**COMPUTER LABORATORY MANUAL**

|  |  |
| --- | --- |
| Logo  Description automatically generated | nust logo |

**SOFTWARE CONSTRUCTION**

**(SE-314)**

**Fall Semester**

**DEPARTMENT OF COMPUTER SOFTWARE ENGINEERING**

**Military College of Signals**

**National University of Sciences and Technology**

[www.mcs.nust.edu.pk](http://www.mcs.nust.edu.pk)

**PREFACE**

This lab manual has been prepared to support students of Software Engineering in developing practical insight into the principles and practices of **software construction**. The primary objective of studying this course is to understand how software systems are systematically designed, implemented, tested, and maintained. Writing reliable and efficient software requires knowledge of programming principles, design patterns, coding standards, and quality assurance techniques. A software engineer aiming to build high-quality systems must study software construction to learn how to write code that is not only functional but also maintainable, reusable, and efficient. Furthermore, working with diverse platforms, tools, and development environments demands familiarity with best practices in software construction and integration.

**PREPARED BY**

Lab manual is prepared by Lecturer Rabia Khan and Demonstrator Kabeer Ahmed, updated by Mehreen Ahmed under the supervision of Head of Department Dr. Adil Masood Siddiqi in year 2016.

**GENERAL INSTRUCTIONS**

a. Students are required to maintain the lab manual with them till the end of the semester.

b. All readings, answers to questions and illustrations must be solved on the place provided. If more space is required, then additional sheets may be attached. You may add screen print to the report by using the ‘Print Screen’ command on your keyboard to get a snapshot of the displayed output.

c. It is the responsibility of the student to have the manual graded before deadlines as given by the instructor

d. Loss of manual will result in re submission of the complete manual.

e. Students are required to go through the experiment before coming to the lab session. Lab session details will be given in training schedule.

f. Students must bring the manual in each lab.

g. Keep the manual neat clean and presentable.

h. Plagiarism is strictly forbidden. No credit will be given if a lab session is plagiarised and no re submission will be entertained.

i. Marks will be deducted for late submission.

j. In the exercises, you have to put the output in your Lab report.

k. Name your reports using the following convention:

***Lab#\_Rank\_Your FullName***

(1) ‘#’ replaces the lab number.

(2) ‘Rank’ replaces Maj/Capt/TC/NC/PC

(3) ‘YourFullName’ replaces your complete name.

l. You need to submit the report even if you have demonstrated the exercises to the lab engineer/instructor or shown them the lab report during the lab session.

**VERSION HISTORY**

|  |  |  |
| --- | --- | --- |
| **Date** | **Update By** | **Details** |
| July 2013 | Lec Ayesha Naseer | First Version Created |
| Sep 2014 | Lec Ayesha Naseer | Second version created. Labs improved |
| Sep 2015 | Kabeer Ahmed | Labs improved &Updated exercises |
| Sept 2017 | Lab Engr Marium Hida | Labs added, improved &Updated exercises |
| March 2018 | Lab Engr Marium Hida | Labs added, improved &Updated exercises |
| Feb 2020 | Lab Engr Sehrish Ferdous | Labs, CLO’s and lab rubrics updated |
| March 2021 | Lab Engr Saba Siddique | Labs improved and updated exercises |
| February 2022 | Lab Engr Saba Siddique | Lab Rubrics updated |
| February 2023 | LE Laraib Zainab/  Lec. Fawad Khan | Include Open Ended Lab |
| April 2025 | LE Hafsa Ahmad/Lec Fawad Khan | Revision of Lab Titles and objectives of 14 experiments, also changed format of whole manual |

**Department of Computer Software Engineering**

**Lab Rubrics   
Group 1: Programming Tasks**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Criteria** | **Unacceptable  (Marks=0)** | **Substandard Marks=1** | **Adequate Marks=2** | **Proficient Marks=3** |
| **R1 Completeness**  **And Accuracy** | The program failed to produce the right accurate result | The program execution let to inaccurate or incomplete results. It was not correctly functional or not all the features were implemented | The program was correctly functional and most of the features were implemented | The program was correctly functional, and all the features were implemented |
| **R2**  **Syntax and Semantics** | The student fails to figure out the syntax and semantic errors of the incorrect program | Student successfully figures out few of syntax and semantic errors of the program with extensive guidance | Student successfully figures out most of syntax and semantic errors of the program with minimum guidance | Student successfully figures out all syntax and semantic errors of the program without any guidance |
| **R3**  **Demonstration** | Student failed to demonstrate a clear understanding of the assigned task | Student has basic understanding, but asked questions were not answered. | Student has basic knowledge of understanding. Provides fundamental answers to asked questions | Student has demonstrated on accurate understanding of the lab objective and concepts. All the questions are answered completely and correctly |
| **R4**  **Complexity and Readability** | The code is poorly organized and very difficult to read | The code is readable only by someone who knows what it is supposed be doing | The code is fairly easy to read | The code is exceptionally well organized and very easy to follow |
| **R5**  **Perseverance and plagiarism** | Complete working program is copied indicating no effort on student’s part resulting in a total score of zero for all rubrics | Most of working program is copied. Minor contribution by the student | Most of working program is contributed by the student. Minor copied components | Complete working program is contributed by the student |

**Department of Computer Software Engineering**

**Lab Rubrics   
Group 3: Group Tasks**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Criteria** | **Unacceptable  (Marks=0)** | **Substandard Marks=1** | **Adequate Marks=2** | **Proficient Marks=3** |
| **R1 Completeness and Accuracy** | The system failed to produce the right accurate result | The system execution let to inaccurate or incomplete results. It was not correctly functional or not all the features were implemented | The system was correctly functional and most of the features were implemented | The system was correctly functional, and all the features were implemented |
| **R2**  **Demonstration** | The student failed to demonstrate a clear understanding of the assigned task | The student has basic knowledge of understanding, but asked questions were not answered. | The student has moderate knowledge of understanding. Answer to the question is basic | The student has demonstrated on accurate understanding of the lab objective and concepts. All the questions are answered completely and correctly |
| **R3**  **Plagiarism** | Complete working program is copied indicating no effort on student’s part resulting in a total score of zero for all rubrics | Most of working program is copied. Minor contribution by the student | Most of working program is contributed by the student. Minor copied components | Complete working program is contributed by the student |
| **R4**  **Contribution/ Group participation** | Shows little commitment to group goals and fails to perform assigned roles | Demonstrates commitment to group goals, but has difficulty performing assigned roles | Demonstrates commitment to group goals and carries out assigned roles effectively | Actively helps to identify group goals and works effectively to meet them in all roles assumed |
| **R5**  **Presentation skills** | Poor presentation; cannot explain topic; scientific terminology lacking or confused; lacks understanding of topic | Presentation lacks clarity and organization; little use of scientific terms and vocabulary; poor understanding of topic | Presentation acceptable; adequate use of scientific terms; acceptable understanding of topic | Well-organized, clear presentation; good use of scientific vocabulary and terminology; good understanding of topic |

![A red circle with a book and lightning bolt and crescent moon

Description automatically generated](data:image/jpeg;base64,/9j/4AAQSkZJRgABAQEAYABgAAD/4SUERXhpZgAATU0AKgAAAAgABgALAAIAAAAmAAAIYgESAAMAAAABAAEAAAExAAIAAAAmAAAIiAEyAAIAAAAUAAAIrodpAAQAAAABAAAIwuocAAcAAAgMAAAAVgAAEUYc6gAAAAgAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAFdpbmRvd3MgUGhvdG8gRWRpdG9yIDEwLjAuMTAwMTEuMTYzODQAV2luZG93cyBQaG90byBFZGl0b3IgMTAuMC4xMDAxMS4xNjM4NAAyMDIxOjEwOjA2IDEzOjQ1OjMxAAAGkAMAAgAAABQAABEckAQAAgAAABQAABEwkpEAAgAAAAMzNwAAkpIAAgAAAAMzNwAAoAEAAwAAAAEAAQAA6hwABwAACAwAAAkQAAAAABzqAAAACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAMjAyMToxMDowNiAxMzo0NDowOAAyMDIxOjEwOjA2IDEzOjQ0OjA4AAAAAAYBAwADAAAAAQAGAAABGgAFAAAAAQAAEZQBGwAFAAAAAQAAEZwBKAADAAAAAQACAAACAQAEAAAAAQAAEaQCAgAEAAAAAQAAE1gAAAAAAAAAYAAAAAEAAABgAAAAAf/Y/9sAQwAIBgYHBgUIBwcHCQkICgwUDQwLCwwZEhMPFB0aHx4dGhwcICQuJyAiLCMcHCg3KSwwMTQ0NB8nOT04MjwuMzQy/9sAQwEJCQkMCwwYDQ0YMiEcITIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIyMjIy/8AAEQgAhgCFAwEhAAIRAQMRAf/EAB8AAAEFAQEBAQEBAAAAAAAAAAABAgMEBQYHCAkKC//EALUQAAIBAwMCBAMFBQQEAAABfQECAwAEEQUSITFBBhNRYQcicRQygZGhCCNCscEVUtHwJDNicoIJChYXGBkaJSYnKCkqNDU2Nzg5OkNERUZHSElKU1RVVldYWVpjZGVmZ2hpanN0dXZ3eHl6g4SFhoeIiYqSk5SVlpeYmZqio6Slpqeoqaqys7S1tre4ubrCw8TFxsfIycrS09TV1tfY2drh4uPk5ebn6Onq8fLz9PX29/j5+v/EAB8BAAMBAQEBAQEBAQEAAAAAAAABAgMEBQYHCAkKC//EALURAAIBAgQEAwQHBQQEAAECdwABAgMRBAUhMQYSQVEHYXETIjKBCBRCkaGxwQkjM1LwFWJy0QoWJDThJfEXGBkaJicoKSo1Njc4OTpDREVGR0hJSlNUVVZXWFlaY2RlZmdoaWpzdHV2d3h5eoKDhIWGh4iJipKTlJWWl5iZmqKjpKWmp6ipqrKztLW2t7i5usLDxMXGx8jJytLT1NXW19jZ2uLj5OXm5+jp6vLz9PX29/j5+v/aAAwDAQACEQMRAD8A9/ooAKKACmSzRQRmSaRI416s7AAfiaBpNuyOZv8A4ieGbA7f7QFy3paqZB/30Pl/WsCf4v2Cswt9Ju5AOhkkVM/lmuWeLhHRanvYXh3FVlzVLQXnv93+dit/wuE7udB+X1+2c/8AoFXLf4vaY7hbnTL2IHqyFHA/UGoWNXVHVU4XqJfu6ib81b/M6DTvHvhrUiqpqccDk4CXIMRz6Zbg/nXSKyuoZGDKeQQcg11QqRmrxZ4GKwVfCS5a0bfk/Ri0VZyhRQAUUAFFABTZJEhjaSR1RFGWZjgAepNAJX0R5x4j+KkEBe20KJbiQcG6k/1Y/wB0dW+vA+tea6nrGpazL5mpXs1yc5Ac/Kv0UcD8BXl4jEOb5Y7H3uT5RHCxVWqr1H+H/B7v5Lzo0VzHvBRQAVp6R4i1fQXB06+kiTPMLHdGf+Anj8sGqjNwd4mNfD08RTdOqrp/1956j4b+J1jqbpa6qi2N0xwsmf3Ln6n7p9jx713gIIyD1r1qNVVI3PzrMsvnga3I9U9n3/4PcWitTzwooAKKAKuo6ha6VYS3t7MIreJcsx/kB3PtXh/i3xpe+J5jCN1vpqn5Lfu/u/qfboP1rjxdXljyLqfS8O4D2tV4ia0jt6/8D87HMUV5x9sFOSN5GwiMxAJwozwOtITaSuyaCRIxG09ossL9yWUkf7LDj9CKt6hpSwWcOoWcjTWM3AZgNyN3Vsd/esJ1HTqRu/dlp6P/AIP5mEqjhNXd4vT0f9fiZlFdB0ARkYPSu18G+PrjQnjsNRZp9MzhSeXg+nqvt27ela0ans53PPzLBLGYd0+u69f+Dse0QTxXMEc8EiyRSKGR1OQwPcVJXsH5q007MKKBBTZJEhiaSRgiICzMTgADqaASvojwfxr4tk8T6lthJXTYG/cJ/fPTefc9vQfU1zFeNVnzzcj9PwGFWFw8aXVLX16hRWZ2ATgZNaelzwaTrImu45ne3kKqI2CgHoSfUYzxWVZOcHCOjaZjXjKdNwju0y14ov7O/wBR/wBHWRWg/c5yPLZQTyo7ck1d0aEyeBtY84HywzvHkd1RTkfiBXn1YypYWnGb1Tj+ZwSjKnhYKW91+ZytFeqeqFFAHc/DzxgdFvF0q+l/4l9w37tmP+ocn/0Enr6Hn1r2ivUws+aFn0PgeIML7HF862nr8+v+fzCiuk8IK83+KfiM29rHoVs+JLhfMuSD0j7L+JH5D3rHES5abPUyah7fGwT2Wv3f8E8moryD9HCigA6jBrftrO18QxoFuFttURQjCT7lwAMBs9mxwf5Vy4qcqUVVir23Xl1/RnNiJSppVYq9t/Tr+jLdr4Fvnl/0y5giiHUxEsx+mQAKm8R6tZ2emLoemEFQAsrKchR1xnuSev415n1n6/iIRpr3Iu7v+B5/t/rteMYL3Y6s4+ivdPZCigAIBGCMg17b8N/Eba1oZs7mQteWOEZieXQ/db9CD9PeurCStUt3Pn+I6HtMJ7TrF/g9P8jtKK9M+EGSyJDE8sjBURSzMegA6mvm7WNUk1rWLvUpCc3EhZQf4U6KPwUAVxY2WiR9VwvSvUqVeyS+/wD4YpUV559iFFAGnp2gahqbSrAiIYgpYTEp97pjg+lTXPhLWYI8taLMvfynDfocH8hXDPMsPCr7KTs+/T7zilj6EKns5PXv0M5Te3DfYw11I2dvklmP4Fa1IPB+syxhvs8UPossgB/TNVXxeHwq952v0RVbE0MMtdL9F+ZR1DSL3TbgwTorMIxITESwC5xknHHNUa3pVY1YKcNmb0qsasVKIUVqaBW/4L1g6J4rsrhn2wSt5E3ptfj9DtP4VdOXLNM5cbS9th50+6f/AAD6Cor2j8uOW+Id+1h4Kvihw8+23HPZzhv/AB3dXg9ebjH76R9xwzC2FlLvL9EFFch9GFNk4jY+xoQLc9TsP+Rh1j/th/6Aa1q+ExHxr0j/AOko+NrfEvRfkjEtQP8AhMtROBn7LFz+J/wH5Vt1WK+KP+GP5Iqv8S9F+SOZ1wD+0NTProsv8zXn1fSZU/3fyX5HvZb/AA/uCivVPSCkIypGSMjqKAPo7w3qDar4a02+c5kmt0aT/exhv1zRXtxd4pn5TXh7OrKHZtHG/F6dl0fTYAeHuS7D/dQ//FV5JXmYr+Kz7vh9WwMX3b/MfFH5sgTcqluAWOBn69qR0eKRo5EZHU4ZWGCD6Gue2lz2eZc3KNpsn+qf/dNCKR6tYD/idas3+1EP/HK06+ExHx/Jfkj42r8XyX5IxLX/AJHHUf8Ar1h/m1bdVifij/hj+SKr/EvRfkjnNdGLrUG9dHmH6153X0eU/wAL7j3cs/h/cFPhhknlEcS7mP6DuT6Aeteslc9GUlFXY1gFYgMGA7joaSga2Pb/AIYTtN4KhRiT5M0qD6bs/wBaK9ii/wB3H0PzLMopYyqv7z/MxPjBnyNIPbfL/Ja8rrzsV/FZ9tkP+4Q+f5sK2rD7Prix6feSrBfgBLS7b7r+kUn/ALK3bp6VnCzfK+p34rmjD2sFdx19V1X3bedjMvLO40+7ktLuFop4zhkb+fuPeq0n+qf/AHTU2admb05xnFTi7pno1+YLHUZ7n+31s2n2loNiPyFA6de1VP8AhLo4Acahb3gH8JtZYWP44I/QV8vTwksVFN0mtF71+2l7Pf5Hz1PDSxEU/Z221v8AjZ7/ACKcXiu1j1m71AQTETQRxrGcAhgTnJ9Oau/8JTHOQG1e3tc9orOSQj/gTAD/AMdrarlc7qTXNZJWTS2Xd/obVMune7XNolZO2y8xZliuNP1O9XWhqEgsZYggVVKgjPQc9u9cJXdlzdpJw5baWOzL72knHlt0LmmaXd6xera2ce9+rMThY1/vMewqzqFxa2kT6Zpj+bD0ubsjBuWHYekYPQd+p7V66XLHmfU2nP2ldUVtHV/+2r9fl5mVRWZ1ntHwoB/4RGX0+1vj/vlaK9ih/DR+aZr/AL7V9WVfi5bs+hWFwBxHdbWPoGU/1AryGvPxa/en2HD8r4GK7N/mFIRkYIzmuc9s7TSLuz8W2ceiaw4j1GJdtjfHlm/2G9T7E8/Xmua1bR73RL5rS9iKsOVcfdkHqp7/ANK3qLmgqi9GeThJ/V8RLBy2+KPo918nf5FHvzRWB6wUUAHfPfpmtbQPD174ivPJtRshQ/vrhxlYx/U+1VCDnJRRz4rEQw1GVaey/qxq+INWstOs38PaDgW3S8ugfmuGHVcjqPXt26deUq6sk5WWyMMvpThR56vxz95/Pp8lZBRWR3nuXwztzD4HtWb/AJbSSSD3BYgfoKK9mlpTXofmOZS5sZVf95/mXfHWnNqng3UYUXMiRiZAOpKENj8QCPxrwCuHGL30z6rhmpfDTh2l+aX+QUVyH0gAkEEEgg5BBwQa9K0HVrHxnpf9i64A19GMxy8Kz4/iU9mA6jv6YyB0YeS5uSWzPGzqjP2SxNL46bv8upm6d4Pk07xWdM1OPz7C9gkjinUYDEYYf7rjbn8OO9c54g0G68PaibW4O+N8tBMBxIv+I4yP8RRUouML9mLB5nCviVBbTimvJq6a/D8DKroPCvhebxFe5fdHYRH99KOrH+4vufXsPwrKlDnmonfjsUsLh5Vn029ehrx+D5Ne8V35jT7JpFvN5IdRjcEAUqg+oOT/ADNWfFfiG20iyPhzQQsKplZ5I/4fVQf7x7n+vTqaVKEp9XseEqjx+IpYbeNNJy83Zfr+p5+AAMAYAoriPqAowzHailnPCqO59KA9T6S0TTxpWhWGn9TbwJGT6kAAn880V7kVZJH5RVn7SpKfdtl5gGUqwBBGCD3r508S6M+geIbvTyMRo++E+sbcr+Q4+oNceNj7qZ9HwxWtWnS7q/3f8OZVFeefaBTo5JIZkmido5Y2DI6nBUjoRQnYTSasz2Dwh4ri8QWvkXJSPUYR86DpIP76/wBR2/GtHUrPTPEtndabJIrvC21ip+eF8cH9fxr14yjVpq/U/OcRh6uX4xuH2HdenT/Jnmdn4G1SbxC2mToY4YsNJdAfKU7Ffc+nbnPTn1S0GnaV9n0i2KRN5ZaOEHLFR1Y/ievc1jhqXs7uXoejnmPWM5KVHVJcz+79F+ZzHjbxeNLjbTNNkH25x+9kX/lgD/7Mf06+leV1z4qpzTstke1kGD9hhfaS+Kevy6f5/MKK5j3Arqfh/orax4rt2Zc29mRcSkjjI+4PxbB/4Ca0pR5ppHHmFX2OFqT7J/e9F+J7xRXsn5gFcL8S/DTatpK6naoTd2SksqjJki6kfUdR+PrWVeHNTaPQyrEfV8ZCb2vZ/PQ8YByMjpRXjn6WFFAE1rdT2V3FdW0hjnibcjjqDXQ6PNrOveMXvtMb7Lcyssk7jmNFwAcjuDjhf14yNabk2oR7nBjadCMZ16y0UWn5rf8APb1PT21myuby50mz1CBdSSM4BG4K2PTuR1IzmvJzf6v4b8RXk12XfUjG0fmSnP3sYcccjjjt+WK68TN6Si9F+Z87kmFj+8oVo2lOKa/wvt/X5GEzNI7O7FnYlmZjkknqTSV559gkkrIKKBhySAASScAAZJNe9eBfDZ8OaAqTgfbbk+bcH+6eyf8AARx9c114OF5uXY+c4lxHJho0lvJ/gv8Ag2Onor0j4cKKAPFviB4NbRbttUsIydOnf50Uf6hz/wCyk9PQ8elcPXj16fJNo/Ssrxf1rCxqdVo/Vf57/MKKyPRLWnafPql6trbhdxBZnc4WNR1Zj2ArfvPENvpOmtovh1mWMn/Sb/GHnbuV/uj39OnqdYPki5dXsediqf1qrGg/hj70vPsv1focwjvHIskbskincrKcEH1Brq11e28VWUena06QalEMWuoEYVj/AHJPTPr0+ncpS3i9mPHUG+XEU178NfVdV81t5nMXVrPZXUlrcxmOaJtroe3/ANb3qGsmmnZndCSnFSjswooKPTPhv4MZ5Itf1KMqqndaQt3P/PQ+3p+fpXqtethqfJT9T87zvF/WcW+XaOi+W/4hRW55AUUAMmhiuIXhmjWSKRSrowyGB6givHfGPw8uNJZ77SI5Liw+88Q+aSH/AOKX9R3z1rmxNLnjdbo93IswWGr+zm/dl+D6P9H/AMA4MEEZHINFeWffFlrxxYmyhzHA53TDPMx7bvYdl6d+TzVam3ciEOVPu9X/AF+HyCikWWZb2S4tY4bgmQwjbDIT8yr/AHCe6+np244qtTbuRCCgrLYBkkAAkk4AAySfSvTPBnw4d3j1LXoiig7orNup95P/AIn8/StsPS9pPXZHl5zj1hMO+V+9LRfq/l+dj1QAAAAYA7UtesfnYUUAFFABRQBx3iT4daXrjPcWx+wXh5MkSAo5/wBpf6jB+teYaz4J17RCzTWT3EC/8trYGRce4HI/EV5+IwzvzQPscmzqLisPiXZrZv8AJ/5/0+dBBJAPQ4PtS1xH1QUUACgs6ooLO3RQMk/hXU6L8Ptf1gq7W32G3PJlugVOPZOp/HFaU6UqjsjjxuOo4Onz1Xr0XV/13PUvDfgbSfDm2ZE+1Xo/5eZgMj/dH8P8/eumr1qdNU48qPzrGYypi6zq1P8Ahl2CirOUKKACigAooAKKAM3UPD+j6rzfabazt/feMbv++utYE3wx8LysWS1nhJ/uXLkf+PE1jOhTm7tHpYXN8Zho8tOenZ6/nt8iD/hVPh7OfMvvp5w/+Jq1b/DPwvAwZrKWYj/nrcOR+QIFQsJSR1T4hx0lZNL0X+dzoLDRtM0tdthYW1sPWKIKT9T3q9XQkoqyPHqVZ1ZOdR3b7hRTMwooAKKAP//Z/+Ex6Gh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8APD94cGFja2V0IGJlZ2luPSfvu78nIGlkPSdXNU0wTXBDZWhpSHpyZVN6TlRjemtjOWQnPz4NCjx4OnhtcG1ldGEgeG1sbnM6eD0iYWRvYmU6bnM6bWV0YS8iPjxyZGY6UkRGIHhtbG5zOnJkZj0iaHR0cDovL3d3dy53My5vcmcvMTk5OS8wMi8yMi1yZGYtc3ludGF4LW5zIyI+PHJkZjpEZXNjcmlwdGlvbiByZGY6YWJvdXQ9InV1aWQ6ZmFmNWJkZDUtYmEzZC0xMWRhLWFkMzEtZDMzZDc1MTgyZjFiIiB4bWxuczp4bXA9Imh0dHA6Ly9ucy5hZG9iZS5jb20veGFwLzEuMC8iPjx4bXA6Q3JlYXRvclRvb2w+V2luZG93cyBQaG90byBFZGl0b3IgMTAuMC4xMDAxMS4xNjM4NDwveG1wOkNyZWF0b3JUb29sPjx4bXA6Q3JlYXRlRGF0ZT4yMDIxLTEwLTA2VDEzOjQ0OjA4LjM2NzwveG1wOkNyZWF0ZURhdGU+PC9yZGY6RGVzY3JpcHRpb24+PC9yZGY6UkRGPjwveDp4bXBtZXRhPg0KICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgCiAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAKICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgIAogICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgICAgPD94cGFja2V0IGVuZD0ndyc/Pv/bAEMAAwICAwICAwMDAwQDAwQFCAUFBAQFCgcHBggMCgwMCwoLCw0OEhANDhEOCwsQFhARExQVFRUMDxcYFhQYEhQVFP/bAEMBAwQEBQQFCQUFCRQNCw0UFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFP/AABEIAIYAhQMBIgACEQEDEQH/xAAfAAABBQEBAQEBAQAAAAAAAAAAAQIDBAUGBwgJCgv/xAC1EAACAQMDAgQDBQUEBAAAAX0BAgMABBEFEiExQQYTUWEHInEUMoGRoQgjQrHBFVLR8CQzYnKCCQoWFxgZGiUmJygpKjQ1Njc4OTpDREVGR0hJSlNUVVZXWFlaY2RlZmdoaWpzdHV2d3h5eoOEhYaHiImKkpOUlZaXmJmaoqOkpaanqKmqsrO0tba3uLm6wsPExcbHyMnK0tPU1dbX2Nna4eLj5OXm5+jp6vHy8/T19vf4+fr/xAAfAQADAQEBAQEBAQEBAAAAAAAAAQIDBAUGBwgJCgv/xAC1EQACAQIEBAMEBwUEBAABAncAAQIDEQQFITEGEkFRB2FxEyIygQgUQpGhscEJIzNS8BVictEKFiQ04SXxFxgZGiYnKCkqNTY3ODk6Q0RFRkdISUpTVFVWV1hZWmNkZWZnaGlqc3R1dnd4eXqCg4SFhoeIiYqSk5SVlpeYmZqio6Slpqeoqaqys7S1tre4ubrCw8TFxsfIycrS09TV1tfY2dri4+Tl5ufo6ery8/T19vf4+fr/2gAMAwEAAhEDEQA/AP1TooooAKKKgvLyDTrWa6up47a2hUvJNM4REUckkngAepoGrt2RPVbUNStNJtJLq+uobO1jGXmuJBGij1LE4FfI3xo/b80/SWuNK+HdpHrF2uUOt3gP2RTyMxICGl+pKr0I3Cvjjx18SfFPxMvjdeKdevNbk3bljuHxDGc8bIlARP8AgKivBxOcUaL5afvP8PvP1zI/DbNMziq2MfsIPurzf/bulv8At5p+R+jXi39s34T+E2MY8SDXJgcbdEha7U/SRf3f/j1eXar/AMFHfDsUki6Z4L1i7VSQr3lxDbhv++TIRXwjRXhVM5xUvhsvl/mfrWE8Mshw6/fKdR+crf8ApPL+bPtVf+Ckjebg/DrEX94a5lvy+zf1rodH/wCCjXhO4uI49U8J67YI33pbd4LhV493Qn8q+CaKyjm2MW8r/JHfV8OuG6itHDuPpOf6yZ+png39rj4U+Nmjjt/FltpdzIdqwayrWTFj0AaQBSfYMTXr1vcR3UKSwyLLE4yskbBlYeoIr8Va7T4c/Gbxr8JrhX8L+ILqwtwctp8jedaP7GFsqPqu1vcV6VHPHe1aH3f5f8E+HzTwopOLnleIaf8ALPVf+BRSt/4Cz9eqK+XPgn+3V4d8c3Fto/jG3j8K61Kwjju9+bCdj0G9jmIn0fI5ADknFfUKsHUMpBBGQR0NfTUcRTxEeek7o/Cs0yfHZLX+r4+k4S6dn5prRr0HUUUV0HjBRRRQAUUVDeXkGnWc91dSpb20CNLLLIcKiKMliewABNA1duyMrxl4y0fwB4avte169TT9LsozJLM+T9FUDlmJwAo5JIAr81P2hv2ntf8AjtfSWKeZpHg+NswaQCN0xB4kuCCd7eij5VwOpG6l/af/AGhrr46eLvKsXeHwfpshGm25yDO2CDcuP7zAkKP4VPYs1eLV8PmWZOu3SpP3fz/4B/VvBHBNPKKccwzCF8Q9Un9j/wC27vpsurZRRRXz5+xBU1rZz3shS3hkndVZysaliFAyTgdgASfYVAzBVLE4AGSTXa+A9W074eePkvdZtb24uNMuWjjS1lEQR8mN2bIyyhS3AIznvXPXqSp05ShHmlZtLvb+kcuJqyo0pSpx5pJNpd7GBpd5BZLbSahosd/p0xyJGaWF3UHkxSKdpI91YdiK3vGHgGLStB0/xJod1Lqfhq+PlrNMAJ7eUZDRSheM5B5HBwenGdr48eLNE8WeKidLiuYpNNzp4YOptZYkZiGjUDK/MW9iPwrpPhpprXn7OPxBF8rfY1kmubbeuQHjgjcMuf8AbUfiD7185icdVw9ChjnFwcpRUoN30k7fer30t1T8vnqmOq08PRxzi4OUknBu+knbTzW91a+zXbw6iiivqz6sRlDKVYAqRgg9DX0f+zP+11qnwnmtfDvieabVfBhISORsyXGmj1Q9WiHdOSAPl/un5xorooV6mHmqlN2Z5Ga5Tg86wssJjYc0X96fdPo/+Gemh+0el6pZ63pttqGn3MV7ZXMayw3EDhkkQjIZSOoIq3X50/sb/tIN8Mtej8H+IbvHhPU5sWs0p40+5du3pE5PzejEN0LV+itfoWDxcMZT5479UfxlxNw7iOG8c8LV1g9Yy/mX6NbNdPRpi0UUV3HyYV8fft9fGh9F0e0+HelT7bvVI/tWqsh5W1yQkR5/5aMCT/soQeGr66vr2DTbG4u7qVYLa3jaWWV+iIoyWPsADX4+fEjx5c/E7x7rviq6Zi2qXTzRKxP7uH7sKf8AAYwg+oNeDnGJdGh7OO8vy6n654bZHHM80eMrK9OhZ+s38P3WcvVI5uiiivhT+tAooooAGUMpVhkHgg16lofhvSPjJaQJDqceieNreFIJVvAfI1NVAVJAR0k2gK2Mk4ztNYHg34ReJfHUt9Hp0FvbSWaxtKuou8BxICU2jYc5AJ7cEetaGt/s8+OdKti8mjRalF1YWFwkpA9drbWP0APSvnMdjcFOoqKxUaVaOzuuvRp6NNW09GtUmeBjMXhJzVJYlU6sdnddejT0aatp6Na2Ot0H9k/xBcXoGt6pYWNmp+ZrB3mlYe25FUfU5+lX/jR8QdC8NeD4vh94SdXjQLHdywvvWKMHcUL/AMUjty3sTk5NeKwtr2sSDREl1a9kz5R01pZnOf7pjJ6jPQj+Vdvpf7N/jvULVJP7NtdOz0ivLpVYD6Jvx9K8avho08RTxGd42LUHeMLKCv0k1dt2/DvbQ8qth4wr06+b4uLUXeMdIq/R7tv+teh5nRXTeL/hzrvgfVH0/UYI5p0thdu1izTIkRYqGY7Rj5hjkdx61zNfbUa1LEQVWjJSi9mj66jWp4iCqUpJp9gooorY2EZQylWAZSMEHvX6SfsUfGqX4nfDl9E1W4afxD4d2W8skhy9xbsD5Mp9ThWRj6pn+Kvzcr1L9mT4kv8AC741eHdTkn8nTLyYaZf5J2mGYhQT/uyeW/0U+tepluJeGxCfR6P+vI+C41yOOeZRUhFfvad5Q73W6/7eWlu9n0P1copKK/RD+LDxL9srxdJ4R/Z78SmBitzqQj0tMMQSszhZOn/TLzPyr8v6+8/+CjeqSw+A/B+nI2EuNWeeRfURwOB+sgr4Mr4XOajlieXsl/mf1r4Y4WNDIvbdak5P7rRt+D+8KKsafZ/b7pLcSxwvJ8qNMwVC3YFjwuemTxnGSByGXVrPY3U1tcwyW9zC5jlhlUq6MDgqwPIIPavCs7XP1jnjzcl9SKorxttpOeOEY8jPapahvf8Ajzn/AOubfyojujSO6Pt/wioHxS+IOOg/s8D/AL8NXdVxfhJQfiF48kzkmayUj0xbg/1rtK/k3Mv48V/cp/8ApuJ/NONd6q/ww/8ASInm+gRovx88WsFUN/ZFkd2OclnBP5Kv5D0r0ivONB/5L14t/wCwPY/+hyV6PW2bfxaX/Xun/wCkI1zD+JD/AAQ/9IR438U4x/wk/jJiOW8BXg/J3xXyhX1r8WI9uteK5Mfe8DXy5+jH/Gvkqv2vg582Eb8o/kfrPCrvhm/KP5BRRVnTtNudWvI7W0iM0z5OMgAKBlmYnhVABJY8AAk1+gq70R9rKSinKTskVqbInmRsm4ruBG5Tgj3FTTRrFM6JIsqqcCRc4b3Ge1R0ik+qP14+CfjCXx98I/CHiC4YPd32mQSXBBz++CASf+PhqK8v/YR1STUv2edOhlYsbG/vLcey+aZAPykFFfp+Gqe0owm+qR/Bee4RYHNcVhYLSE5Jeibt+B5t/wAFIQ39m+ASM7PPvQfrthx/WviCvvv/AIKLaRJc/DfwvqSJlLTV/Kkb+6skL/8AsyLXwJXxObq2Ll52/I/qjw5qKfDlGK+y5r/yZv8AUK9H8J/2Z8VI7Xw1rl5HpXidVW30XXpjiKcAYWzuz6HpHL1UkKcjArzikkUSKVIDA8EGvLpz5Hqrrqj7/FYb6zC0Zcs18Mlun+q7p6NGn4j8N6p4P1y60bWrKXT9TtW2y28wwR6EHoVOOGGQexrHvv8Ajzn/AOubfyr6O+HXiLQv2htAtPAHjm4Fl4ssojH4e8TN80sox/qJSSN5GPuk/OBkEONx8e+Inw4174XeI5tG160aCZfmhuEBMFyn9+N8cj1HUHggV0VaHJFVqesH+Hk/P8zyMuzj22Ill+MXJiYa26Tj/PB9Yvqt4u6e139H+LZLDwr4q1LVf+Fkx+GpdS8uSXTPs8NwSyRKgO0gueF7Adawl/aJtdKViniXTvEiqMiGTSLywlf1Ak2umcdAVUZ7jPHzOzbmyTkk9T1NJX5vT4RwsoRji5uo0kvhhHRaLVR5ttPiuefT4Xw7go4mbm0ktoLRab8rl/5Nc94sP2gNHsfH+ueJk069ddQ061torR9issiM27cwJAUBuoyT6V0R+PdtqzqsvjXTdA3jAjs9Du7tlOe80iqp7/8ALMetfMtFddXhTLarUrNNJK+j0SSStJSXTtfzOmpw1gKjUrNNJLo9EklpJSXTtfzPp3Urez1jwv401+Lx4njC5Hhy+sVt1ihjMKsm4/KgBzlf4h3r5ioX724fewRuHXB6iu6+Efwb1/4za8bDSFFrYQEfbdVnQtDbKewGRvkI6ICPUkDmvZyfKZ4JypQnz8zVvdUbJLb3Ul53sjoo0aGQ4erXxNZKmtbtKKSWltNH8ldt2s2Y3gXwFrXxI8QR6PoVr9ouSN8ssh2w20feSV+iqOeT16DJ4rX8Zaxo/h6zuPCXhS4N/p+dmq666bX1WVSMrGP4LZSPlX+M4Zs8Gu8+MHxC0HwboNz8Mvhxtj0cHy9c1qNv32pyqcGMuANyAjDEcH7gAUEHwuvpKvLQ/dwd5dX+i/V9fTeMBLEZs1jcRFwpbwg932nPz6xj9n4neVuUoooriPpj9FP+Cfat/wAKNvSc7TrdwV+nlw/1zRW3+wzpD6X+zpoksg5vrq7u1PqpnZVP5KKK/SsDHlw1NPsj+GeK6iqZ9jZR/wCfkl9zt+h0f7V3guTx38AfF1hBGZLu3t11GBVGWL27rNtHuwRl/wCBV+VtftXLGs0bxyKrowKsrDIIPUEV+Rvxu+Gs3wj+KWveGXTbawTGexbGA9rIS0RH0HyH3Rq8DPKLvCsvR/p+p+x+FGaR9niMrm9U+ePztGX3Wj97OGooor5Q/oMVWZHR0Zo3Rg6ujFWVgcggjkEHkEdK+xfhJ8QvD/7TPg7/AIQP4hIk/iS2Qva3vEctxgECaJh92ZR95RwwycFSyj45qayvLnTb62vbO4ktL21kWaC4hba8UinKsp7EGu3C4l4eeqvF7ruj5nPsjp51QSUvZ1oO9Oa3jL/J9V19Uj6Q8Gfs33Hgv41N4T8W2v8AavhbxBpt3aWmpwgqkjrsmTnrFOgiLD6EgkZA8h+L/wAI9X+Dfit9J1I/aLSbdJp9+owt1CDjJHZ1yAy9iQejCvs/9nH9oK0+L+i/2dqjQ2ni6xQGe3XhbqMDHnxD07Mv8JI7EGus8b+G/Cfxw0HXPC11cx3E9jL5UrQEfaNPuNuUcAjg4YHkYYEjkZr6KWX0MRh74d6vWP8A8i/60PxSlxpnGTZy6ecUnyxjGNW22jaVWPRXTSdtJeTsl+Zler/s/wDwHvvjP4h3z+ZaeFrJ/wDT71OGkbAIgjP98gjJ/hBz1K1o+HP2U/GGqfFKfwnqMLWWn2ZE11rcafuXtyflaHOQXfBAU/dIO7IHP3B4dj8L+ATpPgnSnt7GYWzy2unK26VolI3yt3OWblj1JNcGAy2VSfPiFaKfXq+x9hxfxvTwOGWHyiftK1SPNzR1UIWvzettUum76X+S7P8AZvuvi18avFD2sH/CP+AdLvxYLcQoF8xbeNIWhtx6ho2DOeAc9WyBr/tBfGTSfh34fb4XfDhI9MhgDQ6jdWZ/1QOd8SPnJkYn535I5GdxJXtf2o/2jl8B2k3hLwrcqPElwn+l3kOP+Jejc8H/AJ7MDkf3Qdx5Kg/Dyrtz1OTkknJJPUk9zVYytTwjlRw/xP4n+i/r8TLhvL8dxHHD5jnCtQpKPsqb+1ZJe0nfdvdX330XxCqEUKoCqBgADAFLRRXzp+zBSrHLMwjgiM07nbHGvV2PAUfU8Ule2/sffDKT4kfGzSZpIt+k6Ay6reMwBXch/cJ9TIFb6RtW1Gk61SNOO7PNzLH08rwVXG1vhpxb9bbL5uyXmz9Gfhf4PX4f/Dfwx4aBDNpWm29o7D+N0jAdvxYE/jRXUUV+oRioxUVsj+B61aeIqyrVHeUm2/V6sK+ZP24fgfJ8QfBMfi3SYWk13w9GzSxRoWe5szy6ADqyHLjrxvA5avpukrHEUY4im6U9mellGaV8mx1LHYf4oP71s0/JrQ/FFWDKGUgqRkEdDS19LftgfszyfDDXJvF3h22Z/CWpTZnt41402duo9onP3f7rHbwNtfNNfm+IoTw9R05rVH9v5Rm2FzrBwxuEleMvvT6xfmv+CtGgooornPZL+g69qHhfXLHWNKunstSspRLb3EfVG6fiCCQQeCCQeDXrPw31Lxz8Wfj3c+IfCUv9haveSR3Oo3UYL2tvEERH8xf4kbYdsZ5J4BG3ePM/Bfg7UfHniCDSNLWMTOrSyzztshtoV5eaVv4UUHk+4A5IFeo+JfjHpvw98JS+A/hfNLDalsap4qIMdzqUuMO0XeNOwbqAMLj7zehh/dXPUk1BO+m7fl+r/wCAfH50vbVHhsFRjPETi4tyV4wpyern3V17sdXJ3tpzH2bP8TNC1zXtX8GaJ4msIvGMFqxRHTzVil2nB25AcocFow2QCM4zXws3izxt8Efit4gvtbaa58YvazWpvLxy4PmldtwmRh1AU7AAFHTA2la8zt7qezuorm3mkt7mJxJHNE5V0cHIYMOQc9xXuUHxG0r4/wDh608MePLiHS/GFopXRvFjKEilb/nhdYHyh8D5gNuecKfvehUxzx1teScfh10fl5Psz47AcKw4V57R9vhqsUqt4rnjbeStrKm/tQ3S1XNZnhlxcS3dxLcXErz3EztJLNIxZndiSzMTySSSST3NR1oa9oOoeF9avNJ1W1ey1Gzk8uaCTqp6g+4IIII4III4NZ9eA7p2e5+vU5QnBSpu8WtLbW6W8gooozjk8CkWKAzMqqrO7EKqIpZmYnAAA5JJ4AHWv1D/AGUPgk3wX+GMMOoIo8SaswvdTYc+WxHyQZ9I1OPQsXPevB/2J/2Z5bm6s/iR4ptmigibzNE0+QYLt2unHp/zzH/A/wC7X3FX2WUYF019YqLV7enf5n8y+JHFUMbP+xsFK8IO82tnJbR9I9f73+EKKKK+mPwkKKKKAKupabaa1p9zYX9tFeWVzG0U1vMgdJEYYKsD1BFfn3+0n+xrqfw9muPEPgm2uNY8L4Mk9ggMt1YdSSAOZIvcZZe+Rlh+h1JXDisHTxkOWe/R9j6vh7iTHcN4j22Fd4v4ovaX+TXRrbzV0/xRVg6hlIZSMgjoaWv0r+Nf7GPg/wCKks+p6Uf+ES8QyEs11ZQhredvWWHgEk9WUqxPJJ6V8W/Ez9lv4jfC95pb3QJtX0yPJGpaKrXUW3PBZVG9OOu5QB618TictxGGd2rruv60P6nyPjfKM8ioxqezq/yTdnfye0vlr3SPP5vEk0fh1tCsd1pps7CW+UNl76Qfd8090X+GMfKPvHc3zVjUiyKzMFYEqdrAHofQ0tebKTlufdU6UaSait9X5vu/62stkFFFFSaGxfeJrnWtHtbHVGa9ksE8qxvGb99DF/zwZj9+MZyoPKdFO0lax6dErT3EcEStLPIcJFGCzsfQAcmvbPhj+x58SPiQ8M02l/8ACK6S4yb3W0aKTHqkH32P+9sHvXRCnVxErQTbPIxeOy/JqTqYqpGlDfV29bLd97Jb3Z4kqs7oiKzyOwVERSzMxOAABySTwAOtfZX7Mv7FNxcz23ir4j2bW8CESWXh2UDc5ByHuR2HpF1/v45WvfPgn+yr4N+C/lXsEJ13xGoOdY1BFMkeeohXpEO3GWxwWNez19VgcoVNqpiNX26fPufz5xV4kTxsJYPJ7wg9HN6Sf+FfZXn8X+Hq2NFjVVVQqqMBQMACnUUV9MfhIUUUUAFFFFABRRRQAUUUUAcd4w+Dvgjx+S3iHwppOrTZz9ouLRPOB9pANw/A15dqX7Cfwjvpnkg0jUdOZjnFtqk7KPoJGYUUVzTw1Go7zgn8j28JnmaYGPJhcTOC7KUkvuvYyx/wT9+GXmbvtXiAj+79uTH/AKL/AM5rb0j9hz4Q6XMssmgXeoyKcj7Zqdwy590Vwp+hGKKKyWBwsdqa+476nFWe1FyyxtS3lNr8mj1bwj8NfCfgKER+HPDel6IoGN1jaJEx+rAZP4muloorsjFRVoqyPmqtapXm6lWTlJ9W7v72FFFFUZBRRRQAUUUUAf/Z)**Department of Computer Software Engineering**

**Lab Rubrics   
Open Ended Lab**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Criteria** | **Unacceptable  (Marks=0)** | **Substandard Marks=1** | **Adequate Marks=2** | **Proficient Marks=3** |
| **Functionality**  **CLO-2 (C-3)** | The software does not fulfill any of the required functionality. | The software only partially fulfills the required functionality, or the functionality is very limited. | The software fulfills most of the required functionality, but with some errors or issues. | The software fully and correctly implements all the required functionality. |
| **Layout**  **CLO-4**  **(P-3)** | The user interface is unusable and difficult to understand. It is not intuitive and does not allow the user to complete tasks effectively. | The user interface is somewhat usable, but there are major issues that hinder the user's ability to complete tasks effectively. The design may be unappealing. | The user interface is generally usable. However, there may be some minor issues with the layout or design that could be improved | The user interface is highly usable and intuitive, allowing the user to complete tasks with ease and the design is aesthetically pleasing. |
| **Creativity**  **and**  **Design**  **CLO-2**  **(C-3)** | Use of Elements detracts from presentation.  Graphics, color schemes, transition and backgrounds disrupt flow. | Little evidence of original thinking.  Minimal use of design elements or graphics. | Graphic elements moderately enhance and support the presentation of content. | Excellent sense of creativity and design.  Graphic elements enhance and support the presentation of content. |
| **Testing**  **CLO-2 (C-3)** | The student did not conduct any testing or validation, or the testing was insufficient or inadequate. | The student conducted some testing, but it was not comprehensive or thorough enough to ensure the correctness and robustness of the program. | The student conducted adequate testing and validation of their code, but there were some minor issues or oversights that could have been improved upon. | The student conducted thorough and comprehensive testing and validation of their code and ensuring the correctness and robustness of their program. |
| **Code Quality**  **CLO-4**  **(P-3)** | Code is difficult to understand and follow.  Code is not commented, or the comments are insufficient.  The code does not follow best practices or coding conventions. | Code is somewhat understandable and followable.  and can be navigated.  Code contains some comments, but they could be more detailed.  The code follows some best practices or coding conventions. | Code is easy to understand and follow.  Code contains detailed comments that make it easy to understand.  The code follows most best practices or coding conventions. | Code is extremely easy to understand and follow.  Code contains detailed comments that make it effortless to understand.  The code follows all best practices and coding conventions. |

|  |  |  |  |
| --- | --- | --- | --- |
| SE-314 Software Construction | | | |
| Course Learning Outcomes (CLOs) | | | |
| At the end of the course the students will be able to: | | PLOs | BT Level\* |
| 1. | Explain the principles of Software construction | 1 | C-2 |
| 2. | Apply patterns, frameworks and techniques for software construction | 2 | C-3 |
| 3. | Apply principles of bug-free, ready to change and easy to understand software construction | 3 | C-3 |
| 4. | Practice and adapt modern tools for software construction | 5 | P-3 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Mapping of CLOs to Program Learning Outcomes** | | | | |
| **PLOs/CLOs** | **CLO1** | **CLO2** | **CLO3** | **CLO4** |
| PLO 1 (Engineering Knowledge) | √ |  |  |  |
| PLO 2 (Problem Analysis) |  | √ |  |  |
| PLO 3 (Design/Development of Solutions) |  |  | √ |  |
| PLO 4 (Investigation) |  |  |  |  |
| PLO 5 (Modern tool usage) |  |  |  | √ |
| PLO 6 (The Engineer and Society) |  |  |  |  |
| PLO 7 (Environment and Sustainability) |  |  |  |  |
| PLO 8 (Ethics) |  |  |  |  |
| PLO 9 (Individual and Team Work) |  |  |  |  |
| PLO 10 (Communication) |  |  |  |  |
| PLO 11 (Project Management) |  |  |  |  |
| PLO 12 (Lifelong Learning) |  |  |  |  |

**Lab Experiments CLO Mapping**

|  |  |  |  |
| --- | --- | --- | --- |
| **S No** | **Experiment Title** | **CLO** | **R-G** |
| 1 | UML Diagrams | 1 | 1 |
| 2 | Process of Creation of Detailed Design Document | 1 | 1 |
| 3 | Javadoc Utility | 4 | 1 |
| 4 | Java Package and Java Applet Creation | 4 | 1 |
| 5 | Visibility of Variables in Java | 4 | 1 |
| 6 | Version Control – I | 4 | 1 |
| 7 | Version Control – II | 4 | 1 |
| 8 | Static Binding vs Dynamic Binding | 4 | 1 |
| 9 | Exception Handling | 4 | 1 |
| 10 | Mutability and Immutability | 4 | 1 |
| 11 | Assertions | 4 | 1 |
| 12 | Recursion and Iteration | 4 | 1 |
| 13 | Abstract Data Types | 4 | 1 |
| 14 | Abstract Functions and Interfaces | 4 | 1 |
| 15 | Open Ended Lab | 2,4 | OEL |

**MARKS**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Date** | **Experiment** | **Max. Marks** | **Marks Obtained** | | | | | **Instructor Sign** |
| **R1** | **R2** | **R3** | **R4** | **R5** |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| **Grand Total** | |  |  |  |  |  |  |  |

**CONTENTS**

[Experiment # 1: Constructing UML Diagrams for Software Design 12](#_Toc206658205)

[Experiment # 2: Creating a Detailed Software Design Document 15](#_Toc206658206)

[Experiment # 3: Generating API Documentation Using Javadoc 16](#_Toc206658207)

[Experiment # 4: Java Package and Applet Creation 17](#_Toc206658208)

[Experiment # 5: Managing Variable Scope and Visibility in Java 19](#_Toc206658209)

[Experiment # 6: Version Control with Git – Part I 21](#_Toc206658210)

[Experiment # 7: Advanced Git Operations – Part II 23](#_Toc206658211)

[Experiment # 8: Static Binding vs Dynamic Binding 30](#_Toc206658212)

[Experiment # 9: Exception Handling 33](#_Toc206658213)

[Experiment # 10: Understanding Mutability and Immutability in Java 36](#_Toc206658214)

[Experiment # 11: Validating Code with Assertions 40](#_Toc206658215)

[Experiment # 12: Recursion and Iteration 42](#_Toc206658216)

[Experiment # 13: Implementing Abstract Data Types 43](#_Toc206658217)

[Experiment # 14: Designing Abstract Classes and Interfaces 45](#_Toc206658218)

[Experiment # 15: Open-Ended Lab 48](#_Toc206658219)

# Experiment # 1: Constructing UML Diagrams for Software Design

**Objective:** To explain the fundamentals and purpose of UML diagrams in the context of software construction.

**Programming Language**: Java

**Tool**: MS Visio /Visual Paradigm

**Time:** 3Hrs

**Introduction**: **What is UML?**

UML stands for Unified Modeling Language. UML 2.0 helped extend the original UML specification to cover a wider portion of software development efforts including agile practices.

* Improved integration between structural models like class diagrams and behavior models like activity diagrams.
* Added the ability to define a hierarchy and decompose a software system into components and sub-components.
* The original UML specified nine diagrams; UML 2.x brings that number up to 13. The four new diagrams are called: communication diagram, composite structure diagram, interaction overview diagram, and timing diagram. It also renamed state chart diagrams to state machine diagrams, also known as state diagrams.

**Types of UML Diagrams:**

The current UML standards call for 13 different types of diagrams: class, activity, object, use case, sequence, package, state, component, communication, composite structure, interaction overview, timing, and deployment.

These diagrams are organized into two distinct groups: structural diagrams and behavioral or interaction diagrams.

Structural UML diagrams

* Class diagram
* Package diagram
* Object diagram
* Component diagram
* Composite structure diagram
* Deployment diagram

Behavioral UML diagrams

* Activity diagram
* Sequence diagram
* Use case diagram
* State diagram
* Communication diagram
* Interaction overview diagram
* Timing diagram

1. **UML Use Case Diagrams**

Use case diagrams are usually referred to as behavior diagrams used to describe a set of actions (use cases) that some system or systems (subject) should or can perform in collaboration with one or more external users of the system (actors). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system.

**When to apply use case diagrams:**

A use case diagram doesn't go into a lot of detail—for example, don't expect it to model the order in which steps are performed. Instead, a proper use case diagram depicts a high-level overview of the relationship between use cases, actors, and systems. Experts recommend that use case diagrams be used to supplement a more descriptive textual use case.

**Use case diagram components:**

Common components include:

***Actors***: The users that interact with a system. An actor can be a person, an organization, or an outside system that interacts with your application or system. They must be external objects that produce or consume data.

***System***: A specific sequence of actions and interactions between actors and the system. A system may also be referred to as a scenario.

***Goals***: The result of most use cases. A successful diagram should describe the activities and variants used to reach the goal.

**Use case diagram symbols and notation:**The notation for a use case diagram is straightforward and doesn't involve as many types of symbols as other UML diagrams.

* **Use cases:** Horizontally shaped ovals that represent the different uses that a user might have.
* **Actors:** Stick figures that represent the people employing the use cases.
* **Associations:** A line between actors and use cases. In complex diagrams, it is important to know which actors are associated with which use cases.
* **System boundary boxes:** A box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system.
* **Packages:** A UML shape that allows you to put different elements into groups. Just as with component diagrams, these groupings are represented as file folders.
* **Relationships:**

Illustrate relationships between an actor and a use case with a simple line. For relationships among use cases, use arrows labeled either "uses" or "extends." A "uses" relationship indicates that one use case is needed by another to perform a task. An "extends" relationship indicates alternative options under a certain use case.

1. **Sequence Diagram:**

UML sequence diagrams model the flow of logic within your system in a visual manner, enabling you both to document and validate your logic, and are commonly used for both analysis and design purposes.

Sequence diagrams are typically used to model:

* Usage scenarios. A usage scenario is a description of a potential way your system is used. The logic of a usage scenario may be part of a use case, perhaps an alternate course. It may also be one entire pass through a use case, such as the logic described by the basic course of action or a portion of the basic course of action, plus one or more alternate scenarios. The logic of a usage scenario may also be a pass through the logic contained in several use cases. For example, a student enrolls in the university, and then immediately enrolls in three seminars.
* The logic of methods. Sequence diagrams can be used to explore the logic of a complex operation, function, or procedure. One way to think of sequence diagrams, particularly highly detailed diagrams, is as visual object code.
* The logic of services. A service is effectively a high-level method, often one that can be invoked by a wide variety of clients. This includes web-services as well as business transactions implemented by a variety of technologies.
  + **Class Diagram**

Class diagrams are the main building block of any object-oriented solution. It shows the classes in a system, attributes, and operations of each class and the relationship between each class.

In most modeling tools, a class has three parts. Name at the top, attributes in the middle and operations or methods at the bottom. In a large system with many related classes, classes are grouped together to create class diagrams. Different relationships between classes are shown by different types of arrows.

* + **Package Diagram**

As the name suggests, a package diagram shows the dependencies between different packages in a system.

**Elements**

***Package***: a general-purpose mechanism for organizing model elements & diagrams into groups. It provides an encapsulated namespace within which all the names must be unique. It is used to group semantically related elements. It is a namespace as well as an element that can be contained in other packages' namespaces.

***Class***: a representation of an object that reflects its structure and behavior within the system. It is a template from which running instances are created. Classes usually describe the logical structure of the system.

***Interface***: a specification of behavior. An implementation class must be written to support the behavior of an interface class.

***Object***: an instance of a class. It is often used in analysis to represent an artifact or other item.

Table: a stereotyped class.

* + **Deployment Diagram**

A deployment diagram shows the hardware of your system and the software in that hardware. Deployment diagrams are useful when your software solution is deployed across multiple machines with each having a unique configuration.

**Exercise:**

Q: Briefly explain UML Diagrams?

Q: Describe Purpose of each UML Model?

Q: Explain When to design which particular UML Diagram is based upon given scenario

**Web Resources:**

<https://www.uml-diagrams.org>

<https://www.lucidchart.com>

<https://www.smartdraw.com/uml-diagram/>

<https://www.uml-diagrams.org>

<https://www.lucidchart.com>

<https://www.smartdraw.com/uml-diagram/>

# Experiment # 2: Creating a Detailed Software Design Document

**Objective:** To describe and document detailed architectural and design components for a software system.

**Programming Language**: Java

**Tool**: MS Visio/Visual Paradigm

**Time:** 3Hrs

**Detailed Design Document:**

A software design description or SDD or Software Design Specification, is a written description of a software product, that a software designer writes in order to give a software development team overall guidance to the architecture of the software project.

**Purpose of A Design Document:**

The purpose of the Software Design Document is to provide a description of the design of a system fully enough to allow for software development to proceed with an understanding of what is to be built and how it is expected to build.

**Importance of Design Document:**

A design document is quite simply an effective way for you to communicate to others who may be interested in your product, what your design decisions are and why your decisions are worthy and reasonable decisions.

**EXERCISE:**

Q: Consider any Software Detailed Design Document and explain process of creation of detailed design document with the help of flow chart?

**Web Resources:**

<https://en.wikipedia.org/wiki/Software_design_description>

<https://www.google.com/search?q=Detailed+Design+Document&tbm=isch&source=univ&sa=X&ved=2ahUKEwjlrpHeqO7nAhXIKewKHZysCTcQsAR6BAgJEAE>

<https://www.projectmanagementdocs.com/template/project-documents/system-design-document/>

# Experiment # 3: Generating API Documentation Using Javadoc

**Objective:** To practice generating Javadoc to create structured and reusable API documentation.  
**Programming Language**: Java

**Tool**: Cmd, JDK, Java Applet Viewer

**Time:** 3Hrs

**General (for all OS)**

In DOS command prompt do followings to set the environment for java

* *Set CLASSPATH=%CLASSPATH%;* C:\Program Files\Java\jdk-13.0.2\lib
* *Set Path=%path%;* C:\Program Files\Java\jdk-13.0.2\bin
* Type Set & enter, this should print all the environment variables and check your setting of Classpath & Path.

**Win 2000 or Later versions**

Set Environment variables

* Double click on the system icon in the control panel.
* On the advance tab press Environment variables button
* Under system variables find path and press edit
* At the end of the path string add “C:\Program Files\Java\jdk13.0.2\bin;”
* Press new and enter variable name:” CLASSPATH” (if not already existed) and variable value:” C:\Program Files\Java\jdk1.5.0\bin;”

**JAR Files**

Unzip scr.zip which will open up the source code of all packages (which collectively called java API).

rt.jar in ..\jdk1.5.0\jre\lib contains all the class files for respecting source files in scr.zip. Unjar this file and compare.

**Create Documentation**

1 – Make directory c:\home\html

2- Copy jdk folder from C drive and paste it in your current directory

3- Open jdk folder and goto Bin

4- Create a file Hello.java having “Hello world java code “in Bin folder

5- run the following command to create documentation using javadoc utility.

Working dir>**javadoc hello.java**

**6-** Observe what happened after executing above command

# Experiment # 4: Java Package and Applet Creation

**Objective:**To practice using Java tools for package management and creating applets.

**Programming Language**: Java

**Tool**: Cmd, JDK, Java Applet Viewer

**Time:** 3Hrs

**Create Your own package**

1 – Add statement “package APC.lab1;” as first line of your code

2 – Compile with javac –d . yourProgram.java

3 – This will create your yourProgram.class file in the <workingDir>\ APC\lab1

4 – Make a class to call yourProgram from step3

5 – **create a jar file by following command**

jar cf myapi.jar. **C:\home\html\jdk-13.0.2\bin\APC\lab1\HelloPrg.class**

// bold text is the path where your class file exists; gives that path in above command

6 – **Add this jar file in the lib folder of classpath and change the class path**

(CLASSPATH= C:\Program Files \ Java \ jdk1.5.0 \ lib \ myapi .jar

7 –Your API is added in the classpath and now you are able to make your program calling your own api from the lib folder.

**Java Application**

1 – Write a simple Application to show a message “Hello world”

2 – Write an application using swing package to accept two integers from the user and display a result of addition of input integers.

3 – Try a program with

* System.exit(0);
* System.exit(1);

Explain the differences in the two tried exercises.

**For Compilation:**

1. Goto your working directory.
2. Compile your application by command javac filename.java
3. Execute your application by command java filename

**Java Applet**

1 – Write a simple applet displaying “Hello world” with import java.applet.Applet;

2 – Write a simple applet displaying “Hello world” with import java.swing.JApplet;

**Note (for 1 & 2):**

Run the both applets with appletviewer and Internet browsers.

3 – Write an applet to show the addition of two integers entered by user.

**Web Resources:**

<http://www.skylit.com/javamethods/faqs/createjar.html>

<https://docs.oracle.com/javase/tutorial/deployment/jar/build.html>

<https://www.thoughtco.com/building-your-first-java-applet-2034332>

# Experiment # 5: Managing Variable Scope and Visibility in Java

**Objective:** To practice implementing variables with appropriate scope and visibility for code maintainability.  
**Programming Language**: Java

**Time Required:** 3 hrs

**Software/Tools Required:** IntelliJ IDEA / Eclipse / NetBeans

A variable provides us with named storage that our programs can manipulate. Java provides three types of variables.

* **Class variables** − Class variables also known as static variables are declared with the static keyword in a class, but outside a method, constructor, or a block. There would only be one copy of each class variable per class, regardless of how many objects are created from it.
* **Instance variables** − Instance variables are declared in a class, but outside a method. When space is allocated for an object in the heap, a slot for each instance variable value is created. Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object's state that must be present throughout the class.
* **Local variables** − Local variables are declared in methods, constructors, or blocks. Local variables are created when the method, constructor or block is entered, and the variable will be destroyed once it exits the method, constructor, or block.

**Task 1:** Write program in java to demonstrate

Class variables

Instance variables

Local variables

**Task 2:** Demonstrate that default variables in java would be initialized as local variables, class variables or instance variables? show by working code?

# 

# Experiment # 6: Version Control with Git – Part I

**Objective:** To practice initializing repositories and basic version control commands using Git.  
**Programming Language**: Java

**Time Required:** 3 hrs

**Software/Tools Required:** Git Bash / GitHub / Git GUI

Required Reading (from the Java Tutorial):

● Hello World!

● Primitive Data Types

● Assignment/Arithmetic Ops

● Equality/Relational/Conditional Ops

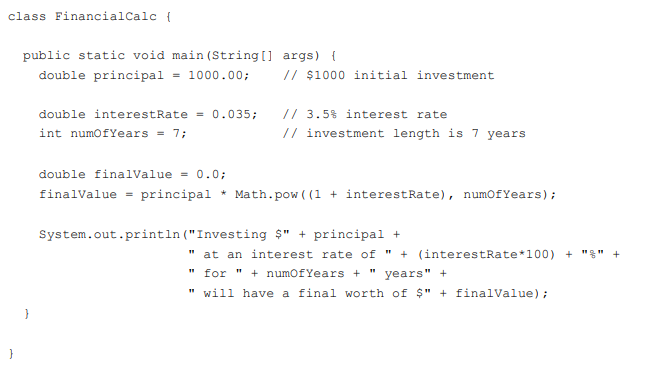
● Math class

● Defining methods

● Calling methods

● Returning a value from a method

Look at the code below which calculates the value of investing an initial sum of money at a specified interest rate and for a specified number of years. Create FinancialCalc.java file in NetBeans with code given below.



**Task 1: Run the code in NetBeans to see what it does.** Look for the output in the Console pane at the bottom of the screen.

**Task 2: Change the program so that it calculates the final value in a separate method** computeFinalValue, which should be a function of three arguments (the principal, interest rate, and number of years).

Your **main** method should still print the same result on the output.

In general, the final value (V) of an investment (principal P) compounded yearly for Y years at interest rate I is given by the formula:

Given any three of these variables, you can write code that computes the fourth.

**Task 3: Write methods that compute each of the variables (P, I, and Y)** **when given values for the other three, and demonstrate by having your main method print the following:**

1. The amount of money you should invest today (at an interest rate of 5%) to have a total amount of $1000.00 at the end of 4 years.

2. The interest rate you need to turn an initial investment of $500.00 into $525.00 at the end of 3 years **(hint: be careful when carrying out division on integers! Cast the integer to a double when necessary).**

3. The number of years you need to invest an initial sum of $100.00 at an interest rate of 4.4% to have a final value of $150.00 **(hint: the number of years is not necessarily an integer).**

# Experiment # 7: Advanced Git Operations – Part II

**Objective:** To practice initializing repositories and basic version control commands using Git.  
**Programming Language**: Java  
**Time Required:** 3 hrs

**Software/Tools Required:** Git Bash / GitHub / Git GUI

**Using Git in Apache NetBeans:**

The Apache NetBeans IDE provides support for the [Git version control system](https://git-scm.com/). The IDE’s Git features let you perform versioning tasks directly from your projects and code within the IDE. Every Git clone is a full-fledged repository with complete history and full revision tracking capabilities, not dependent on network access or a central server. Branching and merging are fast and easy to do. Git is used for version control of files, much like tools such as Mercurial, Subversion, CVS, and Perforce.

**Initializing a Git Repository:**

To initialize a Git repository from existing files that are not in source control yet, you need to complete the following steps:

1. In the Projects window, select an unversioned project and right-click the project name.
2. In the context menu, choose Versioning > Initialize Git Repository (alternatively, in the main menu, choose Team > Git > Initialize).
3. Specify the path to the repository you are going to store your versioned files in the Initialize a Git Repository dialog box or click Browse and browse for the directory required.
4. Click OK.

A. git subfolder is created in the folder you specified in step 3 above (your NetBeans project folder by default), which is your Git repository where all the data of your project snapshots are stored. Git starts versioning all files in the folder you specified.

1. Open Window > Output > Output to view the IDE’s report about the progress of the repository creation under your local working directory.

All the project files are marked Added in your Working Tree.

1. To view a file [status](http://netbeans.apache.org/kb/docs/ide/git.html#viewFileStatus), place the cursor over the file name in the Projects window. The status of the file in the Working Tree displayed green to the right of the slash.

After you initialized the Git repository, you either [add](http://netbeans.apache.org/kb/docs/ide/git.html#add) files or directly [commit](http://netbeans.apache.org/kb/docs/ide/git.html#committing) them to the Git repository.

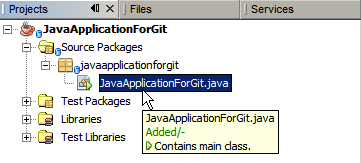
**Adding Files to a Git Repository:**

To start tracking a new file and also to stage changes to an already tracked file in the Git repository, you need to add it to the repository.

When adding files to a Git repository, the IDE composes and saves snapshots of your project first in the Index. After you perform the commit, the IDE saves those snapshots in the HEAD. The IDE allows you to choose between the two workflows described in the following table.

| **Workflow Description** | **Explicitly add new or modified files to the Index and then commit only those that are staged in the Index to the HEAD** | **Skip adding new or modified files to the Index and commit the required files directly to the HEAD** |
| --- | --- | --- |
| Steps to Follow the Workflow | 1. In the Projects window, right-click the file you want to add.  [start=2]. In the context menu, choose Git > Add. This adds the file contents to the Index before you commit it.  [start=3]. In the Projects window, right-click the file you want to commit.  [start=4]. In the Commit dialog box, select the Changes between HEAD and Index ( changes head index ) toggle button. This displays the list of files that are already staged.  [start=5]. Commit the file(s) as described in the [Committing Sources to a Repository](http://netbeans.apache.org/kb/docs/ide/git.html#committing) section below. | 1. In the Projects window, right-click the file you want to commit.  [start=2]. In the context menu, choose Git > Commit.  [start=3]. In the Commit dialog box, select the Select the Changes between HEAD and Working Tree (changes head wt) toggle button. This displays the list of files that are not staged.  [start=4]. Commit the file(s) as described in the [Committing Sources to a Repository](http://netbeans.apache.org/kb/docs/ide/git.html#committing) section below. |

|  |  |
| --- | --- |
|  | The [status](http://netbeans.apache.org/kb/docs/ide/git.html#viewFileStatus) of the file in the HEAD displays in green to the left of the slash like shown in the following picture. |



The action works recursively if invoked on folders while respecting the NetBeans IDE flat folder content structure.

**Editing Files:**

Once you have a Git versioned project opened in the IDE, you can begin making changes to sources. As with any project opened in NetBeans IDE, you can open files in the Source Editor by double-clicking on their nodes, as they appear in the IDE’s windows (e.g., Projects (Ctrl-1), Files (Ctrl-2), Favourites (Ctrl-3) windows).

When working with source files in the IDE, there are various UI components at your disposal, which aid in both viewing and operating version control commands:

* [Viewing Changes in the Source Editor](http://netbeans.apache.org/kb/docs/ide/git.html#_viewing_changes_in_the_source_editor)
* [Viewing File Status Information](http://netbeans.apache.org/kb/docs/ide/git.html#_viewing_file_status_information)
* [Reverting Changes](http://netbeans.apache.org/kb/docs/ide/git.html#_reverting_changes)

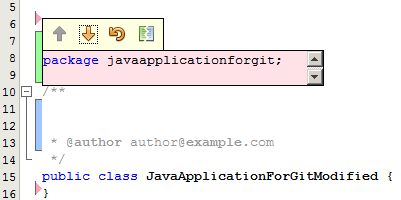
**Viewing Changes in the Source Editor:**

When you open a versioned file in the IDE’s Source Editor, you can view real-time changes occurring to your file as you modify it against the base version from the Git repository. As you work, the IDE uses color coding in the Source Editor’s margins to convey the following information:

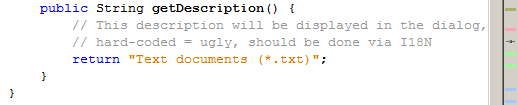
* **Blue.** Indicates lines that have been changed since the earlier revision.
* **Green.** Indicates lines that have been added since the earlier revision.
* **Red.** Indicates lines that have been removed since the earlier revision.

The Source Editor’s left margin shows changes occurring on a line-by-line basis. When you modify a given line, changes are immediately shown in the left margin.

|  |  |
| --- | --- |
|  | You can click on a color grouping in the margin to call versioning commands. For example, the picture below shows widgets available to you when clicking a red icon which indicates that lines have been removed from your local copy: |

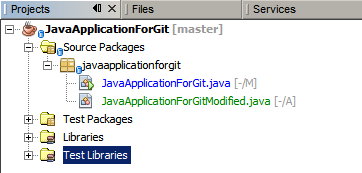


The Source Editor’s right margin provides you with an overview that displays changes made to your file, from top to bottom. Color coding is generated immediately when you make changes to your file.



**Viewing File Status Information:**

When you are working in the Projects (Ctrl-1), Files (Ctrl-2), Favorites (Ctrl-3), or Versioning views, the IDE provides several visual features that aid in viewing status information about your files. In the example below, notice how the badge (e.g. blue badge), color of the file name, and adjacent status label, all coincide with each other to provide you with a simple but effective way to keep track of versioning information on your files:



Badges, color coding, file status labels, and perhaps most importantly, the Git Diff Viewer all contribute to your ability to effectively view and manage versioning information in the IDE.

* [Badges and Color Coding](http://netbeans.apache.org/kb/docs/ide/git.html#_badges_and_color_coding)
* [File Status Labels](http://netbeans.apache.org/kb/docs/ide/git.html#_file_status_labels)
* [Git Versioning View](http://netbeans.apache.org/kb/docs/ide/git.html#_git_versioning_view)

**Badges and Color Coding:** Badges are applied to project, folder, and package nodes and inform you of the status of files contained within that node:

The following table displays the color scheme used for badges:

|  |  |
| --- | --- |
| **UI Component** | **Description** |
| Blue Badge blue badge | Indicates the presence of files that have been modified, added, or deleted in your working tree. For packages, this badge applies only to the package itself and not its sub-packages. For projects or folders, the badge indicates changes within that item, or any of the contained subfolders. |
| Red Badge red badge | Marks projects, folders or packages that contain *conflicting* files. For packages, this badge applies only to the package itself and not its sub-packages. For projects or folders, the badge indicates conflicts within that item, or any of the contained subfolders. |

**File Status Labels:** Color coding is applied to file names to indicate their current status against the repository:

| **Color** | **Example** | **Description** |
| --- | --- | --- |
| No specific color (black) | black text | Indicates that the file has no changes. |
| Blue | blue text | Indicates that the file has been locally modified. |
| Green | green text | Indicates that the file has been locally added. |
| Red | red text | Indicates that the file is in a merge conflict. |
| Gray | gray text | Indicates that the file is ignored by Git and will not be included in versioning commands (e.g., Update and Commit). Files cannot be ignored if they are versioned. |

**Git Versioning View:** The Git Versioning view provides you with a real-time list of all the changes made to files within a selected folder of your local working tree. It opens by default in the bottom panel of the IDE, listing added, deleted, or modified files.

1. To open the Versioning view, select a versioned file or folder (e.g., from the Projects, Files, or Favourite’s window) and either choose Git > Show Changes from the right-click menu, or choose Team > Show Changes from the main menu.

By default, the Versioning view displays a list of all modified files within the selected package or folder in your Working Tree. Using the buttons in the toolbar, you can choose to display the list of files which have differences either between Index and HEAD, Working Tree and Index or Working Tree and HEAD. You can also click the column headings above the listed files to sort the files by name, status or location.

The Versioning view toolbar also includes buttons that enable you to invoke the most common Git tasks on all files displayed in the list. The following table lists the Git commands available in the toolbar of the Versioning view:

|  |  |  |
| --- | --- | --- |
| **Icon** | **Name** | **Function** |
| changes head wt | Changes between HEAD and Working Tree | Displays a list of files that are either already staged or only modified/created and not staged yet. |
| changes head index | Changes between HEAD and Index | Displays a list of files that are staged. |
| changes index wt | Changes between Index and Working Tree | Displays files that have differences between their staged and Working Tree states. |
| refresh | Refresh Statuses | Refreshes the status of the selected files and folders. Files displayed in the Versioning view can be refreshed to reflect any changes that may have been made externally. |
| open diff | Open Diff | Opens the Diff Viewer providing you with a side-by-side comparison of your local copies and the versions maintained in the repository. |
| update | Revert Modifications | Displays the [Revert Modifications](http://netbeans.apache.org/kb/docs/ide/git.html#revertdialog) dialog box. |
| commit button | Commit Changes | Displays the [Commit](http://netbeans.apache.org/kb/docs/ide/git.html#commitdialog) dialog box. |

1. You can access other Git commands in the Versioning view by selecting a table row that corresponds to a modified file, and choosing a command from the right-click menu.

**Comparing File Revisions:**

Comparing file versions is a common task when working with versioned projects. The IDE enables you to compare revisions by using the Diff command:

|  |  |
| --- | --- |
|  | Several comparing modes - Diff To HEAD, Diff To Tracked, and Diff To - are available in the IDE. |

1. Select a versioned file or folder (e.g. from the Projects, Files , or Favorites window).
2. Choose Team > Diff > Diff to HEAD from the main menu. A graphical Diff Viewer opens for the selected file(s) and revisions in the IDE’s main window. The Diff Viewer displays two copies in side-by-side panels. The more current copy appears on the right side, so if you are comparing a repository revision against your working tree, the working tree displays in the right panel.

The Diff Viewer makes use of the same [color coding](http://netbeans.apache.org/kb/docs/ide/git.html#color-coding-table) used elsewhere to display version control changes. In the screen capture displayed above, the green block indicates content that has been added to the more current revision. The red block indicates that content from the earlier revision has been removed from the later. Blue indicates that changes have occurred within the highlighted line(s).

|  |  |
| --- | --- |
|  | Other revisions can be selected from the Diff and to drop-down lists below the Diff Viewer toolbar. |

The Diff Viewer toolbar also includes buttons that enable you to invoke the most common Git tasks on all files displayed in the list. The following table lists the Git commands available in the toolbar of the Diff Viewer:

|  |  |  |
| --- | --- | --- |
| **Icon** | **Name** | **Function** |
| changes head wt | Changes between HEAD and Working Tree | Displays a list of files that are either already staged or only modified/created and not staged yet. |
| changes head index | Changes between HEAD and Index | Displays a list of files that are staged. |
| changes index wt | Changes between Index and Working Tree | Displays files that have differences between their staged and working tree states. |
| nextdiff | Go to Next Difference | Displays next difference in the file. |
| prevdiff | Go to Previous Difference | Displays previous difference in the file. |
| refresh | Refresh Statuses | Refreshes the status of the selected files and folders. Files displayed in the Versioning window can be refreshed to reflect any changes that may have been made externally. |
| update | Revert Modifications | Displays the [Revert Modifications](http://netbeans.apache.org/kb/docs/ide/git.html#revertdialog) dialog box. |
| commit button | Commit Changes | Displays the [Commit](http://netbeans.apache.org/kb/docs/ide/git.html#commitdialog) dialog box. |

**Reverting Changes:**

To throw away local changes made to selected files in your Working Tree and replace those files with the ones in the Index or HEAD:

1. Select a versioned file or folder (e.g., from the Projects, Files, or Favourite’s window).
2. Choose Team > Revert Modifications from the main menu. The Revert Modifications dialog box displays.
3. Specify additional options (e.g., Revert only Uncommitted Changes in Index to HEAD).
4. Click Revert.

The IDE replaces the selected files with those specified in [step 3](http://netbeans.apache.org/kb/docs/ide/git.html#three) above.

**Committing Sources to a Repository:**

To commit files to the Git repository:

1. In the Projects window, right-click the file(s) you want to commit.
2. In the context menu, choose Git > Commit.

The Commit dialog box displays. It contains the following components:

* Commit Message text area intended for describing the change being committed.
* Author and Committer drop-down lists that allow to differentiate between those who made the change and who physically committed the file if necessary.
* Files to Commit section that lists:
* all files modified,
* all files that have been deleted in the Working Tree (locally),
* all new files (i.e. files that do not yet exist in the Git repository),
* all files that you have renamed.

Two toggle buttons that switch the mode in which the actual commit is to be performed are available here:

| **UI Component** | **Name** | **Description** |
| --- | --- | --- |
| changes head index | Changes between HEAD and Index | Displays a list of files that are staged. |
| changes head wt | Changes between HEAD and Working Tree | Displays a list of files that are either already staged or only modified/created and not staged yet. |

**Note**: To specify here whether to exclude individual files from the commit, either deselect the checkbox in the first column called Commit or right-click a file row in the Commit Action column and choose Exclude from commit from the pop-up menu. To display the Diff Viewer here, right-click a file row in the Commit Action column and choose Diff from the pop-up menu.

* Update Issue section intended for tracking issues related to the change being committed.

1. Type in a commit message in the Commit Message text area. Alternatively, you can do any of the following:
   * click the Recent Messages ( recent msgs ) icon located in the upper right corner to view and select from a list of messages that you have previously used,
   * click the Load Template (msg template) icon located in the upper right corner to select a message template.
     1. After specifying actions for individual files, click Commit. The IDE executes the commit and stores your snapshots to the repository. The IDE’s status bar, located in the bottom right of the interface, displays as the commit action takes place. Upon a successful commit, versioning badges disappear in the Projects, Files and Favourites windows, and the colour coding of committed files returns to black.

**Final Task:**

Perform commit on the java source file (not class file) which you created in task 1 and modified in task 2 of your lab 07. Also compare the file revisions after modifying your code and provide your analysis based on your understanding.

# Experiment # 8: Static Binding vs Dynamic Binding

**Objective:** To practice dynamic method dispatch and static method binding using Java examples.  
**Programming Language**: Java

**Time Required:** 3hrs

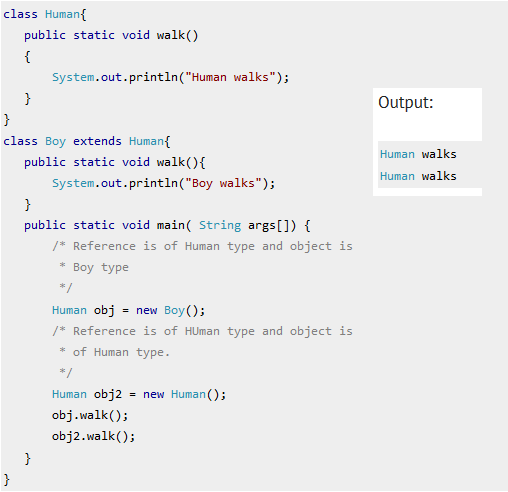
**Software/Tools Required:** IntelliJ IDEA / Eclipse

Association of method call to the method body is known as binding. There are two types of binding: Static Binding that happens at compile time and Dynamic Binding that happens at runtime.

**Static Binding or Early Binding:**

The binding which can be resolved at compile time by compiler is known as static or early binding. The binding of static, private, and final methods is compile-time. The reason is that these methods cannot be overridden, and the type of the class is determined at the compile time. Let’s see an example to understand this:

**Static binding examples:**



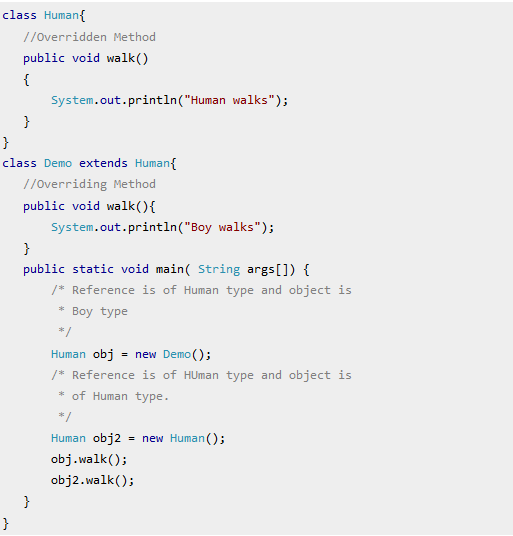
Here we have two classes, Human and Boy. Both the classes have the same method walk () but the method is static, which means it cannot be overridden so even though I have used the object of Boy class while creating object obj, the parent class method is called by it. Because the reference is of Human type (parent class). So whenever a binding of static, private and final methods happen, type of the class is determined by the compiler at compile time and the binding happens then and there.

**Dynamic Binding or Late Binding**

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or late binding. **Method Overriding** is a perfect example of dynamic binding as in overriding both parent and child classes have same method and in this case the **type of the object** determines which method is to be executed. The type of object is determined at the run time so this is known as dynamic binding.

**Dynamic binding example:**

This is the same example that we have seen above. The only difference here is that in this example, overriding is actually happening since these methods are **not** static, private and final. In case of overriding the call to the overridden method is determined at runtime by the type of object thus late binding happens.



**Task1:** Execute following codes and provide output? Also explain which type of binding is implemented in each code?

Code 1:

public class NewClass

{

public static class superclass

{

static void print ()

{

System.out.println("print in superclass.");

}

}

public static class subclass extends superclass

{

static void print ()

{

System.out.println("print in subclass.");

}

}

public static void main (String[] args)

{

superclass A = new superclass ();

superclass B = new subclass ();

A.print ();

B.print ();

}

}

Code 2:

public class NewClass

{

public static class superclass

{

void print()

{

System.out.println("print in superclass.");

}

}

public static class subclass extends superclass

{

@Override

void print()

{

System.out.println("print in subclass.");

}

}

public static void main(String[] args)

{

superclass A = new superclass();

superclass B = new subclass();

A.print();

B.print();

}

}

References:

<https://www.javatpoint.com/static-binding-and-dynamic-binding>

<https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/>

# Experiment # 9: Exception Handling

**Objective:** To practice writing robust code using Java's try-catch-finally and custom exception handling.  
**Programming Language**: Java  
**Time Required:** 3 hrs

**Software/Tools Required:** Windows OS, NetBeans /Eclipse Kepler

An **exception** is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an **exception object**, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called **throwing an exception**.

After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the **call stack***.*

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an **exception handler.** The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to **catch the exception**. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates.

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc. The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application that is why we use exception handling. Suppose there are 10 statements in your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e., statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

**Java Exception Keywords:**

1. **try:** The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
2. **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.
3. **finally:** The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.
4. **throw:** The "throw" keyword is used to throw an exception.
5. **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.

**Example:**

Consider the following code:

Text

Description automatically generated

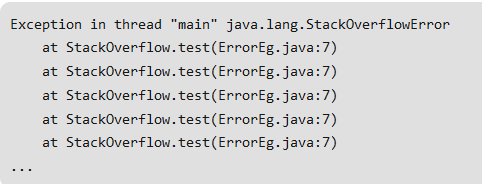
The output of above program will be:

A picture containing text

Description automatically generated

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

**Task 1:** For the registration of a student to online library, there is a registration form with basic details. You are required to create your own exceptions to be raised if the data for the above form in invalidated. In this task you make the exceptions and test your exceptions by a simple driver program. You are not required to create the form or to link the exceptions with the form.

**Task 2:** Write a program that shows that the order of catch blocks is important. If you try to catch a superclass exception type before a subclass type, the compiler should generate errors.

**Task 3:** Write a program that shows a constructor passing information about constructor failure to an exception handler. Define class SomeException, which throws an Exception in the constructor. Your program should try to create an object of type SomeException and catch the exception that is thrown from the constructor.

**Task 4:** Write a program that illustrates rethrowing an exception. Define methods someMethod and someMethod2. Method someMethod2 should initially throw an exception. Method someMethod should call someMethod2, catch the exception and rethrow it. Call someMethod from method main and catch the rethrown exception. Print the stack trace of this exception.

# Experiment # 10: Understanding Mutability and Immutability in Java

**Objective:** To practice creating mutable and immutable objects and analyzing their effect on software behavior

**Programming Language**: Java  
**Time Required:** 3 hrs

**Software/Tools Required:** Windows OS, NetBeans /Eclipse Kepler

**Mutable objects:** The objects whose value can be changed after initialization. We can change the object's values, such as field and states, after the object is created. For example, [Java.util.Date](https://www.javatpoint.com/java-util-date), [StringBuilder](https://www.javatpoint.com/StringBuilder-class), [StringBuffer](https://www.javatpoint.com/StringBuffer-class), etc.

When we made a change in existing mutable objects, no new object will be created; instead, it will alter the value of the existing object. These object's classes provide methods to make changes in it. The Getters and Setters (get() and set() methods) are available in mutable objects.

**Immutable objects:** The objects whose value cannot be changed after initialization. We cannot change anything once the object is created. For example, primitive objects such as [int](https://www.javatpoint.com/int-keyword-in-java), [long](https://www.javatpoint.com/long-keyword-in-java), [float](https://www.javatpoint.com/float-keyword-in-java), [double](https://www.javatpoint.com/double-keyword-in-java), all [legacy classes](https://www.javatpoint.com/legacy-class-in-java), [Wrapper class](https://www.javatpoint.com/wrapper-class-in-java), [String class](https://www.javatpoint.com/methods-of-string-class), etc.

Once the immutable objects are created, their object values and state cannot be changed. Only Getters (get() method) are available not Setters (set() method) for immutable objects.

**Creating a mutable class:**

To create a mutable class in Java, the following requirements must be satisfied:

1. [Provide a method](https://www.edureka.co/blog/method-overloading-and-overriding-in-java/) to modify the field values.
2. Getter and Setter method

Consider the following code:

Text

Description automatically generated

**Creating an immutable class:**

To create an immutable class in Java, the following requirements must be satisfied:

1. A class should be declared as [final](https://www.edureka.co/blog/final-finally-and-finalize-in-java/) so that it can’t be extended.
2. All the fields should be made private so that direct access is not allowed.
3. No setter methods
4. Make all mutable fields final, so that they can be assigned only once.

Consider the following code:

Text

Description automatically generated

**Task 1:** **Run both codes given above for mutable and immutable class. Provide the outputs with brief analysis on it based on your understanding.**

**Risks of mutation:** Mutable types seem much more powerful than immutable types, but Immutable types are safer from bugs, easier to understand, and more ready for change. Mutability makes it harder to understand what your program is doing, and much harder to enforce contracts.

**Risky example: passing mutable values**

Simple method that sums the integers in a list:

/\*\* @return the sum of the numbers in the list \*/

public static int sum(List<Integer> list) {

int sum = 0;

for (int x : list)

sum += x;

return sum; }

We also need a method that sums the absolute values.

/\*\* @return the sum of the absolute values of the numbers in the list \*/

public static int sumAbsolute(List<Integer> list) {

for (int i = 0; i < list.size(); ++i)

list.set(i, Math.abs(list.get(i)));

return sum(list); }

Notice that this method does its job by mutating the list directly.

// meanwhile, somewhere else in the code...

public static void main(String[] args) {

// ...

List<Integer> myData = Arrays.asList(-5, -3, -2);

System.out.println(sumAbsolute(myData));

System.out.println(sum(myData)); }

* **Safe from bugs?** In this example, it’s easy to blame the implementer of sum­Absolute() for going beyond what its spec allowed. But really, **passing mutable objects around is a latent bug**. It’s just waiting for some programmer to inadvertently mutate that list, often with very good intentions like reuse or performance, but resulting in a bug that may be very hard to track down.
* **Easy to understand?** When reading main(), what would you assume about sum() and sum­Absolute()? Is it clearly visible to the reader that myData gets changed by one of them?

**Task 2: Implement a Java program to demonstrate the problems of risk given above.**

# Experiment # 11: Validating Code with Assertions

**Objective:** To practice applying Java assertions to enforce program correctness during development.  
**Time Required:** 3 hrs  
**Programming Language**: Java

**Software/Tools Required:** Windows OS, NetBeans /Eclipse Kepler

An **assertion** is a statement in Java which ensures the correctness of any assumptions which have been done in the program. When an assertion is executed, it is assumed to be true. If the assertion is false, the JVM will throw an Assertion error. It finds it application primarily for the testing purposes. Assertion statements are used along with Boolean expressions.

Assertions in Java can be done with the help of the assert keyword. There are two ways in which an assert statement can be used.

**First way:** assert expression;

**Second way:** assert expression1 : expression2;

**Enabling Assertions:** By default, assertions are disabled. The syntax for enabling assertion statement in Java source code is:

java –ea Test OR java –enableassertions Test

Here, Test is the file name.

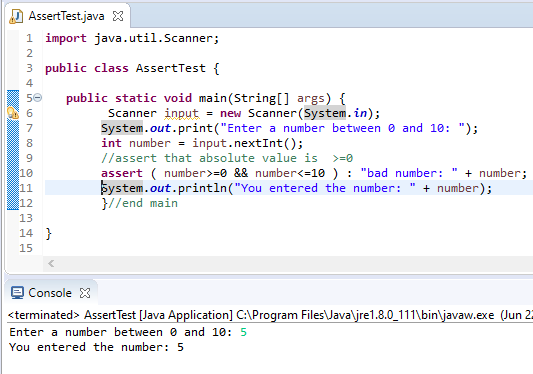
**Disabling Assertions:** The syntax for disabling assertions in java are:

java –da Test OR java –disableassertions Test

Here, Test is the file name.

**Task 1:** Run the code given below in which user is asked to enter number between 0 and 10. You must enter values in following ways and provide your output with brief analysis regarding assertion behaviour for each case:

1. Enter any number between 0 and 10.
2. Enter any number greater than 10 e.g., you can enter 50.



**Task 2:** Make a program that requires the user to enter his age to demonstrate the concept of assertion. The age must be greater than or equal to 18. If the user enters age less than 18, your program must throw an exception with the statement “Not valid”. Provide the output of both cases.

# Experiment # 12: Recursion and Iteration

**Objective:** To practice implementing recursive and iterative solutions for programming problems.  
**Time Required:** 3 hrs  
**Programming Language**: Java

**Software/Tools Required:** Windows OS, NetBeans /Eclipse Kepler

The Recursion and Iteration both repeatedly execute the set of instructions. **Recursion** is when a statement in a function calls itself repeatedly. The **iteration** is when a loop repeatedly executes until the controlling condition becomes false. The primary difference between recursion and iteration is that recursion is a process, always applied to a function and iteration is applied to the set of instructions which we want to get repeatedly executed.

**Task 1:** Write two programs using recursion and iteration mechanisms to print the Fibonacci series. The program should ask the user about the count of numbers he requires to have in Fibonacci series e.g., if he enters 10, your program should print the first 10 numbers in Fibonacci series starting from 0.

**Task 2:** Write two programs using recursion and iteration mechanisms to find the factorial of any number entered by user between 3 and 10.

**Task 3:** Write two programs using recursion and iteration mechanisms to print series of any number entered by user e.g., when the user enters 9, the following will be printed on the screen:

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

# Experiment # 13: Implementing Abstract Data Types

**Objective:** To practice creating and using ADTs such as stacks, queues, and lists using Java classes.  
**Time Required:** 3 hrs  
**Programming Language**: Java

**Software/Tools Required:** Windows OS, NetBeans /Eclipse Kepler

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of value and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. The behavior of each operation is determined by its inputs and outputs. It is called “abstract” because it gives an implementation-independent view.

**Autoboxing:**Converting a primitive value into an object of the corresponding [wrapper class](https://www.geeksforgeeks.org/wrapper-classes-java/) is called autoboxing. For example, converting int to [Integer class](https://www.geeksforgeeks.org/wrapper-classes-java/). The Java compiler applies autoboxing when a primitive value is:

* Passed as a parameter to a method that expects an object of the corresponding wrapper class.
* Assigned to a variable of the corresponding wrapper class.

**Example:**

**Graphical user interface, text, application

Description automatically generated**

**Unboxing:** Converting an object of a wrapper type to its corresponding primitive value is called unboxing. For example, conversion of [Integer](https://www.geeksforgeeks.org/wrapper-classes-java/) to int. The Java compiler applies unboxing when an object of a wrapper class is:

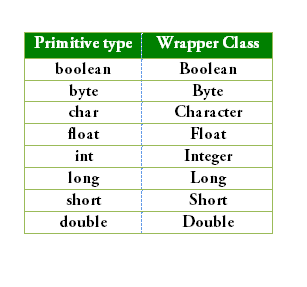
* Passed as a parameter to a method that expects a value of the corresponding primitive type.
* Assigned to a variable of the corresponding primitive type.

**Example:**

Graphical user interface, text, application

Description automatically generated

The following table lists the primitive types and their corresponding wrapper classes, which are used by the Java compiler for autoboxing and unboxing:



**Task 1:** Run the codes given above in two examples. What do you infer from the outputs?

**Task 2**: Write a Java program to demonstrate the concept of Autoboxing and Unboxing.

**Task 3**: Consider the following method which finds the sum of odd numbers in a list:

Text, letter

Description automatically generated

Write a complete Java program by using above method to demonstrate the concept of unboxing. The remainder (%) and unary plus (+=) operators do not apply to Integer objects. However, the Java compiler compiles the method without issuing any errors. Why?

# Experiment # 14: Designing Abstract Classes and Interfaces

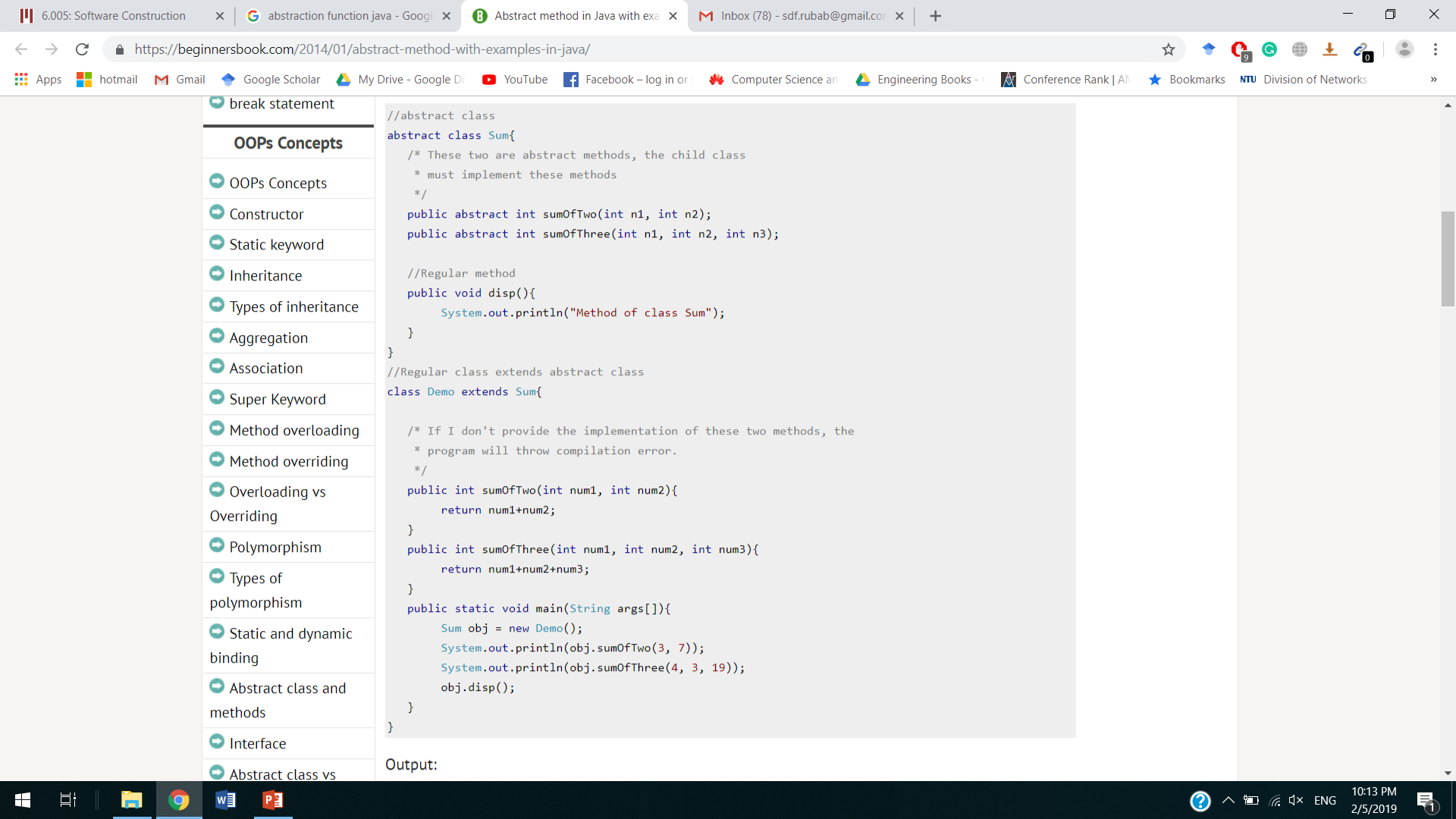
**Objective:** To practice abstraction and modularity using abstract classes and interfaces in software construction.  
**Time Required:** 3 hrs  
**Programming Language**: Java

**Software/Tools Required:** Windows OS, NetBeans /Eclipse Kepler

**Abstract Function:** A method without body (no implementation) is known as abstract method. A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well. This is how an abstract method looks in java:

public abstract int myMethod(int n1, int n2);

**Example:**



Output:

10

26

Method of class Sum

**Interface:** Interface looks like a class but it is not a class. An interface can have methods and variables just like the class but the methods declared in interface are by default abstract (only method signature, no body). Also, the variables declared in an interface are public, static & final by default.Since methods in interfaces do not have body, they have to be implemented by the class before you can access them. The class that implements interface must implement all the methods of that interface. Also, java programming language does not allow you to extend more than one class, However you can implement more than one interfaces in your class. The syntax for interfaces is as follows:

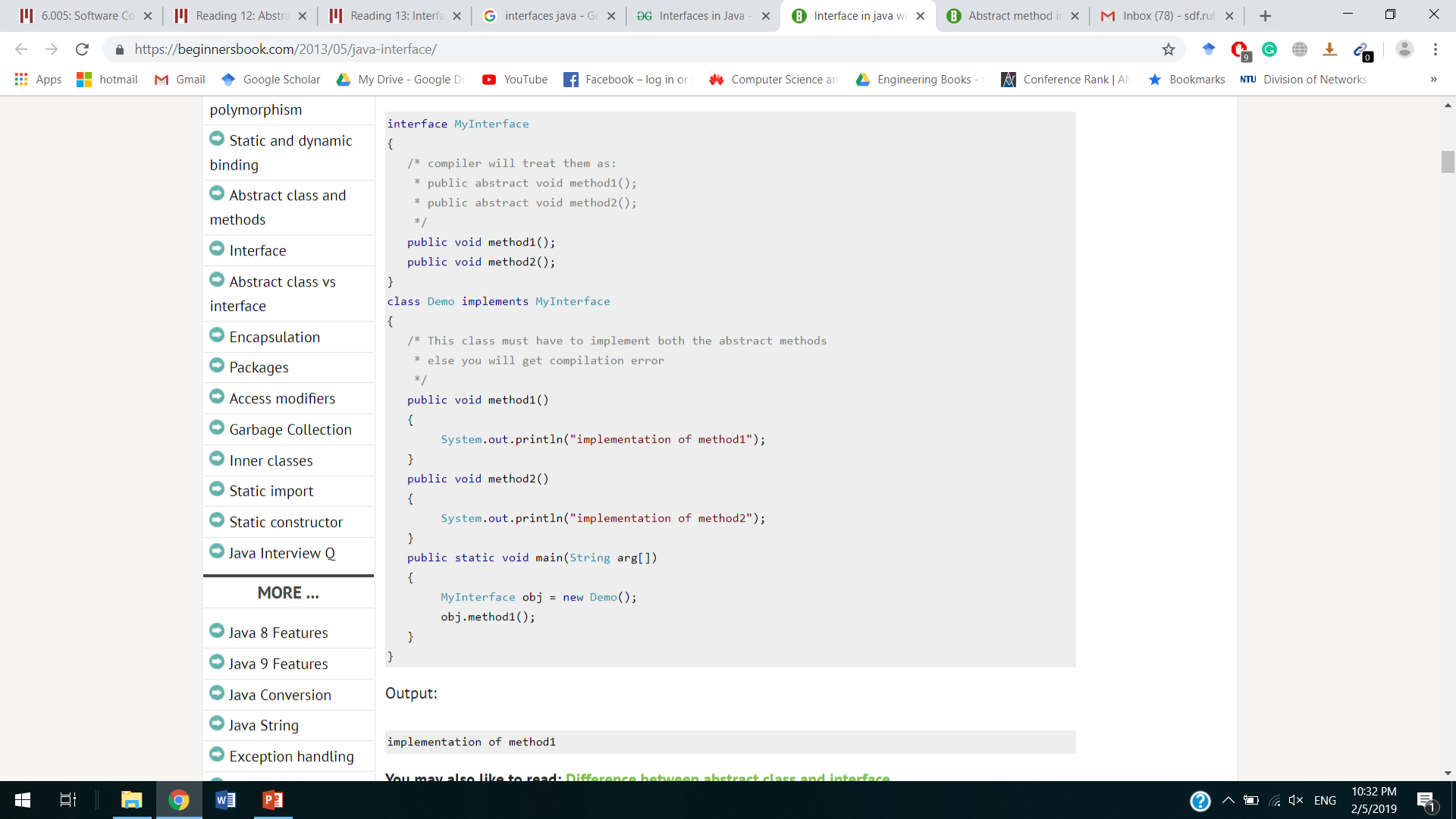
interface MyInterface {

/\* All the methods are public abstract by default

\* As you see they have no body \*/

public void method1();

public void method2(); }

**Example:**

Output:

implementation of method1

**Task 1:** Implement the Shape hierarchy shown in the figure below:



Each TwoDimensionalShape should contain method getArea to calculate the area of the two-dimensional shape. Each ThreeDimensionalShape should have methods getArea and getVolume to calculate the surface area and volume, respectively, of the three-dimensional shape. Create a program that uses an array of Shape references to objects of each concrete class in the hierarchy. The program should print a text description of the object to which each array element refers. Also, in the loop that processes all the shapes in the array, determine whether each shape is a TwoDimensionalShape or a ThreeDimensionalShape. If it is a TwoDimensionalShape, display its area. If it is a ThreeDimensionalShape, display its area and volume.

|  |
| --- |
| * + Targets WA4 (Investigations) / WA5 (Modern Tool Usage)   CLO 2 is mapped to PLO 2 (WA4) and CLO 4 is mapped to PLO 5 (WA5)   * + Lab Experiment Objective without giving any details about the conduct procedure, list of equipment, tables for recording values, etc.     1. The lab task can be achieved in multiple manners        - Multiple manners for finding best policy     2. The output of the lab is not fixed        - The reward mapping effects the result        - Selection different discount rates effects the policy. Discount rates are to selected by students as per understanding of the problem . |

# Experiment # 15: Open-Ended Lab

**Developing a TODO List Application in Java**

This open-ended lab is not a guided lab, and students are required to fulfill all the requirements on their own.

**Objective:**

Develop a simple TODO list desktop application (like Microsoft Notes/Google Keep) in Java using object-oriented programming concepts, such as classes, objects, inheritance, and polymorphism.

**Tools and Technologies:**

Java, Integrated Development Environment (IDE), such as Eclipse or IntelliJ, JavaFX or Swing for GUI, Git for version control.

**Tasks:**

1. Define the requirements and features of the TODO list application, including the ability to perform CRUD (i.e. Create/add, Read/open, Update/edit, Delete/remove) and prioritize tasks. **(CLO 2)**
2. Plan and design the application architecture, including the classes, methods, and data structures needed to implement the functionality. **(CLO 2)**
3. Use an Integrated Development Environment (IDE), such as Eclipse or IntelliJ, to create a new Java project and set up the necessary dependencies, such as JavaFX or Swing for GUI. Implement the TODO list functionality by creating classes for tasks, categories, and the main application, and writing the necessary methods to add, edit, delete, and prioritize tasks. **(CLO 4)**
4. Create a graphical user interface (GUI) for the application using JavaFX or Swing, including buttons, text fields, and lists for displaying the tasks and categories. **(CLO 4)**
5. Use Git for version control to track changes to the code and collaborate with others on the project. **(CLO 4)**

**Deliverables:**

1. Report CLO 2 (15 Marks)

2. Lab Demonstration as per lab rubrics CLO 4 (15 Marks)