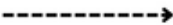







<p>Singleton Class</p> <pre> class Logger { private List<String> logs; private static Logger instance; private Logger() {this.logs = new ArrayList<String>();} public static Logger getInstance() { if (instance == null) {instance = new Logger();} return instance;} public void log(String message) {this.logs.add(message);} public ArrayList<String> getLogs(){...}} </pre> <p>Composite Pattern</p> <pre> private double volume = 0.0; public double getCabinVolume() { for (OfficeSpace o : childSpaces) { if (o instanceof Cabin) { Cabin child = (Cabin) o; volume += child.getCabinVolume();} else { CompositeSpace cs = (CompositeSpace) o; volume += cs.getCabinVolume(); } } return this.volume; } </pre> <p>Exceptions</p> <p>ArrayIndexOutOfBoundsException.</p> <p>ClassCastException</p> <p>NumberFormatException Integer.parseInt() invalid parsable format</p> <p>ArithmeticException division by zero</p> <p>IllegalArgumentException inappropriate argument is passed to a method</p> <p>NoSuchElementException element is requested but not found</p> <p>.next()</p> <p>Checking if String is numeric / letter</p> <p>Character.isDigit(string.charAt(index))</p> <p>Character.isLetter(string.charAt(index))</p> <p>Template Pattern</p> <pre> abstract class ...{ protected String ...= ""; public final void ...(String line) { readInput(line); operate(); writeOutput(); } abstract void readInput(String line); abstract void operate(); private void writeOutput() {...}} </pre> <p>Linked List Iterator</p> <pre> private class LinkedListIterator implements Iterator<ListNode> { private ListNode current; public LinkedListIterator() { this.current = head; } public boolean hasNext() { return current != null; } public ListNode next() { if (!hasNext()) { throw new java.util.NoSuchElementException(); } ListNode temp = current; current = current.getNext(); return temp; } } </pre>	<p>Observable</p> <pre> public void update(Observable o, Object arg) { if (o instanceof ParkingSpace) { ParkingSpace space = (ParkingSpace) o; System.out.println(String.format("The current status for Parking Slot %s is true", space.getID())); if (space.getStatus()) { if (_parkingTypeNeeded == ((ParkingSlot) space).getType()) { double charge = Math.round(space.getRent() * _hours); System.out.println(String.format("Parking Slot %s is available to customer %s for \$%.1f", space.getID(), _name, charge)); } else { System.out.println(String.format("Parking Slot %s is not available to customer %s", space.getID(), _name));}} } } </pre> <p>Swingworker <Long, Void></p> <pre> private int _n; private int _r; public PermutationsWorker() {} protected Long doInBackground() { Long x = factorial(_n); Long z = factorial(_n - _r); return x / z;} public Long factorial(int n){ Long total = 1L; for (int i=1; i<=n; i++){ total *= i; } return total;} protected void done() { try { Long result = get();setText(...); } catch (...) {... } } } </pre> <p>Course[1..5] → Student[0..10]</p> <pre> class Student { private static final int MAX.. = 5; private List<Course> enrolledCourses; private String name; public Student(String name, Course course) { this.name = name; this.enrolledCourses = new ArrayList<>(); if (!course.enrollStudent(this)) {throw new ... } enrolledCourses.add(course); } public boolean enrollCourse(Course course) { if (enrolledCourses.size() >= MAX !course.enrollStudent(this)) {return false; } enrolledCourses.add(course); return true; }} class Course { private static final int MAX.. = 10; private List<Student> students; private String title; public Course(String title) { students = new ArrayList<Student>(); this.title = title; } boolean enrollStudent(Student student) { if (students.size() < MAX...) { students.add(student); return true; } return false; }} </pre>	<p>File Reading</p> <pre> try { File myObj = new File("filename.txt"); Scanner myReader = new Scanner(myObj); while (myReader.hasNextLine()) { String data = myReader.nextLine(); System.out.println(data); }myReader.close(); } catch (FileNotFoundException e) { System.out.println(...); e.printStackTrace(); } </pre> <p>Actor[0..3] → Student[0..*]</p> <pre> class Actor { private List<Movie> movies = new ArrayList<Movie>(); String name; public Actor(String name) {this.name = name;} public boolean signUpFor(Movie movie) { if (movie.getActors().contains(this) && !movies.contains(movie)) { movies.add(movie); return true; }return false; }} class Movie { private static final int MAX_ACTORS = 3; private List<Actor> actors = new ArrayList<Actor>(); private String name; public Movie(String name) {this.name = name;} public boolean signUp(Actor actor) { if (actors.size() < MAX_ACTORS && !actors.contains(actor)) { actors.add(actor); return true; } return false; }} </pre> <p>Iterator</p> <pre> private class AscendingIterator implements Iterator<Integer> { private int index = 0; // current index in the iterator private Integer nextValue = getNextValue(); // get the first ascending value private Integer getNextValue() { while (index < numbers.size()) { Integer value = numbers.get(index); index++; if (value != null && (index == 1 value > nextValue)) { return value; } } return null; // no more ascending values } public boolean hasNext() { return nextValue != null; } public Integer next() { if (!hasNext()) { throw new java.util.NoSuchElementException(); } Integer returnValue = nextValue; // store the current value nextValue = getNextValue(); // prepare the next ascending value return returnValue; // return the current value } } </pre>
---	--	--

Association Types			
Dependency	Definition: A temporary relationship where one class depends on another to function (typically for a short time). Example: A Car needs Fuel to drive. The Car does not own the Fuel, but it needs it.		Thread Life Cycles Thread.sleep() or Thread.wait(long timeout) = Time waiting thread.start() = runnable Thread.currentThread().interrupt() = Terminated Thread t = new Thread: New born synchronized (lock): Blocked Robustness Software is robust if it can handle unusual conditions such as corrupt data, user error, programmer error, and environmental conditions. <ul style="list-style-type: none"> - Use of polymorphism - Use of protected fields - Use of type-safety: List<Staff> = ArrayList<Staff> - Exception handling - Interfaces - Access Modifiers / Encapsulation
Association	Definition: A basic "has-a" relationship between classes where one class uses or references another. Example: A Driver uses a Car. The Car may exist without the Driver.		
Generalization	Definition: Represents an "is-a" relationship, typically used in inheritance. All uses of inheritance where the superclass is a concrete class Example: A Dog is an Animal. Here, Dog inherits from Animal.		
Realization	All uses of inheritance where the superclass is an abstract class or interface. One model element realizes (implements or executes) the behavior that the other model element (the supplier) specifies		
Aggregation	Definition: A "has-a" relationship where the contained object (part) can exist independently of the container (whole). Example: A School has Students. If the School is closed, Students still exist.		
Composition	Definition: A stronger "has-a" relationship where the contained object cannot exist independently of the container. When the container is destroyed, the contained object is too. Example: A House has Rooms. If the House is destroyed, the Rooms are destroyed too.		

SOLID Design Principles

Single Responsibility Principle <pre> class Stationery{ class print{ class scan{ class copy{ class Stationery{ public void print() {} public void scan() {} public void copy() {} </pre>	Open-Closed Principle <pre> interface Shape {double area();} class Circle implements Shape {public double area() {} } class Square implements Shape {public double area() {} } class AreaCalculator {} class Shape { public double area(String shapeType) }} </pre> Software entities (like classes, modules, and functions) should be open for extension but closed for modification. This means we should be able to add new functionality to a class without altering its existing implementation	Liskov Substitution Principle <pre> class Bird {public void move(){} } class Penguin extends Bird{public void move(){} } class Sparrow extends Bird{public void move(){} } class Bird {public void fly() {} } class Penguin extends Bird{public void fly() {} } </pre> objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program. In other words, subclasses should behave in a way that is consistent with the expectations set by their superclass	Interface Segregation Principle <pre> interface Eater{ void eat();} interface Flyer {void fly();} class Dog implements Eater {public void eat() {} } class Bird implements Eater, Flyer {public void eat(){} public void fly(){} } interface Animal{void eat(); void fly();} class Dog implements Animal { public void eat() {} public void fly() {} } </pre>	Dependency Inversion Principle <pre> class Order{ PaymentMethod payment; } abstract class PaymentMethod{ class CreditCard extends PaymentMethod{ class DebitCard extends PaymentMethod{ class Order{ CreditCard debit; CreditCard credit; } class CreditCard {} class DebitCard {} </pre>
--	--	---	---	---

Composite Design Pattern: Implementing the composite pattern lets clients treat individual objects and compositions uniformly. **Template-Method Design Pattern:** defines the program skeleton of an algorithm in an operation, deferring some steps to subclasses. This allows subclasses to redefine certain steps of the algorithm without altering its overall structure. The concrete class implements the primitive operations to carry out subclass-specific steps of the algorithm. It's fairly easy to create concrete implementations of an algorithm because you're removing common parts of the problem domain by the use of an abstract class. Clean code because you avoid duplicate code and we separate the algorithm into private methods/functions. **Observer Pattern Design:** The observer pattern facilitates a one-to-many relationship between objects, ensuring that when one object changes, its dependent objects are notified automatically. Subject - represents the core abstraction. Observer represents the variable (or dependent or optional or user interface) abstraction. Observer - represents the variable abstraction. Each Observer can call back to the Subject as needed. **Factory Method Design Pattern:** Factory Method is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special factory method. Don't worry: the objects are still created via the new operator, but it's being called from within the factory method. Objects returned by a factory method are often referred to as products.**Realization vs Generalization:** All uses of inheritance where the superclass is a concrete class, the relationship between the superclass and subclass is known as **generalization**. All uses of inheritance where the superclass is an abstract class or interface, the relationship between the superclass and subclass is known as **realization**.

DefaultTreeModel Stack Example

```

public void buildTree(String expression) {
    Stack<DefaultMutableTreeNode> stack = new Stack<>();
    for (char ch : expression.toCharArray()) {
        if (Character.isDigit(ch)) {
            DefaultMutableTreeNode node = new DefaultMutableTreeNode(ch);stack.push(node); } else if (ch == '+' || ch == '-' || ch == '*') {
            DefaultMutableTreeNode rightNode = stack.pop(); // Right child
            DefaultMutableTreeNode leftNode = stack.pop(); // Left child
            DefaultMutableTreeNode operatorNode = new DefaultMutableTreeNode(ch);
            operatorNode.add(leftNode);
            operatorNode.add(rightNode);
            stack.push(operatorNode); } }    root = stack.pop();}

```