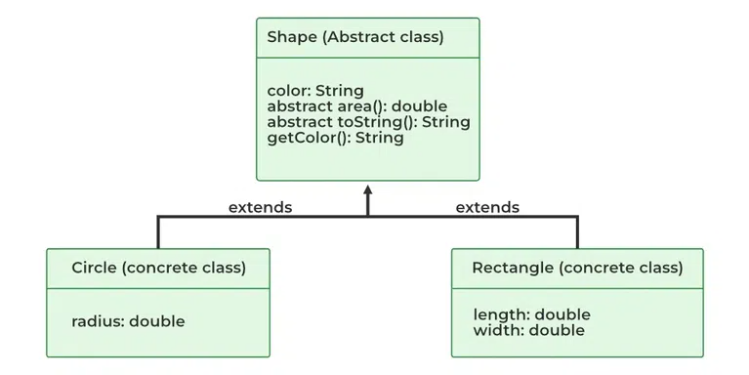
**Technical Documentation Module 3**

|  |
| --- |
| [M03 L014 1](#_Toc1159570335)  [M03 L015 1](#_Toc659120842)  [M03 L016 2](#_Toc1968344842)  [M03 L017 3](#_Toc750423324)  [M03 L018 3](#_Toc910848080)  [M03 L019 3](#_Toc164275897)  [M03 L020 3](#_Toc288555571) |

## **M03 L014**

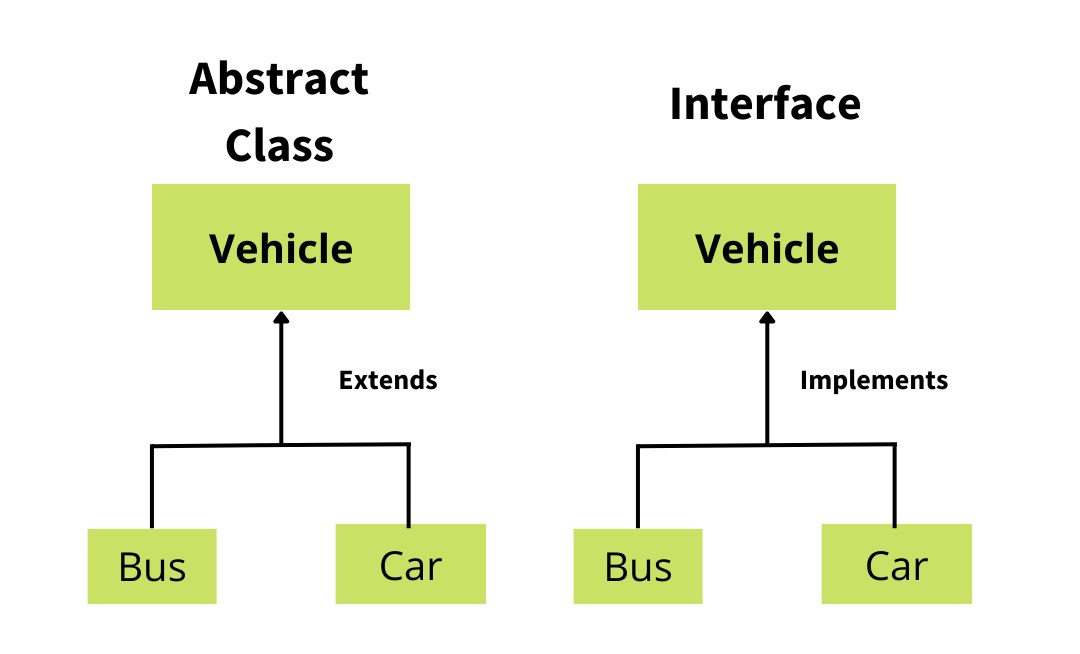
* **Abstract Classes**
  + Similar to regular classes, but for more general concepts for classes with the only differences is that it cannot be Instantiated
    - it must be extended for other classes to inherit if you are using (Inheritance/Polymorphism)
    - It is unable to create an instance object due to its nature and vagueness
    - Uses abstract keyword, and can have both concrete and or Abstract methods
* **Abstract Methods**
  + A type of method unique to an Abstract class that is not implementing anything, is declaring something
    - It can only be used with override in the subclasses
    - It has no body and to use it you need to *Override* in the subclass
    - Uses the abstract keyword and cannot use the final or static keywords

***Example***



* **Interface**
  + A type of class that would only contain abstract classes which helps in what it should do and how it should be implemented
    - This allows the code to be flexible since it can be used with other abstract classes due to its vagueness
    - Classes implementing an interface needs to provide implementations for all methods in its protocol (contract)
    - Can’t use instance variables only static variables

***Example***

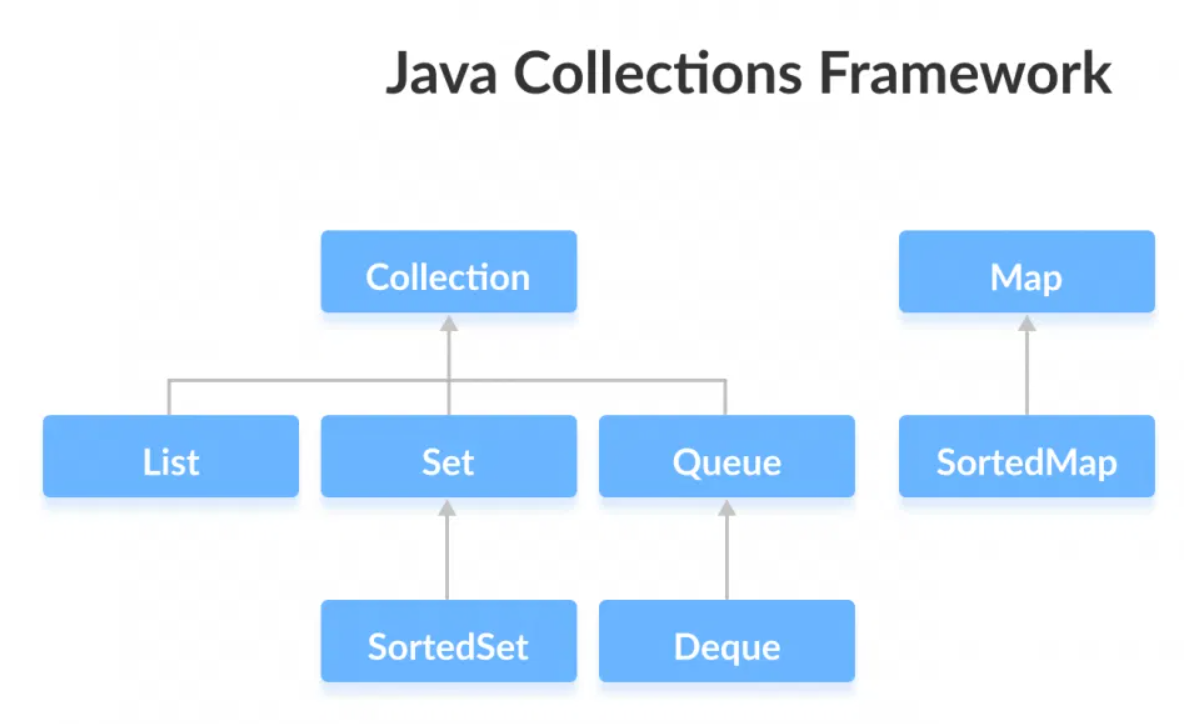


|  |
| --- |
| public static void main(String[] args)  {  Bulldog myDog = new Bulldog("Max", 30.5, "Dog Food", "Backyard");  myDog.eat();  myDog.beFriendly();  myDog.play();  }  }  //Interface defining behaviors for a Pet,  interface Pet {  void beFriendly();  void play();  }  //Abstract Class  abstract class Animal {  //concrete encapsulation  private String name;  private double weight;  private String food;  private String location;  // Constructor  public Animal(String name, double weight, String food, String location) {  this.name = name;  this.weight = weight;  this.food = food;  this.location = location;  }  // Abstract method - must be implemented by subclasses  public abstract void eat();  // Getters  public String getName() { return name; }  public double getWeight() { return weight; }  public String getFood() { return food; }  public String getLocation() { return location; }  }  //Abstract class Canine extending Animal  abstract class Canine extends Animal {  //Constructor  public Canine(String name, double weight, String food, String location) {  super(name, weight, food, location);  }  }  //Concrete class Bulldog extending Canine and implementing Pet  class Bulldog extends Canine implements Pet {  //Constructor  public Bulldog(String name, double weight, String food, String location) {  super(name, weight, food, location);  }  // To use the Abstract methods, you use the Override  *@Override*  public void eat() {  System.***out***.println(getName() + " is eating " + getFood() + ".");  }  // to utilize the Interface, you need to use the implement keyword on the subclass and in that subclass, you must use Override to utilize the methods within  *@Override*  public void beFriendly() {  System.***out***.println(getName() + " is wagging its tail happily!");  }  *@Override*  public void play() {  System.***out***.println(getName() + " is playing with a ball.");  }  } |

## **M03 L015**

* **Data Structure**
  + A way to define how data should be stored, organized and manipulated in a program allowing for a more efficient program, a fundamental in Java
    - Some examples of a data structure are Arrays, Stacks, or Tree’s
* **Collections**
  + A data structure that is used to store and manage many elements and interfaces efficiently
    - It's an abstract data type
    - Tip: All these collections are contained in Java’s Collection Framework (JCF)

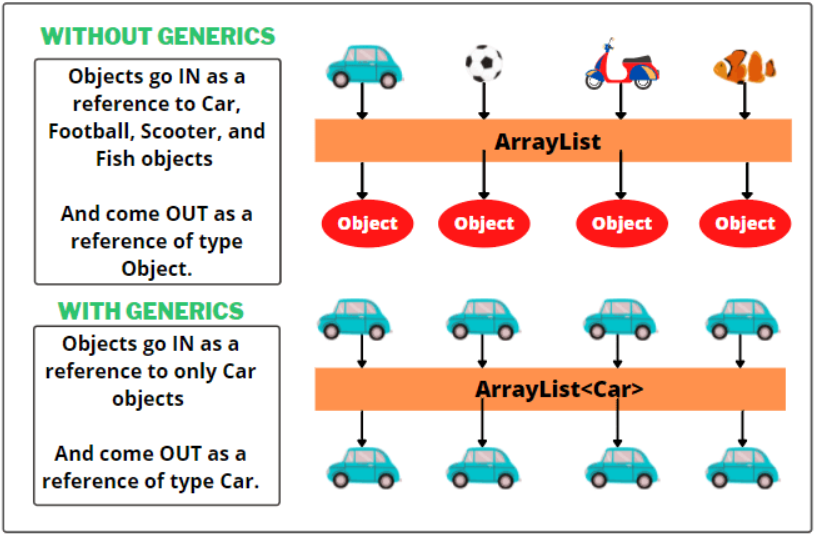
***Example***



* **Generic**
  + A class or method that the compiler can replace with concrete types to allow for more control, reusability, and efficiency in code using <, >.
    - **Generic Type-** A parameterized data type that fills in a Generic class or method with a placeholder value
    - **Concrete Type-** You're filling in a Generic class or method with an actual value like using an integer or string
      * It can use all datatypes because it only stores the address to the reference variable

|  |
| --- |
| public class Main {  public static void main(String[] args) {  // Generics = A concept where you can write a class, interface, or method  // that is compatible with different data types.  // <T> type parameter (placeholder that gets replaced with a real type)  // <String> type argument (specifies the type)  Box<String> box = new Box<>();  box.setItem("banana");  System.***out***.println(box.getItem());  Product<String, Double> product1 = new Product<>("apple", 0.50);  Product<String, Integer> product2 = new Product<>("ticket", 15);  }  }  public class Box<T> {  T item;  public void setItem(T item) {  this.item = item;  }  public T getItem() {  return item;  }  }  public class Product<T, U> {  T item;  U price;  public Product(T item, U price) {  this.item = item;  this.price = price;  }  public T getItem() {  return this.item;  }  public U getPrice() {  return this.price;  }  } |

***Example***



* + **ArrayList-** A Dynamic generic class that is used to store objects like an array without the issue of its size, stored in consecutive memory,
    - ***(Extra Info)-*** it's stored in static memory even though it's dynamic. It creates a new array every time the size changes and moves the data to the new array as it deletes the old array from memory.

***(ArrayList Example w/ Collections.sort & using Java’s ArrayList)***

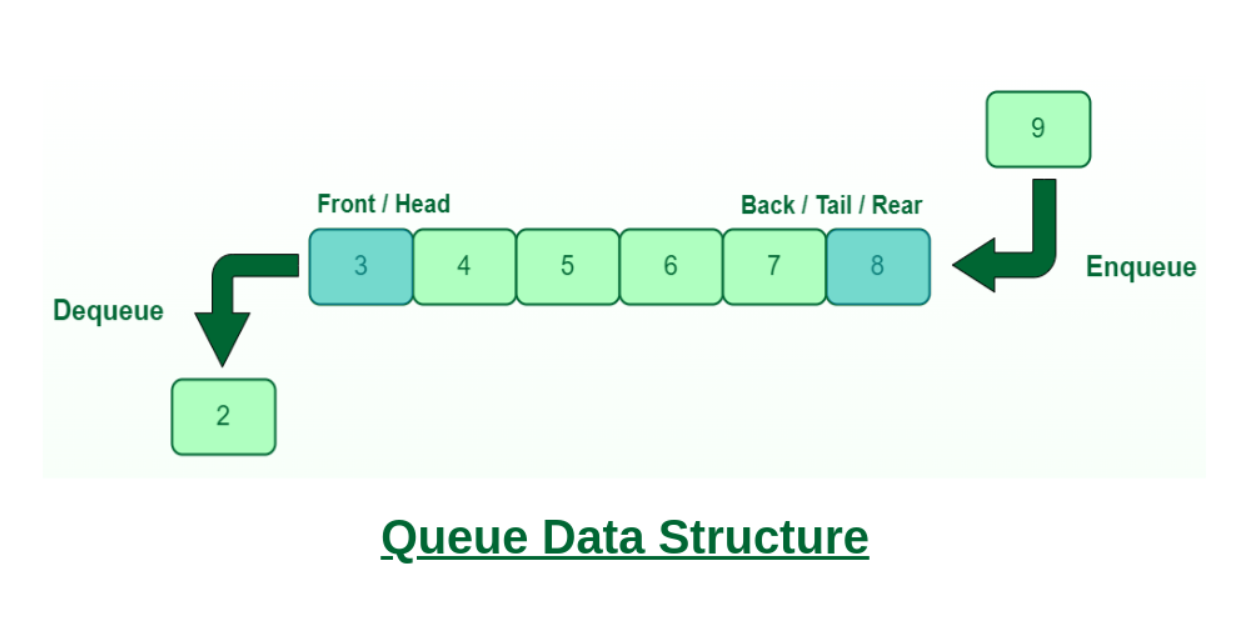
|  |
| --- |
| import java.util.ArrayList;  import java.util.Collections;  public class Main {  public static void main(String[] args) {  // ArrayList = A resizeable array that stores objects (autoboxing)  // Arrays are fixed in size, but ArrayLists can change  ArrayList<String> fruits = new ArrayList<>();  fruits.add("Apple");  fruits.add("Orange");  fruits.add("Banana");  fruits.add("Coconut");  //fruits.remove(0);, removes value stored in that index  //fruits.set(0, "Pineapple");, replaces what is stored in the index with something else  //*object.method*  Collections.*sort*(fruits);  // this is a for-each loop  for(String fruit : fruits){  System.***out***.println(fruit);  }  }  } |

* + **For-Each Loop-** Like a For-loop but made to iterate through an Arraylist, arrays and its collections

## **M03 L016**

* **Upcasting**
  + A type of implicit casting that uses inheritance, when a subclass object gets stored in a superclass reference
    - This can be used to store objects in a single ArrayList instead of creating multiple ArrayList
    - This allows us to take advantage of Polymorphism
* **Downcasting**
  + A type of explicit casting that uses inheritance, when a superclass reference gets stored in a subclass reference
    - It's using the specifics of a subclass but with a superclass reference
    - This allows us to take advantage of Polymorphism
* **Data Structures (*Extras*)**
  + - **Linear Data structures-** Like a data structure but with the added difference of sorting elements in a linear/sequential way.
* **Queues**
  + A type of collection & data structure that has nodes enter the tail and exit from the head and is a specialized linked list
    - **Enqueue-** Adds a node to the tail of the queue
    - **Dequeue-**removes a node from the head of the queue
      * First In First Out (FIFO)
      * Can't create an instance of a queue

***Example***



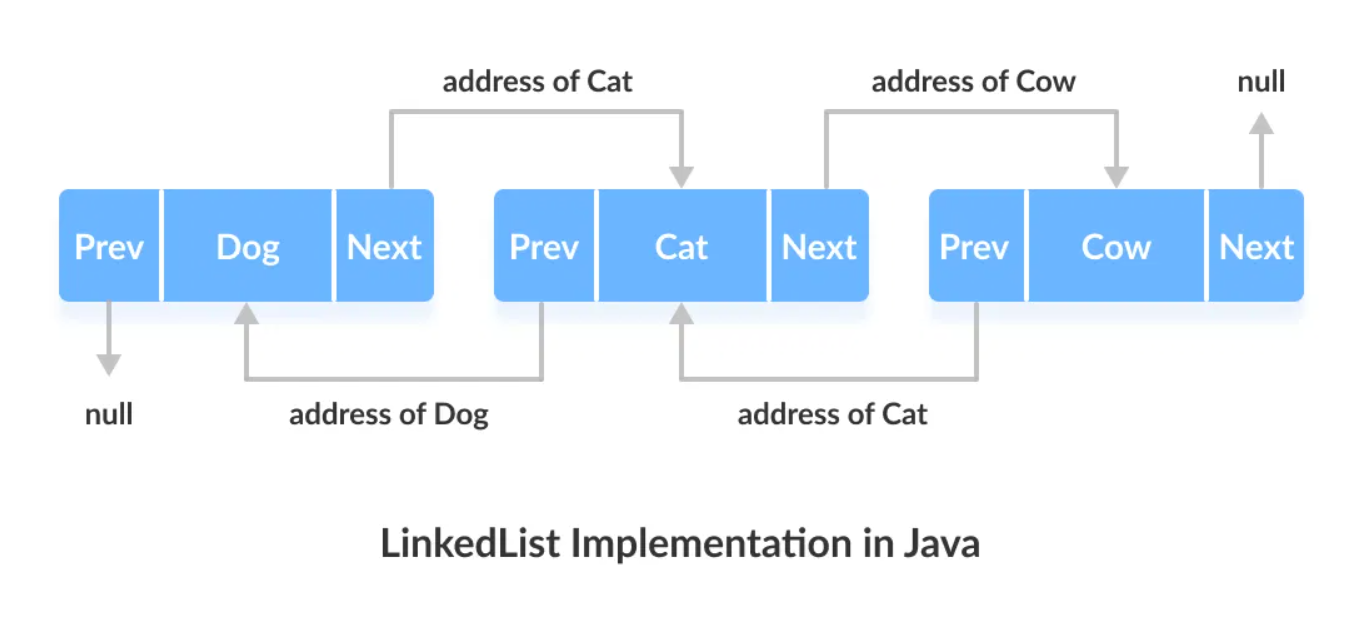
***Code Example***

|  |
| --- |
| public class Main{  public static void main(String[] args) {  // Queue = FIFO data structure. First-In First-Out  // A collection designed for holding elements prior to processing  // Linear data structure  // enqueue() = add, offer()  // dequeue() = remove, poll()  // Where are queues useful?  // 1. Keyboard Buffer (letters should appear on the screen in the order  // they're pressed)  // 2. Printer Queue (Print jobs should be completed in order)  // 3. Used in LinkedLists, PriorityQueues, Breadth-first search  Queue<String> queue = new LinkedList<String>();  queue.offer("Karen");  queue.offer("Chad");  queue.offer("Steve");  queue.offer("Harold");  //System.out.println(queue.isEmpty()); its empty  //System.out.println(queue.size()); the size of the queue  //System.out.println(queue.contains("Harold")); what the queue contains  //System.out.println(queue.peek());  //queue.poll();  //queue.poll();  //queue.poll();  //queue.poll();  System.***out***.println(queue);  }  } |

## **M03 L017**

* **Linked Lists**
  + A dynamic data structure consisting of nodes, where each node contains data and a pointer to the next node in the sequence, its Dynamic memory and uses <, >.
    - **Nodes-** the main object used in containing values and a pointer (reference)
    - **Head-** A pointer to the first node in the linked list serving as the start
    - The pointer is used to know what is inside each node
* **Self-Referential Class/Structure**
  + When a class member is a reference (pointer) to its same class data type
    - The next node is making a reference to itself or the next node

***Example***



***Code Example***

|  |
| --- |
| public class FixSinglyLinkedOrderedList  {  // Test the Singly Linked List  public static void main(String[] args)  {  SinglyLinkedListFix list = new SinglyLinkedListFix();  list.insertNode(4);  list.insertNode(2);  list.insertNode(8);  list.insertNode(3);  list.printList();  list.deleteNode(0);  list.deleteNode(3);  list.printList();  }  }  /\*\*  \* this.next = null; is a self-Referencial  \*/  class NodeFix {  int data;  NodeFix next;  public NodeFix(int data) {  this.data = data; // reference to where it is  this.next = null;  }  }  class SinglyLinkedListFix {  NodeFix head;  /\*\*  \* the issue is that in the else-statement the newNode connects to the previous node  \* but forgot to add a lick to the current node so I implemented  \* newNode.next = current; to fix it.  \*/  public void insertNode(int number) {  NodeFix newNode = new NodeFix(number);  NodeFix current = head;  NodeFix previous = null;  while (current != null && current.data < number) {  previous = current;  current = current.next;  }  if (previous == null) {  newNode.next = head;  head = newNode;  } else {  previous.next = newNode;  newNode.next = current;  }  }  /\*\*  \* The issues that I noted is that in the while loop on line 82  \*/  public void deleteNode(int number) {  NodeFix current = head;  NodeFix previous = null;  while (current.next != null && current.data != number) { // Bug #3: Incorrect loop termination  previous = current;  current = current.next;  }  if (previous == null) {  head = current.next;  } else {  previous.next = null; // Bug #5: Should be previous.next = current.next  }  }  public void printList() {  NodeFix current = head;  while (current != null) {  System.***out***.print(current.data + " → ");  current = current.next;  }  System.***out***.println("null");  }  } |

## **M03 L018**

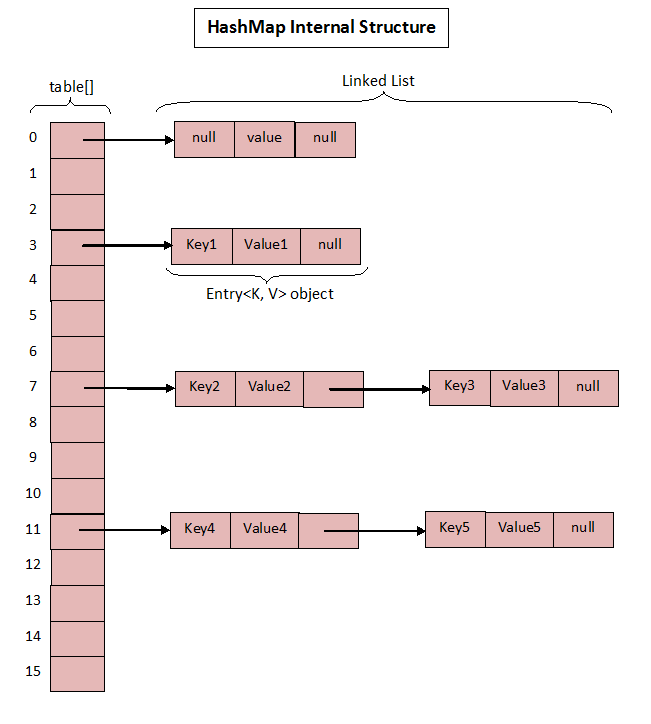
* **Double-Linked List**
  + Like how a normal linked list works, but with the benefit of being able to go backwards and forwards
* **Circular-Linked List**
  + Like how a normal linked list works but with the ability to be able to go back to the tail of a linked list from the head almost in a circle

## **M03 L019**

## **M03 L020**

* **Maps**
  + A collection that stores data in key-value patterns, and uses <,>.
    - **HashMap**- It has a key and value, if you try to use key on another map it will replace old key with new key

***Example***



## ***Extra***

* ***Instanceof-*** *for checking if a reference variable contains a given type of object reference or not, a type of comparator*

**Code Example**

|  |
| --- |
| //returns false for null  class Test {  }  class Main {  public static void main(String[] args)  {  Test tobj = null;  // A simple case  if (tobj instanceof Test)  System.***out***.println("tobj is instance of Test");  System.***out***.println(  "tobj is NOT instance of Test");  }  } |