output stream.

Class declaration

public class PrintWriter extends Writer

Methods of PrintWriter class

Method	Description
void println(boolean x)	It is used to print the boolean value.
void println(char[] x)	It is used to print an array of characters.
void println(int x)	It is used to print an integer.

18

<code>PrintWriter append(char c)</code>	It is used to append the specified character to the writer.
<code>PrintWriter append(CharSequence ch)</code>	It is used to append the specified character sequence to the writer.
<code>PrintWriter append(CharSequence ch, int start, int end)</code>	It is used to append a subsequence of specified character to the writer.
<code>boolean checkError()</code>	It is used to flushes the stream and check its error state.
<code>protected void setError()</code>	It is used to indicate that an error occurs.
<code>protected void clearError()</code>	It is used to clear the error state of a stream.
<code>PrintWriter format(String format, Object... args)</code>	It is used to write a formatted string to the writer using specified arguments and format string.
<code>void print(Object obj)</code>	It is used to print an object.
<code>void flush()</code>	It is used to flushes the stream.
<code>void close()</code>	It is used to close the stream.

19

Eg:

```
import java.io.File;
import java.io.PrintWriter;
public class PrintWriterExample {
    public static void main(String[] args) throws Exception {
        //Data to write on Console using PrintWriter
        PrintWriter writer = new PrintWriter(System.out);
        writer.write("Javatpoint provides tutorials of all technology.");
        writer.flush();
        writer.close();
        //Data to write in File using PrintWriter
        PrintWriter writer1 = null;
        writer1 = new PrintWriter(new File("D:\\testout.txt"));
        writer1.write("Like Java, Spring, Hibernate, Android, PHP etc.");
        writer1.flush();
        writer1.close();
    }
}
```

Output

Javatpoint provides tutorials of all technology.

20

SERIALIZATION

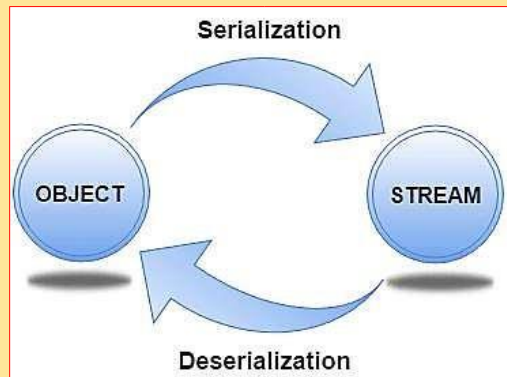
- Serialization in Java is the process of converting the Java code Object into a Byte Stream, to transfer the Object Code from one Java Virtual machine to another and recreate it using the process of Deserialization.
- Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.
- For **serializing** the object, we call the **writeObject()** method of ObjectOutputStream, and for **deserialization** we call the **readObject()** method of ObjectInputStream class.

21

- We must have to **implement** the **Serializable** **interface** for serializing the object.

Advantages of Java Serialization

- It is mainly used to travel object's state on the network (which is known as marshaling).



22

? ObjectOutputStream class

- The `ObjectOutputStream` class is used to write primitive data types, and Java objects to an `OutputStream`.
- Only objects that support the `java.io.Serializable` interface can be written to streams.

Constructor

1) <code>public ObjectOutputStream(OutputStream out) throws IOException {}</code>	creates an <code>ObjectOutputStream</code> that writes to the specified <code>OutputStream</code> .
---	---

Important Methods

Method	Description
1) <code>public final void writeObject(Object obj) throws IOException {}</code>	writes the specified object to the <code>ObjectOutputStream</code> .
2) <code>public void flush() throws IOException {}</code>	flushes the current output stream.
3) <code>public void close() throws IOException {}</code>	closes the current output stream.

23

? ObjectInputStream class

- An `ObjectInputStream` deserializes objects and primitive data written using an `ObjectOutputStream`.

Constructor

1) <code>public ObjectInputStream(InputStream in) throws IOException {}</code>	creates an <code>ObjectInputStream</code> that reads from the specified <code>InputStream</code> .
--	--

Important Methods

Method	Description
1) <code>public final Object readObject() throws IOException, ClassNotFoundException {}</code>	reads an object from the input stream.
2) <code>public void close() throws IOException {}</code>	closes <code>ObjectInputStream</code> .

24

Example of Java Serialization

- In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

Output

```
success
```

```
import java.io.*;
class Persist{
    public static void main(String args[]){
        try{
            //Creating the object
            Student s1 =new Student(211,"ravi");
            //Creating stream and writing the object
            FileOutputStream fout=new FileOutputStream("f.txt");
            ObjectOutputStream out=new ObjectOutputStream(fout);
            out.writeObject(s1);
            out.flush();
            //closing the stream
            out.close();
            System.out.println("success");
        }catch(Exception e){System.out.println(e);}
    }
}
```

25

WORKING WITH FILES

- File handling is an important part of any application.
- Java has several methods for creating, reading, updating, and deleting files.
- The File class from the java.io package, allows us to work with files.
- To use the File class, create an object of the class, and specify the filename or directory name:

Example

```
import java.io.File; // Import the File class

File myObj = new File("filename.txt"); // Specify the filename
```

26

- The File class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
<code>canRead()</code>	Boolean	Tests whether the file is readable or not
<code>canWrite()</code>	Boolean	Tests whether the file is writable or not
<code>createNewFile()</code>	Boolean	Creates an empty file
<code>delete()</code>	Boolean	Deletes a file
<code>exists()</code>	Boolean	Tests whether the file exists
<code>getName()</code>	String	Returns the name of the file
<code>getAbsolutePath()</code>	String	Returns the absolute pathname of the file
<code>length()</code>	Long	Returns the size of the file in bytes
<code>list()</code>	String[]	Returns an array of the files in the directory
<code>mkdir()</code>	Boolean	Creates a directory

27

❓ Create a File

❓ To create a file in Java, you can use the `createNewFile()` method.

❓ This method returns a boolean value: **true** if the file was **successfully created**, and **false** if the **file already exists**.

❓ Note that the method is enclosed in a try...catch block.

❓ This is necessary because it throws an IOException if an error occurs (if the file cannot be created for some reason):

28

Example

```
import java.io.File; // Import the File class
import java.io.IOException; // Import the IOException class to handle errors

public class CreateFile {
    public static void main(String[] args) {
        try {
            File myObj = new File("filename.txt");
            if (myObj.createNewFile()) {
                System.out.println("File created: " + myObj.getName());
            } else {
                System.out.println("File already exists.");
            }
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

The output will be:

```
File created: filename.txt
```

29

❓ Write To a File

- ❓ In the following example, we use the **FileWriter class** together with its **write()** method to write some text to the file we created in the example above.
- ❓ Note that when we are done writing to the file, we should close it with the **close()** method:

30

Example

```
import java.io.FileWriter; // Import the FileWriter class
import java.io.IOException; // Import the IOException class to handle errors

public class WriteToFile {
    public static void main(String[] args) {
        try {
            FileWriter myWriter = new FileWriter("filename.txt");
            myWriter.write("Files in Java might be tricky, but it is fun enough!");
            myWriter.close();
            System.out.println("Successfully wrote to the file.");
        } catch (IOException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

The output will be:

```
Successfully wrote to the file.
```

31

? Read Files

```
import java.io.File; // Import the File class
import java.io.FileNotFoundException; // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files

public class ReadFile {
    public static void main(String[] args) {
        try {
            File myObj = new File("filename.txt");
            Scanner myReader = new Scanner(myObj);
            while (myReader.hasNextLine()) {
                String data = myReader.nextLine();
                System.out.println(data);
            }
            myReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

The output will be:

```
Files in Java might be tricky, but it is fun enough!
```

32

❓ Get File Information

```
import java.io.File; // Import the File class

public class GetFileInfo {
    public static void main(String[] args) {
        File myObj = new File("filename.txt");
        if (myObj.exists()) {
            System.out.println("File name: " + myObj.getName());
            System.out.println("Absolute path: " + myObj.getAbsolutePath());
            System.out.println("Writeable: " + myObj.canWrite());
            System.out.println("Readable " + myObj.canRead());
            System.out.println("File size in bytes " + myObj.length());
        } else {
            System.out.println("The file does not exist.");
        }
    }
}
```

The output will be:

```
File name: filename.txt
Absolute path: C:\Users\MyName\filename.txt
Writeable: true
Readable: true
File size in bytes: 0
```

33

❓ Delete a File

❓ To delete a file in Java, use the delete() method:

```
import java.io.File; // Import the File class

public class DeleteFile {
    public static void main(String[] args) {
        File myObj = new File("filename.txt");
        if (myObj.delete()) {
            System.out.println("Deleted the file: " + myObj.getName());
        } else {
            System.out.println("Failed to delete the file.");
        }
    }
}
```

The output will be:

```
Deleted the file: filename.txt
```

34

Delete a Folder

```
import java.io.File;

public class DeleteFolder {
    public static void main(String[] args) {
        File myObj = new File("C:\\Users\\MyName\\Test");
        if (myObj.delete()) {
            System.out.println("Deleted the folder: " + myObj.getName());
        } else {
            System.out.println("Failed to delete the folder.");
        }
    }
}
```

The output will be:

```
Deleted the folder: Test
```

35

MODULE 4 ADVANCED FEATURES OF JAVA

CHAPTER 1 Java Library & Collections framework

1

STRING

- In Java, string is basically an **object** that represents sequence of char values. An array of characters works same as Java string. For example:

```
char[] ch={'h','a','i','j','a','v','a'};  
String s=new String(ch); is same as:  
String s = "haijava";
```

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

2

❓ Create a string object

❓ There are two ways to create String object:

- By string literal
- By new keyword

1) String Literal

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

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

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

3

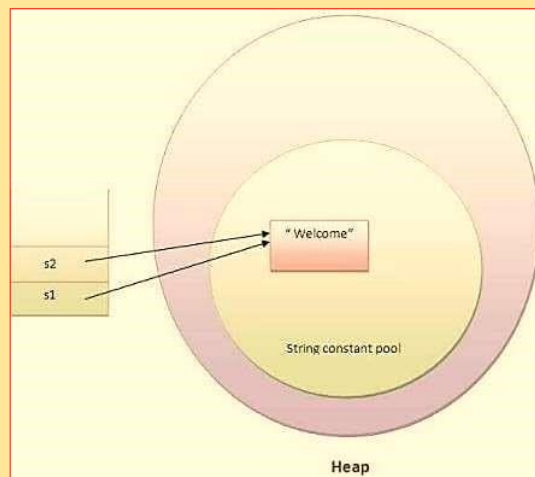
```
String s1="Welcome";
```

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

- In the above example, only one object will be created.
- Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object.
- After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.
- **Note:** String objects are stored in a special memory area known as the "string constant pool".

📌 Why Java uses the concept of String literal

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



2) By new keyword

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

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

6

String Example

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

OUTPUT

```
java  
strings  
example
```

7

STRING CONSTRUCTORS

- The string class supports several types of constructors in Java APIs. The most commonly used constructors of String class are as follows:

1. **String()** : To create an empty String, we will call a default constructor. For example:

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

- It will create a string object in the heap area with no value

8

2. **String(String str)** : It will create a string object in the heap area and stores the given value in it. For example:

```
String s2 = new String("Hello Java");
```

Now, the object contains Hello Java.

3. **String(char chars[])** : It will create a string object and stores the array of characters in it. For example: `char chars[] = { 'a', 'b', 'c', 'd' };`

```
String s3 = new String(chars);
```

The object reference variable s3 contains the address of the value stored in the heap area.

9

- ☐ Let's take an example program where we will create a string object and store an array of characters in it

```
package stringPrograms;
public class Science
{
    public static void main(String[] args)
    {
        char chars[] = { 's', 'c', 'i', 'e', 'n', 'c', 'e' };
        String s = new String(chars);
        System.out.println(s);
    }
}
```

Output:
science

10

4. String(char chars[], int startIndex, int count)

- It will create and initializes a string object with a subrange of a character array.
- The argument **startIndex** specifies the index at which the subrange begins and count specifies the number of characters to be copied.

For example:

```
char chars[ ] = { 'w', 'i', 'n', 'd', 'o', 'w', 's' }; String str = new
String(chars, 2, 3);
```

- The object str contains the address of the value "ndo" stored in the heap area because the starting index is 2 and the total number of characters to be copied is 3

11

EXAMPLE

```
package stringPrograms;
public class Windows
{
    public static void main(String[] args)
    {
        char chars[] = { 'w', 'i', 'n', 'd', 'o', 'w', 's' };
        String s = new String(chars, 0,4);
        System.out.println(s);
    }
}
```

Output:

wind

12

- In this example program, we will construct a String object that contains the same characters sequence as another string object.

```
package stringPrograms;
public class MakeString
{
    public static void main(String[] args)
    {
        char chars[] = { 'F', 'A', 'N' };
        String s1 = new String(chars);
        String s2 = new String(s1);
        System.out.println(s1);
        System.out.println(s2);
    }
}
```

Output:

FAN

FAN

As you can see the output, s1 and s2 contain the same string.

Thus, we can create one string from another string.

13

5. **String(byte byteArr[])** : It constructs a new string object by decoding the given array of bytes (i.e, by decoding ASCII values into the characters) according to the system's default character set.

```
package stringPrograms;
public class ByteArray
{
    public static void main(String[] args)
    {
        byte b[] = { 97, 98, 99, 100 }; // Range of bytes: -128 to 127. These byte
        values will be converted into corresponding characters.
        String s = new String(b);
        System.out.println(s);
    }
}
```

Output:

abcd

14

6. **String(byte byteArr[], int startIndex, int count)**

This constructor also creates a new string object by decoding the ASCII values using the system's default character set.

```
package stringPrograms;
public class ByteArray
{
    public static void main(String[] args)
    {
        byte b[] = { 65, 66, 67, 68, 69, 70 }; // Range of bytes: -128 to 127.
        String s = new String(b, 2, 4); // CDEF
        System.out.println(s);
    }
}
```

Output:

CDEF

15

STRING LENGTH

- The java string length() method gives length of the string. It returns count of total number of characters.

- **Internal implementation**

```
public int length() {  
    return value.length;  
}
```

Signature - The signature of the string length() method is given below:

```
public int length()
```

16

String length() method example - 1

```
public class LengthExample{  
    public static void main(String args[]){  
        String s1="javatpoint";  
        String s2="python";  
        System.out.println("string length is: "+s1.length());//10 is the length of javatpoint string  
        System.out.println("string length is: "+s2.length());//6 is the length of python string  
    }}
```

Output

```
string length is: 10  
string length is: 6
```

17

String length() method example - 2

```
public class LengthExample2 {  
    public static void main(String[] args) {  
        String str = "Javatpoint";  
        if(str.length()>0) {  
            System.out.println("String is not empty and length is: "+str.length());  
        }  
        str = "";  
        if(str.length()==0) {  
            System.out.println("String is empty now: "+str.length());  
        }  
    }  
}
```

Output

```
String is not empty and length is: 10  
String is empty now: 0
```

18

STRING COMPARISON

- We can compare string in java on the basis of content and reference
- There are three ways to compare string in java:

By `equals()` method

By `=` operator

By `compareTo()` method

String compare by equals() method

- The String `equals()` method compares the original content of the string.
- It compares values of string for equality. String class provides two methods

19

public boolean equals(Object another)
compares this string to the specified object.

public boolean equalsIgnoreCase(String another) compares this String to another string, ignoring case.

```
class Teststringcomparison1{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="Sachin";  
        String s3=new String("Sachin");  
        String s4="Saurav";  
        System.out.println(s1.equals(s2));//true  
        System.out.println(s1.equals(s3));//true  
        System.out.println(s1.equals(s4));//false  
    }  
}
```

```
Output:true  
true  
false
```

20

Example 2

```
class Teststringcomparison2{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="SACHIN";  
  
        System.out.println(s1.equals(s2));//false  
        System.out.println(s1.equalsIgnoreCase(s2));//true  
    }  
}
```

Output

```
false  
true
```

21

? String compare by == operator

- The == operator compares references not values.

```
class Teststringcomparison3{  
    public static void main(String args[]){  
        String s1="Sachin";  
        String s2="Sachin";  
        String s3=new String("Sachin");  
        System.out.println(s1==s2);//true (because both refer to same instance)  
        System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)  
    }  
}
```

Output:true
false

22

□ String compare by compareTo() method

- The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

s1 == s2 : 0 s1 > s2 : positive
value s1 < s2: negative value

```

class Teststringcomparison4{
    public static void main(String args[]){
        String s1="Sachin";
        String s2="Sachin";
        String s3="Ratan";
        System.out.println(s1.compareTo(s2)); //0
        System.out.println(s1.compareTo(s3)); //1(because s1>s3)
        System.out.println(s3.compareTo(s1)); //-1(because s3 < s1 )
    }
}

```

Output:0

1

-1

24

Eg:

```

public class CompareToExample{
    public static void main(String args[]){
        String s1="hello";
        String s2="hello";
        String s3="meklo";
        String s4="hemlo";
        String s5="flag";
        System.out.println(s1.compareTo(s2)); //0 because both are equal
        System.out.println(s1.compareTo(s3)); //-5 because "h" is 5 times lower than "m"
        System.out.println(s1.compareTo(s4)); //-1 because "l" is 1 times lower than "m"
        System.out.println(s1.compareTo(s5)); //2 because "h" is 2 times greater than "f"
    }}

```

Output

0

-5

-1

2

25

SEARCHING STRINGS

String contains()

- The java string `contains()` method searches the sequence of characters in this string.
- It returns true if sequence of char values are found in this string otherwise returns false.

Internal implementation public boolean

```
contains(CharSequence s) { return  
    indexOf(s.toString()) > -1;  
}
```

26

Signature

- The signature of string contains() method is given below:

```
public boolean contains(CharSequence sequence)
```

```
class ContainsExample{  
    public static void main(String args[]){  
        String name="what do you know about me";  
        System.out.println(name.contains("do you know"));  
        System.out.println(name.contains("about"));  
        System.out.println(name.contains("hello"));  
    }  
}
```

Output

```
true  
true  
false
```

27

Eg 2 - The contains() method searches case sensitive char sequence. If the argument is not case sensitive, it returns false. Let's see an example below.

```
public class ContainsExample2 {  
    public static void main(String[] args) {  
        String str = "Hello Javatpoint readers";  
        boolean isContains = str.contains("Javatpoint");  
        System.out.println(isContains);  
        // Case Sensitive  
        System.out.println(str.contains("javatpoint")); // false  
    }  
}
```

Output

```
true  
false
```

28

Eg 3 - The contains() method is helpful to find a char-sequence in the string. We can use it in control structure to produce search based result. Let us see an example below.

```
public class ContainsExample3 {  
    public static void main(String[] args) {  
        String str = "To learn Java visit Javatpoint.com";  
        if(str.contains("Javatpoint.com")) {  
            System.out.println("This string contains javatpoint.com");  
        }else  
            System.out.println("Result not found");  
    }  
}
```

Output:

```
This string contains javatpoint.com
```

29

CHARACTER EXTRACTION

□ String charAt()

- The java string charAt() method returns a char value at the given index number.
- The index number starts from 0 and goes to n-1, where n is length of the string.
- It returns **StringIndexOutOfBoundsException** if given index number is greater than or equal to this string length or a negative number.
- **Signature** - The signature of string charAt() method is given below:

```
public char charAt(int index)
```

30

Example:

```
public class CharAtExample{  
    public static void main(String args[]){  
        String name="javatpoint";  
        char ch=name.charAt(4);//returns the char value at the 4th index  
        System.out.println(ch);  
    }  
}
```

Output

t

31

❓ StringIndexOutOfBoundsException with charAt()

- Let's see the example of charAt() method where we are passing greater index value.
- In such case, it throws StringIndexOutOfBoundsException at run time.

```
public class CharAtExample{  
    public static void main(String args[]){  
        String name="javatpoint";  
        char ch=name.charAt(10);//returns the char value at the 10th index  
        System.out.println(ch);  
    }  
}
```

32

Output:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:  
String index out of range: 10  
at java.lang.String.charAt(String.java:658)  
at CharAtExample.main(CharAtExample.java:4)
```

❓ Java String charAt() Example 3

- Let's see a simple example where we are accessing first and last character from the provided string.

33

```

public class CharAtExample3 {
    public static void main(String[] args) {
        String str = "Welcome to Javatpoint portal";
        int strLength = str.length();
        // Fetching first character
        System.out.println("Character at 0 index is: "+ str.charAt(0));
        // The last Character is present at the string length-1 index
        System.out.println("Character at last index is: "+ str.charAt(strLength-1));
    }
}

```

Output:

```

Character at 0 index is: W
Character at last index is: l

```

34

Java String charAt() Example 4

- Let's see an example where we are accessing all the elements present at odd index.

```

public class CharAtExample4 {
    public static void main(String[] args) {
        String str = "Welcome to Javatpoint portal";
        for (int i=0; i<=str.length()-1; i++) {
            if(i%2!=0) {
                System.out.println("Char at "+i+" place "+str.charAt(i));
            }
        }
    }
}

```

Output:

```

Char at 1 place e
Char at 3 place c
Char at 5 place m
Char at 7 place 
Char at 9 place o
Char at 11 place J
Char at 13 place v
Char at 15 place t
Char at 17 place o
Char at 19 place n
Char at 21 place 
Char at 23 place o
Char at 25 place t
Char at 27 place l

```

35

Java String charAt() Example 5

- Let's see an example where we are counting frequency of a character in the string.

```
public class CharAtExample5 {  
    public static void main(String[] args) {  
        String str = "Welcome to Javatpoint portal";  
        int count = 0;  
        for (int i=0; i<=str.length()-1; i++) {  
            if(str.charAt(i) == 't') {  
                count++;  
            }  
        }  
        System.out.println("Frequency of t is: "+count);  
    }  
}
```

Output:

Frequency of t is: 4

36

MODIFY STRINGS

- The java string replace() method returns a string replacing all the old char or CharSequence to new char or CharSequence. **Signature**
- There are two type of replace methods in java string.
 public String replace(char oldChar, char newChar) and
 public String replace(CharSequence target, CharSequence replacement) • The second replace method is added since JDK 1.5.

37

? String replace(char old, char new) method example

```
public class ReplaceExample1{  
    public static void main(String args[]){  
        String s1="java is a very good language";  
        // replaces all occurrences of 'a' to 'e'  
        String replaceString=s1.replace('a','e');  
        System.out.println(replaceString);  
    }  
}
```

Output

jeve is e very good leneuge

38

? String replace(CharSequence target, CharSequence replacement) method example

```
public class ReplaceExample2{  
    public static void main(String args[]){  
        String s1="my name is khan my name is java";  
        String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "was"  
        System.out.println(replaceString);  
    }  
}
```

Output

my name was khan my name was java

39

❓ String replace() Method Example 3

```
public class ReplaceExample3 {  
    public static void main(String[] args) {  
        String str = "oooooo-hhhh-oooooo";  
        String rs = str.replace("h","s"); // Replace 'h' with 's'  
        System.out.println(rs);  
        rs = rs.replace("s","h"); // Replace 's' with 'h'  
        System.out.println(rs);  
    }  
}
```

Output

```
oooooo-ssss-oooooo  
oooooo-hhhh-oooooo
```

40

- The java string replaceAll() method returns a string replacing all the sequence of characters matching regex and replacement string.

Internal implementation public String replaceAll(String regex, String replacement) {
return Pattern.compile(regex).matcher(this).replaceAll(replacement);
}

Signature public String replaceAll(String regex, String replacement)

41

❑ String replaceAll() example: replace character

- Let's see an example to replace all the occurrences of a single character.

```
public class ReplaceAllExample1{  
    public static void  
    main(String args[]){
```



```
String s1="java is a very good language";  
String replaceString=s1.replaceAll("a","e");//replaces all occurrences of "a" to "e"  
  
System.out.println(replaceString);  
}}
```

Output jeve is e very good lenuge

42

🔗 String replaceAll() example: replace word

- Let's see an example to replace all the occurrences of single word or set of words.

```
public class ReplaceAllExample2{  
    public static void main(String args[]){  
        String s1="My name is Khan. My name is Bob. My name is Sonoo."  
        String replaceString=s1.replaceAll("is","was");//replaces all occurrences of "is" to "was"  
        System.out.println(replaceString);  
    }  
}
```

Output

My name was Khan. My name was Bob. My name was Sonoo.

43

? String replaceAll() example: remove white spaces

- Let's see an example to remove all the occurrences of white spaces.

```
public class ReplaceAllExample3{  
    public static void main(String args[]){  
        String s1="My name is Khan. My name is Bob. My name is Sonoo."  
        String replaceString=s1.replaceAll("\\s","");  
        System.out.println(replaceString);  
    }  
}
```

Output

MynameisKhan.MynameisBob.MynameisSonoo.

44

STRING VALUE OF ()

- The java string `valueOf()` method converts different types of values into string.
- By the help of string `valueOf()` method, we can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

Internal implementation public static String valueOf(Object obj) {
 return (obj == null) ? "null" : obj.toString();
}

45

? Signature

- The signature or syntax of string valueOf() method is given below:

public static	String valueOf(boolean b)
public static	String valueOf(char c)
public static	String valueOf(char [] c)
public static	String valueOf(int i)
public static	String valueOf(long l)
public static	String valueOf(float f)
public static	String valueOf(double d)
public static	String valueOf(Object o)

46

? valueOf() method example

```
public class StringValueOfExample{  
    public static void main(String args[]){  
        int value=30;  
        String s1=String.valueOf(value);  
        System.out.println(s1+10); //concatenating string with 10  
    }  
}
```

Output

3010

47

❓ valueOf(boolean bol) Method Example

❓ This is a boolean version of overloaded valueOf() method. It takes boolean value and returns a string. Let's see an example.

```
public class StringValueOfExample2 {  
    public static void main(String[] args) {  
        // Boolean to String  
        boolean bol = true;  
        boolean bol2 = false;  
        String s1 = String.valueOf(bol);  
        String s2 = String.valueOf(bol2);  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

Output

true
false

48

❓ valueOf(char ch) Method Example

❓ This is a char version of overloaded valueOf() method. It takes char value and returns a string. Let's see an example.

```
public class StringValueOfExample3 {  
    public static void main(String[] args) {  
        // char to String  
        char ch1 = 'A';  
        char ch2 = 'B';  
        String s1 = String.valueOf(ch1);  
        String s2 = String.valueOf(ch2);  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

Output

A
B

49

❓ `valueOf(float f)` and `valueOf(double d)` Example

❓ This is a float version of overloaded `valueOf()` method. It takes float value and returns a string. Let's see an example.

```
public class StringValueOfExample4 {  
    public static void main(String[] args) {  
        // Float and Double to String  
        float f = 10.05f;  
        double d = 10.02;  
        String s1 = String.valueOf(f);  
        String s2 = String.valueOf(d);  
        System.out.println(s1);  
        System.out.println(s2);  
    }  
}
```

Output

10.05

10.02

50

❓ `String valueOf()` Complete Examples

```
public class StringValueOfExample5 {  
    public static void main(String[] args) {  
        boolean b1=true;  
        byte b2=11;  
        short sh = 12;  
        int i = 13;  
        long l = 14L;  
        float f = 15.5f;  
        double d = 16.5d;  
        char chr[]={'j','a','v','a'};  
        StringValueOfExample5 obj=new StringValueOfExample5();  
        String s1 = String.valueOf(b1);  
        String s2 = String.valueOf(b2);  
        String s3 = String.valueOf(sh);  
        String s4 = String.valueOf(i);  
        String s5 = String.valueOf(l);  
        String s6 = String.valueOf(f);  
        String s7 = String.valueOf(d);  
        String s8 = String.valueOf(chr);  
        String s9 = String.valueOf(obj);  
    }  
}
```

```
System.out.println(s1);  
System.out.println(s2);  
System.out.println(s3);  
System.out.println(s4);  
System.out.println(s5);  
System.out.println(s6);  
System.out.println(s7);  
System.out.println(s8);  
System.out.println(s9);
```

Output

```
true  
11  
12  
13  
14  
15.5  
16.5  
java  
StringValueOfExample5@2a139a55
```

51

❓ Immutable String in Java

- In java, string objects are immutable . Immutable simply means **unmodifiable** or **unchangeable** . Once string object is created its data or state can't be changed but a new string object is created.

Example

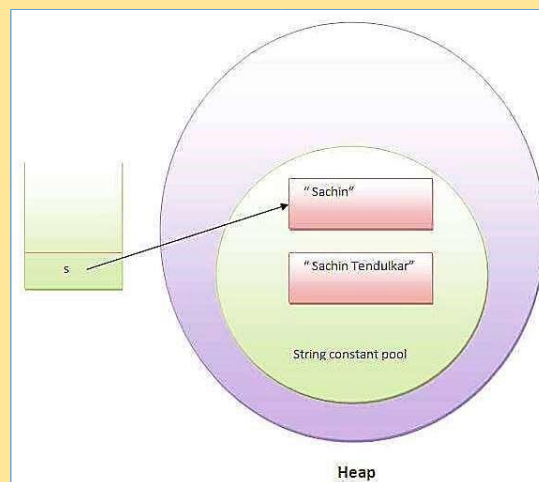
```
class Testimmutablestring{  
    public static void main(String args[]){  
        String s="Sachin";  
        s.concat(" Tendulkar");//concat() method appends the string at the end  
        System.out.println(s);//will print Sachin because strings are immutable objects  
    }  
}
```

Output

Sachin

52

It can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



53

- As you can see in the figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".
- But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar" object. For example:

```
class Testimmutablestring1{  
    public static void main(String args[]){  
        String s="Sachin";  
        s=s.concat(" Tendulkar");  
        System.out.println(s);  
    }  
}
```

Output

Sachin Tendulkar

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

54

❓ **Why string objects are immutable in java** • Because java uses the concept of string literal.

- Suppose there are 5 reference variables, all refer to one object "sachin".
- If one reference variable changes the value of the object, it will be affected to all the reference variables.
- That is why string objects are immutable in java.

String and StringBuffer

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you concat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

- Java **StringBuffer class** is used to create **mutable** (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

56

? Important Constructors of StringBuffer class

Constructor	Description
StringBuffer()	creates an empty string buffer with the initial capacity of 16.
StringBuffer(String str)	creates a string buffer with the specified string.
StringBuffer(int capacity)	creates an empty string buffer with the specified capacity as length.

Mutable string - A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

57

? StringBuffer append() method

```
class StringBufferExample{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello ");  
        sb.append("Java");//now original string is changed  
        System.out.println(sb);//prints Hello Java  
    }  
}
```

58

? StringBuffer insert() method

? The insert() method inserts the given string with this string at the given position.

```
class StringBufferExample2{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello ");  
        sb.insert(1,"Java");//now original string is changed  
        System.out.println(sb);//prints HJavaello  
    }  
}
```

59

? StringBuffer replace() method

? The replace() method replaces the given string from the specified beginIndex and endIndex.

```
class StringBufferExample3{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello");  
        sb.replace(1,3,"Java");  
        System.out.println(sb);//prints HJavallo  
    }  
}
```

60

? StringBuffer delete() method

? The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class StringBufferExample4{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello");  
        sb.delete(1,3);  
        System.out.println(sb);//prints Hlo  
    }  
}
```

61

? StringBuffer reverse() method

? The reverse() method of StringBuffer class reverses the current string.

```
class StringBufferExample5{  
    public static void main(String args[]){  
        StringBuffer sb=new StringBuffer("Hello");  
        sb.reverse();  
        System.out.println(sb);//prints olleH  
    }  
}
```

62

COLLECTIONS IN JAVA

? The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

? Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

? Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

63

? **Collection in Java** - Represents a single unit of objects, i.e., a group.

? **framework in Java**

- It provides readymade architecture.

- It represents a set of classes and interfaces.
- It is optional.

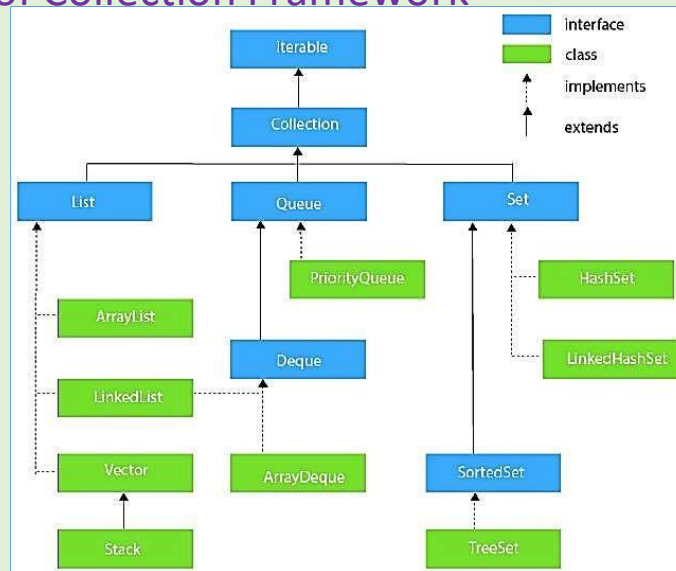
❓ Collection framework

❓ The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

- Interfaces and its implementations, i.e., classes
- Algorithm

64

❓ Hierarchy of Collection Framework



65

❓ The java.util package contains all the **classes** and **interfaces** for the Collection framework.

□ Collection Interface

- The Collection interface is the interface which is implemented by all the classes in the collection framework.
- It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.
- Some of the methods of Collection interface are Boolean add (Object obj), Boolean addAll (Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

66

LIST INTERFACE

- List interface is the child interface of Collection interface.
- It inhibits a list type data structure in which we can store the ordered collection of objects.
- It can have duplicate values.
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
- To instantiate the List interface, we must use :

67

```
List <data-type> list1= new ArrayList();  
List <data-type> list2 = new LinkedList();  
List <data-type> list3 = new Vector();  
List <data-type> list4 = new Stack();
```

❑ There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

❑ The classes that implement the List interface are given below. **ArrayList**

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types.

68

- The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;  
class TestJavaCollection1{  
    public static void main(String args[]){  
        ArrayList<String> list=new ArrayList<String>();//Creating arraylist  
        list.add("Ravi");//Adding object in arraylist  
        list.add("Vijay");  
        list.add("Ravi");  
        list.add("Ajay");  
        //Traversing list through Iterator  
        Iterator itr=list.iterator();  
        while(itr.hasNext()){  
            System.out.println(itr.next());  
        }  
    }  
}
```

Output:

```
Ravi  
Vijay  
Ravi  
Ajay
```

69

❑ Java ArrayList class uses a dynamic array for storing the elements.

❑ It is like an array, but there is no size limit. We can add or remove elements anytime.

❑ So, it is much more flexible than the traditional array. It is found in the java.util package. It is like the Vector in C++.

❑ The ArrayList in Java can have the duplicate elements also. It implements the List interface so we can use all the methods of List interface here.

❑ The ArrayList maintains the insertion order internally.

❑ It inherits the AbstractList class and implements List interface.

70

❑ The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.
- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In ArrayList, manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

71

ArrayList Example

```
import java.util.*;

public class ArrayListExample1{
    public static void main(String args[]){
        ArrayList<String> list=new ArrayList<String>();//Creating arraylist
        list.add("Mango");//Adding object in arraylist
        list.add("Apple");
        list.add("Banana");
        list.add("Grapes");
        //Printing the arraylist object
        System.out.println(list);
    }
}
```

Output:

```
[Mango, Apple, Banana, Grapes]
```


Iterating ArrayList using Iterator

```
import java.util.*;
public class ArrayListExample2{
    public static void main(String args[]){
        ArrayList<String> list=new ArrayList<String>();//Creating arraylist
        list.add("Mango");//Adding object in arraylist
        list.add("Apple");
        list.add("Banana");
        list.add("Grapes");
        //Traversing list through Iterator
        Iterator itr=list.iterator();//getting the Iterator
        while(itr.hasNext()){//check if iterator has the elements
            System.out.println(itr.next());//printing the element and move to next
        }
    }
}
```

Output:

```
Mango
Apple
Banana
Grapes
```

73

MODULE 4

CHAPTER 2 MULTITHREADED PROGRAMMING

THREAD

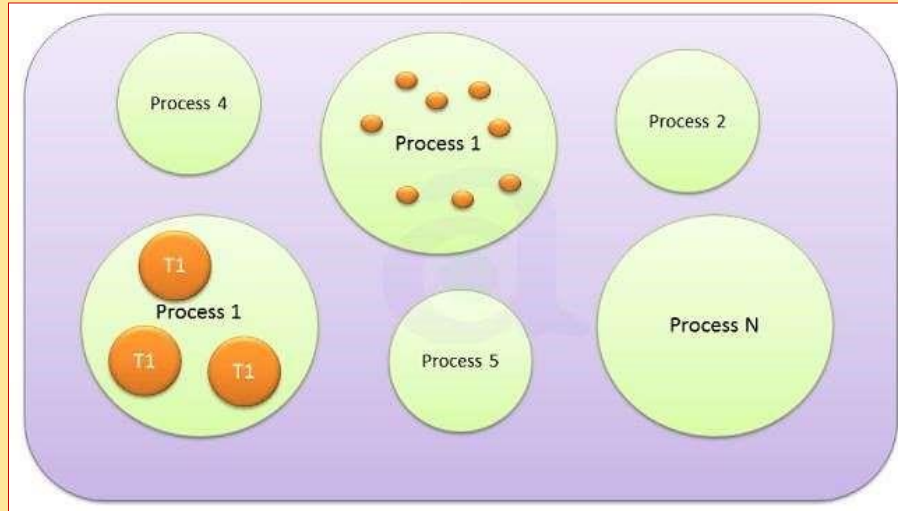
- JAVA is a multi-threaded programming language which means we can develop multi-threaded program using Java.
- A multi-threaded program contains **two or more parts** that can run concurrently and each part can **handle a different task** at the same time making optimal use of the available resources specially when your computer has multiple CPUs.
- **Each part of such program is called a thread.** So, threads are **lightweight processes** within a process.

2

- Multiprocessing and multithreading, both are used to achieve **multitasking** But we use multithreading than multiprocessing because **threads share a common memory area**.
- They don't allocate separate memory area so **saves memory**, and **context-switching** between the threads takes less time than process.
- Java Multithreading is mostly used in **games, animation** etc..
- A thread is a **lightweight sub process**, a smallest unit of processing.
- It is a **separate path of execution**.
- They are **independent**, if there occurs exception in one thread, it doesn't affect other threads.

3

- ❑ At least one process is required for each thread.



4

❑ Advantages of Java Multithreading

- ❑ It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.
- ❑ You **can perform many operations together so it saves time**.
- ❑ Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.
- ❑ Note: At a time one thread is executed only.

5

LIFE CYCLE OF THREAD

- A thread can be in one of the **five states**.
- According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated.
- There is no running state. But for better understanding the threads, we can explain it in the 5 states.

❓ New

❓ Runnable

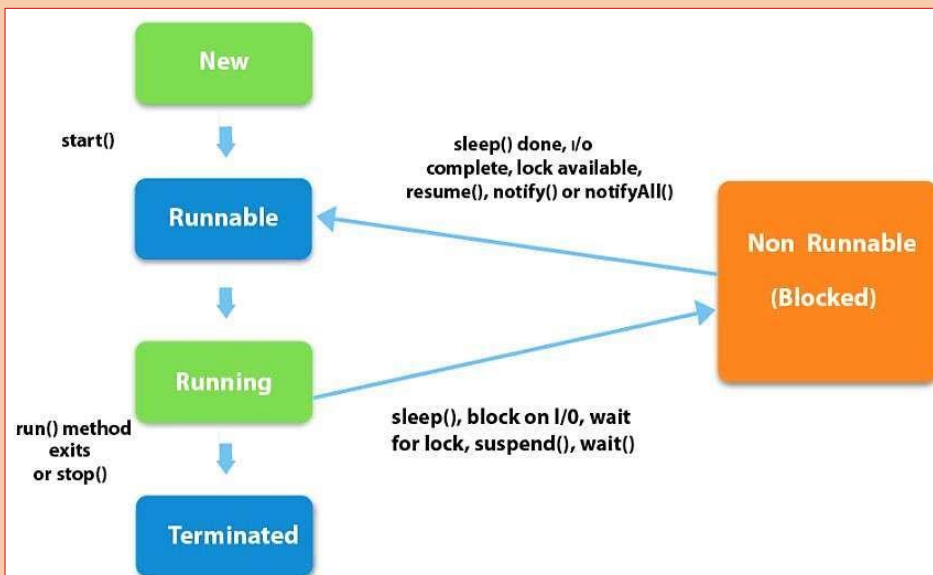
❓ Running

❓ Non-Runnable (Blocked)

❓ Terminated

6

Life Cycle



7

❏ **New** - The thread is in new state if you **create an instance of Thread class** but before the invocation of **start()** method.

❏ **Runnable** - The thread is in runnable state after invocation of **start()** method, but the thread scheduler has not selected it to be the running thread.

❏ **Running** - The thread is in running state if the thread scheduler has selected it.

❏ **Non-Runnable (Blocked)** - This is the state when the thread is still **alive**, but is currently **not eligible to run**.

❏ **Terminated** - A thread is in terminated or dead state when its **run()** method exits.

8

❏ A Running Thread transit to one of the non-runnable states, depending upon the circumstances.

- **Sleeping**: The Thread sleeps for the specified amount of time.
- **Blocked for I/O**: The Thread waits for a blocking operation to complete.
- **Blocked for join completion**: The Thread waits for completion of another Thread.
- **Waiting for notification**: The Thread waits for notification another Thread.
- **Blocked for lock acquisition**: The Thread waits to acquire the lock of an object.

❏ JVM executes the Thread, based on their priority and scheduling.

9

CREATING THREAD

? There are two ways to create a thread:

- By extending **Thread class**
- By implementing **Runnable interface** .

? Extending Thread class:

- Thread class provide constructors and methods to create and perform operations on a thread.
- Thread class extends Object class and implements Runnable interface.

10

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

11

Thread Methods - Following is the list of important methods available in the Thread class.

- `public void run()` : is used to perform action for a thread.
- `public void start()` : starts the execution of the thread. JVM calls the `run()` method on the thread.
- `public void sleep(long milliseconds)` : Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- `public void join()` : waits for a thread to die.
- `public int getPriority()` : returns the priority of the thread.
- `public int setPriority(int priority)` : changes the priority of the thread.

12

- `public String getName()`: returns the name of the thread.
- `public Thread currentThread()` : returns the reference of currently executing thread.
- `public int getId()` : returns the id of the thread.
- `public Thread.State getState()` : returns the state of the thread.
- `public boolean isAlive()` : tests if the thread is alive.
- `public void suspend()` : is used to suspend the thread(deprecated). • `public void resume()` : is used to resume the suspended thread
- `public void stop()` : is used to stop the thread(deprecated).
- `public boolean isDaemon()` : tests if the thread is a daemon thread.

13

□ Thread.start() & Thread.run()

❓ In Java's multi-threading concept, **start()** and **run()** are the two most important methods.

- When a program calls the start() method, a new thread is created and then the run() method is executed.
- But if we directly call the run() method then no new thread will be created and run() method will be executed as a normal method call on the current calling thread itself and no multi-threading will take place.

14

❓ Let us understand it with an example:

```
class MyThread extends Thread {  
    public void run()  
    {  
        System.out.println("Current thread name: "  
            + Thread.currentThread().getName());  
        System.out.println("run() method called");  
    }  
}  
class Xyz {  
    public static void main(String[] args)  
    {  
        MyThread t = new MyThread();  
        t.start();  
    }  
}
```

Output

Output:

```
Current thread name: Thread-0  
run() method called
```

15

start ()

When we call the start() method of our thread class instance, a new thread is created with default name **Thread-0** and then run() method is called and everything inside it is executed on the newly created thread.

run ()

When we called the run() method of our MyThread class, no new thread is created and the run() method is executed on the current thread i.e. main thread. Hence, no multi-threading took place. The run() method is called as a normal function call.

16

Let us try to call run() method directly instead of start() method:

```
class MyThread extends Thread {
    public void run()
    {
        System.out.println("Current thread name: "
                           + Thread.currentThread().getName());
        System.out.println("run() method called");
    }
}
class Xyz {
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.run();
    }
}
```

Output

```
Current thread name: main
run() method called
```

17

Difference

START()	RUN()
Creates a new thread and the run() method is executed on the newly created thread.	No new thread is created and the run() method is executed on the calling thread itself.
Can't be invoked more than one time otherwise throws <i>java.lang.IllegalStateException</i>	Multiple invocation is possible
Defined in <i>java.lang.Thread</i> class.	Defined in <i>java.lang.Runnable</i> interface and must be overridden in the implementing class.

18

❓ Implementing Runnable interface:

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.
- Runnable interface have only one method named **run()**.

`public void run();` is used to perform action for a thread.

❓ Steps to create a new Thread using Runnable :

- Create a Runnable implementer and implement run() method.
- Instantiate Thread class and pass the implementer to the Thread, Thread has a constructor which accepts Runnable instance.
- Invoke start() of Thread instance, start internally calls run() of the implementer. Invoking start(), creates a new Thread which executes the code written in run().

19

❓ Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{  
    public void run(){  
        System.out.println("thread is running...");  
    }  
  
    public static void main(String args[]){  
        Multi3 m1=new Multi3();  
        Thread t1 =new Thread(m1);  
        t1.start();  
    }  
}
```

Output:thread is running...

30

MAIN THREAD

❓ Every java program has a main method. The main method is the entry point to execute the program.

❓ So, when the JVM starts the execution of a program, it creates a thread to run it and that thread is known as the **main thread**.

❓ Each program must contain at least one thread whether we are creating any thread or not.

❓ The **JVM provides** a default thread in each program.

❓ A program can't run without a thread, so it requires at least one thread, and that thread is known as the main thread.

21

- ❓ If you ever tried to run a Java program with compilation errors you would have seen the mentioning of main thread. Here is a simple Java program that tries to call the non-existent `getValue()` method.

```
public class TestThread {  
    public static void main(String[] args) {  
        TestThread t = new TestThread();  
        t.getValue();  
    }  
}
```

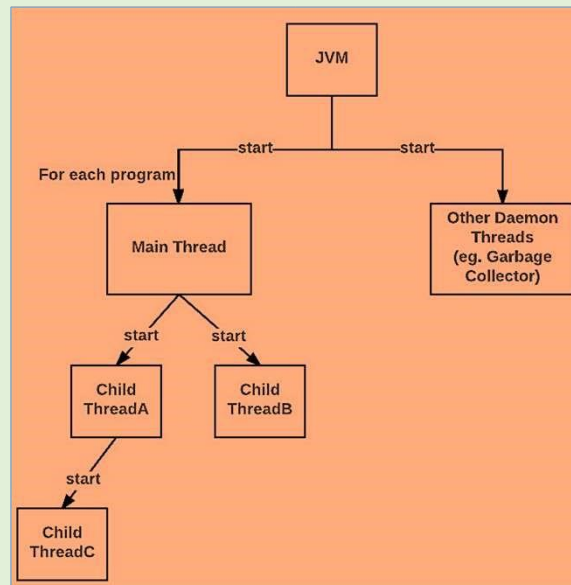
```
Exception in thread "main" java.lang.Error: Unresolved compilation  
The method getValue() is undefined for the type TestThread
```

- ❓ As you can see in the error when the program is executed, main thread starts running and that has encountered a compilation problem.

22

❓ Properties

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions



23

□ How to control Main thread

- The main thread is **created automatically** when our program is started.
- To control it we must obtain a **reference** to it.
- This can be done by calling the method **currentThread()** which is present in Thread class.
- This method returns a reference to the thread on which it is called.
- The **default priority of Main thread is 5** and for all remaining user threads priority will be inherited from parent to child.

24

```
class MainThread
{
    public static void main(String args [] )
    {
        Thread t = Thread.currentThread ( );
        System.out.println ("Current Thread : " + t);
        System.out.println ("Name : " + t.getName ( ) );
        System.out.println (" ");
        t.setName ("New Thread");
        System.out.println ("After changing name");
        System.out.println ("Current Thread : " + t);
        System.out.println ("Name : " + t.getName ( ) );
        System.out.println (" ");
        System.out.println ("This thread prints first 10 numbers");
        try
        {
            for (int i=1; i<=10;i++)
            {
                System.out.print(i);
                System.out.print(" ");
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println(e);
        }
    }
}
```

Output

```
Current Thread : Thread[main,5,main]
Name : main

After changing name
Current Thread : Thread[New Thread,5,main]
Name : New Thread

This thread prints first 10 numbers
1 2 3 4 5 6 7 8 9 10
```

25

- The program first creates a Thread object called 't' and assigns the reference of current thread (main thread) to it. So now main thread can be accessed via Thread object 't'.
- This is done with the help of `currentThread()` method of Thread class which return a reference to the current running thread.
- The Thread object 't' is then printed as a result of which you see the output Current Thread : Thread [main,5,main].
- The first value in the square brackets of this output indicates the name of the thread, the name of the group to which the thread belongs.

- The program then prints the name of the thread with the help of `getName()` method.
- The name of the thread is changed with the help of `setName()` method.
- The thread and thread name is then again printed.
- Then the thread performs the operation of printing first 10 numbers.
- When you run the program you will see that the system wait for sometime after printing each number.
- This is caused by the statement `Thread.sleep (1000)`.

CREATING MULTIPLE THREADS

```
class ThreadA extends Thread
{
    public void run()
    {
        for (int i=1;i<=5;i++)
        {
            System.out.println("ThreadA i="+(-1*i));
        }
        System.out.println("Exiting ThreadA");
    }
}
class ThreadB extends Thread
{
    public void run()
    {
        for (int j=1;j<=5;j++)
        {
            System.out.println("ThreadB j="+(-2*j));
        }
        System.out.println("Exiting ThreadB");
    }
}
```

```
class ThreadC extends Thread
{
    public void run()
    {
        for (int k=1;k<=5;k++)
        {
            System.out.println("ThreadC k="+(-2*(k-1)));
        }
        System.out.println("Exiting ThreadC");
    }
}
class MultiThreadDemo
{
    public static void main (String args [])
    {
        ThreadA t1 = new ThreadA();
        ThreadB t2 = new ThreadB();
        ThreadC t3 = new ThreadC();
        t1.start();
        t2.start();
        t3.start();
    }
}
```

28

Output

ThreadA i = -1	ThreadA i = -4
ThreadB j = 2	ThreadB j = 8
ThreadC k = 1	ThreadC k = 7
ThreadA i = -2	ThreadA i = -5
ThreadB j = 4	ThreadB j = 10
ThreadC k = 3	ThreadC k = 9
ThreadA i = -3	Exiting ThreadA
ThreadB j = 6	Exiting ThreadB
ThreadC k = 5	Exiting ThreadC

THREAD SYNCHRONIZATION

- When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues.
- For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

30

- So there is a **need to synchronize the action of multiple threads** and make sure that only one thread can access the resource at a given point in time.

Following is the general form of the synchronized statement :

```
Syntax synchronized(object identifier) {  
    // Access shared variables and other shared resources  
}
```

❓ Understanding the problem without Synchronization

- In this example, we are not using synchronization and creating multiple threads that are accessing display method and produce the random output.

31


```

class First {
    public void display(String msg)
    {
        System.out.print ("["+msg);
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println ("]");
    }
}

class Second extends Thread {
    String msg;
    First fobj;
    Second (First fp,String str) {
        fobj = fp;
        msg = str;
        start();
    }
    public void run() {
        fobj.display(msg);
    }
}

```

```

public class Syncro
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new Second(fnew, "welcome");
        Second ss1= new Second(fnew,"new");
        Second ss2 = new Second(fnew, "programmer");
    }
}

```

In the above program, object fnew of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(void display). Hence the result is nonsynchronized and such situation is called Race condition

OUTPUT:

```

[welcome [ new [ programmer]
]
]

```

☑Synchronized Keyword

- To synchronize above program, we must synchronize access to the shared display() method, making it available to only one thread at a time. This is done by using keyword **synchronized** with display() method.
- With a synchronized method, the lock is obtained for the duration of the entire method.
- So if you want to lock the whole object, use a synchronized method **synchronized void display (String msg)**

Example : implementation of synchronized method

```

class First
{
    synchronized public void display(String msg) {
        System.out.print ("["+msg);
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println ("]");
    }
}

class Second extends Thread {
    String msg;
    First fobj;
    Second (First fp,String str) {
        fobj = fp;
        msg = str;
        start();
    }
    public void run() {
        fobj.display(msg);
    }
}

```

```

public class MyThread
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new Second(fnew, "welcome");
        Second ss1= new Second(fnew,"new");
        Second ss2 = new Second(fnew, "programmer");
    }
}

```

OUTPUT:

```

[welcome]

[programmer]

[new]

```

34

❑ Using Synchronized block

- If we want to synchronize access to an object of a class or only a part of a method to be synchronized then we can use synchronized block for it.
- It is capable to make any part of the object and method synchronized.
- With synchronized blocks we can specify exactly when the lock is needed. If you want to keep other parts of the object accessible to other threads, use synchronized block. [Example](#)
- In this example, we are using synchronized block that will make the display method available for single thread at a time.

35

```

class First {
    public void display(String msg) {
        System.out.print ("["+msg);
        try {
            Thread.sleep(1000);
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println ("]");
    }
}
class Second extends Thread {
    String msg;
    First fobj;
    Second (First fp,String str) {
        fobj = fp;
        msg = str;
        start();
    }
    public void run() {
        synchronized(fobj)    //Synchronized block
        {
            fobj.display(msg);
        }
    }
}

```

```

public class MyThread
{
    public static void main (String[] args)
    {
        First fnew = new First();
        Second ss = new Second(fnew, "welcome");
        Second ss1= new Second (fnew,"new");
        Second ss2 = new Second(fnew, "programmer");
    }
}

```

OUTPUT:

```

[welcome]
[new]
[programmer]

```

36

Which is more preferred - Synchronized method or Synchronized block?

- In Java, synchronized keyword causes a performance cost.
- A synchronized method in Java is very slow and can degrade performance.
- So we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.

37

Thread suspend() method

- The suspend() method of thread class puts the thread from running to waiting state.

- This method is used if you want to stop the thread execution and start it again when a certain event occurs.
- This method allows a thread to temporarily cease execution.
- The suspended thread can be resumed using the resume() method.

Syntax `public final void suspend()`

38

Example

```
public class JavaSuspendExp extends Thread
{
    public void run()
    {
        for(int i=1; i<5; i++)
        {
            try
            {
                // thread to sleep for 500 milliseconds
                sleep(500);
                System.out.println(Thread.currentThread().getName());
            } catch (InterruptedException e) { System.out.println(e); }
            System.out.println(i);
        }
    }
}
```

```
public static void main(String args[])
{
    // creating three threads
    JavaSuspendExp t1=new JavaSuspendExp ();
    JavaSuspendExp t2=new JavaSuspendExp ();
    JavaSuspendExp t3=new JavaSuspendExp ();
    // call run() method
    t1.start();
    t2.start();
    // suspend t2 thread
    t2.suspend();
    // call run() method
    t3.start();
}
```

39

Output

```
Thread-0  
1  
Thread-2  
1  
Thread-0  
2  
Thread-2  
2  
Thread-0  
3  
Thread-2  
3  
Thread-0  
4  
Thread-2  
4
```

40

Thread resume() method

- The resume() method of thread class is only used with suspend() method.
- This method is used to resume a thread which was suspended using suspend() method.
- This method allows the suspended thread to start again.

Syntax

```
public final void resume()
```

41

Example

```
public class JavaResumeExp extends Thread
{
    public void run()
    {
        for(int i=1; i<5; i++)
        {
            try
            {
                // thread to sleep for 500 milliseconds
                sleep(500);
                System.out.println(Thread.currentThread().getName());
            } catch (InterruptedException e) { System.out.println(e); }
            System.out.println(i);
        }
    }
}
```

```
public static void main(String args[])
{
    // creating three threads
    JavaResumeExp t1=new JavaResumeExp ();
    JavaResumeExp t2=new JavaResumeExp ();
    JavaResumeExp t3=new JavaResumeExp ();
    // call run() method
    t1.start();
    t2.start();
    t2.suspend(); // suspend t2 thread
    // call run() method
    t3.start();
    t2.resume(); // resume t2 thread
}
```

42

Output

```
Thread-0
1
Thread-2
1
Thread-1
1
Thread-0
2
Thread-2
2
Thread-1
2
Thread-0
```

```
3
Thread-2
3
Thread-1
3
Thread-0
4
Thread-2
4
Thread-1
4
```

43

Thread stop() method

- The stop() method of thread class terminates the thread execution.
- Once a thread is stopped, it cannot be restarted by start() method.

Syntax

```
public final void stop()
public final void stop(Throwables obj)
```

44

Example

```
public class JavaStopExp extends Thread
{
    public void run()
    {
        for(int i=1; i<5; i++)
        {
            try
            {
                // thread to sleep for 500 milliseconds
                sleep(500);
                System.out.println(Thread.currentThread().getName());
            } catch (InterruptedException e) { System.out.println(e); }
            System.out.println(i);
        }
    }
}
```

```
public static void main(String args[])
{
    // creating three threads
    JavaStopExp t1=new JavaStopExp ();
    JavaStopExp t2=new JavaStopExp ();
    JavaStopExp t3=new JavaStopExp ();
    // call run() method
    t1.start();
    t2.start();
    // stop t3 thread
    t3.stop();
    System.out.println("Thread t3 is stopped");
}
```

45

MODULE 4

CHAPTER 3

EVENT HANDLING

1

EVENT

- Change in the state of an object is known as event i.e. event describes the change in state of source.
- Events are generated as result of user interaction with the graphical user interface components.
- For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

Types of Event

The events can be broadly classified into two categories:

2

Foreground Events

- Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

Background Events

- Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

3

EVENT HANDLING

- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- This mechanism have the code which is known as **event handler** that is executed when an event occurs.
- Java Uses the **Delegation Event Model** to handle the events.
- This model defines the standard mechanism to generate and handle the events.
- Let's have a brief introduction to this model.

4

The Delegation Event Model has the following key participants namely:

Source - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

Listener - It is also known as **event handler**. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event and then returns.

5

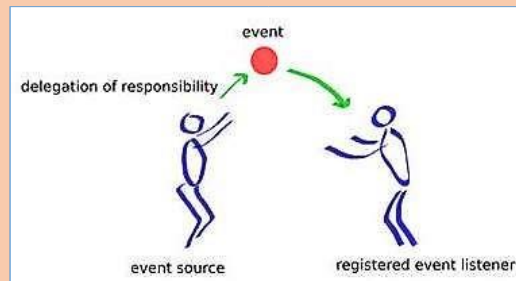
- The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event.
- The user interface element is able to delegate the processing of an event to the separate piece of code.

- In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification.
- This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

6

How Events are handled

- A source generates an Event and send it to one or more listeners registered with the source.
- Once event is received by the listener, they process the event and then return.
- Events are supported by a number of Java packages, like java.util, java.awt and java.awt.event



? Event classes and interface

Event Classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved, clicked, pressed or released and also when it enters or exits a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of text area or text field is changed	TextListener
MouseEvent	generated when mouse wheel is moved	MouseWheelListener

8

WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

Steps to handle events:

- ? Implement appropriate interface in the class.
- ? Register the component with the listener.

9

□ Steps involved in event handling

1. The User clicks the button and the event is generated.

2. Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.

3. Event object is forwarded to the method of registered listener class.

4. The method is now get executed and returns.

□ Points to remember about listener

- In order to design a listener class we have to develop some listener interfaces.
- These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.
- If we do not implement the predefined interfaces then your class can not act as a listener class for a source object.

11

SOURCES OF EVENT

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

12

EVENT LISTENER INTERFACES

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

13

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

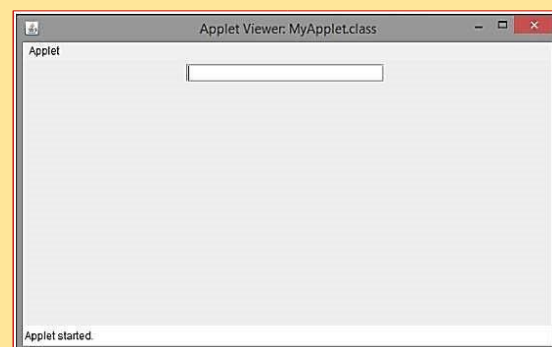
public class MyApplet extends JApplet implements KeyListener
{
    JTextField jtf;
    JLabel label;

    public void init()
    {
        setSize(600,300);
        setLayout(new FlowLayout());
        jtf = new JTextField(20);
        add(jtf);
        jtf.addKeyListener(this);
        label = new JLabel();
        add(label);
    }

    public void keyPressed(KeyEvent ke){}
    public void keyReleased(KeyEvent ke){}
    public void keyTyped(KeyEvent ke)
    {
        label.setText(String.valueOf(ke.getKeyChar()));
    }
}
```

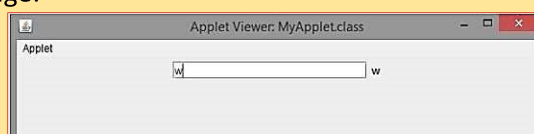
Key Event Handling

Initial output of the program



14

After the user enters a character into the text field, the same character is displayed in the label beside the text field as shown in the below image:



MODULE 5

Graphical User Interface & support of Java

CHAPTER 1

SWING

1

SWING FUNDAMENTALS

- Java Swing is a **GUI Framework** that contains a set of classes to provide more powerful and flexible GUI components than AWT.
- Swing provides the look and feel of modern Java GUI.
- Swing library is an official Java GUI tool kit released by Sun Microsystems.
- It is used to create graphical user interface with Java.
- Swing classes are defined in **javax.swing** package and its subpackages.
- Java Swing provides **platform-independent** and **lightweight** components.

2

Java Swing is a part of Java Foundation Classes (JFC) that is **used to create window-based applications**.

It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java

JFC

- The Java Foundation Classes (JFC) are a set of GUI components which simplify the development of desktop applications.

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

3

☐Features of Swing

Platform Independent:

- It is platform independent, the swing components that are used to build the program are not platform specific.
- It can be used at **any platform** and **anywhere**.

Lightweight:

- Swing components are lightweight which helps in creating the UI lighter.
- Swings component allows it to plug into the operating system user interface framework that includes the mappings for screens or device and other user interactions like key press and mouse movements.

4

Plugging:

- It has a powerful component that can be extended to provide the support for the user interface that helps in good look and feel to the application.
- It refers to the highly **modular-based architecture** that allows it to plug into other customized implementations and framework for user interfaces.

Manageable: It is easy to manage and configure. Its mechanism and composition pattern allows changing the settings at run time as well. The uniform changes can be provided to the user interface without doing any changes to application code.

MVC:

- They mainly follow the concept of MVC that is **Model View Controller**.
- With the help of this, we can do the changes in one component without impacting or touching other components.
- It is known as loosely coupled architecture as well.

Customizable:

- Swing controls can be easily customized. It can be changed and the visual appearance of the swing component application is independent of its internal representation.

Rich Controls :

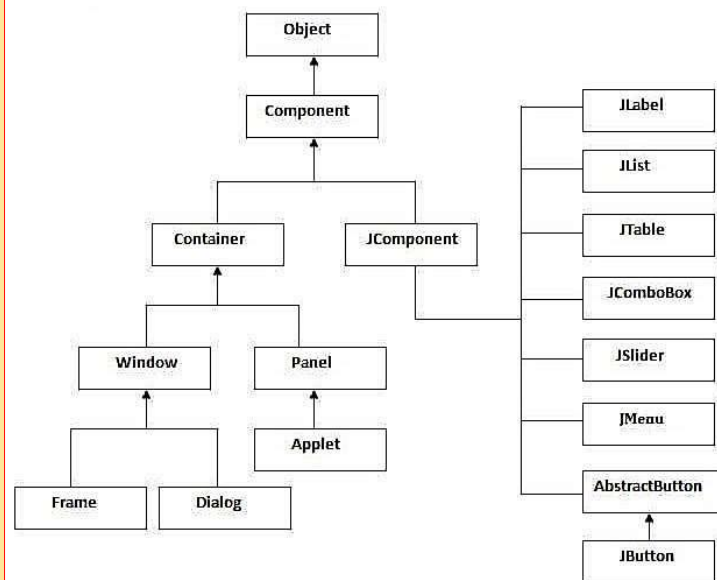
- Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, and table controls.

❓ Difference between AWT and Swing

No.	Java AWT	Java Swing
1)	AWT components are platform-dependent .	Java swing components are platform-independent .
2)	AWT components are heavyweight .	Swing components are lightweight .
3)	AWT doesn't support pluggable look and feel .	Swing supports pluggable look and feel .
4)	AWT provides less components than Swing.	Swing provides more powerful components such as tables, lists, scrollpanes, colorchooser, tabbedPane etc.
5)	AWT doesn't follows MVC (Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing follows MVC .

7

❓ Hierarchy of Java Swing classes



8

The Model-View-Controller Architecture

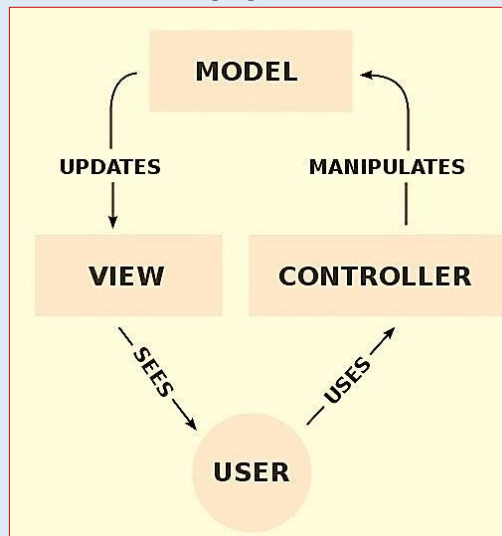
- Swing uses the model-view-controller architecture (MVC) as the fundamental design behind each of its components
- Essentially, MVC breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves.
- The Model-View-Controller is a well known **software architectural pattern** ideal to implement user interfaces on computers by dividing an application into three interconnected parts

9

- **Main goal** of Model-View-Controller, also known as MVC, is to separate internal representations of an application from the ways information are presented to the user.
- Initially, MVC was designed for desktop GUI applications but it's quickly become an extremely popular pattern for designing web applications too.
- MVC pattern has the **three components** :
 - Model** that manages data, logic and rules of the application **View** that is used to present data to user
 - Controller** that accepts input from the user and converts it to commands for the Model or View.

10

- the MVC pattern defines the interactions between these three components like you can see in the following figure :



11

- The Model receives commands and data from the Controller. It stores these data and updates the View.
- The View lets to present data provided by the Model to the user.
- The Controller accepts inputs from the user and converts it to commands for the Model or the View.

12

COMPONENTS & CONTAINERS

- A **component** is an independent visual control, such as a push button or slider.
- A **container** holds a group of components. Thus, a container is a special type of component that is designed to hold other components.
- Swing components inherit from the **javax.Swing.JComponent** class, which is the root of the Swing component hierarchy.

13

□ COMPONENTS

- Swing components are derived from the **JComponent** class.
- JComponent provides the functionality that is common to all components. For example, JComponent supports the pluggable look and feel.
- JComponent inherits the AWT classes Container and Component. Thus, a Swing component is built on and compatible with an AWT component.
- All of Swing's components are represented by classes defined within the package **javax.swing**.
- The following table shows the class names for Swing components

14

- JApplet
- JColorChooser
- JDialog
- JFrame
- JLayeredPane
- JMenuItem
- JPopupMenu
- JRootPane
- JSlider
- JTable
- JToggleButton
- JViewport
- JButton
- JComboBox
- JEditorPane
- JInternalFrame
- JList
- JOptionPane
- JProgressBar

15

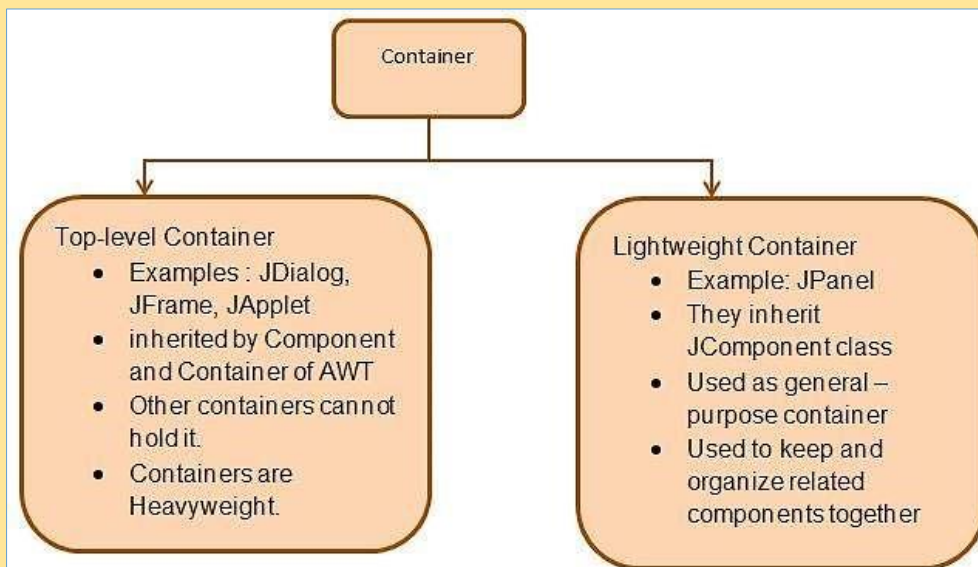
- Notice that all component classes begin with the letter J.
- For example, the class for a label is JLabel; the class for a push button is JButton; and the class for a scroll bar is JScrollBar

□ CONTAINERS

- Swing defines two types of containers. The first are top-level containers: JFrame, JApplet, JWindow, and JDialog. These containers do not inherit JComponent. They inherit the AWT classes Component and Container.
- The second type container are lightweight and the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.

16

In Java, Containers are divided into two types as shown below:



17

Following is the list of commonly used containers while designed GUI using SWING.

Sr.No.	Container & Description
1	Panel JPanel is the simplest container. It provides space in which any other component can be placed, including other panels.
2	Frame A JFrame is a top-level window with a title and a border.
3	Window A JWindow object is a top-level window with no borders and no menubar.

18

Swing Example : A window on the screen.

```
import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class Example extends JFrame {

    public Example() {
        setTitle("Simple example");
        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public static void main(String[] args) {
        Example ex = new Example();
        ex.setVisible(true);
    }
}
```

Output



19

EVENT HANDLING IN SWINGS

- The functionality of Event Handling is what is the further step if an action performed.
- Java foundation introduced “**Delegation Event Model**” i.e describes how to generate and control the events.
- The key elements of the Delegation Event Model are as **source** and **listeners**.
- The listener should have registered on source for the purpose of alert notifications.
- **All GUI applications are event-driven**

20

□ Java Swing event object

- When something happens in the application, an **event object** is created.
- For example, when we click on the button or select an item from a list.

- There are several types of events, including `ActionEvent`, `TextEvent`, `FocusEvent`, and `ComponentEvent`.
- Each of them is created under specific conditions.
- An event object holds information about an event that has occurred.

21

SWING LAYOUT MANAGERS

- Layout refers to the arrangement of components within the container.
- Layout is placing the components at a particular position within the container. The task of laying out the controls is done automatically by the Layout Manager.
- The **layout manager automatically positions all the components within the container.**
- Even if you do not use the layout manager, the components are still positioned by the default layout manager. It is possible to lay out the controls by hand, however, it becomes very difficult

22

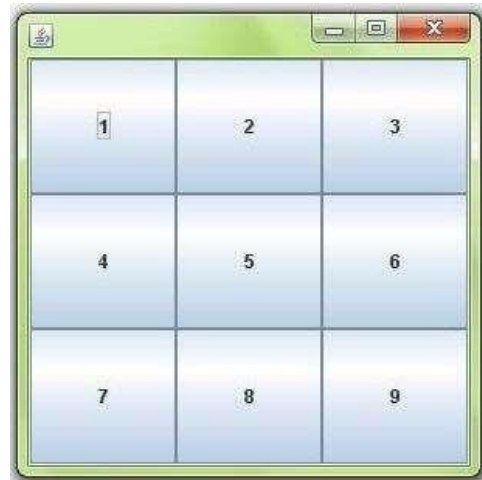
- Java provides various layout managers to position the controls. Properties like size, shape, and arrangement varies from one layout manager to the other.
- There are following classes that represents the **layout managers**:
 - java.awt.BorderLayout
 - java.awt.FlowLayout
 - java.awt.GridLayout
 - java.awt.CardLayout
 - java.awt.GridBagLayout
 - javax.swing.BoxLayout
 - javax.swing.GroupLayout
 - javax.swing.ScrollPaneLayout
 - javax.swing.SpringLayout etc.

23

BorderLayout

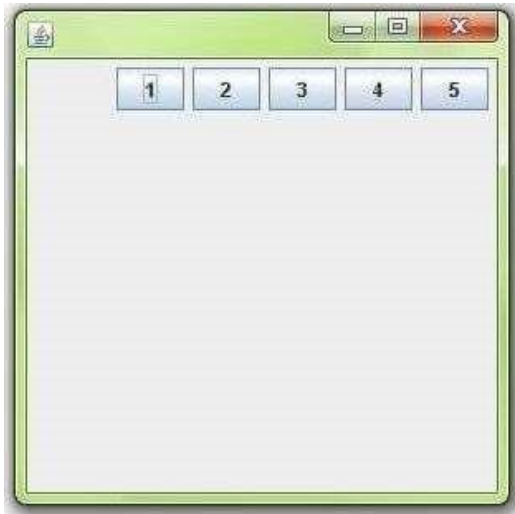


GridLayout

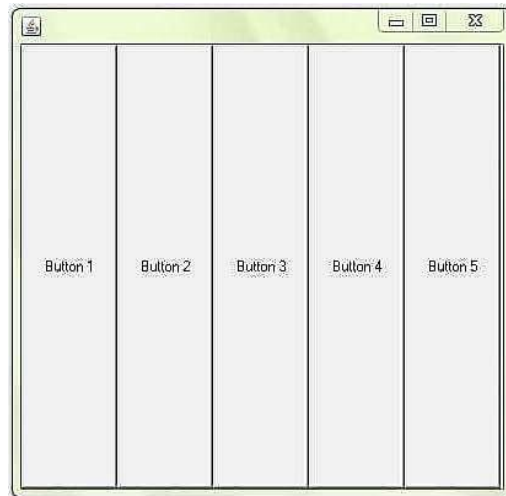


24

FlowLayout

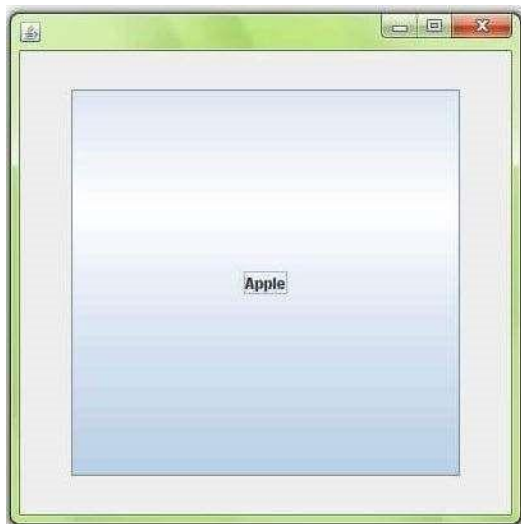


BoxLayout



25

CardLayout



GridLayout

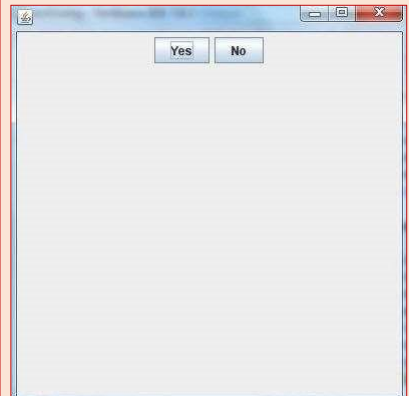


26

Example of JButton

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class testswing extends JFrame
{
    testswing()
    {
        JButton bt1 = new JButton("Yes");           //Creating a Yes Button.
        JButton bt2 = new JButton("No");           //Creating a No Button.
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE) //setting close operation.
        setLayout(new FlowLayout());               //setting layout using FlowLayout object
        setSize(400, 400);                          //setting size of JFrame
        add(bt1);                                    //adding Yes button to frame.
        add(bt2);                                    //adding No button to frame.

        setVisible(true);
    }
    public static void main(String[] args)
    {
        new testswing();
    }
}
```



27

Example of JTextField

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class MyTextField extends JFrame
{
    public MyTextField()
    {
        JTextField jtf = new JTextField(20); //creating JTextField.
        add(jtf);                             //adding JTextField to frame.
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 400);
        setVisible(true);
    }
    public static void main(String[] args)
    {
        new MyTextField();
    }
}
```

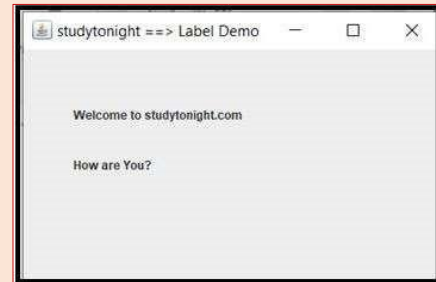


28

Example of JLabel

- It is used for placing text in a box

```
import javax.swing.*;
class SLabelDemo1
{
    public static void main(String args[])
    {
        JFrame label_f= new JFrame("studytonight ==> Label Demo");
        JLabel label_11,label_12;
        label_11=new JLabel("Welcome to studytonight.com");
        label_11.setBounds(50,50, 200,30);
        label_12=new JLabel("How are You?");
        label_12.setBounds(50,100, 200,30);
        label_f.add(label_11);
        label_f.add(label_12);
        label_f.setSize(300,300);
        label_f.setLayout(null);
        label_f.setVisible(true);
    }
}
```



29

MODULE 5

CHAPTER 2

JDBC

Java DataBase Connectivity (JDBC)

❑ JDBC stands for Java Database Connectivity, which is a standard **Java API** for database-independent connectivity between the Java programming language and a wide range of databases.

❑ The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database
- Creating SQL or MySQL statements
- Executing SQL or MySQL queries in the database
- Viewing & Modifying the resulting records

2

❓JDBC Architecture

❓JDBC Architecture consists of **two layers**

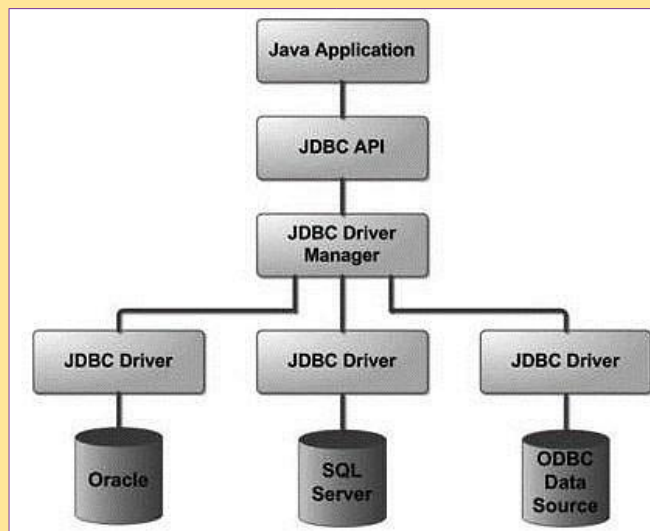
- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

❓The JDBC API uses a **driver manager** and **database-specific drivers** to provide transparent connectivity to heterogeneous databases.

❓The JDBC driver manager ensures that the correct driver is used to access each data source.

3

- Following is the **architectural diagram** , which shows the location of the driver manager with respect to the JDBC drivers and the Java application



Java Database Connectivity with 5 Steps

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

- 1 . Register the Driver class
- 2 . Create connection
- 3 . Create statement
- 4 . Execute queries
- 5 . Close connection

The `forName()` method is used to register the driver class.

The `getConnection()` method of `DriverManager` class is used to establish connection with the database

The `createStatement()` method of `Connection` interface is used to create statement. The object of statement is responsible to execute queries with the database.

The `executeQuery()` method of `Statement` interface is used to execute queries to the database. This method returns the object of `ResultSet` that can be used to get all the records of a table.

By closing connection object statement and `ResultSet` will be closed automatically. The `close()` method of `Connection` interface is used to close the connection.

Java Database Connectivity with MySQL

- To connect Java application with the MySQL database, we need to follow 5 following steps.
- In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is `com.mysql.jdbc.Driver`.
2. **Connection URL:** The connection URL for the mysql database is `jdbc:mysql://localhost:3306/sonoo` where `jdbc` is the API, `mysql` is the database, `localhost` is the

server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, we need to replace the sonoo with our database name.

3. **Username:** The default username for the mysql database is **root**.

4. **Password:** It is the password given by the user at the time of installing the mysql database. In this example, we are going to use **root** as the password.

Let's first create a table in the mysql database, but before creating table, we need to create database first.

```
create database sonoo; use sonoo;
create table emp(id int(10),name varchar(40),age int(3));
```

8

Example to Connect Java Application with mysql database

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection(
"jdbc:mysql://localhost:3306/sonoo","root","root");
//here sonoo is database name, root is username and password
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

This example will
fetch all the records of
emp table.

Creating a sample MySQL database

```
create database SampleDB;
use SampleDB;
CREATE TABLE `users` (
  `user_id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(45) NOT NULL,
  `password` varchar(45) NOT NULL,
  `fullname` varchar(45) NOT NULL,
  `email` varchar(45) NOT NULL,
  PRIMARY KEY (`user_id`)
);
```

10

Connecting to the database

```
String dbURL = "jdbc:mysql://localhost:3306/sampledbs";
String username = "root";
String password = "secret";
try {
    Connection conn = DriverManager.getConnection(dbURL, username,
password);
    if (conn != null) {
        System.out.println("Connected");
    }
} catch (SQLException ex )
{
    ex.printStackTrace();
}
```

11

-
- Once the connection was established, we have a **Connection** object which can be used to create statements in order to execute SQL queries.
- In the above code, we have to close the connection explicitly after finish working with the database: `conn.close();`

❏ INSERT Statement Example

Let's write code to insert a new record into the table Users with following details:

username: bill password: secretpass
fullname: Bill Gates email:
bill.gates@microsoft.com

12

```
String sql = "INSERT INTO Users (username, password, fullname, email) VALUES (?, ?, ?, ?)";  
PreparedStatement statement = conn.prepareStatement(sql); statement.setString(1,  
"bill"); statement.setString(2, "secretpass"); statement.setString(3, "Bill Gates");  
statement.setString(4, "bill.gates@microsoft.com"); int rowsInserted =  
statement.executeUpdate(); if (rowsInserted > 0) {  
    System.out.println("A new user was inserted successfully!");  
}
```

SELECT Statement Example

```
String sql = "SELECT * FROM Users";  
Statement statement = conn.createStatement();  
ResultSet result = statement.executeQuery(sql);  
  
int count = 0;  
while (result.next()){  
    String name = result.getString(2);  
    String pass = result.getString(3);  
    String fullname = result.getString("fullname");  
    String email = result.getString("email");  
  
    String output = "User #%d: %s - %s - %s - %s";  
    System.out.println(String.format(output, ++count, name, pass, fullname, email));  
}
```

Output

User #1: bill - secretpass - Bill Gates - bill.gates@microsoft.com

14

? UPDATE Statement Example

```
String sql = "UPDATE Users SET password=?, fullname=?, email=? WHERE username=?";

PreparedStatement statement = conn.prepareStatement(sql);
statement.setString(1, "123456789");
statement.setString(2, "William Henry Bill Gates");
statement.setString(3, "bill.gates@microsoft.com");
statement.setString(4, "bill");

int rowsUpdated = statement.executeUpdate();
if (rowsUpdated > 0) {
    System.out.println("An existing user was updated successfully!");
}
```

15

? DELETE Statement Example

- The following code snippet will delete a record whose username field contains "bill"

```
String sql = "DELETE FROM Users WHERE username=?";

PreparedStatement statement = conn.prepareStatement(sql);
statement.setString(1, "bill");

int rowsDeleted = statement.executeUpdate();
if (rowsDeleted > 0) {
    System.out.println("A user was deleted successfully!");
}
```

16