|  |
| --- |
| **1. Inheritance Definition**: Allows a subclass to inherit fields and methods from a superclass, establishing an "is-a" relationship. **Syntax**: class Subclass extends Superclass { ... } **Key Points**: - **Single inheritance only** in Java (one superclass). - **Protected** members can be accessed by subclasses, **private** members cannot. - **super()**: Calls the superclass’s constructor. - **Method Overriding**: Subclass provides specific implementations for superclass methods (using @Override annotation). |

|  |
| --- |
| **4. Exceptions Definition**: Objects that represent runtime errors, disrupting normal flow. They are handled to make applications robust. **Types**:  **- Checked**: Must be handled or declared in method (IOException).  **- Unchecked (Runtime)**: Do not require handling (NullPointerException).  **- Errors**: Serious issues usually not handled (OutOfMemoryError).  **try-catch-finally**: Example java Code try {      // Code that might throw an exception  } catch (ExceptionType e) {      // Handle exception  } finally {      // Code that always runs  }  **throw**: Explicitly throw an exception (throw new IllegalArgumentException("error");).  **throws**: Declares exceptions a method can throw  (void myMethod() throws IOException).  **Custom Exceptions**: Extend Exception or RuntimeException to create specific exceptions. Example Java Code class MyException extends Exception { } |

|  |
| --- |
| **2. Polymorphism Definition**: Allows objects of different subclasses to be treated as instances of a superclass, enabling one interface, many implementations. **Dynamic Binding**: Determines which method to execute at runtime based on the object’s actual type.  **Example Java code**: Animal myAnimal = new Dog();  myAnimal.sound(); // Calls Dog's overridden method  **Upcasting**: Casting a subclass object to a superclass type (implicit). **Downcasting**: Casting a superclass reference back to a subclass type (explicit, requires instanceof for safety).  Example Java Code if (myAnimal instanceof Dog) {     Dog myDog = (Dog) myAnimal;} |

|  |
| --- |
| **3. Interfaces Definition**: A contract that defines a set of methods that a class must implement. **Syntax**: interface InterfaceName { void methodName(); } **Key Points**: - Methods in interfaces are implicitly **public** and **abstract**. - A class can implement multiple interfaces  (e.g., class Dog implements Animal, Moveable). - **Default Methods** (Java 8+): Allows methods in interfaces with an implementation  (default void methodName() { }). - **Static Methods**: Can be defined in interfaces. **Use Case**: Defines common behavior across unrelated classes (e.g., List and Set both implement Collection). |

|  |  |  |
| --- | --- | --- |
| **5. Collections Framework List - Characteristics**: Ordered, allows duplicates. - **Implementations**: **--- ArrayList**: Fast access, slower insert/delete in the middle. --- **LinkedList**: Fast insert/delete, slower access by index. - **Methods**: add(element), get(index), remove(index), size() - **Example Java Code**: List<String> list = new ArrayList<>(); list.add("Apple"); | **Queue - Characteristics**: FIFO order (usually), can be priority-based. **- Implementations**: --- **LinkedList**: Implements Queue for FIFO. --- **PriorityQueue**: Orders elements based on priority. - **Methods**: offer(element), poll(), peek()  **Example Java Code**:  Queue<String> queue = new LinkedList<>();  queue.offer("Task1"); | **Iterating Over Collections**  **for-each Loop (Ex code)**: for (String item : list) {      System.out.println(item);  }  **Iterator (Ex code)**: Iterator<String> it = list.iterator();  while (it.hasNext()) {      System.out.println(it.next());  } |
| **Set - Characteristics**: Unordered, no duplicates. **- Implementations**: **--- HashSet**: No order, fast operations. **--- LinkedHashSet**: Maintains insertion order. **---TreeSet**: Sorted order. **- Methods**: add(element), contains(element), remove(element), size()  **Example Java Code:**  Set<Integer> set = new HashSet<>();  set.add(1); | **Map Characteristics**: Key-value pairs, unique keys. **Implementations**: **- HashMap**: Unordered, fast. **- LinkedHashMap**: Maintains insertion order. **- TreeMap**: Sorted by keys. **Methods**: put(key, value), get(key), remove(key), containsKey(key) **Example Java Code**: Map<String, Integer> map = new HashMap<>();  map.put("Alice", 30); | **Utility Classes Collections** (Utility Class): Provides static methods for collection manipulation.  - Collections.sort(list)  - Collections.reverse(list)  - Collections.max(list)  **Arrays** (Utility Class): Provides methods for array manipulation.  - Arrays.asList(array) – Converts array to List.  - Arrays.sort(array) |

A screenshot of a computer program

Description automatically generatedA screenshot of a computer code

Description automatically generated

**Understanding / Remembering Concepts:**   
- Interfaces: You can have infinite amount of interfaces and they can all infinitely interact with eachother  
- Exceptions: An **UNCHECKED** exception is a RuntimeException. **Un**checked == r**UN**timeExceptions. Should be caught, could be declared, recoverable errors. Errors are also here though a separate thing (not caught, can declare, irrecoverable).   
A **CHECKED** exception is all else (IO. SQL, etc). You must handle these. Must be caught, must be declared, and are recoverable.

**Design Patterns Hail Mary:**  
**Factory Method Pattern:** Centralizes object creation for flexibility and decouples client code from specifics. Use AccountFactory.createAccount("Checking"). **Adapter Pattern** Enables incompatible interfaces to work together. Ideal for legacy systems or third-party libraries without modifying client or server code. **Proxy Pattern** Controls access and adds behavior like security before requests. Use BankProxy.updateAccount(transaction) to add control while keeping client code the same. **Builder Pattern** Simplifies complex object creation by allowing separate field setting. Example: new Student.Builder("John", "Doe", "E12345").age(22).build(). **Singleton Pattern** Ensures a single instance of a class globally, useful for resources like database connections. Access with Singleton.getInstance(). **Facade Pattern**  
Provides a simplified interface to complex systems, useful in APIs. Helps clients interact with multiple subsystems through a single interface. **Strategy Pattern** Encapsulates algorithms in separate classes for runtime flexibility, ideal for interchangeable strategies (e.g., sorting, payment methods). **Model-View-Controller (MVC)** Separates data, UI, and control logic. Controller updates Model ➔ Model notifies View ➔ View updates Controller, enhancing maintainability and scalability.