CAPITAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, ISLAMABAD.

LAB MANUAL

V2.0 Spring 2024

SOFTWARE CONSTRUCTION AND DEVELOPMENT

(SE3511)

DEPARTMENT OF SOFTWARE ENGINEERING

Capital University of Science & Technology

Islamabad

Department of Software Engineering,

Faculty of Computing

Lab Course Development Team

■ Lab Designer

Ms. Fatima Waseem
Associate Lecturer
Faculty of Computing

Table of Contents

Lab 1: Overview of Unified Modeling Language (UML)

Lab 2: Project Management Tools for Agile Project.

Lab 3: Applying Software Project Construction Management Fundamentals.

Lab 4: Converting Design Diagrams into Meaningful Code.

Lab 5: Introduction to Concurrency and Concurrency Management.

Lab 6: Ensuring Robustness and Flexibility in your code: Introducing concepts of Mutability and Immutability and Generics in Java.

Lab 7: Mastering Recursion-Exploring Recursive Algorithms

Lab 8: Exception Handling

Lab 9: Refactoring

Lab 10: Understanding and applying SOLID Principles

Lab 11: Application Lab: Applying Previously Studied Concepts

Lab 12: JUnit Testing

Lab 13: JUnit Testing: Introduction to Test Suite

Lab 14: Introduction to GitHub and cloning repository

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 1: Overview of Unified Modeling Language (UML)

Unified Modeling Language (UML) Overview

Introduction Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of object-oriented software engineering. UML includes a set of graphic notation techniques to create visual models of object-oriented software systems. UML combines techniques from data modeling, business modeling, object modeling, and component modeling and can be used throughout the software development life-cycle and across different implementation technologies.

UML diagrams represent two different views of a system model: Static (or structural) view This view emphasizes the static structure of the system using objects, attributes, operations, and relationships. Ex: Class diagram, Composite Structure diagram. Dynamic (or behavioral) view This view emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. Ex: Sequence diagram, Activity diagram, State Machine diagram.

Diagrams Overview UML has 14 types of diagrams divided into multiple categories as shown in the figure below.

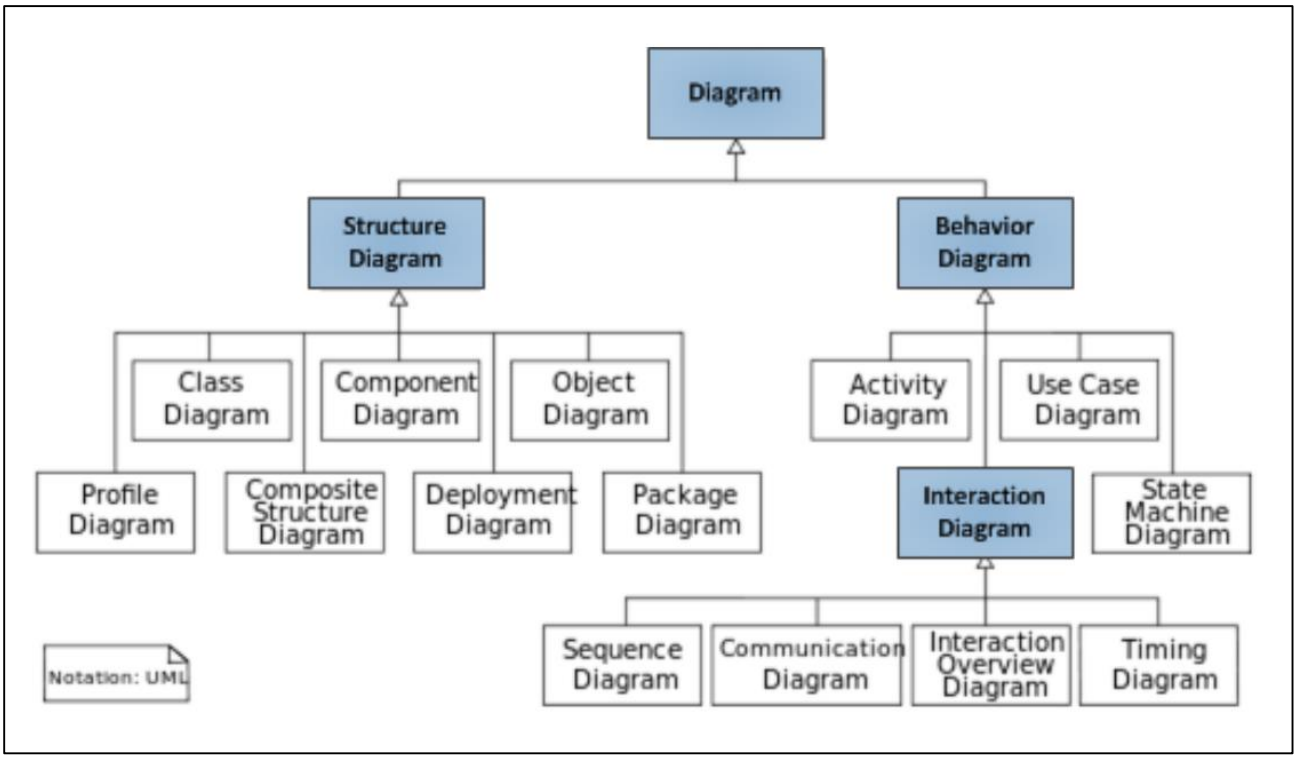


Figure 1 Overview UML

Structure Diagrams

These diagrams emphasize the things that must be present in the system being modeled. Since they represent the structure, they are used extensively in documenting the software architecture of software systems.

1. **Class Diagram**

   Describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.

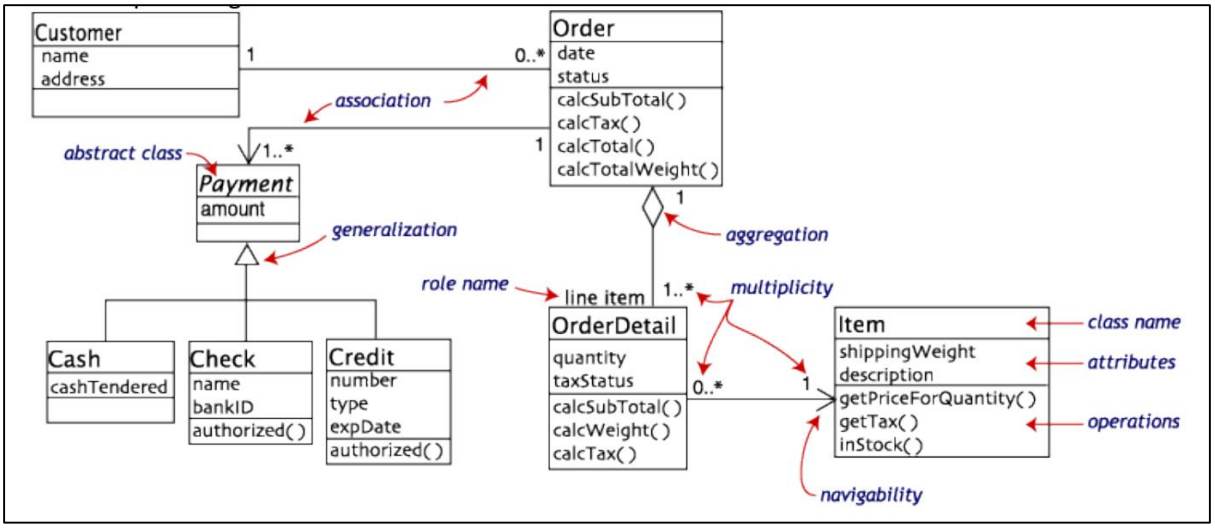


Figure 2 Class Diagram

2. Component Diagram

Describes how a software system is split-up into components and shows the dependencies among these components.

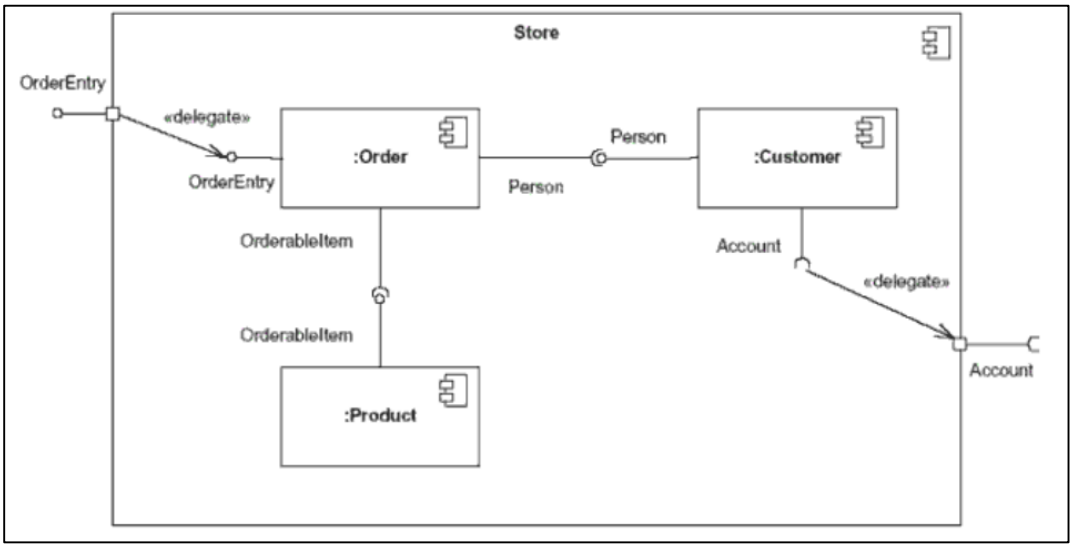


Figure 3 Component Diagram

Behavior Diagrams

These diagrams emphasize what must happen in the system being modeled. Since they illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

1. **Activity Diagram**
   Describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.

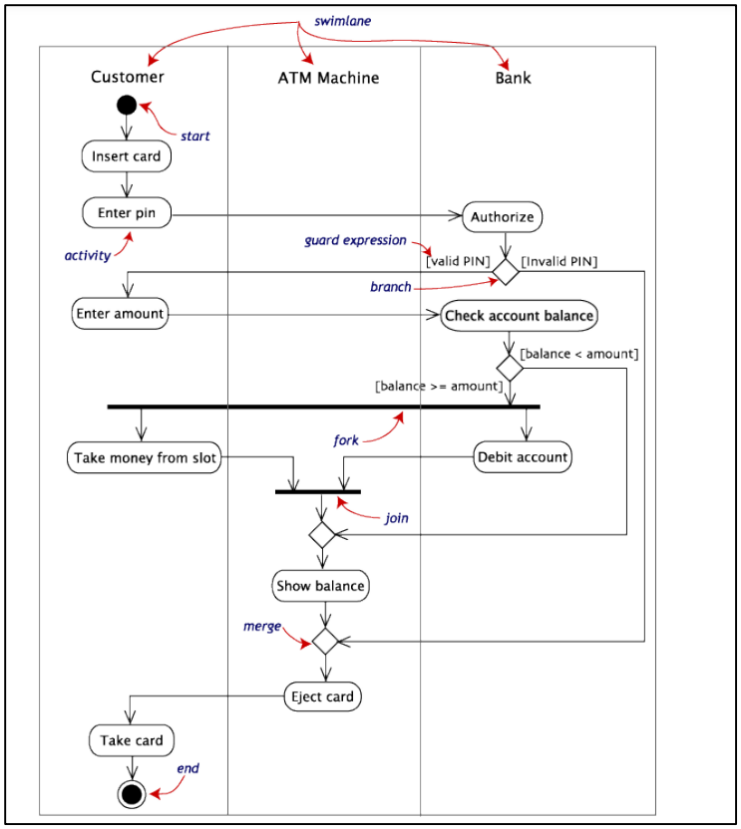


Figure 4 Activity Diagram

2. State Machine Diagram

Describes the states and state transitions of the system.

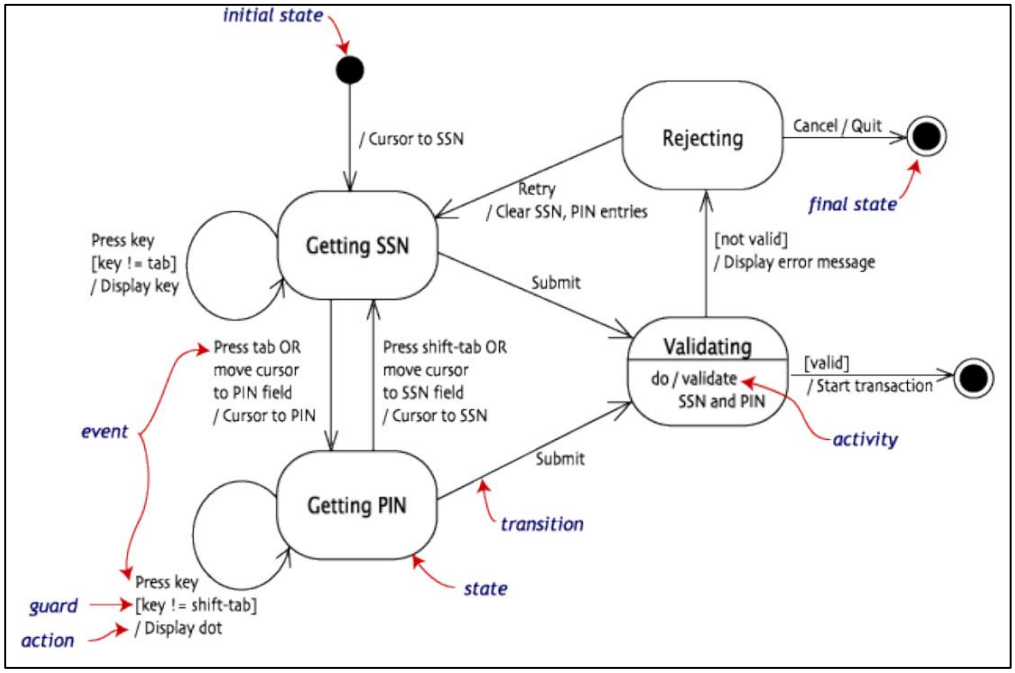


Figure 5 State Machine Diagram

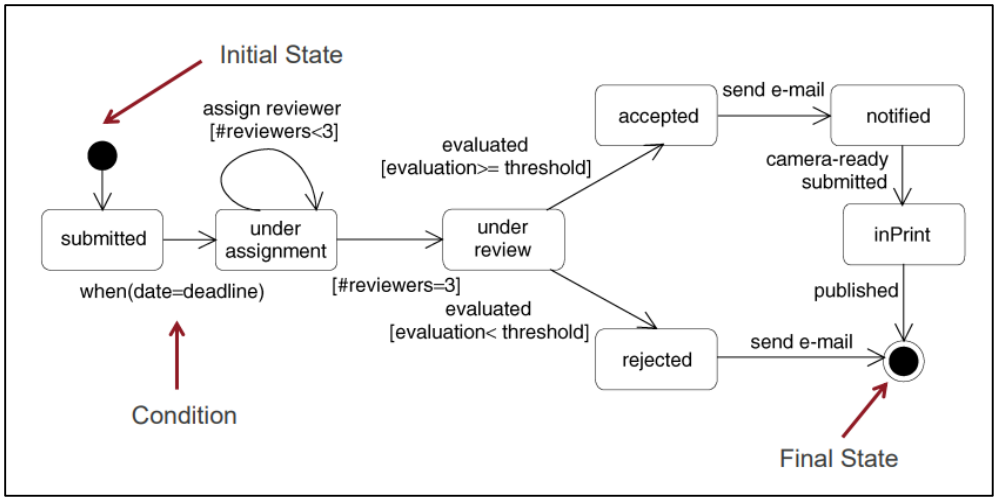


Figure 6 State Machine Diagram

3. **Use Case Diagram**
   Describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.

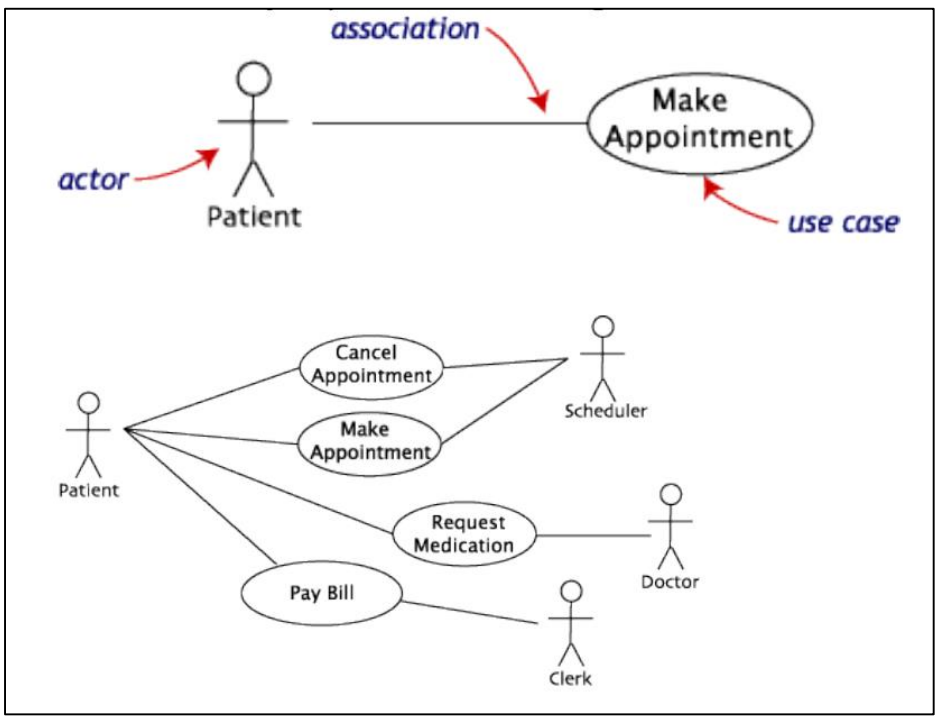


Figure 7 Use Case Diagram

<u>Interaction Diagrams</u>

These diagrams are a subset of behavior diagrams, emphasizing the flow of control and data among the things in the system being modeled.

1. **Sequence Diagram**

Shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.

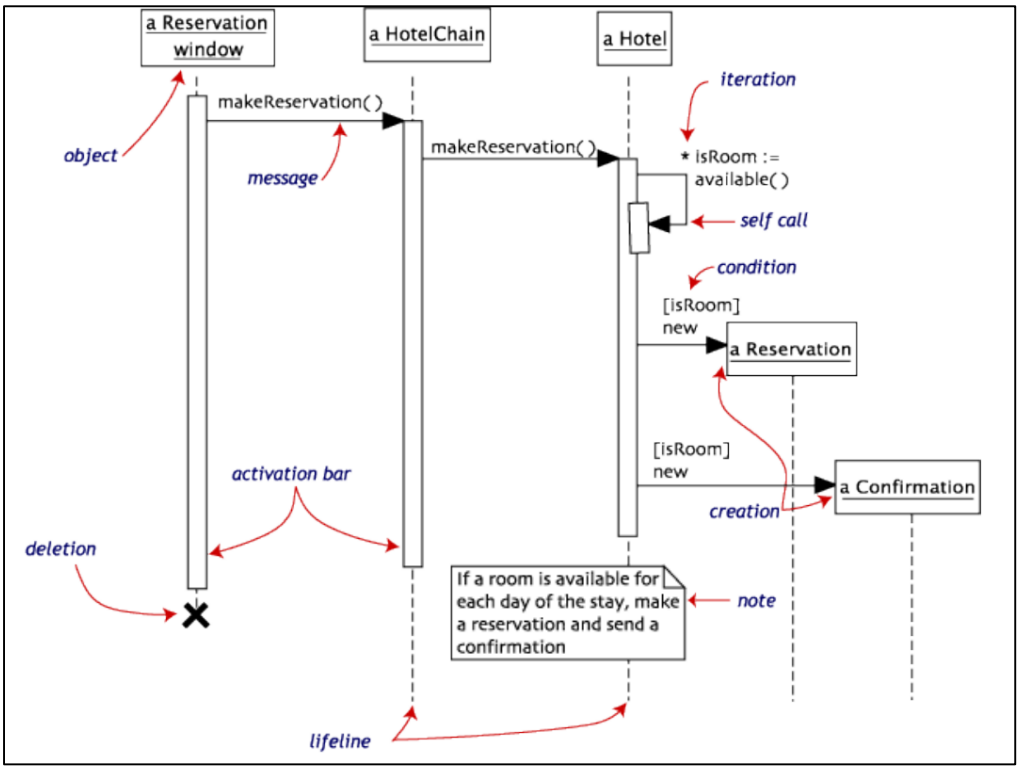


Figure 8 Sequence Diagram

Practice task 1:

Scenario: Online Shopping System Requirements and UML Diagrams

Background:

You are a software engineer working for a prominent e-commerce company, ShopNShop. The company is planning to revamp its existing Online Shopping System (OSS) to provide a better shopping experience for customers and streamline internal operations. Your team has been tasked with conducting a comprehensive requirements analysis and creating UML diagrams to visualize the system's structure and behavior.

Phase 1: Requirements Identification

In this phase you are required to do Requirement Gathering Step which will include gathering Functional and Non Functional Requirements.

Phase 2: UML Diagrams

To visually represent the system's structure and behavior, your team will create UML diagrams such as: Use Case diagram, Class diagram, Component diagram, Activity diagram and Sequence diagram.

Walkthrough Task:

Introduction to JIRA

JIRA is a bug tracking tool that allows software developers to plan, track and work faster. JIRA is the main source of information for future software release. Developers can plan new features to be added and bugs to be fixed in the next release, organize documentation tasks, track documentation progress.

Basic Tasks that we have to do in JIRA:

Issue Tracking:

Jira is primarily known for its issue tracking capabilities. You can create, view, and manage issues, which can represent tasks, bugs, user stories, or any other work items.

Each issue can have various attributes, including a summary, description, priority, assignee, status, and more.

Project Management:

Jira allows you to create projects to organize and categorize your work. Projects can represent different teams, departments, or initiatives.

You can define project-specific workflows, permissions, and configurations.

Customizable Workflows:

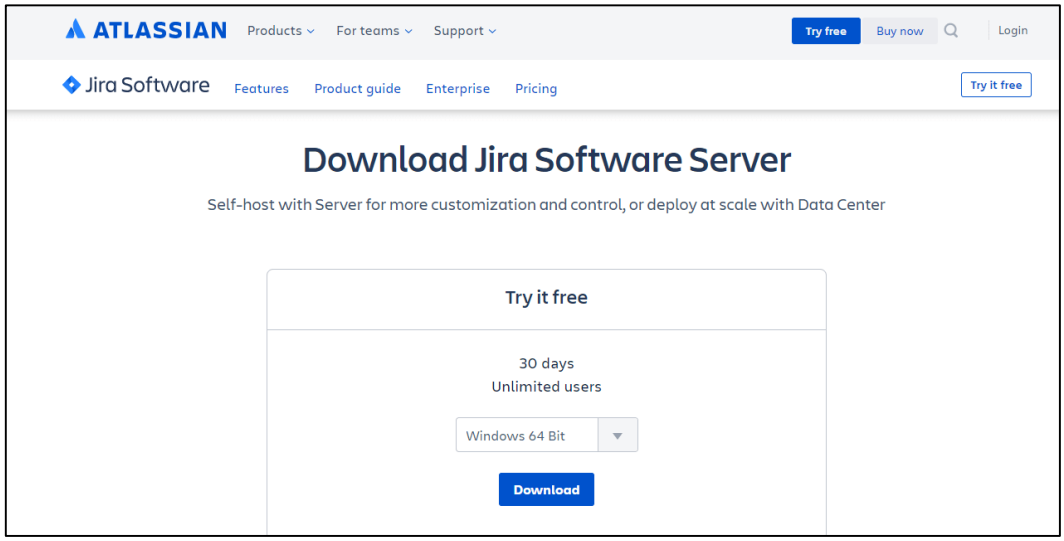
Jira offers customizable workflows that define the lifecycle of an issue. You can design workflows to match your team's unique processes.

Workflows can include states like "To Do," "In Progress," "Testing," and "Done," with transitions between them.

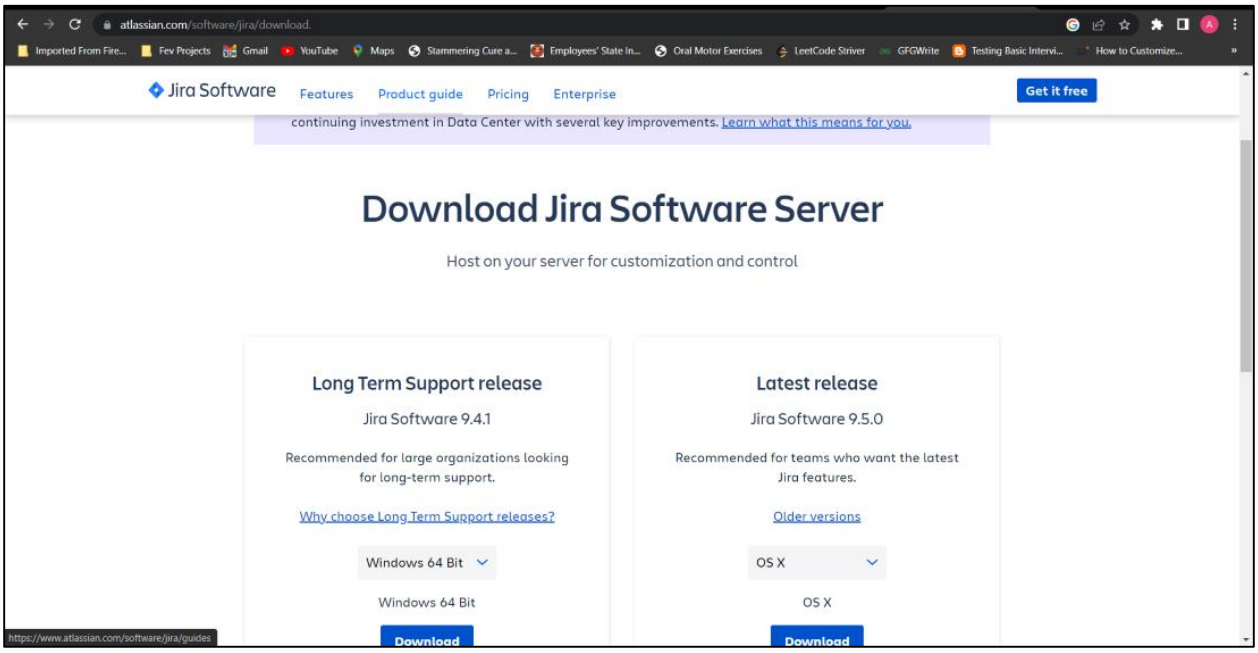
Steps for Jira Download:

Go to the Atlassian website at https://**www.atlassian.com/software/**jira and navigate to the Jira product page.

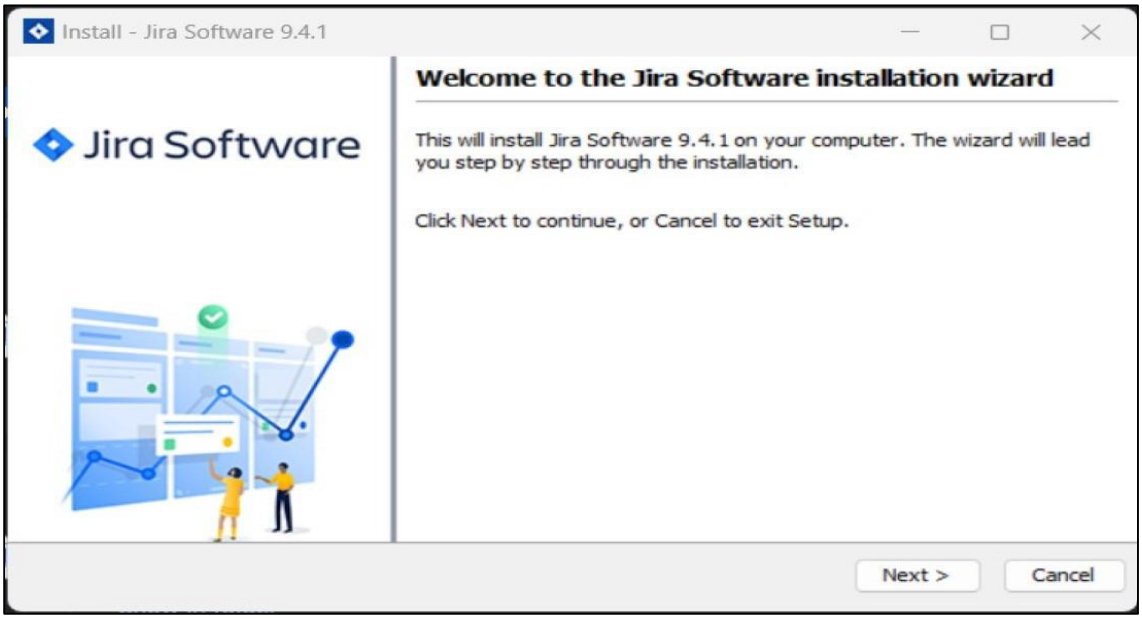
STEP 1:



STEP 2: Visit Downloads



STEP 3 and 4: Click setup and Install



Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 2: Project Management Tools for Agile Project.

In this lab students will be able to:

1. Efficiently create and manage tasks, issues, and projects using JIRA's features and workflows.
2. Proficiently collaborate with team members, tracking progress, and optimizing project management processes within JIRA.

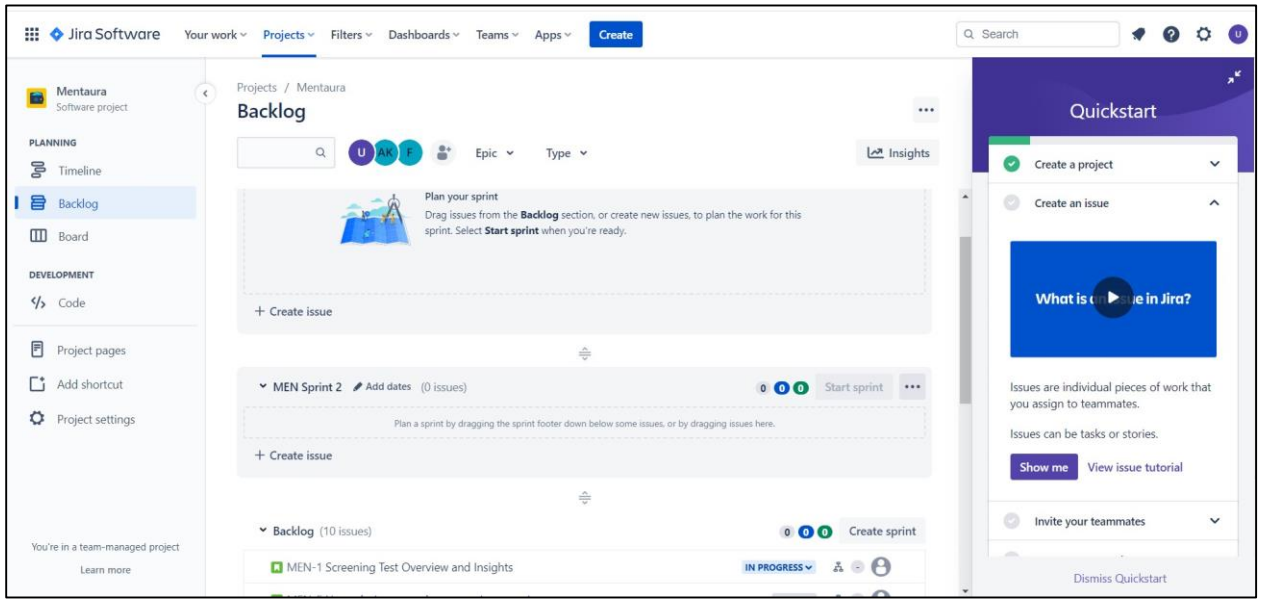
INTRODUCTION TO JIRA:

JIRA is a Project Management and bug tracking tool that allows software developers to plan, track and work faster. JIRA is the main source of information for future software release. Developers can plan new features to be added and bugs to be fixed in the next release.

Role of JIRA

A role is a set of permissions granted to Jira users or groups to view or modify data with Assets. Roles can have three scopes: across all of Assets (Jira admin) for a single object schema (Object Schema Manager, Developer, or User) and all object types within that schema.

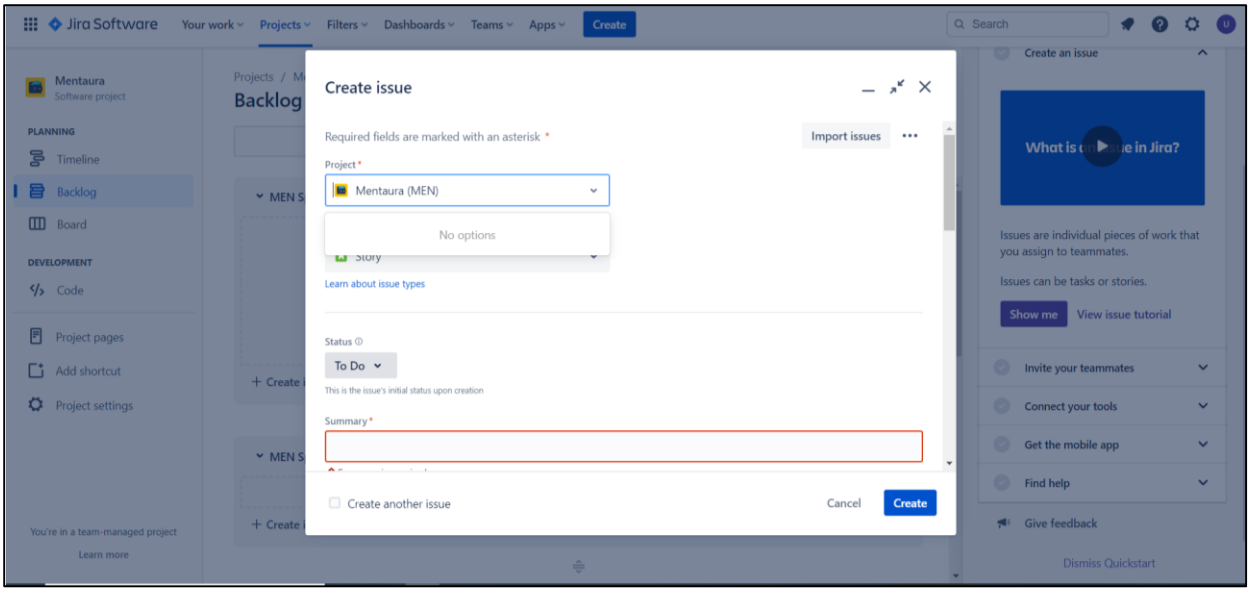
JIRA INTERFACE:



<u>Walkthrough Task:</u>

Create a new project:

- Click on create then go to the new project
- There are two options in Jira
    - Create a new project /Story
    - Create a new project

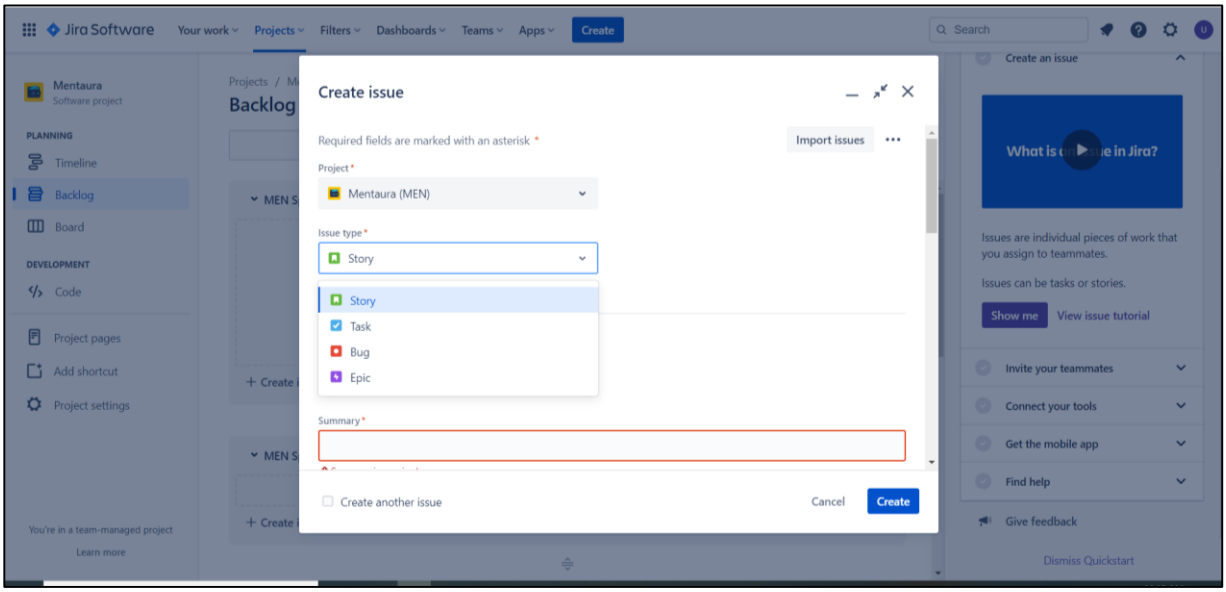


Create Story:

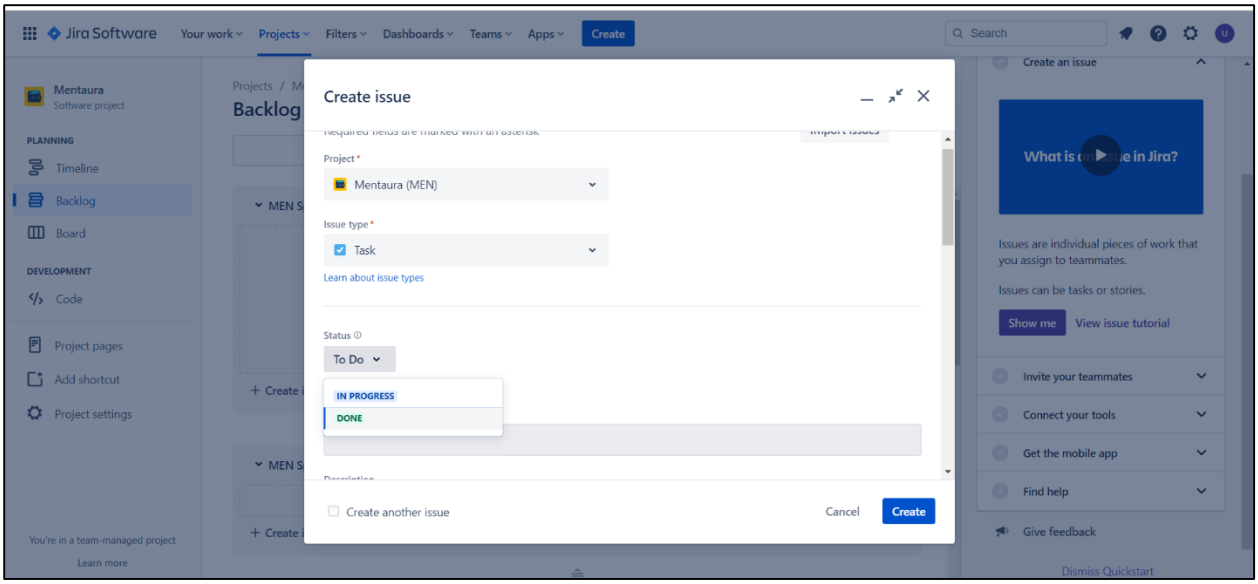
- Task
- Bug
- Epic

Bugs, and tasks describe a single piece of work, while epics are used to describe a group of issues that all relate to the same, larger body of work. Epics are typically completed over several sprints, or a longer time frame if you don't use sprints.

Click on Task:

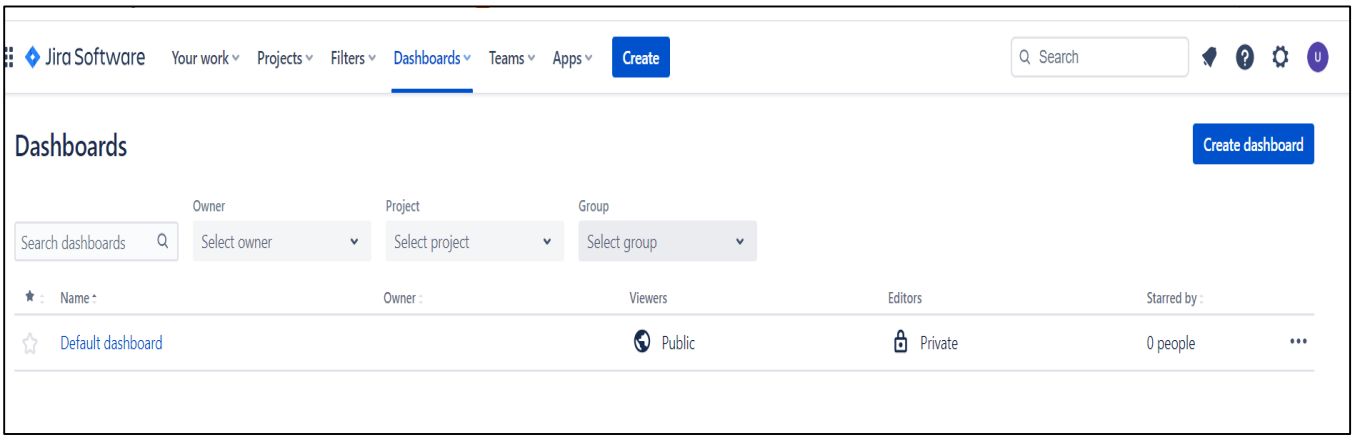


- In progress
- To Do



Dashboard

Create Dashboard:

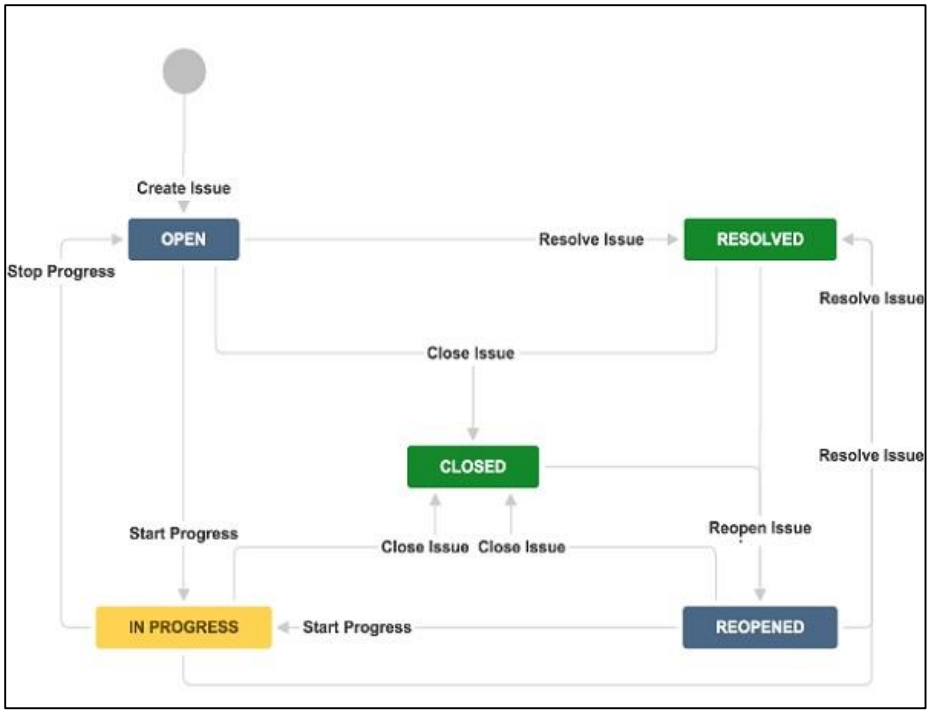


For Dashboard

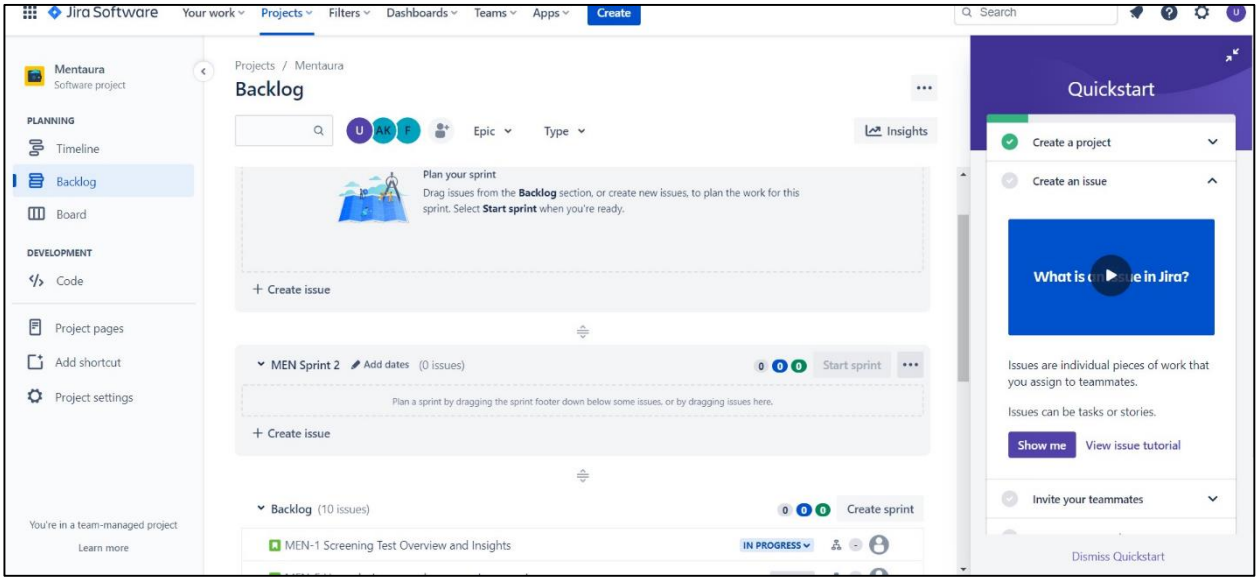
- Select Owner
- Project
- Group

Workflow:

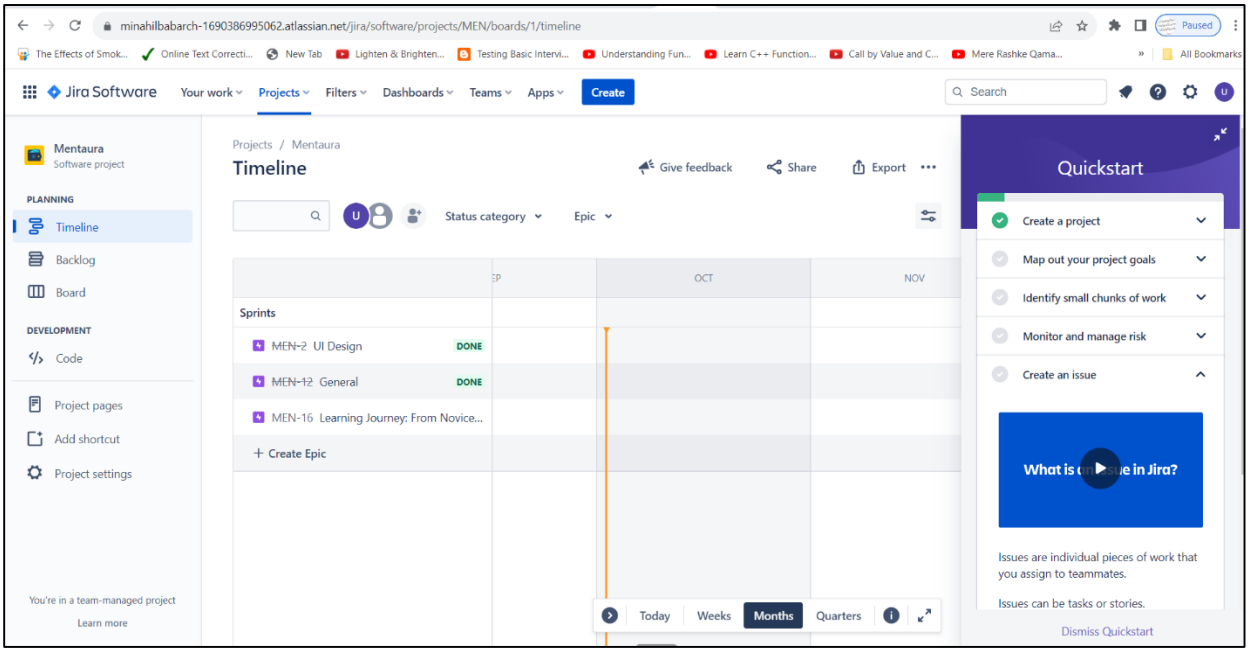
- In progress
- Resolved
- To do
- Closed



Backlog: The backlog of a Scrum board shows the issues for your project grouped into a backlog and sprints. In the Scrum backlog, you can create and update issues, drag and drop issues to rank them, or assign them to sprints.



Time line of Work Flow:



<u>Practice Task 1:</u>

Creating a Project:

Task: Create a new project in Jira with a suitable name and description. Choose an appropriate project template (e.g., Scrum or Kanban).

Creating Issues:

Task: Within your newly created project, create three different types of issues (e.g., Task, Bug, Story) with relevant details, including a summary, description, and priority.

Assigning Issues:

Task: Assign one of the issues you created to a specific user in your Jira instance.

Setting Due Dates:

Task: Set due dates for two of the issues you created. Ensure they have realistic deadlines.

<u>Practice Task 2</u>:

Scenario Title: Bug Tracking and Issue Resolution in Jira

Scenario Overview:

In this scenario, you will play the role of a software development team member working on a project. Your team uses Jira to track and manage tasks, bugs, and new feature requests. You will be assigned various tasks related to bug tracking and issue resolution using Jira.

Scenario Steps:

- **Login to Jira:**

Access the Jira platform using the provided login credentials.

- **View the Project Dashboard:**

After logging in, you will be taken to the project dashboard. Familiarize yourself with the layout, including the project summary, recent issues, and activity feed.

- **Create a New Issue:**

Click on the "Create" button and select "Issue" to create a new bug report.

Fill in the necessary details such as issue type (bug), summary, description, priority, and assignee.

- **Assign an Issue:**

Assign the issue you just created to a team member of your choice.

- **Add Comments:**

Navigate to the issue you created, and add a comment providing additional details about the bug or issue. Mention any steps to reproduce it.

- **Transition the Issue:**

Move the issue through the workflow. Change its status from "Open" to "In Progress" to indicate that you're actively working on it.

- **Resolve the Issue:**

Once you believe the bug is fixed, change the issue status to "Resolved." Be sure to explain how you resolved the issue in the comment section.

- **Create a Sub-task:**

Create a sub-task related to the issue you resolved. This could be a task for testing the fix.

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 3: Applying Software Project Construction Management Fundamentals.

Lab Objectives:

- **Case Study Discussion**
- **Project Selection**
- **Research and Investigation**
- **Report Generation**
- **Planning and Scheduling with Jira**
- **Version Control with GitHub**
- **Problem-Solving and Adaptability**

By addressing these objectives, students will have a structured and comprehensive learning experience that encompasses project selection, research, project management tools (Jira), and version control (GitHub), while also honing critical skills necessary for real-world project development.

Executive Summary: Course management system (CMS)

A course management system (CMS) is a collection of software tools providing an online environment for course interactions. A CMS typically includes a variety of online tools and environments, such as:

- An area for faculty posting of class materials such as course syllabus and handouts
- An area for student posting of papers and other assignments
- A grade book where faculty can record grades and each student can view his or her grades
- An integrated email tool allowing participants to send announcement email messages to the entire class or to a subset of the entire class
- A chat tool allowing synchronous communication among class participants
- A threaded discussion board allowing asynchronous communication among participants.

In addition, a CMS is typically integrated with other databases in the university so that students enrolled in a particular course are automatically registered in the CMS as participants in that course.

The Course Management System (CMS) is a web application for department personnel, Academic Senate, and Registrar staff to view, enter, and manage course information formerly Submitted via paper. Departments can use CMS to create new course proposals, submit changes for existing courses, and track the progress of proposals as they move through the stages of online approval.

<u>Detailed Report: COURSE MANAGEMENT SYSTEM</u>

1.1 OBJECTIVE:

A course management system (CMS) is a collection of software tools providing an online environment for course interactions. A CMS typically includes a variety of online tools and environments, such as:

- An area for faculty posting of class materials such as course syllabus and handouts
- An area for student posting of papers and other assignments
- A grade book where faculty can record grades and each student can view his or her grades
- An integrated email tool allowing participants to send announcement email messages to the entire class or to a subset of the entire class
- A chat tool allowing synchronous communication among class participants
- A threaded discussion board allowing asynchronous communication among participants. In addition, a CMS is typically integrated with other databases in the university so that students enrolled in a particular course are automatically registered in the CMS as participants in that course.

The Course Management System (CMS) is a web application for department personnel, Academic Senate, and Registrar staff to view, enter, and manage course information formerly Submitted via paper. Departments can use CMS to create new course proposals, submit changes for existing courses, and track the progress of proposals as they move through the stages of online approval.

PROBLEM ANALYSIS AND PROJECT PLANNING:

A course management system is a set of tools that enables an online environment for course interaction i.e. to create online course content and post it on the Web without having to handle HTML or other programming languages. Course management system become an integral a part of the upper education system. They create teaching and course management easier by providing a framework and set of tools for faculties and for students. The executive aspects of such systems could include class rosters (a group of people or things) and therefore the ability to record students' grades. With relevance the teaching aspects, however, it might include learning objects, class exercises, quizzes and tests. The CMS might also include tools for real-time chat, integrated email tool allowing participants to send announcement email messages to entire class or to a subset of the entire class. The CMS tool additionally focuses on all aspects of teaching, learning and teacher-student interaction.

1.2 RESOURCE:

<u>Software Requirement Analysis</u>

(1) Module Summary:

(1.1) Administrator Module:

Admin can produce accounts for college students and faculties and make course programmed list and add faculties and students to it course list. Admin can produce course details exploitation course creation kind that consists in fact name, course id, and choose student. Using Student creator kind student details are entered to information. User name, adapt username, password, given name and name, ID. After accounts are produced supported every students and instructors are divided and accessorial to list exploitation create missing students' kind.

(1.2) Faculty Module:

It can check student's papers, their assignments and assign grades for work. This module accommodates preparation menu, choose student for grades.

(1.3) Students Module:

Student can register with application or the proposed system and login with user name and password. He will check and submit assignment and his/her grade. Every student can have id.

1.3 PROCEDURE:

(2) Functional and Non-Functional Requirements

(2.1) Functional Requirements:

(2.1.1) Creating Courses

Integration with registration system: The system shall periodically upload the latest registrar's classes list to determine courses that offered in the current semester. The system shall generate course for each class that registered and determine the current set of students that enrolled in that class. The system shall allow course instructor to update course content.

(2.1.2) Grade Management

a. Allow grades to be entered online: The system shall allow instructors to enter and modify grades online.

b. Allow students to access their grades online: The system shall allow student to log in their account and check their grades at any time.

c. The system shall provide statistical information such as averages, standard deviation, and median about student's grades.

d. Track and Handle Re-Grade Requests: The system shall be able to track and handle requests for re- grades, and all information about re-grades shall be available to the student, and the course instructor.

(2.1.3) Paper and Assignment Submission

a. Accept submissions in multiple formats: The system shall accept submissions in multiple formats, including .zip, .cpp, .txt, .doc,etc.

b. Support for late submissions: The system shall provide information about late submissions, and also disallow submissions after a certain period of time.

c. Integration with grade management: The homework submission system shall be integrated with the grade management by using online grading templates that can be filled out, and automatically annotating code with line numbers.

1. Assignment grades can be automatically posted to student account.

2. Grader comments can be sent along with the grades.

(2.1.4) Create Accounts

a. The system shall automatically create accounts for each class.

1. Create one account for course instructor regardless to the number of classes that he/she teaches.

2. The account username is course name and its number.

3. The account password is the same password that in Academic Information System (AIS).

4. Any change in the password in AIS the system shall reflect it on the instructor account password in CMS.

5. Create one account for each student that registered in this class.

6. The account username is course name and its number.

7. The account password is the same password that in Student Information System (SIS).

8. Any change in the password in SIS the system shall reflect it on the student account password in CMS.

-Instructor account contain the classes that he/she teach, each class contain list of student that ordered based on student serial number.

-Instructor can modify student grades from his/her account.

(2.2) Non-Functional Requirements:

(2.2.1) Response Time

a. Average response time shall be less than 2 second.

(2.2.2) Throughput

a. The system shall accommodate 1000 booked per minute.

(2.2.3) Recovery Time

a. In case of a system failure, redundant system shall resume operations within 30 sec.

b. Average repair time shall be less than 1 hour.

(2.2.4) Start-up/Shutdown Time

a. The system shall be operational within 1 minute of starting-up.

(2.2.5) Capacity

a. The system accommodates 4000 concurrent users.

(2.2.6) Utilization of Resources

a. The system shall store in the database no more than one million transactions.

b. If the database grows over this limit, old transaction shall be backed up and deleted from the operational database.

(2.2.7) Security

a. Firewall Protection: The course management software system shall run inside a firewall.

b. Support different roles: The system shall support different roles for users, such as Instructors, Students, and administrative staff, the user logged in with given role should only be allowed access consistent with that role. For example, a student shall only be allowed to see he/she grades not to modify it.

(2.2.8) Reliability

a. The system shall not be down more 2 times in year.

(2.2.9) Scalability

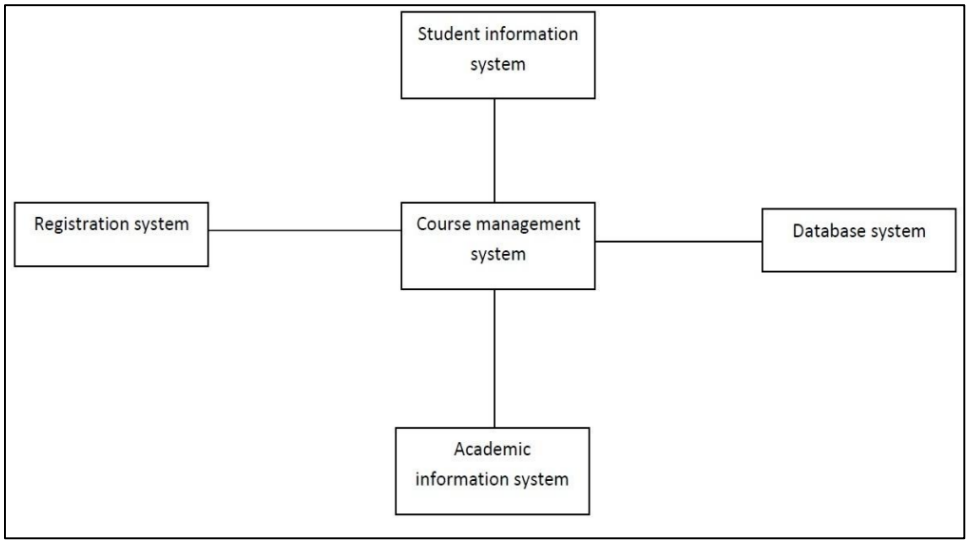
a. Scaling the system to large number of users: large courses will have hundreds of students.

b. The system shall be able to handle the load for such courses, especially near assignment deadlines when many students can be expected to access the course management system.

1.4 DATA MODELING and DESIGN

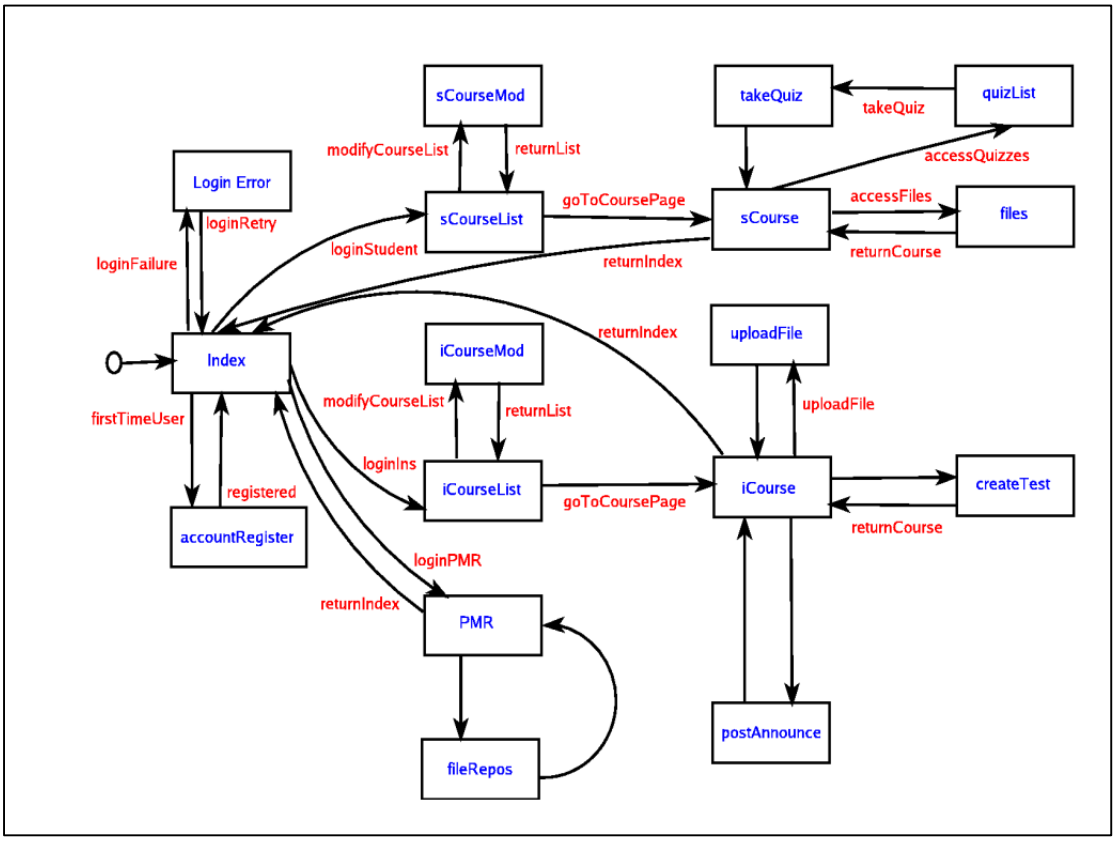
(1) Product Perspective

The system will be operating within university environment. This environment has another system that will interact with this system so we need interfaces between these systems.



(2) Flow Chart

The below diagram will provide the overall flow of the project.



(3) Data Dictionary

(3.1) Student Details:

FIELD NAME	TYPE	CONSTRAINTS
Sid	Varchar2	Primary key
Name	Varchar2	
Roll_No	Varchar2	Notnull
Regulation	Varchar	
Courseid	Number	Foreign key
grade	Char	
Fid	Varchar2	Foreign Key

(3.2)CourseDetails

FIELD NAME	TYPE	CONSTRAINTS
Courseid	Number	Primary key
CourseName	Varchar2	
Start_date	Date	
End_date	Date	
Subject	Varchar2	not null

(3.3)FacultyDetails

FIELD NAME	TYPE	CONSTRAINTS
Fid	Varchar2	Primary key
Name	Varchar2	
Courseid	Number	Foreign Key
Designation	Varchar	
Subject	Varchar	

(3.4)LoginDetails

FIELD NAME	TYPE	CONSTRAINTS
Userid	Varchar2	Unique
Password	Varchar2	Not null

Software Designing

UML

UML stands for Unified Modeling Language. This object-oriented system of notation has evolved from the work of Grady Booch, James Rum Baugh, Ivar Jacobson, and the Rational Software Corporation. These renowned computer scientists fused their respective technologies into a single, standardized model. Today, UML is accepted by the Object Management Group (OMG) as the standard for modeling object oriented programs.

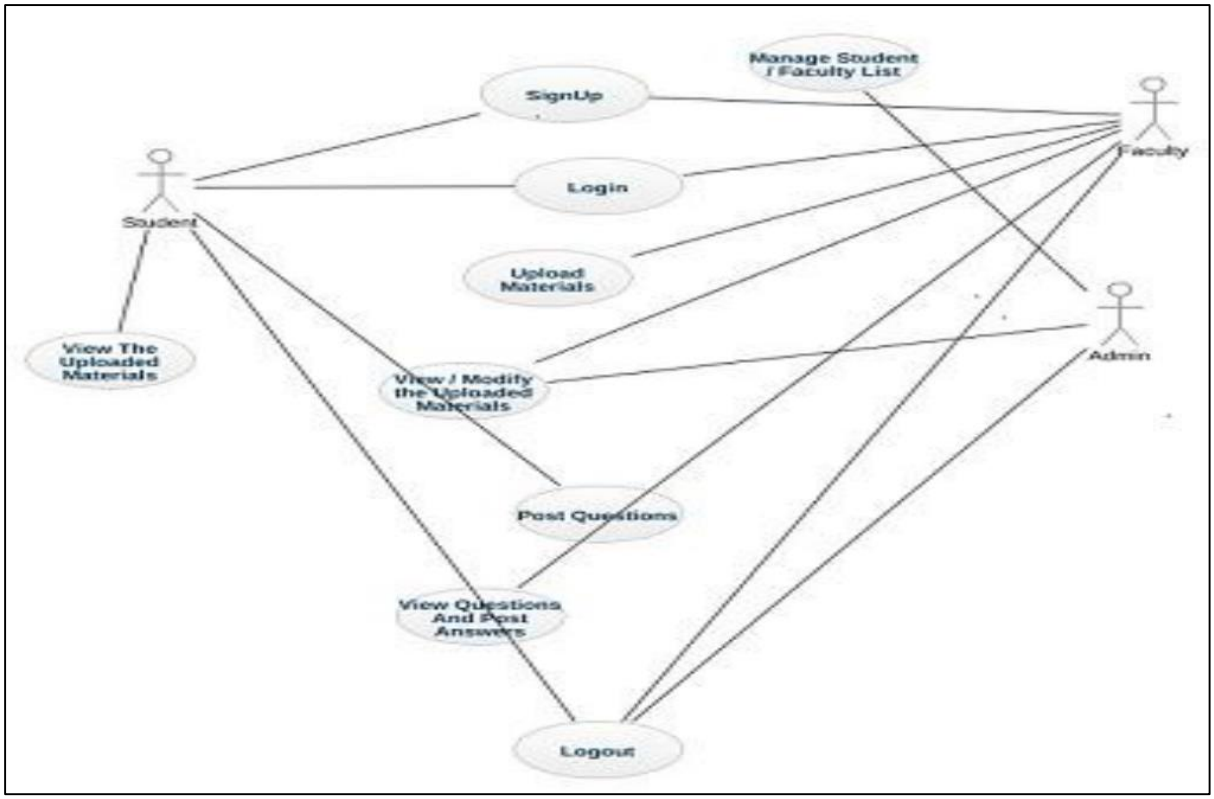
UML Diagrams

UML defines nine types of diagrams: class (package), object, use case, sequence, collaboration, state chart, activity, component, and deployment diagram.

Use Case Diagram

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified. The purposes of use case diagrams can be defined as follows –

- 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 is actors.

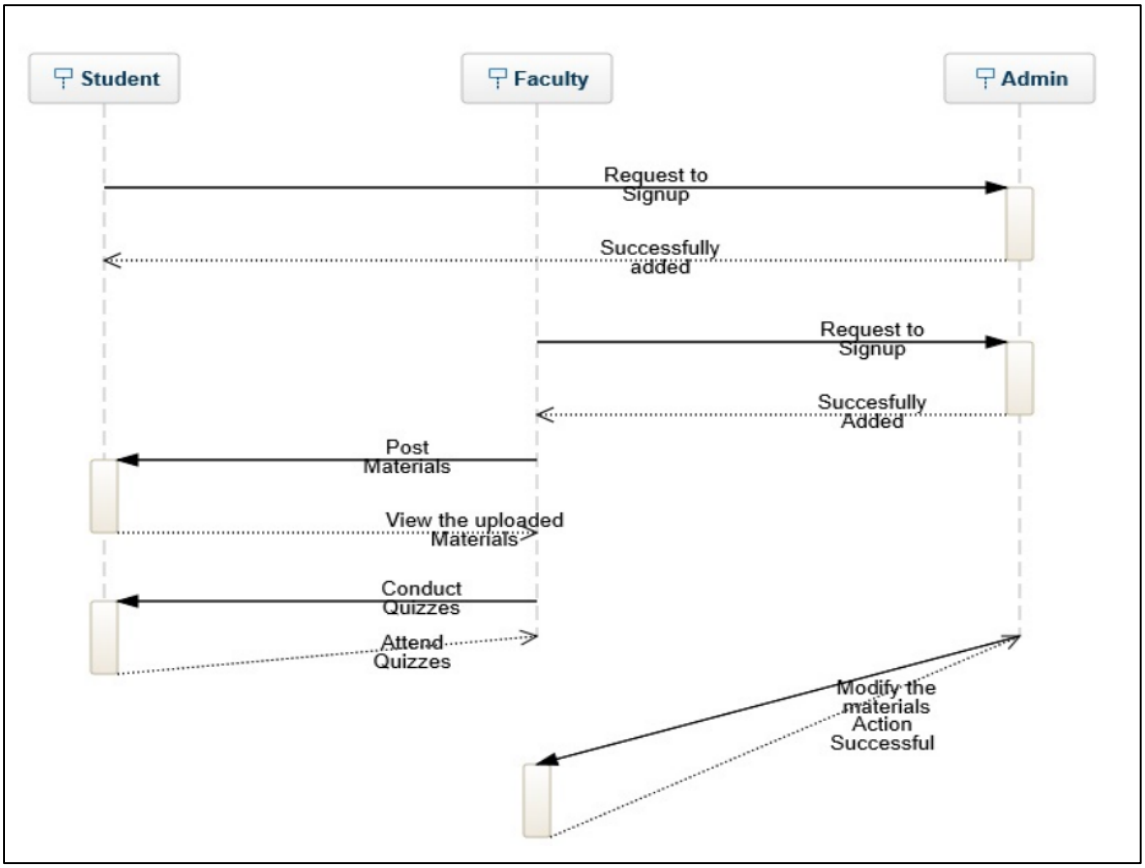


Sequence Diagram

This interactive behavior is represented in UML by Sequence diagram. Sequence diagram emphasizes on time sequence of messages that send and receive messages.

Following things are to be identified clearly before drawing the sequence diagram

- Objects taking part in the interaction.
- Message flows among the objects.
- The sequence in which the messages are flowing.
- Object organization.

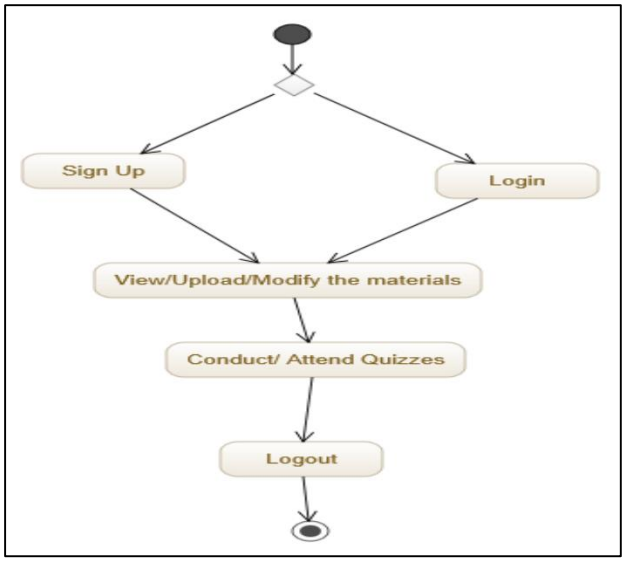


Activity Diagram

The basic purposes of activity diagrams are to captures the dynamic behavior of the system. Activity is a particular operation of the system. 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. The only missing thing in the activity diagram is the message part.

The purpose of an activity diagram can be described as –

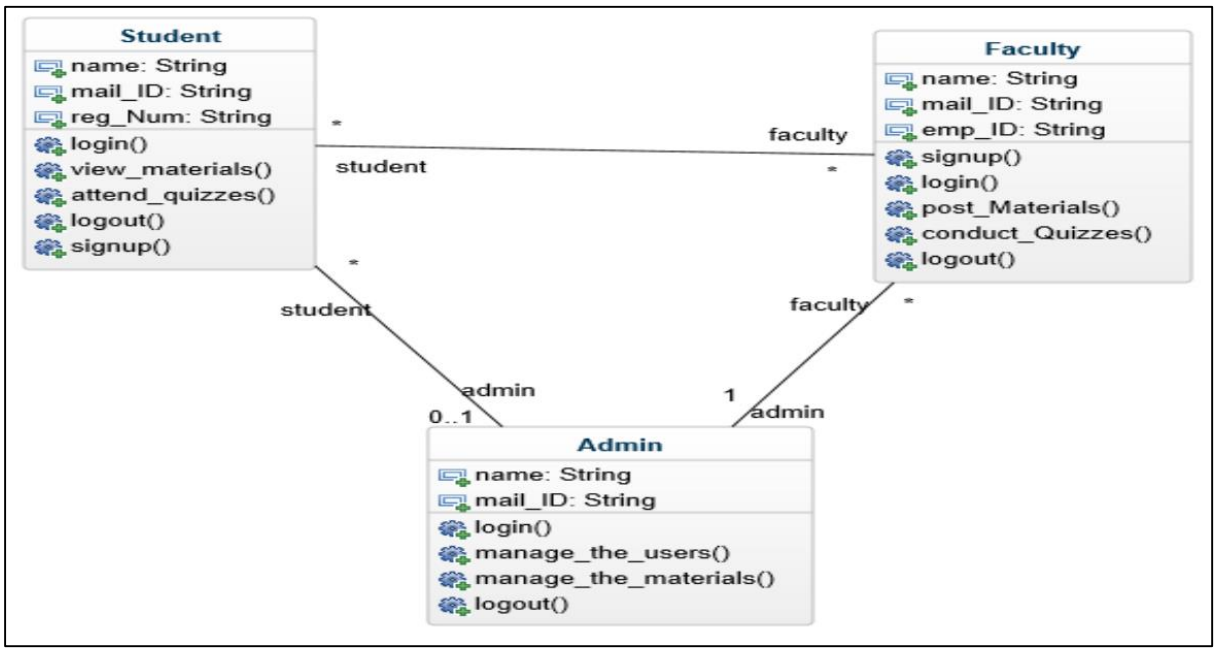
- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.



Class Diagram

The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction. The purpose of the class diagram can be summarized as –

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.



Prototype model

Prototype is a working model of software with some limited functionality. The prototype does not always hold the exact logic used in the actual software application and is an extra effort to be considered under effort estimation. Prototyping is used to allow the users evaluate developer proposals and try them out before implementation. It also helps understand the requirements which are user specific and may not have been considered by the developer during product design.

Practice Task:

1. **Please select a project from the list provided, which you will be committed to throughout the entire semester's lab sessions.**

-E-commerce Website

-Health and Fitness App

-Social Networking Platform

-Online Learning Management System

-Weather Forecasting App

-Smart Home Automation System

-Content Recommendation System

-Employee Management System

-Community Forum or Q&A Website

-Fleet Management System

-Human Resources Management System (HRMS)

-Customer Relationship Management (CRM) System

-Other (Project of your choice)

2. **For the selected project, generate a detailed report in the format given above (section 1.1-1.4)**
3. **For Project Planning phase, you'll work on JIRA**
   - **Project Timeline**
   - **Project Backlog**
   - **Sprints creation module wise**
   - **Team members' roles and responsibilities (Task Assignment)**

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 4: Converting Design Diagrams into Meaningful Code.

Lab Objectives:

- Understanding the Design Process
- Mapping Design Elements to Code
- Coding Guidelines and Best Practices
- Process Model Application
- Hands-On Coding
- Testing and Debugging

Java Naming conventions

Java naming convention is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method, etc.

But, it is not forced to follow. So, it is known as convention not rule. These conventions are suggested by several Java communities such as Sun Microsystems and Netscape.

All the classes, interfaces, packages, methods and fields of Java programming language are given according to the Java naming convention. If you fail to follow these conventions, it may generate confusion or erroneous code.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and other programmers. Readability of Java programs is very important. It indicates that less time is spent to figure out what the code does.

The following are the key rules that must be followed by every identifier:

- The name must not contain any white spaces.
- The name should not start with special characters like & (ampersand), $ (dollar), _ (underscore).

Let's see some other rules that should be followed by identifiers.

Class

Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Info in the name of a class. A class name should not be a verb.

- It should start with the uppercase letter.
- It should be a noun such as Color, Button, System, Thread, etc.
- Use appropriate words, instead of acronyms.

Example: -

1. **public class** Employee

```
{
//code snippet
}
```

Interface

- It should start with the uppercase letter.
- It should be an adjective such as Runnable, Remote, Action Listener.
- Use appropriate words, instead of acronyms.

Example: -

```
interface Printable
{
//code snippet
}
```

Method

Methods should have verb or verb phrase names like postPayment, deletePage, or save. Accessors, mutators, and predicates should be named for their value and prefixed with get, set, and is according to the java bean standard 4.

string name = employee.getName();

customer.setName("mike");

if (paycheck.isPosted())...

- It should start with lowercase letter.
- It should be a verb such as main(), print(), println().
- If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as actionPerformed().

Example: -

```
class Employee
{
//method
void draw()
{
//code snippet
} }
```

Variable

- It should start with a lowercase letter such as id, name.
- It should not start with the special characters like & (ampersand), $ (dollar), _ (underscore).
- If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as firstName, lastName.
- Avoid using one-character variables such as x, y, z.

Example: -

```
1. class Employee
2. {
3. //variable
4. int id;
5. //code snippet
6. }
```

Package

- It should be a lowercase letter such as java, lang.
- If the name contains multiple words, it should be separated by dots (.) such as java.util, java.lang.

Example: -

```
1. package com.javatpoint; //package
2. class Employee
3. {
4. //code snippet
5. }
```

Constant

- It should be in uppercase letters such as RED, YELLOW.
- If the name contains multiple words, it should be separated by an underscore (_) such as MAX_PRIORITY.
- It may contain digits but not as the first letter.

Example: -

```
1. class Employee
2. {
3. //constant
4. static final int MIN_AGE = 18;
5. //code snippet
```

Mapping Design Elements to Code

A UML (Unified Modeling Language) class diagram is often used as a blueprint for designing the structure and relationships of classes in an object-oriented software system. The UML class diagram can be used to guide the development of Java code, as Java is an object-oriented programming language. The conversion from a UML class diagram to Java code typically involves mapping the elements in the UML diagram to their Java counterparts.

Here's how a UML class diagram can be converted into Java code:

Classes and Attributes:

- UML Classes are directly mapped to Java classes. Each class in the UML diagram becomes a Java class.
- Class attributes in the UML diagram become instance variables (fields) in the Java class.

Methods:

- UML class methods become Java class methods (functions).
- Method signatures in the UML diagram, including method names and parameter lists, are replicated in the Java code.

Inheritance and Generalization:

- In UML, inheritance and generalization relationships represent class hierarchies. In Java, these relationships are expressed through the "extends" and "implements" keywords.
- A UML generalization relationship is mapped to Java inheritance, where a subclass extends a superclass.

Interfaces:

- Interfaces defined in the UML diagram are directly implemented in Java using the "implements" keyword.

Visibility and Access Control:

- UML defines visibility levels for attributes and methods (e.g., public, private, protected). These access levels are reflected in Java through access modifiers (e.g., public, private, protected).

Static Members:

- UML may indicate static attributes or methods with an underline. In Java, these members are marked as "static."

Dependencies:

- Dependencies in UML can be translated to Java imports. If one class depends on another (e.g., uses a class), you include an import statement for that class.

In practice, converting a UML class diagram into Java code involves detailed analysis and mapping of UML elements to their Java equivalents. The process can be facilitated by using modeling and code generation tools that can automate parts of this transformation. These tools can help ensure consistency and accuracy in the conversion process.

Basic Class Structure in Java

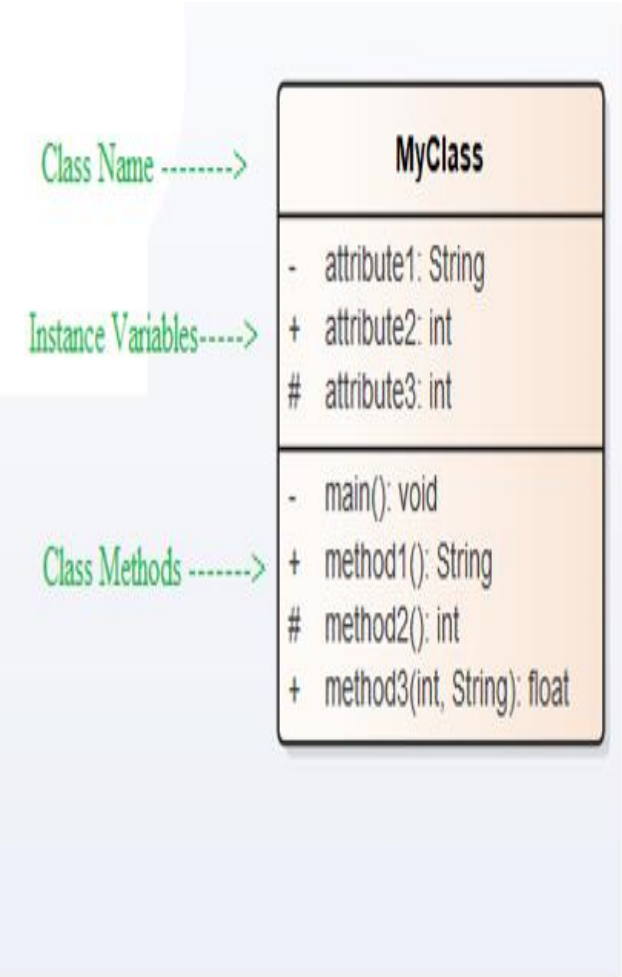
Every application created in Java must contain a class that has a method called main() which acts as an entry point like the main() we used to define in C++. After compiling the program/ classes you will use the name of class that contains the main to start/ run your program.

There are no restrictions for the name of your class, you can give any name to the class, for better understanding see naming conventions for class, variable and method name in java from the given link: https://www.javatpoint.com/java-naming-conventions.

The name of entry point function is always called main(). When you run your program, main() method will be used to call other methods from the classes in your program. For your first lab, you will only be writing a program which has only one class and only one method called main().

UML Class Structure

You are familiar with the following UML class structure shown in figure below:




```
// class name
public class MyClass {
        // instance Variables
        private String attribute1;
        public int attribute2;
        protected int attribute3;
        // class methods
        public String method1(){
                return "";
        }
        protected int method2(){
                return 0;
        }
        public float method3(int reg, String name){
                return 0;
        }
        public static void main(String[] args){

        }
}
```

UML Class Structure and Equivalent OO code in Java

A class notation consists of three parts:

I. **Class Name**

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

II. **Instance Variable**

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

III. **Class Operations** (Methods)

- Operations are shown in the third partition in figure above. 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 is shown after the colon following the parameter name.
- Operations map onto class methods in code

Below figure shows the UML class of Student

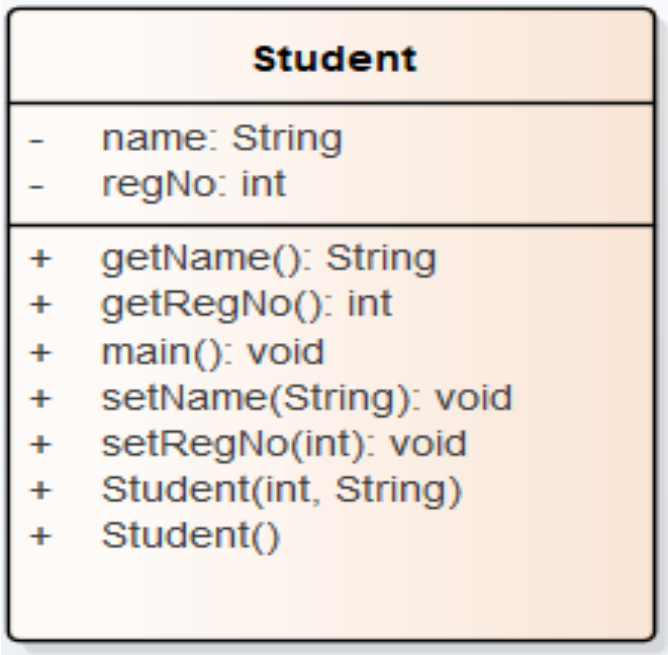


Figure: Example of Student Class

<u>Instance Variables</u>

Instance variable in Java is used by Objects to store their states. Variables that are defined without the STATIC keyword and are Outside any method declaration are Object-specific and are known as instance variables. They are called so because their values are instance specific and are **not** shared among instances.

If a class has an instance variable, then a new instance variable is created and initialized to a default value as part of the object creation of the class or subclass, figure 12 shows instance variables.

```
public class Student {
    //instanceS variables
    private String name;
    private int regNo;
```

Figure: Code snip Instance Variables

Setter and Getter Methods

In Java, getter and setter are two conventional methods that are used for retrieving and updating value of a variable, in figure 13 shows setter and getter methods are mentioned.

```
//setters and getters
public void setName(String name) {
    this.name = name;
}
public void setRegNo(int regNo) {
    this.regNo = regNo;
}
public String getName() {
    return name;
}
public int getRegNo() {
    return regNo;
}
```

Figure9: Code snip of setter and getter methods

Constructor

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

```
//default constructor
public Student() {
    name=null;
    regNo=0;
}
```

Figure 10: Code snip of Constructor

Every time an object is created using the new() keyword, at least one constructor is called. In figure 11 and figure 14 method **Student()** is default constructor.

main() Method in Java

The main() is the starting point for JVM to start execution of a Java program. Without the main() method, JVM will not execute the program.

public: It is an access specifier. We should use a public keyword before the main() method so that JVM can identify the execution point of the program. If we use private, protected, and default before the main() method, it will not be visible to JVM.

static: You can make a method static by using the keyword static. We should call the main() method without creating an object. Static methods are the method which invokes without creating the objects, so we do not need any object to call the main() method.

void: In Java, every method has the return type. Void keyword acknowledges the compiler that main() method does not return any value.

main(): It is a default signature which is predefined in the JVM. It is called by JVM to execute a program line by line and end the execution after completion of this method. We can also overload the main() method.

String args[]: The main() method also accepts some data from the user. It accepts a group of strings, which is called a string array. It is used to hold the command line arguments in the form of string values

If the main() method is written without String args[], the program will compile, but not run, because JVM will not recognize the main() method. Remember JVM always looks for the main() method with a string type array as a parameter.

Below figure shows code snip of main() method in Java.

```java
// main methdd
    public static void main(String[] args){
        Student student1=new Student("ALi",123445);
        Student student2=new Student();
        student2.setName("Qasim");
        student2.setRegNo(158977);
        student1.getName();
        student1.getRegNo();
    }
```

Figure: main Method in Java

Figure below shows default constructor, parametrize constructor, setter and getter methods of class Student.

```java
public class Student {
    //member variables
    private String name;
    private int regNo;
    //default constructor
    public Student() {
        name=null;
        regNo=0;
    }
    //paremetrize constructor
    public Student(String name, int regNo) {
        this.name = name;
        this.regNo = regNo;
    }
    //setters and getters
    public void setName(String name) {
        this.name = name;
    }
    public void setRegNo(int regNo) {
        this.regNo = regNo;
    }
    public String getName() {
        return name;
    }
    public int getRegNo() {
        return regNo;
    }
    // main methdd
        public static void main(String[] args){
            Student student=new Student("ALi",123445);
        }
}
```

Figure 11 Code snip of Student Class

Variables & Operator

Java Unary Operators

Unary operators are used with only one operand. For example, `++` is a unary operator that increases the value of a variable by **1**. That is, `++5` will return **6**.

Different types of unary operators are:

Operator	Meaning
+	**Unary plus**: not necessary to use since numbers are positive without using it
-	**Unary minus**: inverts the sign of an expression
++	**Increment operator**: increments value by 1
--	**Decrement operator**: decrements value by 1
!	**Logical complement operator**: inverts the value of a boolean

Increment and Decrement Operators

Java also provides increment and decrement operators: `++` and `- -`respectively. `++` increases the value of the operand by **1**, while `--` decrease it by **1**. For example,

```
intnum = 5;

// increase num by 1
++num;
```

Here, the value of *num* gets increased to **6** from its initial value of **5**.

Example: Increment and Decrement Operators

```
classMain{
publicstaticvoidmain(String[] args){

// declare variables
int a = 12, b = 12;
int result1, result2;

// original value
System.out.println("Value of a: " + a);

// increment operator
    result1 = ++a;
System.out.println("After increment: " + result1);

System.out.println("Value of b: " + b);

// decrement operator
```

```
    result2 = --b;
System.out.println("After decrement: " + result2);
  }
}
```

Java Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

```
a + b;
```

Here, the `+` operator is used to add two variables *a* and *b*. Similarly, there are various other arithmetic operators in Java.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

Example: Arithmetic Operators

```
classMain{
publicstaticvoidmain(String[] args){

// declare variables
int a = 12, b = 5;

// addition operator
System.out.println("a + b = " + (a + b));

// subtraction operator
System.out.println("a - b = " + (a - b));

// multiplication operator
System.out.println("a * b = " + (a * b));

// division operator
System.out.println("a / b = " + (a / b));

// modulo operator
System.out.println("a % b = " + (a % b));
  }
```

```
}
```

Java Ternary Operator

The ternary operator (conditional operator) is shorthand for the `if-then-else` statement, For example

variable = Expression ? expression1 : expression2

Here's how it works.
If the `Expression` is `true`, `expression1` is assigned to the *variable*.
If the `Expression` is `false`, `expression2` is assigned to the *variable*.

Example:
Java Ternary Operator

```
classJava{
publicstaticvoidmain(String[] args){

intfebruaryDays = 29;
    String result;

// ternary operator
    result = (februaryDays == 28) ?"Not a leap year" :"Leap year";
System.out.println(result);
  }

}
```

Java Logical Operators

Logical operators are used to check whether an expression is `true` or `false`. They are used in decision making.

Operator	Example	Meaning
&& (Logical AND)	expression1 **&&** expression2	true only if both expression1 and expression2 are true
|| (Logical OR)	expression1 || expression2	true if either expression1 or expression2 is true
! (Logical NOT)	!expression	true if expression is false and vice versa

Example: Logical Operators

```
classMain{
publicstaticvoidmain(String[] args){

// && operator
System.out.println((5>3) && (8>5));   // true
```

```
System.out.println((5>3) && (8<5));   // false

// || operator
System.out.println((5<3) || (8>5));   // true
System.out.println((5>3) || (8<5));   // true
System.out.println((5<3) || (8<5));   // false

// ! operator
System.out.println!(5 == 3));   // true
System.out.println!(5>3));   // false
   }
}
```

Java Relational Operators

Relational operators are used to check the relationship between two operands. For example,

```
// check is a is less than b
a < b;
```

Here, `>` operator is the relational operator. It checks if *a* is less than *b* or not.It returns either `true` or `false`.

Operator	Description	Example
==	Is Equal To	3 == 5 returns **false**
!=	Not Equal To	3 != 5 returns **true**
>	Greater Than	3 > 5 returns **false**
<	Less Than	3 < 5 returns **true**
>=	Greater Than or Equal To	3 >= 5 returns **false**
<=	Less Than or Equal To	3 <= 5 returns **false**

Example: Relational Operators

```
classMain{
publicstaticvoidmain(String[] args){

// create variables
int a = 7, b = 11;

// value of a and b
System.out.println("a is " + a + " and b is " + b);
```

```
// == operator
System.out.println(a == b);   // false

// != operator
System.out.println(a != b);   // true

// > operator
System.out.println(a > b);   // false

// < operator
System.out.println(a < b);   // true

// >= operator
System.out.println(a >= b);   // false

// <= operator
System.out.println(a <= b);   // true
   }
}
```

Control Flows

Control flow statements let you control the flow of the execution of the code in your program. In Java programming language, you can control the flow of execution of the code by placing the decision making, looping, and adding conditional blocks.

Based on this, we can classify the types of control flow statements as follows:

1. Decision Making Statements
2. Looping Statements

Decision Making Statements

We generally use decision-making statements when we have to decide which block of the code will be executed. There are three types of decision-making statements.

1. if statement
2. if-else statement
3. The switch statement

if Statement

if statement is the most basic decision-making statement in the java programming language. There are two possible values of the condition, i.e. true and false. The code block after the if statement will execute only if the value of the condition statement is true. If the value of the condition statement is false, the code block will not be executed.

if-else Statement

if-else statement is somewhat similar to the if statement, we are just adding an extra block of the code after the if statement.If the value of the condition statement is true, the if block will be executed, else the else block will be executed.

Example: 'if-else' statement, shown in figure 24

```
// main methdd
    public static void main(String[] args){
        Student student1=new Student("ALi",123445);
        if(student1.getName().equals("Ali")){
            System.out.println("Registration#: "+student1.getName());
        }
        else{
            System.out.println("Student 1 is not Ali");
        }
    }
```

Figure: Code snip of if-else

Nested if-else

In this type of if-else statement, we are nesting one if-else block in another if-else block. Either you can put your if-else block in the outer if block or else block.

Example: ‘nested if-else’ statement, shown in figure 25

```
// main methdd
    public static void main(String[] args){
        //set student name,reg#,gpa and attendence
        Student student1=new Student("Ahmed",123445,3.5,85);
        //student who get scholarship
        if(student1.getGpa()>=3.5){
            if(student1.getAttendencePer()>=85)
                System.out.println("You got scholarship");
            else
                System.out.println("attendence % is less than 85");
        }
        else
            System.out.println("GPA is less than 3.5");
    }
```

Figure: Code snip of Nested if-else

switch Statement

We have seen the if-else statement above. If the condition is true, then only the if block will be executed, else the else block will be executed. In the switch statement, there could be several execution paths.

Example: ‘switch’ Statement, shown in figure 26

```java
public static void main(String[] args){
    Scanner in=new Scanner(System.in);
    int choice,regNo;String name;
    Student student1=new Student();
    System.out.println("Press 1 to enter student name");
    System.out.println("Press 2 to enter student regNo");
    choice=in.nextInt();
    //switch statement
    switch(choice){
        case 1:
            System.out.println("Enter student name");
            name=in.next();
            student1.setName(name);
            break;
        case 2:
            System.out.println("Enter student reg#");
            regNo=in.nextInt();
            student1.setRegNo(regNo);
            break;
        default:
            System.out.println("incorrect choice");
    }
}
```

Figure: Code snip of switch statement

Looping Statements

We have seen all the decision-making statement, these statements make a decision and execute the block of the code based on that decision. In looping statements, we are making a decision and executing the block of code multiple times. Until the condition is true, we are looping over the block of the code.

for Loop

The Java for loop is a control flow statement that iterates a part of the programs multiple times. If the number of iteration is fixed, it is recommended to use for loop.

Example: for loop, shown in figure 27

```java
// main methdd
    public static void main(String[] args){
        Scanner in=new Scanner(System.in);
        int regNo;String name;

        System.out.println("Enter data of 10 student");
        for(int i=1;i<=10;i++){
            System.out.println("Enter data of student"+i);
            System.out.println("Enter regNo");
            regNo=in.nextInt();
            System.out.println("Enter name");
            name=in.next();
        }
    }
```

Figure: Code snip for loop

While Loop

The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition. **If the number of iterations is not fixed, it is recommended to use while loop**

while(condition){

//code to be executed

}

Example: while loop, shown in figure 28

```java
public static void main(String[] args){
    Scanner in=new Scanner(System.in);
    int regNo,strength,i=1;String name;

    System.out.println("Enter strength of class");
    strength=in.nextInt();

    System.out.println("Enter data of all student");
    while(i<=strength){
        System.out.println("Enter data of student"+i);
        System.out.println("Enter regNo");
        regNo=in.nextInt();
        System.out.println("Enter name");
        name=in.next();
        i++;
    }
}
```

Figure: Code snip of while loop

do-while Loop

The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.

If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

do{

//code to be executed

}while(condition);

Example: do-while loop, shown in figure 29

```java
    do{
        System.out.println("Press 1 to show student name");
        System.out.println("Press 2 to show student reg#");
        System.out.println("Press 3 to show student gpa");
        choice=in.nextInt();
        switch(choice){
            case 1:
                System.out.println(student.getName());
                break;
            case 2:
                System.out.println(student.getRegNo());
                break;
            case 3:
                System.out.println(student.getGpa());
                break;
            default:
                System.out.println("incorrect choice");
        }
     System.out.println("Press 1 to show menu");
     System.out.println("Press 0 to exit");
    }while(choice!=0);
}
```

Figure: Code snip of do-while loop

Practice Task 1: Executing Basic program

Write a java program that will ask the user to enter two numbers (integers). Then perform following tasks:

1) Calculate average of given numbers.
2) Sum of all even numbers between num1 and num2.
3) The sum of all squares between num1 and num2.

Practice Task 2:

Write a program that displays the following menu to the user:

Press 1 for Permanent Employee
Press 2 for Daily Wages Employee
After selecting the appropriate employee calculate salary for the employees.

The daily wages employees are paid 400 per hour while permanent employees are paid 800 per hour. First you need to ask for the number of hours for which the employee has worked so far and then calculate and display the salary of each type of employee.

Perform the above-mentioned task using switch statement.

Practice Task 3:

Create a design for the mark sheet by taking runtime value of student name, total marks, obtained marks and calculate its percentage, grade and GPA. Use good practices of programming that we have studied and ensure that the outcomes should be presented in a proper Viewable approach.

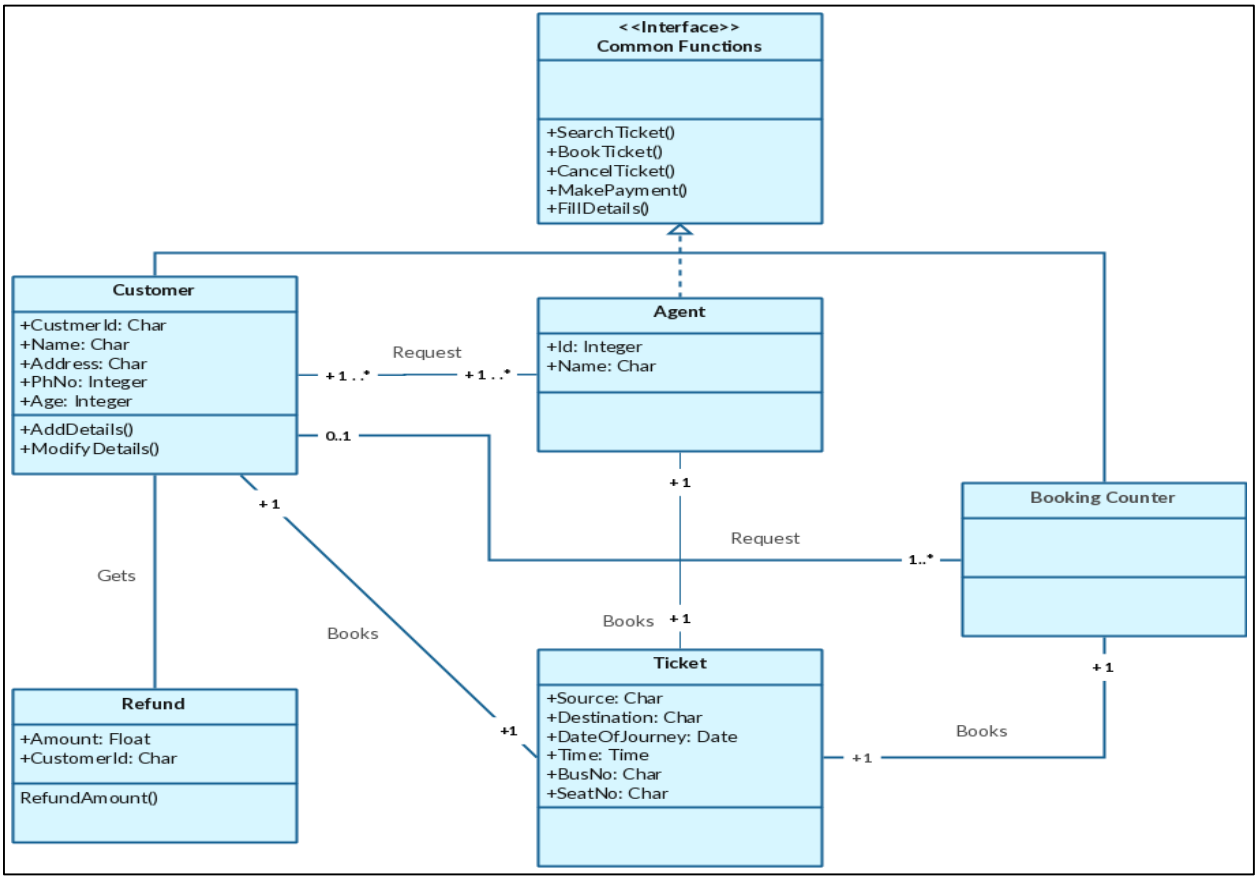
Practice Task 4: Setting Up Class Structure

Convert the following scenario into java code.

Design a university management system, an interface of 'Employees' that will have two/three method declarations. This interface will be implemented by two classes i.e. Admin and Faculty. The faculty class will be extended by Professor, Assistant Professor, and Lecturer. On the other hand, admin class will be extended by examination and admission classes. All these inheritances will be called in the main class by instantiation. (The internal mechanism or method implementation is choice dependent; you can perform whatever you want).

Practice Task 5: Setting Up Class Structure- Mapping Design Elements to Code

Convert the following class diagram into optimized and meaningful java code.



Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 5: Introduction to Concurrency and Concurrency Management.

Lab Objectives:

Introduction to Concurrency

Start, Sleep and Stop Methods of Multi-Threading

Inter-Thread Communication

Deadlock in concurrency

INTRODUCTION TO CONCURRENCY

Threading:

A thread is a lightweight sub-process, the smallest unit of processing. It is a separate path of execution. Threads are independent. If there occurs exception in one thread, it doesn't affect otherthreads. It uses a shared memory area. Consider the following example:

```
public class Main {

      public static void main(String[] args) {


            int i, j;

            for(i=0;i<=5;i++)
            {
            System.out.println(i);
            }
            for(j=0;j<=5;j++)
            {
```

Output:

```
0
1
2
3
4
5
0
1
2
3
4
5
```

Multithreading:

Multithreading in Java is a process of executing multiple threads simultaneously. However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takesless time than process. Java Multithreading is mostly used in games, animation, etc.

Advantages of Java Multithreading

- It doesn't block the user because threads are independent, and you can perform multipleoperations at the same time.
- You can perform many operations together, so it saves time.
- Threads are independent, so it doesn't affect other threads if an exception occurs in a singlethread.

Methods of creating threads:

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

Java Thread Example by extending Thread class

```
class Main extends Thread{
public void run(){
System.out.println("thread is running...");

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

```
Main t1=new Main();
t1.start();
 }
   }
```

Java Thread Example by implementing Runnable interface

```
class Main implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Main m1=new Main();
Thread t1 =new Thread(m1);
t1.start();
 }
}
```

Some simple examples of threading with built-in methods:

Naming Thread:

The Thread class provides methods to change and get the name of a thread. By default, each threadhas a name i.e. thread-0, thread-1 and so on. We can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

public String getName(): is used to return the name of a thread.

public void setName(String name): is used to change the name of a thread.

```
class Main extends Thread{
        public void run(){
```

```
        System.out.println("running...");
       }
      public static void main(String args[]){Main
       t1=new Main();

       Main t2=new Main();
       System.out.println("Name of t1:"+t1.getName());
       System.out.println("Name of t2:"+t2.getName());

       t1.start();
       t2.start();

       t1.setName("Saad");
       System.out.println("Name oft1:"+t1.getName());
      }
     }
```

Output:

```
Problems   @ Javadoc   Declaration
<terminated> Main [Java Application] C:\Use
Name of t1:Thread-0
Name of t2:Thread-1
Name of t1:Saad
Name of t2:Thread-1
```

Priority of a Thread (Thread Priority):

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases,thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that whichscheduling it chooses.

3 constants defined in Thread class:

public static int MIN_PRIORITY

public static int NORM_PRIORITY

public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). **The value of MIN_PRIORITY is 1 andthe value of MAX_PRIORITY is 10.**

```
class Main extends Thread{
      public void run(){
               System.out.println("running thread name
is:"+Thread.currentThread().getName());

               System.out.println("running thread priority
is:"+Thread.currentThread().getPriority());


              }
             public static void main(String args[]){
              Main m1=new Main();

              Main m2=new Main();
              m1.setPriority(Thread.MIN_PRIORITY);
              m2.setPriority(Thread.MAX_PRIORITY);
              m1.start();

              m2.start();
             }
            }
```

Output:

```
running thread name is: Thread-1
running thread priority is:10
running thread name is: Thread-0
running thread priority is: 1
```

START, SLEEP AND STOP METHODS OF MULTI-THREADING

Start Method:

thread.start() method causes this thread to begin execution, the Java Virtual Machine calls the run method of this thread. The result is that two threads are running concurrently: the current thread(which returns from the call to the start method) and the other thread (which executes its run method).

- *Syntax: public void start()*

```
public class Main extends Thread
{
    public void run()
    {
        System.out.println("Thread is running...");
    }
    public static void main(String args[])
    {
        Main t1=new Main();
        // this will call run() method
        t1.start();
    }
}
```

Sleep Method:

thread.sleep(long mills) method causes the currently executing thread to sleep for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.

- *Syntax: public static void sleep(long milliseconds)*

```
class Main extends Thread{
      public void run(){
      for(int i=1;i<5;i++){
         try{Thread.sleep(500);}catch(InterruptedException
e){System.out.println(e);}
         System.out.println(i);
       }
      }
      public static void main(String args[]){
       Main t1=new Main();
       Main t2=new Main();

       t1.start();
       t2.start();
      }
     }
```

Stop Method:

The stop() method of thread class terminates the thread execution. Once a thread is stopped, it cannot be restarted by start() method. Whenever we want to stop a thread from running state by calling stop() method of Thread class in Java. This method stops the execution of a running threadand removes it from the waiting threads pool and garbage collected. A thread will also move to the dead state automatically when it reaches the end of its method. The stop() method is deprecatedin Java due to thread-safety issues.

- *Syntax: public final void stop()*

```java
public class Main 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
        Main t1=new Main ();
        Main t2=new Main ();
        Main t3=new Main ();
        // call run() method
        t1.start();
        t2.start();
        // stop t3 thread
        t3.stop();
        System.out.println("Thread t3 is stopped");
    }
}
```

INTER-THREAD COMMUNICATION:

Synchronization in Multi-Threading:

Synchronization is a process of handling resource accessibility by multiple thread requests. The main purpose of synchronization is to avoid thread interference. At times when more than one thread tries to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.

Why we need Synchronization?

If we do not use synchronization, and let two or more threads access a shared resource at the sametime, it will lead to distorted results.

Consider an example, suppose we have two different threads T1 and T2, T1 starts execution and save certain values in a file temporary.txt which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the filetemporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using temporary.txt file, this file will be locked (LOCK mode), and no other threadwill be able to access or modify it until T1 returns.

Types of Thread Synchronization:

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Synchronized method.
2. Synchronized block.

Example with no Synchronization:

In this example, we are not using synchronization and creating multiple threads that are accessingdisplay method and produce the random output.

```
class Table{
void printTable(int n){//method not synchronized
    for(int i=1;i<=5;i++){
        System.out.println(n*i);try{
          Thread.sleep(400);
        }catch(Exception e){System.out.println(e);}
    }

  }
}
class MyThread1 extends Thread{Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}
}
class MyThread2 extends Thread{Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}     }
```

```
class TestThread{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

Output:

```
5
100
10
200
300
15
20
400
500
25
```

Solution with Synchronization Method:

```
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
```

```
      }
  }
}
class MyThread1 extends Thread{

Table t;

MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);

}

}
class MyThread2 extends Thread{

Table t;

MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);

}      }
public class TestThread{
public static void main(String args[]){ Table obj = new
Table();//only one object

MyThread1 t1=new MyThread1(obj);

MyThread2 t2=new MyThread2(obj);t1.start();
t2.start();
}
}
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

Solution with Synchronization Block:

```
class Table{
void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
     System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

class MyThread1 extends Thread{
```

```
Table t;

MyThread1(Table t){

this.t=t;

}

public void run(){

t.printTable(5);

}
}

class MyThread2 extends Thread{

Table t;

MyThread2(Table t){

this.t=t;

}

public void run(){

t.printTable(100);

}       }
public class TestThread{

public static void main(String args[]){ Table obj = new

Table();//only one objectMyThread1 t1=new

MyThread1(obj); MyThread2 t2=new MyThread2(obj);

t1.start();

t2.start();

}       }
```

Output:

```
5
10
15
20
25
100
200
300
400
500
```

Example of inter thread communication in java:

Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running inits critical section and another thread is allowed to enter (or lock) in the same critical section to beexecuted. It is implemented by following methods of Object class:

- wait()
- notify()
- notifyAll()

wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on thisobject, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

```
public final void notify()
```

notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:public

final void notifyAll()

Let's see the simple example of inter thread communication.

```
class Customer{
int amount=10000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw...");
if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}    }
class Thread1 extends Thread
```

```
{
        Customer c=new Customer();

        public void run()
        {
                c.withdraw(15000);
        }
}
class Thread2 extends Thread
{
        Customer c=new Customer();

        public void run()
        {
                c.deposit(10000);
        }

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



        Thread1  t1=new  Thread1();Thread2
        t2=new Thread2();t1.start();
        t2.start();

      }


       }
```

Output:

```
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
```

DEADLOCK IN CONCURRENCY

Deadlock:

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because thesynchronized keyword causes the executing thread to block while waiting for the lock, or monitor,associated with the specified object.

Here is an example:

```
public class TestThread {
    public static Object Lock1 = new Object();
    public static Object Lock2 = new Object();
    public static void main(String args[]) {ThreadDemo1
        T1 = new ThreadDemo1(); ThreadDemo2 T2 =
        new ThreadDemo2(); T1.start();

        T2.start();
    }
    private static class ThreadDemo1 extends Thread {
        public void run() {
            synchronized (Lock1) {
                System.out.println("Thread 1: Holding lock 1...");
                try { Thread.sleep(10); }
                catch (InterruptedException e) {} System.out.println("Thread 1: Waiting for
                lock 2...");

                synchronized (Lock2) {
                    System.out.println("Thread 1: Holding lock 1 & 2...");
                }
            }
        }
```

```
    private static class ThreadDemo2 extends Thread {
        public void run() {
            synchronized (Lock2) {
                System.out.println("Thread 2: Holding lock 2...");
                try { Thread.sleep(10); }
                catch (InterruptedException e) {} System.out.println("Thread 2: Waiting for lock
                1...");
                synchronized (Lock1) {
                    System.out.println("Thread 2: Holding lock 1 & 2...");
                }
            }
        }
    }
}
```

When you compile and execute the above program, you find a deadlock situation and following isthe output produced by the program:

Output

```
Thread 1: Holding lock 1...

Thread 2: Holding lock 2...

Thread 1: Waiting for lock 2...
```

The above program will hang forever because neither of the threads in position to proceed andwaiting for each other to release the lock, so you can come out of the program:

Deadlock Solution:

Let's change the order of the lock and run of the same program to see if both the threads still waitfor each other:

Example

```
public class TestThread {
    public static Object Lock1 = new Object();
    public static Object Lock2 = new Object();
    public static void main(String args[]) {ThreadDemo1
        T1 = new ThreadDemo1(); ThreadDemo2 T2 =
        new ThreadDemo2(); T1.start();

        T2.start();
    }
    private static class ThreadDemo1 extends Thread {
        public void run() {
            synchronized (Lock1) {
                System.out.println("Thread 1: Holding lock 1...");
                try { Thread.sleep(10); }
                catch (InterruptedException e) {} System.out.println("Thread 1: Waiting for lock
                2...");

                synchronized (Lock2) {
                    System.out.println("Thread 1: Holding lock 1 & 2...");
                }
            }
        }
    }
```

```
    private static class ThreadDemo2 extends Thread {
        public void run() {
            synchronized (Lock1) {
                System.out.println("Thread 2: Holding lock 1...");
                try { Thread.sleep(10); }
                catch (InterruptedException e) {} System.out.println("Thread 2: Waiting for lock
                2...");
                synchronized (Lock2) {
                    System.out.println("Thread 2: Holding lock 1 & 2...");
                }
            }
        }
    }
}
```

So just changing the order of the locks prevent the program in going into a deadlock situation and completes with the following result −

Output

```
Thread 1: Holding lock 1...

Thread 1: Waiting for lock 2...

Thread 1: Holding lock 1 & 2...

Thread 2: Holding lock 1...

Thread 2: Waiting for lock 2...
```

Practice Task:

1. Implement the following program on eclipse IDE and answer the following questions:
   - How many threads are running?
   - How many tasks are running?
   - If more tasks are added, then what will be the impact on number of threads?
   - Explain the flow of program:

```
class Main extends Thread{
      public void run(){
        System.out.println("task one");

      }
      public static void main(String args[]){
       Main t1=new Main();

       Main t2=new Main();
       Main t3=new Main();
       t1.start();
       t2.start();
       t3.start();
      }
```

2. With the help of threading print two tables concurrently, print one table number of student rollnumber e.g. 2020-SE-092 and second number should be date of birth e.g. 05-April.

3. By using start, stop and sleep methods of threading, print alphabets of English from A-Z. (Hint: use math.random method for getting random numbers and then convert them into characters, print26 characters under run method loop with fluctuating visualization through sleep method).

4. Design a simple program of concurrency by implementing the scenario of two account holdersin a joint bank account. (Hint: Total amount will be 50000, if ‘user A’ wants to withdraw 45,000 and ‘user B’ wants to withdraw 20,000) Apply mechanism of synchronization e.g. Block or Method for handling accessibility of multi-threads:

5. Create an inter thread communication program of printer job by implementing two threads, one for calculating the remaining pages in printer tray and other one will print the pages that are pending on queue. (Hint: If total pages are 10 and user sends job for 15 pages than print thread will be on wait and will be notified once available pages are equal or greater than printing pages).

6. Create three threads by implementing thread synchronization block through 3 locks. (Hint: Apply un-sequenced lock to analyze deadlock and solve it through provided solution)

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 6: Ensuring Robustness and Flexibility in your code: Introducing concepts of Mutability and Immutability and Generics in Java.

Part 1: Mutability and Immutability

Objective:

Understanding and implementing the concept of mutability and immutability.

Mutable means the objects can be changes and immutable means cannot be changed. In the Java programming language there are many mechanisms through which mutability and immutability are achieved.

Mutable object:

The objects in which you can change the fields and states after the object is created are known as Mutable objects. Example: java.util.Date, StringBuilder, and etc.

Immutable object:

The objects in which you cannot change anything once the object is created are known as Immutable objects. Example: Boxed primitive objects like Integer, Long and etc.

Difference:

Mutable	**Immutable**
Fields can be changed after the objectcreation.	Fields cannot be changed after object creation.
Generally provides a method to modify the field value.	Does not have any method to modify thefield value.
Has Getter and Setter methods.	Has only Getter method
Example: StringBuilder	Example: String, Boxed primitive objectslike Integer, Long and etc.

How to create a Mutable class?

To create a mutable class in Java you have to make sure the following requirements aresatisfied:

- Provide a method to modify the field values
- Getter and Setter Method Consider the following code:

```java
public class Use {
private String coursename;
Use(String coursename)
{
      this.coursename = coursename;
}
public String getName()
{
      return coursename;
}
public void setName(String coursename)
{
      this.coursename = coursename;
}
public static void main(String[] args)
{
      System.out.println("Adding  Previous  course!");
      Use obj = new Use("Software Design Architecture");
      System.out.println("" +obj.getName()+"\n");

// update the name, this object is mutable
      System.out.println("Updating current course!");
      obj.setName("Software Construction and Development");
     System.out.println("" +obj.getName());
           }
     }
```

```
Adding Previous course!
Software Design Architecture

Updating current course!
Software Construction and Development
```

How to create an Immutable class?

- To create an immutable class in Java you have to make sure the following requirements are satisfied:
- A class should be declared as final so that it can’t be extended.
- All the fields should be made private so that direct access is not allowed
- No setter methods
- Make all mutable fields final, so that they can be only assigned once.

```java
public final class Use {
      final String courseName;
      final int rollNumber;

public Use (String courseName, int rollNumber) {
      this.courseName = courseName;
      this.rollNumber = rollNumber;
}
public String getCourseName() {
      return courseName;
}
public int getRollNumber() {
      return rollNumber;
}
public static void main(String[] args) {
      Use obj = new Use  ("Software Construction and Development"
, 100);
// Since no setters are available contents cannot be modified.
// Also as variables are declared final they cannot be modified
      System.out.println("Course is " + obj.getCourseName());
      System.out.println("Roll Number is " +
obj.getRollNumber());
}
}
```

```
Course is Software Construction and Development
Roll Number is 100
```

Practice Task 1:

Write a program that will calculate the below 4 formulas. Decide what to make mutable and what to make immutable and perform task operations. Formulas are: Circumference of circle: $C = 2 \pi r$

Area of circle: $A = \pi r^2$

Volume of sphere: $V = 4/3 \pi r3$

Surface area of sphere: $A = 4 \pi r^2$

(Hint: Value of pie would be constant and value of radius should be variant)

Practice Task 2:

Apply concept of mutability and immutability for the task promoted and failed students. The roll number, subject code, and subject name would have to be entered at time of object creation and with getter method these variables should be printed. (Hint: Those students who are failed in previous semester will be registered in immutable class, and promoted students are registered in mutable class)

Part 2: Generics in Java

Objective: Implementing generic classes and methods.

Generics:

Generics is a type of specified data type in the related context of class, method or object. In order to restrict any class, method or object with specified type then generics should be introduced in the code. Generics means parameterized types. The idea is to allow type (Integer, String etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is called generic entity.

Importance of generics:

Object is the superclass of all other classes and Object reference can refer to any type object. These features lack type safety. Generics adds that type safety feature.

Generics Parameters name:

- E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
- K – Key (Used in Map)
- N – Number
- T – Type
- V – Value (Used in Map)
- S,U,V etc. – 2nd, 3rd, 4th types

Types of generics:

1. **Generic Class**

We use < > to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

```
class Test<T>
{
        // An object of type T is declaredT obj;
        Test(T obj)
  {
this.obj = obj;
  }   // constructor
        public T getObject()
{
  return this.obj;
}
}

// Driver class to test above
class Main
{
        public static void main (String[] args)
        {
                // instance of Integer type
                Test <Integer> iObj = new Test<Integer>(84);
                System.out.println(iObj.getObject());

                // instance of String type
                Test <String> sObj = new Test<String>("Hiba Fatima");
                System.out.println(sObj.getObject());
        }
}
```

We can also pass multiple Type parameters in Generic classes:

```
class Test<T, U>
{
    T obj1;   // An object of type TU obj2;   // An object of
    type U

    // constructor Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }
    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}
// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =
                        new Test<String, Integer>("Hiba Fatima", 84);
                         obj.print();
                        }
    }
```

```
Hiba Fatima
84
```

2. **Generic Functions:**

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

```
class Main
{
    // A Generic method example
    static <T> void genericDisplay (T element)
    {
        System.out.println("Value is" +
                                    " = " + element);
    }
    // Driver method
    public static void main(String[] args)
    {
        // Calling generic method with Integer argument
        genericDisplay(11);
        // Calling generic method with String argument
        genericDisplay("Software Construction and Development");
        // Calling generic method with double argument
        genericDisplay(1.0);
    }
}
```

```
Value is = 11
Value is = Software Construction and Development
Value is = 1.0
```

Example of a program without generic method and classes:

```
public class Main
{ public static void main(String[] args)   {
  // create arrays of Integer, Double and CharacterInteger[]
integerArray = {1, 2, 3, 4, 5, 6};
Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7};
Character[] characterArray = {'H', 'E', 'L', 'L', 'O'};

System.out.printf("Array integerArray contains:%n");
printArray(integerArray);
System.out.printf("%nArray doubleArray contains:%n");
printArray(doubleArray);
System.out.print("Array characterArray contains:\n");
printArray(characterArray);
}
public static void printArray(Integer[] inputArray)
{
for (int element : inputArray)
System.out.printf("%s ", element);

}
public static void printArray(Double[] inputArray)
{
for ( double element : inputArray)
System.out.printf("%s ", element);
System.out.println();
}
public static void printArray(Character[] inputArray)
{
for ( char element : inputArray)
System.out.printf("%s ", element);
System.out.println();
}
}
```

```
Array integerArray contains:
1 2 3 4 5 6
Array doubleArray contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array characterArray contains:
H E L L O
```

In the above code, three overloaded methods are implemented for getting integer, double or character type input. This long described pattern can be transformed into a single method by simplyadding generic data type as below:

```
public static void printArray(E[] inputArray)
{
// display array elements
for (E element : inputArray) System.out.printf("%s ",
element);System.out.println();
}
```

Solution with Generics:

```
public class Main {
        // generic method printArray
        public static < E > void printArray( E[] inputArray ) {
            // Display array elements
            for(E element : inputArray) {
                System.out.printf("%s ", element);
            }
            System.out.println();
        }
        public static void main(String args[]) {
            // Create arrays of Integer, Double and CharacterInteger[] intArray
            = { 1, 2, 3, 4, 5 };
            Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 }; Character[] charArray = { 'H',
            'E', 'L', 'L', 'O' };

            System.out.println("Array integerArray contains:");
            printArray(intArray);                // pass an Integer array
            System.out.println("\nArray doubleArray contains:");
            printArray(doubleArray);             // pass a Double array
            System.out.println("\nArray characterArray contains:");
            printArray(charArray);               // pass a Character array
        }
    }
```

```
Array integerArray contains:
1 2 3 4 5

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

Practice Task 3:

Write a program that takes integer array, double array and character array. Make a generic function that prints these array in reverse order.

Practice Task 4:

Write a generic method to find the maximal element in the range [begin, end] of a list.

Practice Task 5:

Write a generic method to exchange the positions of two different elements in an array.

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 7: Mastering Recursion-Exploring Recursive Algorithms

OBJECTIVE: To study and Implement concept of Recursion in Java

Recursion in java is a process in which a method calls itself continuously. A method in java that callsitself is called recursive method.

Java Recursion Example: Factorial of a Number

```
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));

          }

 }
```

Output:

Factorial of 5 is: 120

Working of above program:

```
factorial(5) factorial(4)
     factorial(3) factorial(2)
          factorial(1)

             return 1

          return 2*1 = 2return 3*2 = 6
     return 4*6 = 24
  return 5*24 = 120
```

Halting Condition/Base Condition: Just as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stopscalling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself. In the previous example, the halting condition is when the parameter n becomes0.

<u>Practice Tasks:</u>

Practice Task 1:

Write a recursive function to add all the numbers from 0 up to 10. Then draw a diagram to show how the stack grows as main calls add function and add function then calls itself, until add(0) does not make a recursive call. Then the call stack unwinds, each call to add function returning its answer to the caller, until add(10) returns to main.

Practice Task 2:

Write a recursive function to compute the value of x raised to the power of y (x^y).

Practice Task 3:

Implement a function that returns the nth number in the Fibonacci sequence using recursion. The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding ones (0, 1, 1, 2, 3, 5, 8, ...).

Practice Task 4:

Implement a recursive function to reverse a string. For example, given the input "hello", the function should return "olleh"

Practice Task 5:

Scenario: Tower of Hanoi Challenge

You're part of a team developing an innovative game that requires solving puzzles using recursive algorithms. Your task is to create a Tower of Hanoi challenge within the game to test players' problem-solving skills.

Scenario Description:

In this game scenario, players encounter a mystical tower consisting of three pegs and a stack of disks, each of varying sizes, initially placed in ascending order on one peg. The objective is to move the entire stack of disks from the source peg to another peg, adhering to the Tower of Hanoi rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or an empty peg.
3. The player must move the entire stack of disks to another peg, using the third peg as an intermediary, adhering to the minimum number of moves required.

Your Objective:

As the game developer, you need to create a challenging Tower of Hanoi puzzle within the game. Design a scenario where players encounter a tower with a specific number of disks and are tasked with moving the entire stack from one peg to another using the least number of moves.

Challenge Parameters:

- Number of disks: 5
- Starting configuration: Disks placed in ascending order on the first peg ('A').
- Destination peg: Second peg ('B').
- Intermediate peg: Third peg ('C').

Game Execution:

1. The player starts with the stack of 5 disks on peg 'A'.
2. They must utilize their problem-solving skills to move all the disks from peg 'A' to peg 'B' using the Tower of Hanoi rules.
3. The game should display the number of moves taken by the player to complete the puzzle.
4. The game should reward players for completing the challenge in the minimum number of moves required for this configuration (i.e., 31 moves).

Success Criteria:

Players successfully complete the Tower of Hanoi challenge when they manage to move the entire stack of 5 disks from peg 'A' to peg 'B' using the minimum number of moves (31 moves) as per the Tower of Hanoi rules.

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 8: Exception Handling

OBJECTIVE: Constructing a fault tolerant program by implementing exception handling techniques.

Exception:

An exception is an unwanted or unexpected event, which occurs during the execution of a program

i.e. at run time, that interrupts the normal flow of the program's instructions. When an exception occurs, program execution gets terminated. In such cases we get a system generated error message.

Exception Handling in Java:

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 in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of exception handling:

Exception handling ensures that the flow of the program does not break when an exception occurs. For example, if a program has bunch of statements and an exception occurs mid-way after executing certain statements then the statements after the exception will not execute and the program will terminate abruptly. By handling we make sure that all the statements execute, and the flow of program does not break.

Java Exception Keywords:

There are 5 keywords which are used in handling exceptions in Java.

Keyword	Description
Try	The try block contains set of statements where an exception can occur. 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. We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom 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. The Java throws keyword 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.

Try-catch block Example:

```
public class Example1 {

      public static void main(String args[]) {

            int num1, num2;

            try {

               /* We suspect that this block of statement can throw

                * exception so we handled it by placing these statements
                * inside try and handled the exception in catch block
                */

               num1 = 0;

               num2 = 62 / num1; System.out.println(num2);

               System.out.println("Hey I'm at the end of try block");

            }

            catch (ArithmeticException e) {

               /* This block will only execute if any Arithmetic exception

                * occurs in try block
                */

               System.out.println("You should not divide a number by zero");

            }

            catch (Exception e) {

        /* This is a generic Exception handler which means it can handle

            * all the exceptions. This will execute if the exception is not

                * handled by previous catch blocks.

                */

              //This catch block catches all the exceptions


               System.out.println("Exception occurred");

                  }

            System.out.println("I'm out of try-catch block in Java.");

         }

}
```

Output:

```
You should not divide a number by zero
I'm out of try-catch block in Java.
```

```java
public class Use {

    public static void main(String[] args) {

        int i=50; int j=0; int data;

        try

        {  data=i/j; //may throw exception      }

       // handling the exception

        catch(Exception e)

        {   // resolving the exception in catch block

            System.out.println(i/(j+2)); }

       } }
```

Finally block Example:

```
public class FinallyBlock {
      public static void main(String args[]) {
            try{
             int num=121/0;
             System.out.println(num);
            }
            catch(ArithmeticException e){
               System.out.println("Number should not be divided by zero");
            }
            /* Finally block will always execute
             * even if there is no exception in try block
             */
            finally{
             System.out.println("This is finally block");
            }
            System.out.println("Out of try-catch-finally");
         }

}
```

Output:

```
Number should not be divided by zero
This is finally block

Out of try-catch-finally
```

Throw Keyword Example:

```
public class throwExample {
      static void checkEligibilty(int age, int weight){
            if(age<12 && weight<40) {
               throw new ArithmeticException(" Not Eligible for
registration");
            }
            else {
               System.out.println("Entry is Valid!!");
            }
         }

         public static void main(String args[]){
           System.out.println("Welcome to the Registration process!!");
           checkEligibilty(10, 39);

      }
}
```

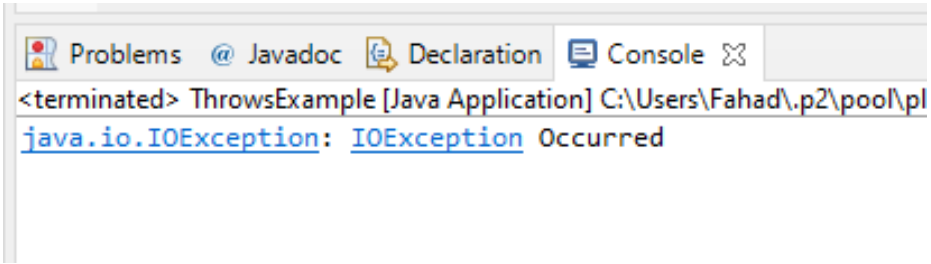
Output:

```
Welcome to the Registration process!!
Exception in thread "main" java.lang.ArithmeticException: Not Eligible for registration
        at throwExample.checkEligibilty(throwExample.java:5)
        at throwExample.main(throwExample.java:14)
```

Throws Keyword example:

```
import java.io.IOException;

public class ThrowsExample {

      void myMethod(int num)throws IOException, ClassNotFoundException{

           if(num==1)

              throw new IOException("IOException Occurred");

           else

              throw new ClassNotFoundException("ClassNotFoundException");

        }

        public static void main(String args[]){

         try {

           ThrowsExample obj=new ThrowsExample();
            obj.myMethod(1);

                  }

         catch(Exception ex)

         {

           System.out.println(ex);

          }

        }

}
```

Output:



Throw vs Throws in java

Throws clause is used to declare an exception, which means it works similar to the try-catch block. On the other hand, **throw** keyword is used to throw an exception explicitly.

Throw:

```
...
void myMethod() {
   try {
      //throwing arithmetic exception using throw
      throw new ArithmeticException("Something went wrong!!");
   }


   catch (Exception exp) {
      System.out.println("Error: "+exp.getMessage());
   }
   }
```

Throws:

```
...
//Declaring arithmetic exception using throws
void sample() throws ArithmeticException{
   //Statements
```

You can **throw** one exception at a time, but you can handle multiple exceptions by declaring them using **throws** keyword.

Throw:

```
void myMethod() {
   //Throwing single exception using throw
   throw new ArithmeticException("An integer should not be divided by zero!!");
```

Throws:

```
//Declaring multiple exceptions using throws
void myMethod() throws ArithmeticException, NullPointerException{
   //Statements where exception might occur
}
```

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked.

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, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime. Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Built-in Exceptions

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

Arithmetic Exception:

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

Output: Can't divide a number by 0

NullPointer Exception:

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

Output: NullPointerException..

StringIndexOutOfBound_Exception:

```
public class StringIndexOutOfBound_Demo {
    public static void main(String args[])
    {
        try {
            String a = "This is like chipping "; // length is 22
            char c = a.charAt(24); // accessing 25th element
            System.out.println(c);
        }
        catch(StringIndexOutOfBoundsException e) {
```

Output: StringIndexOutOfBoundsException

ArrayIndexOutOfBounds Exception:

```
public class ArrayIndexOutOfBound_Demo {
      public static void main(String args[])
    {
        try{
            int a[] = new int[5];
            a[6] = 9; // accessing 7th element in an array of
                      // size 5
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println ("Array Index is Out Of Bounds");
        }
    }
}
```

```
Output: Array Index is Out Of Bounds
```

Practice Task 1:

Write a program that meets the following requirements:

- Creates an array with 10 randomly chosen integers.
- Prompts the user to enter the index of the array, then displays the corresponding element value. If the specified index is out of bounds, display the message Out of Bounds. (ArrayIndexOutOfBoundsException)

Practice Task 2:

Suppose you are developing a game for kids in which they are learning the division operation in math. Your game will take input from kids (2 integers) and then perform the division and displays the answer. Think and apply exception handling in this scenario. For e.g. Arithmetic Exception might occur here. Also suppose, this game has the limitation that it only performs division between integers so if it gets a decimal number as input, it **throws** an exception stating that the input is invalid, please give integer number etc.

Practice Task 3:

Develop a simple banking application that includes a **BankAccount** class. Implement methods for depositing and withdrawing funds from an account. Incorporate exception handling to manage cases such as attempting to withdraw more funds than are available, depositing negative amounts, or any other potential errors that might occur during these operations.

Practice Task 4:

Create a program that simulates a library catalog system. Design a Book class with attributes like title, author, and publication year. Implement a Library class that manages a collection of books and includes a method **findBook(String bookTitle)** which searches for a book by its title. If the book is not found, throw a custom exception **(BookNotFoundException)** that you've created using the throw keyword to handle this scenario gracefully. Additionally, demonstrate the catching of this custom exception in your main program.

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 9: Refactoring

LAB OBJECTIVES:

- Identifying Code Smells
- Understanding and applying various refactoring techniques to improve readability and maintainability.

Refactoring is the process of restructuring existing computer code or software without changing its external behavior. Its primary goal is to improve the code's readability, maintainability, and sometimes performance without altering what it does. Refactoring makes the code easier to understand, modify, and extend while reducing technical debt.

Use of Refactoring:

1. **Code Maintainability:** Refactoring enhances code readability and structure, making it easier for developers to maintain and understand.
2. **Bug Fixing:** It can help identify and eliminate bugs or defects in the code.
3. **Performance Improvement:** Certain refactoring can optimize code performance.
4. **Code Reusability:** Well-refactored code often results in reusable components, which can save time in future development.
5. **Team Collaboration:** It aids in team collaboration as cleaner code is easier for others to comprehend and work on.

Techniques of Refactoring:

- **Extract Method:** Breaks down a chunk of code into a separate method to improve readability and promote reusability.
- **Rename Variables/Methods:** Using more descriptive names that accurately represent their purpose improves code readability.
- **Merge Methods/Classes:** Combines two methods or classes that have similar functionality to eliminate redundancy.
- **Remove Code Duplication:** Identifies repeated code segments and consolidates them into a single function or method.
- **Simplify Conditional Expressions:** Reduces complexity in conditional statements to make the logic clearer.
- **Optimize Loops:** Simplify loops or replace them with more efficient constructs when possible.
- **Code Smells Identification:** Detect and address common indicators (code smells) that suggest areas needing refactoring, like long methods or excessive parameters.

Let’s implement Refactoring techniques in multiple scenarios:

Scenario-1:

You have a small program that manages shapes. There's a class Shape with subclasses **Circle** and **Square**. Each shape has a method to calculate its area, but the implementation is repetitive and can be refactored.

```
// Shape.java
public abstract class Shape {
    public abstract double calculateArea();
}
// Circle.java
public class Circle extends Shape {
    private double radius;
    public Circle(double radius) {
        this.radius = radius;
    }
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}
// Square.java
public class Square extends Shape {
    private double side;
    public Square(double side) {
        this.side = side;
    }
    @Override
    public double calculateArea() {
        return side * side;
    }      }
```

```
// Main.java
public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle(5);
        Square square = new Square(4);
        System.out.println("Circle Area: " + circle.calculateArea());
        System.out.println("Square Area: " + square.calculateArea());
    }
}
```

Practice Task 1: Refactoring Tasks:

1. Extract Common Code: Identify the repetition in the calculateArea() methods of Circle and Square and refactor it to the Shape class.

2. Use Polymorphism: Utilize polymorphism to implement the area calculation directly in each subclass without overriding.

Refactoring Steps:

1. Open Eclipse and create a Java project.
2. Create new classes Shape, Circle, Square, and Main.
3. Copy-paste the initial code into their respective classes.
4. Refactor calculateArea() method: Identify common code, move it to the Shape class, and adjust the subclasses accordingly.
5. Run the program to ensure it still calculates the areas correctly.

Scenario-2:

You have a library system with a Book class that contains information about books and a Library class responsible for managing these books.

// **Book.java**

```
public class Book {
   private String title;
   private String author;
   private int pageCount;
public Book(String title, String author,
int pageCount) {
      this.title = title;
      this.author = author;
      this.pageCount = pageCount;
   }
public String getTitle() {
      return title;
   }
public String getAuthor() {
      return author;
   }
public int getPageCount() {
      return pageCount;
   }
}
```

// Library.java

```
import java.util.ArrayList; import java.util.List;
public class Library {
   private List<Book> books;
public Library() {
      this.books = new ArrayList<>();        }
public void addBook(Book book) {
      books.add(book); }
public void displayBooks() {
      for (Book book : books) {
         System.out.println("Title: " + book.getTitle() +
                        ", Author: " + book.getAuthor() +
                        ", Page Count: " + book.getPageCount());}}}
```

// Main.java

```
public class Main {
   public static void main(String[] args) {
      Library library = new Library();
      Book book1 = new Book("1984", "George Orwell", 328);
      Book book2 = new Book("To Kill a Mockingbird", "Harper
Lee", 281);
      library.addBook(book1);
      library.addBook(book2);
      library.displayBooks();
   }       }
```

<u>Practice Task 2:</u> Refactoring Tasks:

1. Encapsulation: Implement encapsulation by using getters and setters in the Book class.
2. Separation of Concerns: Refactor the Library class to separate book addition and book display functionalities.
3. Enhanced Display: Improve the book display format to provide better information.

Refactoring Steps:

1. Open Eclipse and create a Java project.
2. Create new classes Book, Library, and Main.
3. Copy-paste the initial code into their respective classes.
4. Refactor the Book class to use encapsulation (getters and setters for private fields).
5. Refactor the Library class to provide a method to retrieve books and remove the displayBooks() method.
6. Update the Main class to use the refactored methods and display book information in an enhanced format.

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 10: Understanding and applying SOLID Principles

LAB OBJECTIVES:

- Understanding and applying SOLID Principles
- Identifying Violations of SOLID Principles
- Identifying Code Smells
- Understanding and applying various refactoring techniques to improve readability and maintainability.

SOLID principles are a set of five design principles intended to make software designs more understandable, flexible, and maintainable. Each principle addresses a specific aspect of software design.

1. S - Single Responsibility Principle (SRP):

This principle states that a class should have only one reason to change, meaning it should have only one job or responsibility. When a class or module handles multiple responsibilities, it becomes more difficult to maintain, test, and reason about.

2. O - Open/Closed Principle (OCP):

The Open/Closed Principle states that software entities (classes, modules, functions) should be open for extension but closed for modification. This means that you should be able to extend a system's behavior without modifying its existing code. Achieve this by using inheritance, interfaces, and design patterns.

3. L - Liskov Substitution Principle (LSP):

LSP emphasizes that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In simpler terms, any class that is a subclass of another should be usable in place of its parent class without causing issues in the program.

4. I - Interface Segregation Principle (ISP):

ISP suggests that a client should not be forced to depend on methods it does not use. Instead of creating large interfaces with many methods, it's better to have smaller, more specific interfaces. This principle helps in preventing the creation of "fat" interfaces where implementing classes are forced to provide unnecessary functionality.

5. D - Dependency Inversion Principle (DIP):

DIP promotes two key ideas: high-level modules/classes should not depend on low-level modules/classes but rather on abstractions (interfaces or abstract classes). Additionally, abstractions should not depend on details; details should depend on abstractions. This principle encourages decoupling between modules and promotes easier testing, maintenance, and flexibility in the system.

Let’s implement SOLID principles and Refactoring techniques in multiple scenarios:

Scenario-1:

You have a vehicle rental service with various vehicle types (Cars, Bikes, Vans). Each vehicle has specific functionalities like renting, returning, and getting details. We'll adhere to the following principles:

Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details; details should depend on abstractions.

Single Responsibility Principle (SRP): A class should have only one reason to change.

Interface Segregation Principle (ISP): No client should be forced to depend on methods it does not use.

```
// Vehicle.java
public interface Vehicle {
   void rent();
   void returnVehicle();
   String getDetails();     }
```

```
// Car.java
public class Car implements Vehicle {
   @Override
   public void rent() {
      System.out.println("Car rented."); }
     @Override
   public void returnVehicle() {
      System.out.println("Car returned.");    }
   @Override
   public String getDetails() {
      return "Car details: Sedan, 4 doors."; }}
```

```
// Van.java
public class Van implements Vehicle {
   @Override
   public void rent() {
      System.out.println("Van rented.");
   }
```

```
// Bike.java
public class Bike implements Vehicle {
   @Override
   public void rent() {
      System.out.println("Bike rented."); }
  @Override
   public void returnVehicle() {
      System.out.println("Bike returned."); }
   @Override
   public String getDetails() {
      return "Bike details: Mountain bike.";
   }
}
```

```
// Main.java
public class Main {
   public static void main(String[] args) {
      Vehicle car = new Car();
      RentalService rentalService = new RentalService(car);
      rentalService.rentVehicle();
      rentalService.returnRentedVehicle();
      System.out.println(rentalService.getVehicleDetails());
   }
}
```

```
    @Override
    public void returnVehicle() {
        System.out.println("Van returned.");
    }
    @Override
    public String getDetails() {
        return "Van details: Cargo van.";
    }
}
```

// RentalService.java

```
public class RentalService {
    private Vehicle vehicle;
    public RentalService(Vehicle vehicle) {
        this.vehicle = vehicle;
    }
     public void rentVehicle() {
        vehicle.rent();
    }
    public void returnRentedVehicle() {
        vehicle.returnVehicle();
    }
    public String getVehicleDetails() {
        return vehicle.getDetails();
    }
}
```

<u>Practice Task 1:</u> Refactoring Tasks:

1. Dependency Inversion Principle (DIP): Introduce an abstraction to decouple the RentalService from specific vehicle types.
2. Single Responsibility Principle (SRP): Split the vehicle functionalities into separate interfaces based on responsibilities.

3. Interface Segregation Principle (ISP): Ensure that interfaces are tailored to specific needs.

Refactoring Steps:

1. Introduce Rentable and VehicleDetails interfaces to segregate functionalities.
2. Update vehicle classes to implement the appropriate interfaces (Rentable and VehicleDetails).
3. Modify RentalService to depend on the Rentable interface rather than concrete vehicle types.
4. Update the Main class to use the interfaces instead of concrete classes.

Scenario-2:

We're building a system to handle courses offered by a university. Each course has specific information and functionalities like enrollment and displaying details.

Structure:

- **Dependency Inversion Principle (DIP):** Implement an abstraction layer for courses to decouple higher-level modules from specific course implementations.
- **Single Responsibility Principle (SRP):** Ensure each class has a single responsibility and adheres to handling a specific aspect of the system.
- **Interface Segregation Principle (ISP):** Create interfaces that cater to specific functionalities related to courses.

Classes:

- **Course:** Represents a course with enrollment and details functionalities.
- **EnrollmentService:** Manages the enrollment process for courses.
- **CourseDetails:** Handles the details information for courses.

// Course.java

```
public class Course {

   private String courseName;

   private int courseId;

   private int enrolledStudents;

   public Course(String courseName, int courseId) {

      this.courseName = courseName;

      this.courseId = courseId;

      this.enrolledStudents = 0;

   }

   public void enrollStudent() {

      enrolledStudents++;

      System.out.println("Student enrolled in " + courseName + ".");
```

```
    }
    public void displayCourseDetails() {
        System.out.println("Course ID: " + courseId + ", Name: " + courseName + ", Enrolled Students: " +
enrolledStudents);
    }
}
```

// EnrollmentService.java

```
public class EnrollmentService {
    public void enrollStudentToCourse(Course course) {
        course.enrollStudent();
    }
}
```

// Main.java

```
public class Main {
    public static void main(String[] args) {
        Course mathCourse = new Course("Mathematics", 101);
        EnrollmentService enrollmentService = new EnrollmentService();
        enrollmentService.enrollStudentToCourse(mathCourse);
        mathCourse.displayCourseDetails();
    }
}
```

<u>Practice Task 2:</u>

Task 1 (DIP): Introduce a new course and enroll a student using the CourseEnrollment interface.

Task 2 (SRP): Modify the course details using the CourseDetails interface to display course information.

Task 3 (ISP): Implement a new course that adheres to the CourseEnrollment and CourseDetails interfaces.

<u>Steps:</u>

1. Create new courses implementing CourseEnrollment and CourseDetails.
2. Use the EnrollmentService to enroll students into different courses.
3. Ensure the course details are displayed correctly using the displayCourseDetails method.

Scenario-3:

We're building a simplified banking system that manages different types of accounts (Savings, Checking). Each account has specific functionalities like deposit, withdrawal, and balance inquiry.

Structure:

- **Dependency Inversion Principle (DIP):** Introduce an abstraction layer for accounts to decouple higher-level modules from specific account implementations.
- **Single Responsibility Principle (SRP):** Ensure each class has a single responsibility and handles a specific aspect of the banking system.
- **Interface Segregation Principle (ISP):** Create interfaces catering to specific functionalities related to accounts.

Classes:

- **Account:** Represents a bank account with transaction functionalities.
- **TransactionService:** Manages deposit, withdrawal, and balance inquiry for accounts.

// Account.java

```
public class Account {

   private String accountType;

   private double balance;

   public Account(String accountType) {

      this.accountType = accountType;

      this.balance = 0.0;

   }

   public void deposit(double amount) {

      balance += amount;

      System.out.println("Deposited: $" + amount + " into " + accountType + " account.");

   }

   public void withdraw(double amount) {

      if (balance >= amount) {

         balance -= amount;

         System.out.println("Withdrawn: $" + amount + " from " + accountType + " account.");

      } else {

         System.out.println("Insufficient funds in " + accountType + " account.");       }        }

    public double getBalance() {
```

```
        return balance;
    }    }
```

// TransactionService.java

```
public class TransactionService {
    public void depositAmount(Account account, double amount) {
        account.deposit(amount);
    }
    public void withdrawAmount(Account account, double amount) {
        account.withdraw(amount);
    }
    public void checkBalance(Account account) {
        System.out.println("Current balance in " + account + " account: $" + account.getBalance());  }        }
```

// Main.java

```
public class Main {
    public static void main(String[] args) {
        Account savingsAccount = new Account("Savings");
        TransactionService transactionService = new TransactionService();
        transactionService.depositAmount(savingsAccount, 1000);
        transactionService.checkBalance(savingsAccount);
        transactionService.withdrawAmount(savingsAccount, 500);
        transactionService.checkBalance(savingsAccount);   }        }
```

<u>Refactoring for Principles:</u>

- **Dependency Inversion Principle (DIP):** Introduce an abstraction layer (BankAccount) to handle transactions for different account types.
- **Single Responsibility Principle (SRP):** Split functionalities related to transactions into separate interfaces.
- **Interface Segregation Principle (ISP):** Create interfaces catering to deposit, withdrawal, and balance inquiry functionalities.

<u>Practice Task 3:</u>

Task 1 (DIP): Create a new type of account and perform transactions using the BankAccount interface.

Task 2 (SRP): Ensure that deposit, withdrawal, and balance inquiry functionalities are utilized through interfaces.

Task 3 (ISP): Implement a scenario where a specific account type only uses required transaction functionalities.

Steps:

1. Create a new type of account implementing BankAccount interface.
2. Use the TransactionService to perform transactions (deposit, withdrawal, balance inquiry) on different accounts.
3. Ensure specific accounts adhere to required transaction interfaces, demonstrating Interface Segregation Principle.

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 11: Application Lab: Applying Previously Studied Concepts

This open-ended lab allows students to take ownership of their learning by choosing topics of personal interest and applying previously studied tasks, fostering creativity, critical thinking, and practical application of knowledge.

Objective: To empower students to select a Java-based project and apply previously learned Java concepts in a practical application.

Steps:

Topic Selection:

Choose a topic of personal interest where you can apply Java techniques studied in class and lab sessions. Ensure it aligns with your passion and allows the application of prior knowledge.

Proposal Submission/Review and Refinement:

Submit a brief proposal outlining your chosen topic, the specific Java techniques you plan to apply, and the intended outcome of your project. Refine the proposal if needed after discussions.

Resource Gathering and Task Planning:

Collect Java programming resources and plan the execution of tasks based on your proposal. Ensure you have the necessary materials to implement your chosen Java techniques.

Task Implementation:

Actively work on executing tasks related to your chosen topic using Java techniques learned in class and previous labs. Apply concepts such as OOP, data structures, etc.

Documentation and Presentation Preparation:

Document your task execution progress, methodologies used, challenges faced, and solutions found. Prepare a brief presentation for showcasing your Java-based task implementations.

Task Presentations:

Present your executed tasks to the class, explaining your approach, Java techniques applied, and the impact on your project. Engage in discussions about Java concepts used.

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 12: JUnit Testing

OBJECTIVE: Study the concept of Test Driven Development under the framework of JUnit Testing.

Test Driven Development (TDD):

Test Driven Development (TDD) is software development approach in which test cases are developed to specify and validate what the code will do. In simple terms, test cases for each functionality are created and tested first and if the test fails then the new code is written in order to pass the test and making code simple and bug-free.

Test-Driven Development starts with designing and developing tests for every small functionality of an application. TDD instructs developers to write new code only if an automated test has failed. This avoids duplication of code. The full form of TDD is Test-driven development.

How to perform TDD Test:

Following steps define how to perform TDD:

- Add a test.
- Run all tests and see if any new test fails.
- Write some code.
- Run tests and Refactor code.
- Repeat.

Unit Testing:

Unit is a Regression Testing Framework used by developers to implement unit testing in Java, and accelerate programming speed and increase the quality of code. JUnit Framework can be easily integrated with either of the following −

- Eclipse
- Ant
- Maven

Features of JUnit Test Framework:

JUnit test framework provides the following important features −

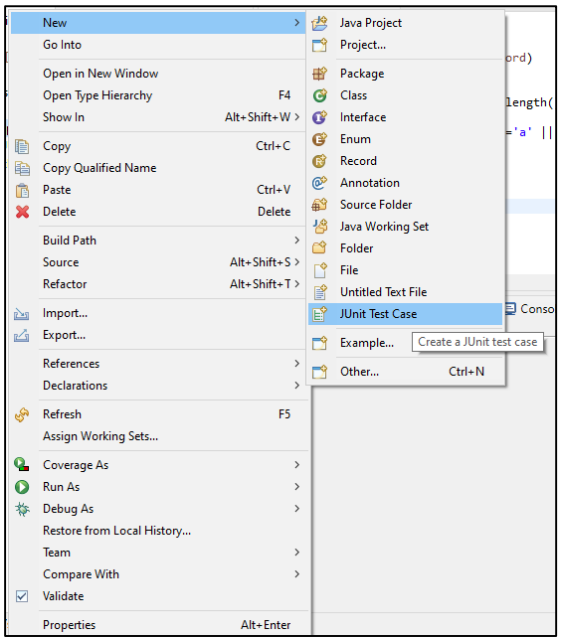
- Test suites: A test suite bundles a few unit test cases and runs them together. In JUnit, both @RunWith and @Suite annotation are used to run the suite test.
- Test runners: Test runner is used for executing the test cases.
- JUnit classes: JUnit classes are important classes, used in writing and testing JUnits.

Steps for creating JUnit Testing on Eclipse:

1. **Create a project and add the following code in class:**

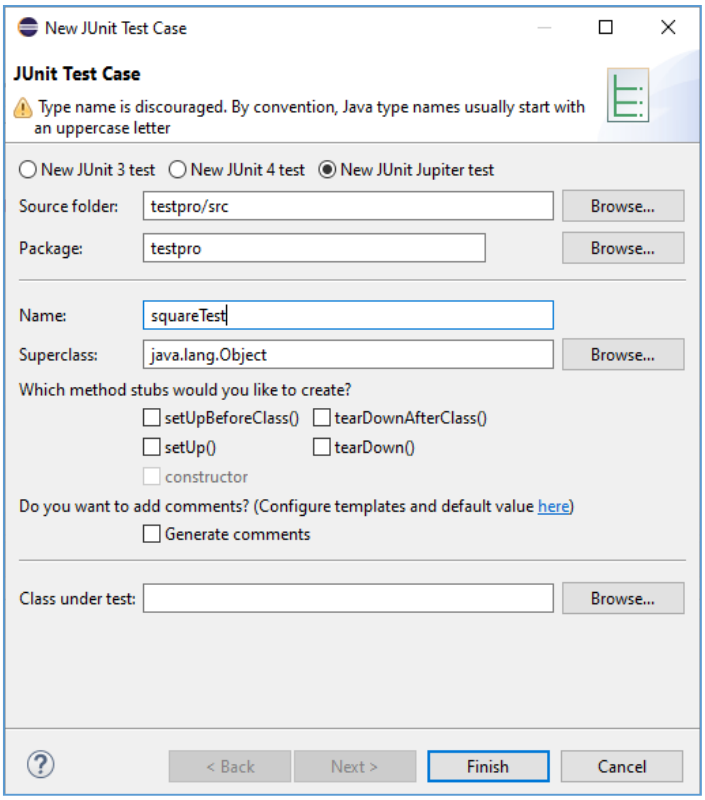
```
public class JUnitTesting {
	public int square (int x) {
		return x*x;
	}
	public int countA(String word)
	{
		int count = 0;
		for (int i=0; i <word.length();i++)
		{
			if(word.charAt(i)=='a' ||
word.charAt(i)=='A')
		{
				count++;
			}
		}
		return count;
	}
}
```

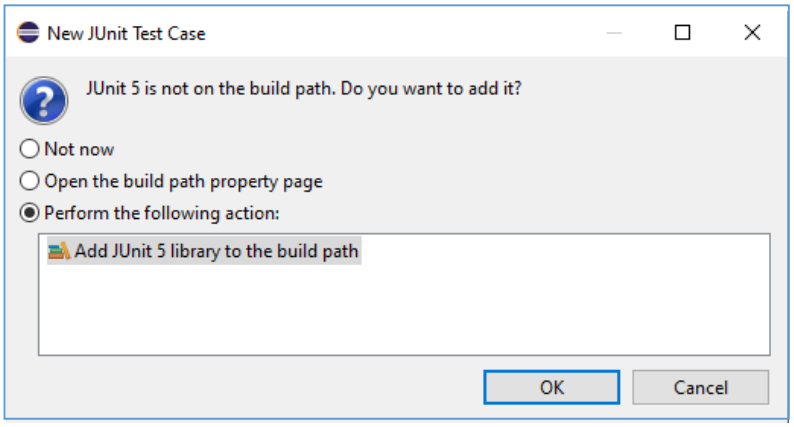
2. **Add JUnit Test case in the same package:**



3. Name the file for testing square function:

Note: Here “New JUnit Jupiter test” is selected in the below dialog box. Select “New JUnit 4test” instead.





4. Now put the test case in test method highlighted below:

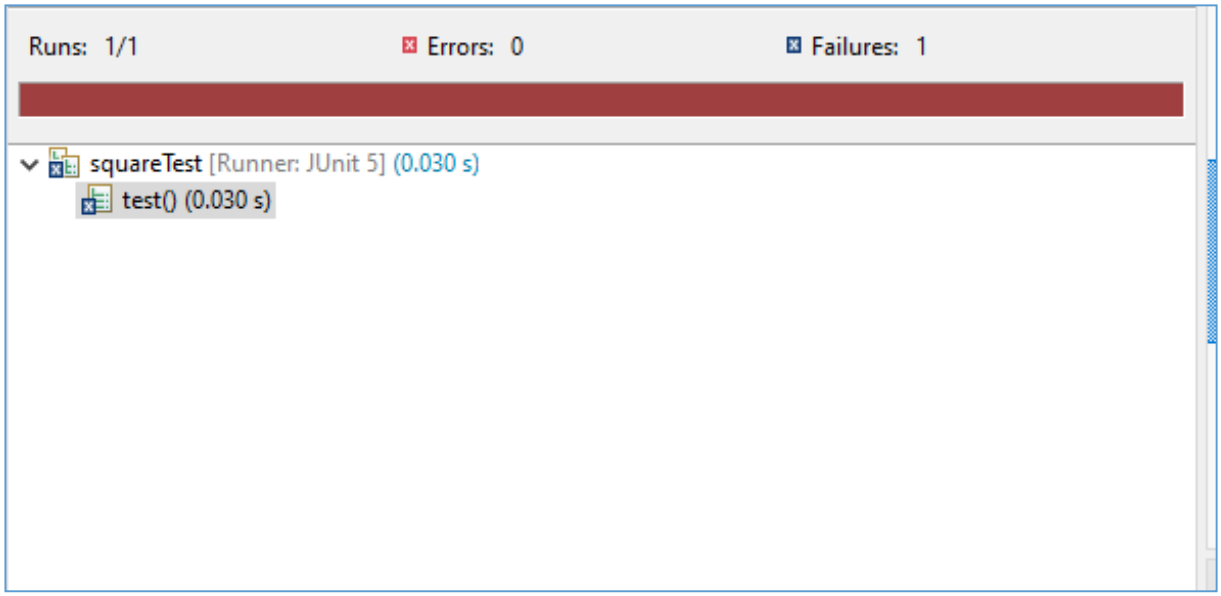
```
*JUnitTesting.java    squareTest.java
package testpro;

import static org.junit.jupiter.api.Assertions.*;

class squareTest {

    @Test
    void test() {
        fail("Not yet implemented");
    }

}
```

```
class squareTest {


    @Test
    void test() {
        JUnitTesting obj=new JUnitTesting();
        int output=obj.square(5);
        assertEquals(25, output);

    }
```

5. Run the code and get result:

```
Runs: 1/1        Errors: 0        Failures: 0

> squareTest [Runner: JUnit 5] (0.061 s)
```

6. If wrong expected value is added or there's a mistake in your logic then the test will be failed and this output will appear:



PRACTICE TASKS

Practice task 1: Add another test case for countA function (which is given in code).

Practice task 2: Make new project, make a class. Add 2 methods in it. One method will find the max integer present in the input integer array. The other method will find the min integer. Now create test cases for both these methods and test your code. Follow all the steps as mentioned above in the manual.

Practice task 3: Calculator Testing:

- Create JUnit tests for basic calculator functions like addition, subtraction, multiplication, and division.
- Check for edge cases like dividing by zero, handling negative numbers, or large number calculations.

Possible Scenarios (but not limited to):

Scenario 1: Test the addition functionality of a calculator with positive numbers.

Scenario 2: Test the subtraction functionality with positive and negative numbers.

Scenario 3: Test the multiplication functionality with zero and non-zero values.

Scenario 4: Test the division functionality with various inputs including division by zero.

Test Exponential Functionality: Test the calculator's ability to handle exponentiation.

Test Cases:

Positive Exponentiation: Test raising a positive number to a positive exponent.

Negative Exponentiation: Test raising a positive number to a negative exponent.

Zero Exponentiation: Test raising a non-zero number to a zero exponent.

Base Zero Exponentiation: Test raising zero to a non-zero exponent.

Expected Outcomes:

Positive Exponentiation: Assert that the result of 2^3 equals 8.

Negative Exponentiation: Assert that the result of 2^-2 equals 0.25 (1/4).

Zero Exponentiation: Assert that the result of 5^0 equals 1.

Base Zero Exponentiation: Assert that the result of 0^4 equals 0.

Practice task 4: String Manipulation:

Write tests for string manipulation methods like reversing a string, converting to uppercase/lowercase, or checking for palindromes.

Possible Scenarios (but not limited to):

Scenario 1: Test a method that reverses a string with both short and long strings.

Scenario 2: Test a method that converts a string to uppercase/lowercase.

Scenario 3: Test a method that checks if a string is a palindrome

Capital University of Science & Technology Islamabad
Department of Software Engineering

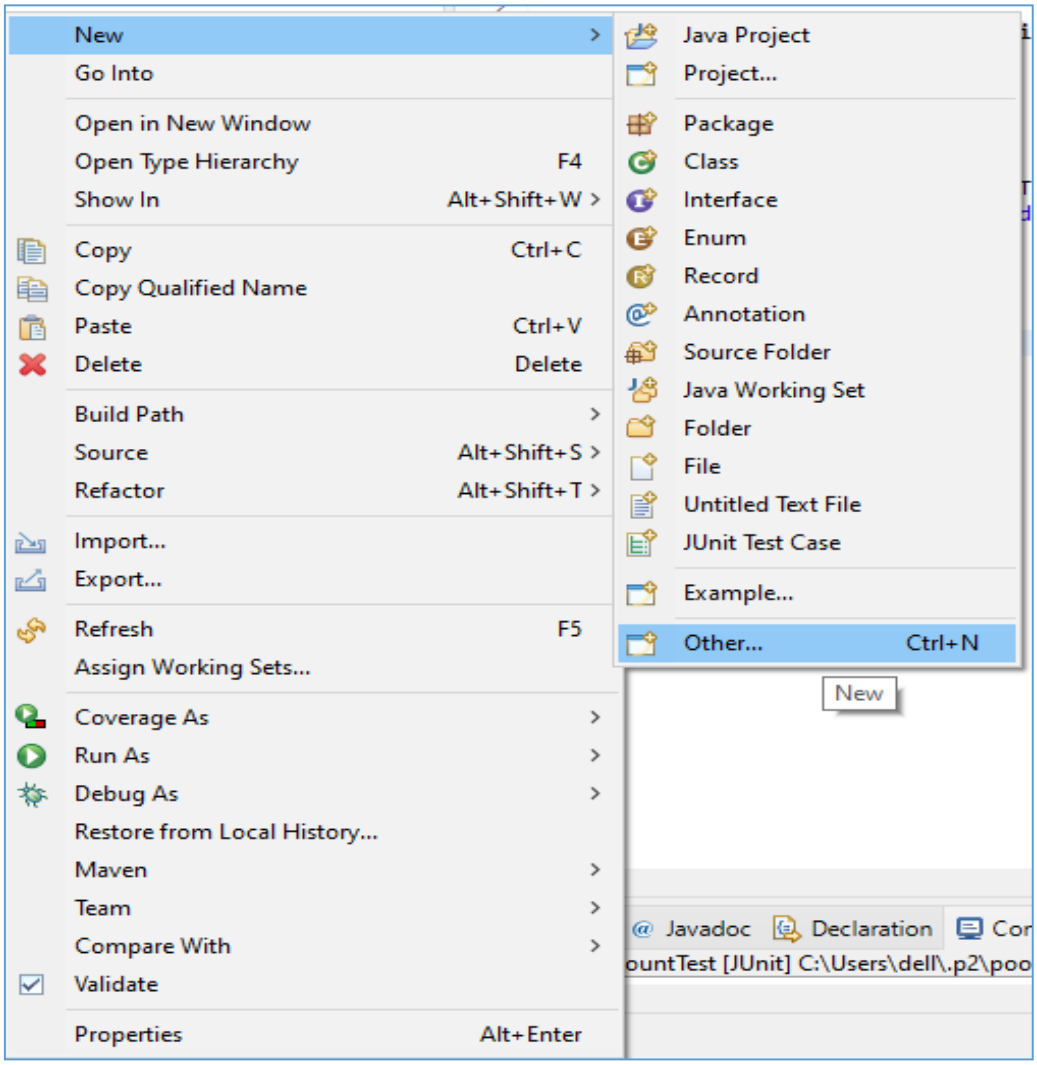
Lab 13: JUnit Testing: Introduction to Test Suite

OBJECTIVE: Grouping the multiple JUnit test cases and constructing a Test Suite program.

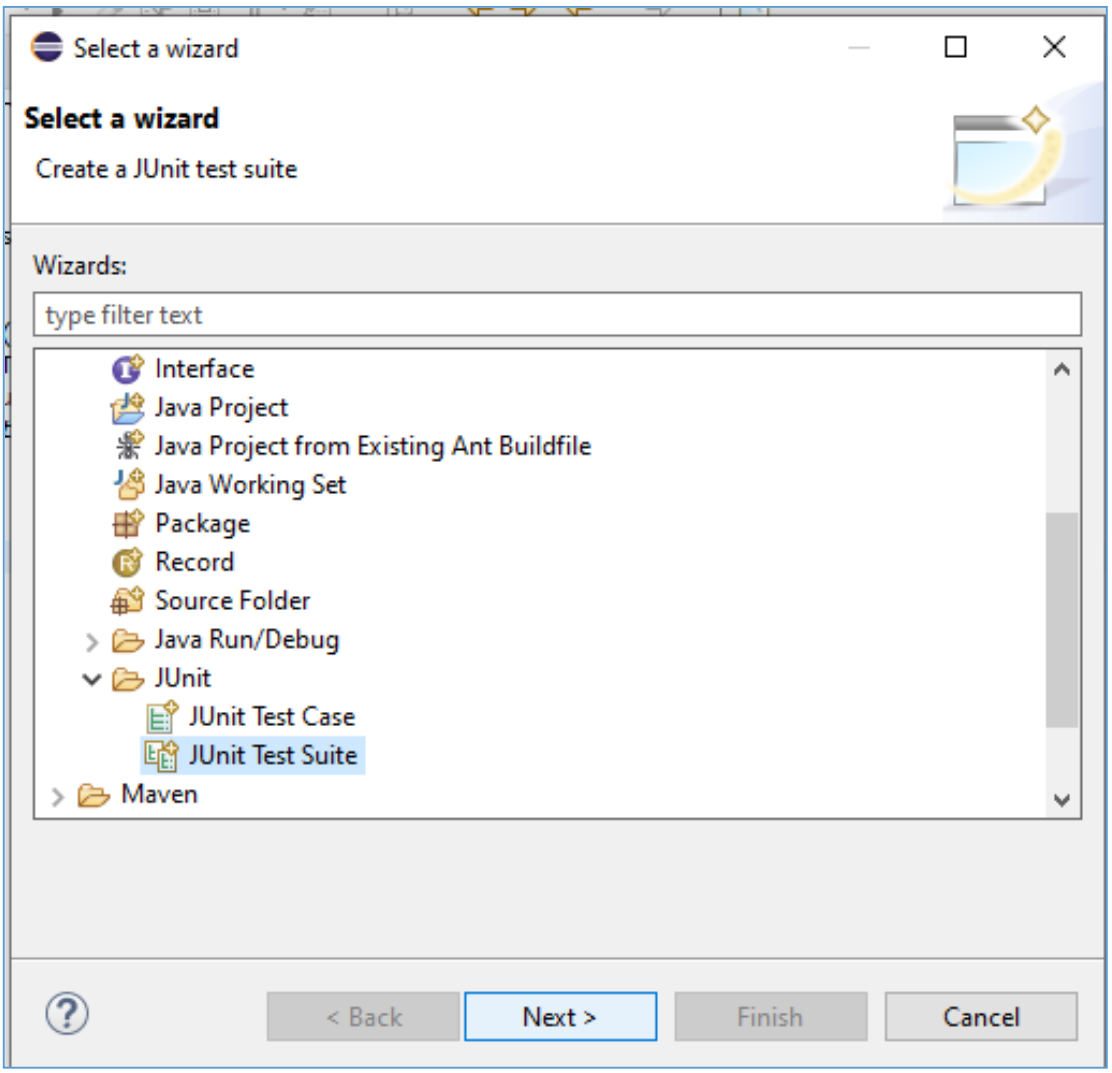
Test suite: Test suite is used to bundle a few unit test cases and run them together. In JUnit,both **@RunWith** and **@Suite** annotations are used to run the suite tests.

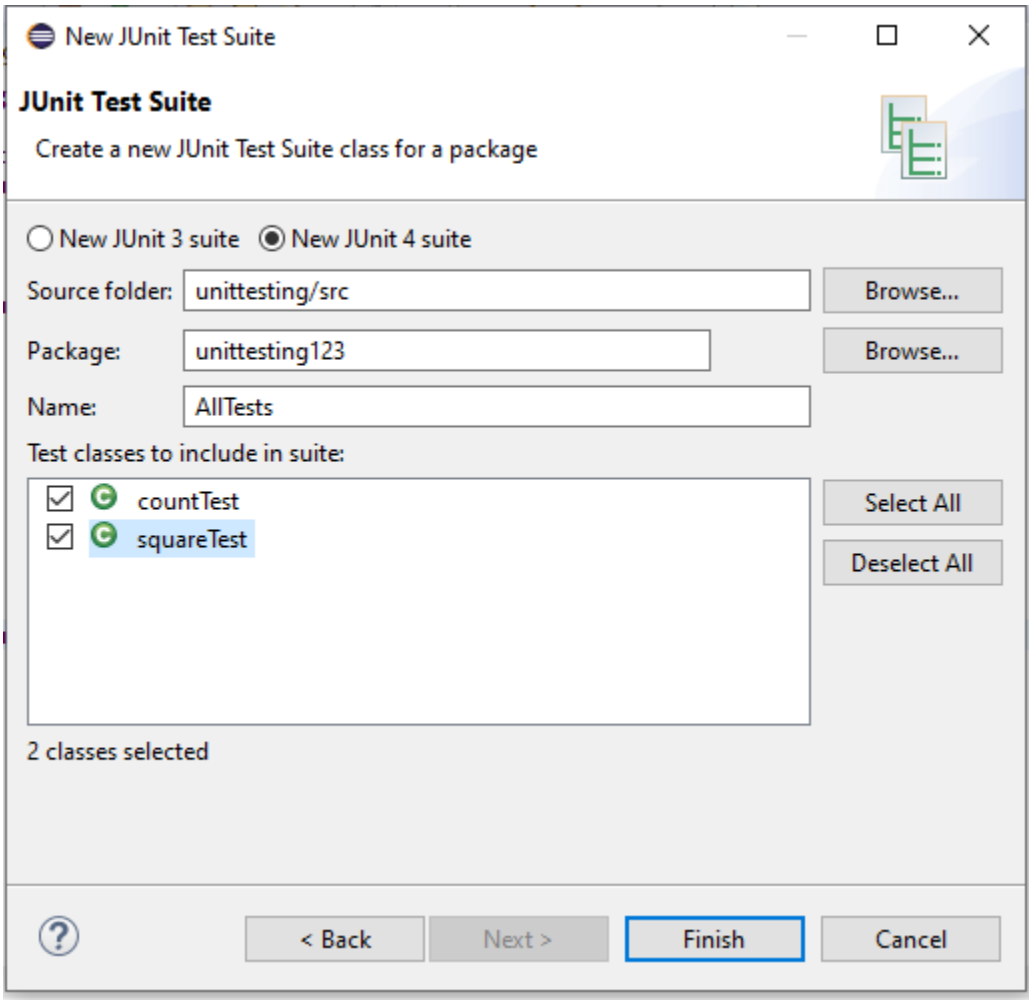
STEPS FOR CREATING TEST SUITE ON ECLIPSE:

Step 1: Right click on 'src' -> New -> Other:



Step 2: Expand JUnit to JUnit Test Suite:

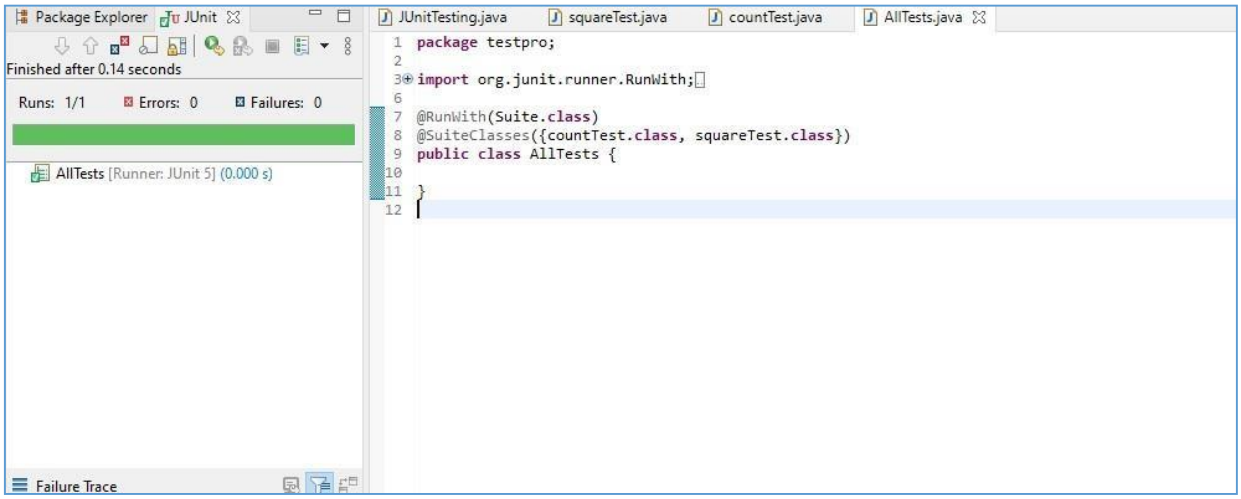

Select a wizard
Select a wizard
Create a JUnit test suite
Wizards:
type filter text
Interface
Java Project
Java Project from Existing Ant Buildfile
Java Working Set
Package
Record
Source Folder
Java Run/Debug
JUnit
JUnit Test Case
JUnit Test Suite
Maven
< Back
Next >
Finish
Cancel



Step 3: Adding test cases in the suite class:

```
import
org.junit.runner.RunWith;import
org.junit.runners.Suite;

import org.junit.runners.Suite.SuiteClasses;



@RunWith(Suite.class)
@SuiteClasses({countTest.class, squareTest.class})
public class AllTests {
```

Step 4: Run the test case suite for getting suite result:



Practice Tasks:

Practice Task 1: Make a class having four functions for determining:

- Whether the input integer is odd.
- Whether the input integer is even.
- Whether the input integer is prime.
- For calculating the factorial of that input integer.

Write their test cases and execute them in a single test suite class. Follow all the steps mentionedabove in the manual.

Practice Task 2: Bank System Transfer Functionality Implementation

Develop the transfer functionality for a bank's account management system. The BankAccount class needs to facilitate transferring funds between accounts securely and accurately.

Task-A:

Your first task is to design and implement the transfer method within the BankAccount class.

Requirements:

BankAccount Class Structure: The BankAccount class consists of properties like accountNumber and balance. It includes methods for deposit, withdraw, and the newly required transfer.

Transfer Method: Implement the transfer method allowing funds to move from one account to another. The transfer method should handle scenarios such as:

-Valid transfer of funds between accounts.

-Insufficient funds when transferring more than the available balance.

-Preventing transfers to the same account or negative transfers.

Error Handling: Ensure proper error handling for invalid scenarios, displaying appropriate messages or exceptions.

Expected Outcome:

- A well-structured transfer method within the BankAccount class that adheres to the specified requirements.
- Valid transfers should successfully update the balances of both the sender and receiver accounts.
- Invalid transfers should be handled gracefully, preventing any balance modifications.

Task B: Test Cases for Bank System's Transfer Functionality

Test Case 1: Valid Transfer

Scenario: Transfer a valid amount from one account to another.

Steps:

- Create two BankAccount instances (sender and receiver) with sufficient balances.
- Transfer a valid amount from the sender to the receiver account.

Expected Outcome:

- Confirm that the transfer is successful.
- Assert that the balances of both accounts are updated accordingly.

Test Case 2: Insufficient Funds

Scenario: Attempt to transfer an amount larger than the available balance.

Steps:

- Create a sender account with a limited balance.
- Attempt to transfer an amount larger than the sender account's balance to a receiver account.

Expected Outcome:

- Ensure that an appropriate error message or exception is raised due to insufficient funds.
- Confirm that no changes occur in the account balances after the failed transfer.

Test Case 3: Negative Amount Transfer

Scenario: Attempt to transfer a negative amount between accounts.

Steps:

- Create a sender and a receiver account.
- Try to transfer a negative amount from the sender account to the receiver account.

Expected Outcome:

- Validate that the system rejects the negative transfer attempt.
- Ensure that the balances remain unchanged after the invalid transfer request.

Test Case 4: Transfer to Same Account

Scenario: Attempt to transfer funds from an account to itself.

Steps:

- Create a sender account.
- Initiate a transfer from the sender account to itself.

Expected Outcome:

- Confirm that the system prevents transfers from an account to itself.
- Ensure that no changes occur in the account balance after the attempted transfer.

Test Case 5: Multiple Concurrent Transfers

Scenario: Simulate multiple concurrent transfers between accounts.

Steps:

- Create multiple instances of sender and receiver accounts.
- Initiate multiple simultaneous transfers between different accounts.

Expected Outcome:

- Ensure that the transfer functionality maintains accuracy and consistency under concurrent transfer scenarios.
- Validate that funds are correctly transferred between the respective accounts without any interference or incorrect modifications.

Test Case 6: Stress Test

Scenario: Perform a stress test with a large number of transfer requests.

Steps:

- Create a substantial number of accounts and initiate transfer requests randomly between these accounts.

Expected Outcome:

- Confirm that the transfer functionality remains stable and does not cause any unexpected issues or errors even with a high volume of simultaneous transfer requests.

Executing these test cases will help ensure that the implemented transfer functionality within the BankAccount class behaves correctly and handles various scenarios effectively.

Task C: Test Suites for Bank System's Transfer Functionality

Suite 1: Basic Transfer Tests

Included Test Cases:

- Valid Transfer
- Insufficient Funds
- Negative Amount Transfer
- Transfer to Same Account

Suite 2: Advanced Transfer Scenarios

Included Test Cases:

- Multiple Concurrent Transfers
- Stress Test

Execution Plan:

Run Suite 1 to ensure basic functionalities of transfer operations and error handling.

Run Suite 2 to evaluate the system's performance under advanced scenarios involving concurrency and stress.

INSTRUCTIONS:

- Consider running Suite 1 before Suite 2 to ensure basic functionality and error handling are intact before stressing the system with advanced scenarios.
- Organizing tests into suites allows for more focused testing and better isolation of test scenarios, making it easier to identify issues in specific functionalities.

Capital University of Science & Technology Islamabad
Department of Software Engineering

Lab 14: Introduction to GitHub and cloning repository