VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

(Autonomous)

Department of Computer Science and Engineering

IV B.Tech – I Semester (Sections – C & D)

Big Data Analytics (Elective – I)

UNIT-I – Data Structures Using JAVA – Session-1

Syllabus

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

UNIT-I

Data structures in Java:

Linked List, Stacks, Queues, Sets, Maps; Generics: Generic classes and Type parameters,

Implementing Generic Types, Generic Methods, Wrapper Classes, Concept of Serialization

Outline of Unit-1

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- Introduction to Data Structures
- Linked List
- Stacks
- Queues
- Sets
- Maps
- Generics: Generic classes and Type parameters
- Implementing Generic Types
- Implementing Generic Methods,
- Wrapper Classes
- Concept of Serialization & De-Serialization
- Example JAVA Programs

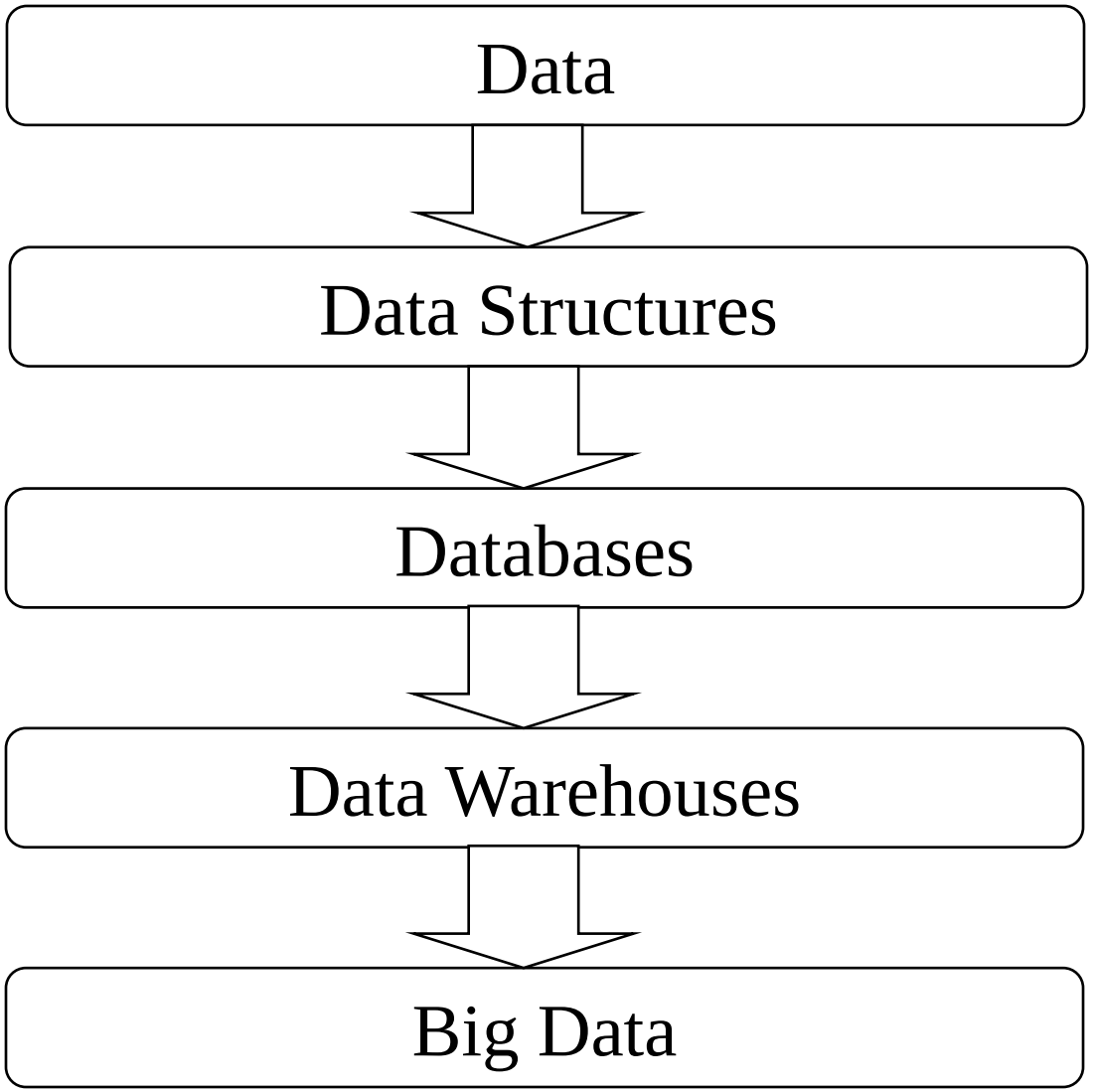
VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

What is Data?

Data are characteristics or information, usually numerical, that are collected through observation. In a more technical sense, data is a set of values of qualitative or quantitative variables ab out one or more persons or objects, while a datum (singular of data) is a single value of a single variable.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Data to Big Data Transformation



VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Revision of Java Language and Discuss the need of Java for our subject.

Why Learning Java is a Starting Point For Big Data Developers Of The Future?

Java is a big friend to Big Data scientists and developers.

Here I am going to tell you why is that so

https://towardsdatascience.com/why-learning-java-is-a-starting-point-for-big-data-developers-of-the-future-9a9b6d240dea

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Data Structures Using JAVA - Introduction

A data structure is a scheme for organizing data in the memory of a computer.

Some of the more commonly used data structures include lists, arrays, stacks, queues, heaps, trees and graphs.

The way in which the data is organized affects the performance of a program for different tasks.

Introduction – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way.

Data Structures provides data elements in terms of some relationship, for better organization and storage.

Introduction – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

For example, our college name “VVIT” and year “2020”. Here “VVIT” is of String data type and “2020” is of integer data type. We can organize this data as a record like College Strength Record. Now we can collect and store college’s records in a file or database as a data structure.

So, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Introduction – Contd.

Basic Types of Data Structures:

As stated above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc., all are data structures.

All these are known as Primitive Data Structures.

Introduction – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

We also have some complex Data Structures, which are used to store large and connected data.

Some examples of Abstract Data Structure are:

Linked List
Tree
Graph
Stack, Queue etc.

Introduction – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

All these data structures allow us to perform different operations on data.

We select these data structures based on which type of operation is required.

The classification of Data Structures was represented in following figure.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Introduction – Contd.

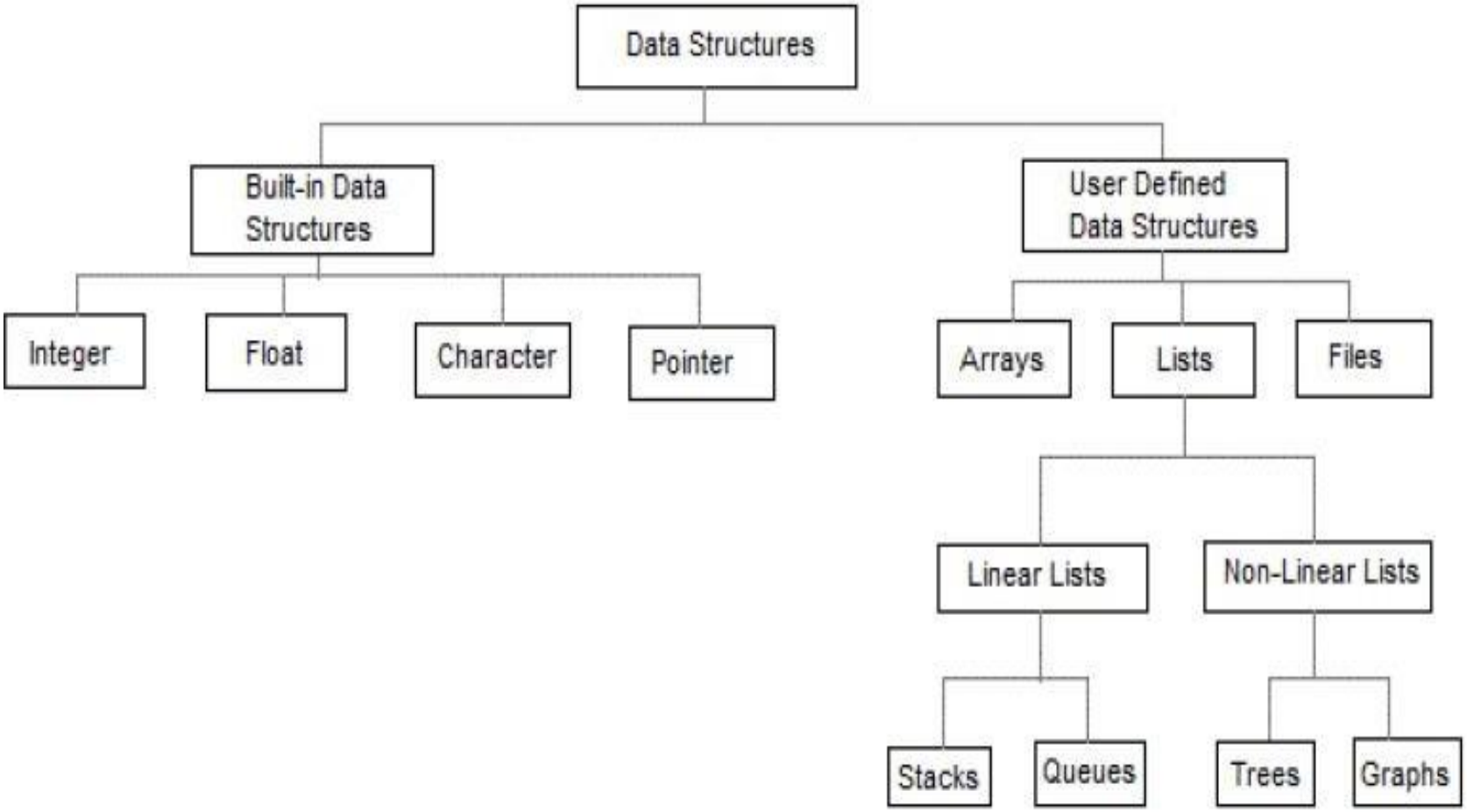


Fig. Classification of Data Structures

Introduction – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

There are basically two types of data structure:

Linear data structure:

Stack, Queue, Linked List

Non-Linear data structure:

Tree and Graph

Introduction – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List

Linked List is a linear collection of data elements, in which linear order is not given by their physical placement in memory.

Elements may be added in front, end of list as well as middle of list.

Introduction – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Stack

Stack is a linear data structure which works on LIFO order. So, that Last In First Out.

In stack element is always added at top of stack and also removed from top of the stack.

Stack is useful in recursive function, function calling mathematical expression calculation, reversing the string etc.

Introduction – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Queue

Queue is also a linear data structure which works on FIFO order. So that First In First Out.

In queue element is always added at rear of queue and removed from front of queue.

Queue applications are in CPU Scheduling, Disk Scheduling, IO Buffers, pipes, file IO.

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Introduction – Contd.

Tree

A tree is a non linear data structure. A root value and sub-trees of children with a parent node, represented as a set of linked nodes.

Nodes can be added at any different node.

Tree applications includes:-

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms.

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Introduction – Contd.

Graph

A graph is a non linear data structure. A set of items connected by edges. Each item is called a vertex or node.

Formally, a graph is a set of vertices and a binary relation between vertices, adjacency.

Graph applications:- finding shortest routes, searching, social network connections, internet routing.

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List

A linked list is a data structure used for collecting a sequence of objects that allows efficient addition and removal of elements in the middle of the sequence.

To understand the need of this data structure, just assume a program that maintains a sequence of employee objects, sorted by the last names of the employees.

When a new employee is hired or joined, an object needs to be inserted into the sequence.

The company maintained their employees in alphabetical order, the new object probably needs to be inserted somewhere near the middle of the sequence.

If we use an array to store the objects, then all objects following the new joined must be moved toward the end.

Linked List – Contd.

Similarly, if an employee left the company, the object must be removed, and the gap (hole) in the sequence needs to be closed up by moving all objects that come after it.

Moving a large number of values can involve a substantial amount of processing time.

Now, we would like to structure the data in a way that minimizes this cost.

Rather than storing the values in an array, a linked list uses a sequence of nodes.

Each node stores a value and a reference to the next node in the sequence, which is show in below Figure

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

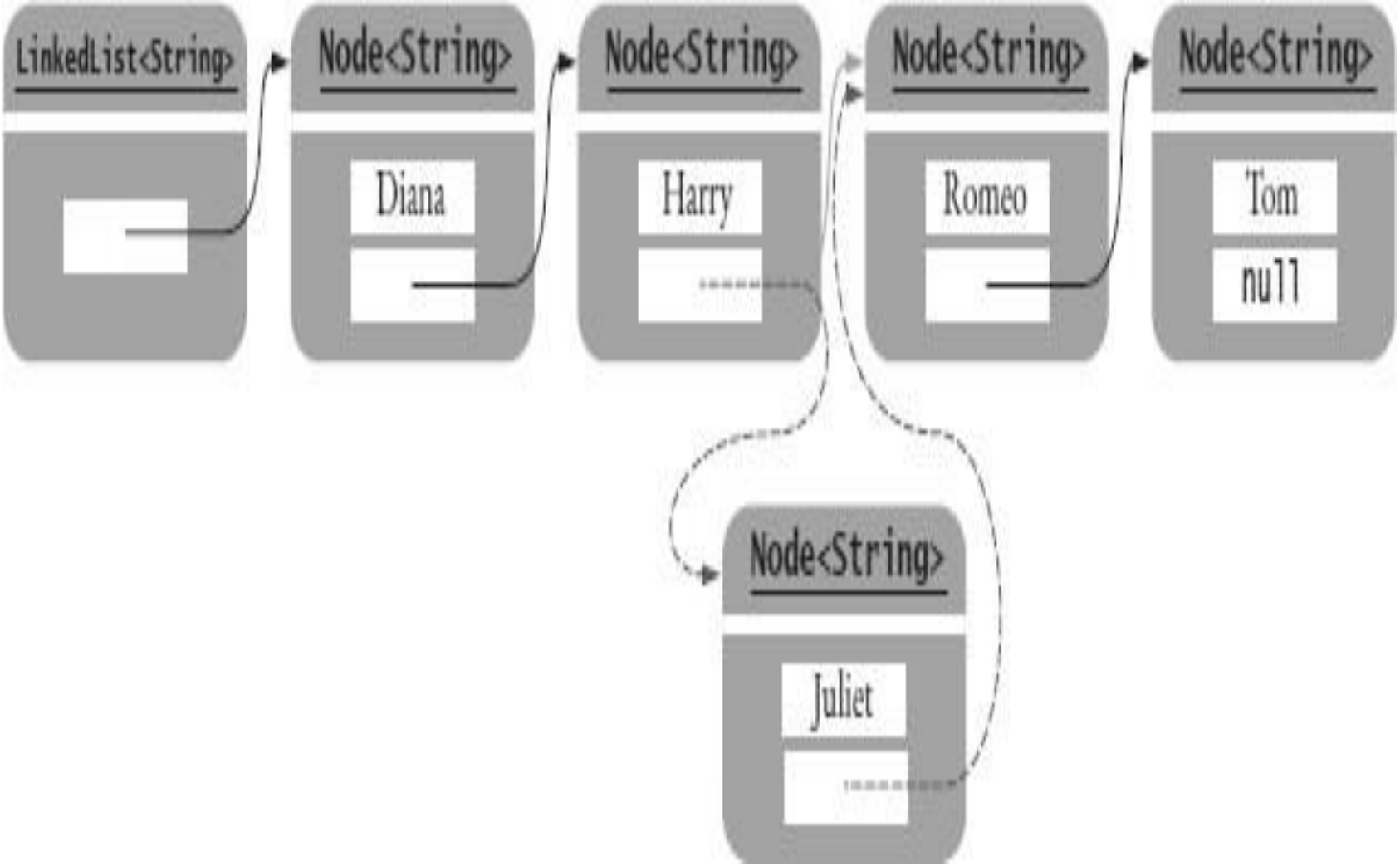


Fig. Inserting an element in a Linked List

Linked List – Contd.

When we insert a new node into a linked list, only the neighboring node references need to be updated.

The same will also applicable for removal of a node.

Linked lists allow speedy insertion and removal, but element access can be slow.

The features of linked list were discussed below:

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

Adding and Removing elements in the middle of a linked list is efficient :

Let us consider, that we want to locate the fifth element in linked list, we must first traverse the first four elements.

This is a problem if we need to access the elements in arbitrary order.

The ‘random access’ describe an access pattern in which elements are accessed in arbitrary (random is not necessary) order.

The sequential access visits the elements in sequence.

For example, a binary search required random access, whereas a linear search required sequential access.

Linked List – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Visiting the elements of a linked list in sequential order is efficient, but random access is not.

If we mostly visit elements in sequence, i.e. to display or print the elements, we don't need to use random access.

We can use linked list when we are concerned about the efficiency of inserting or removing elements and we rarely needs the access of elements in a random order.

Linked List – Contd.

Each element in the LinkedList is called the **Node**.

Each Node of the LinkedList contains two items:

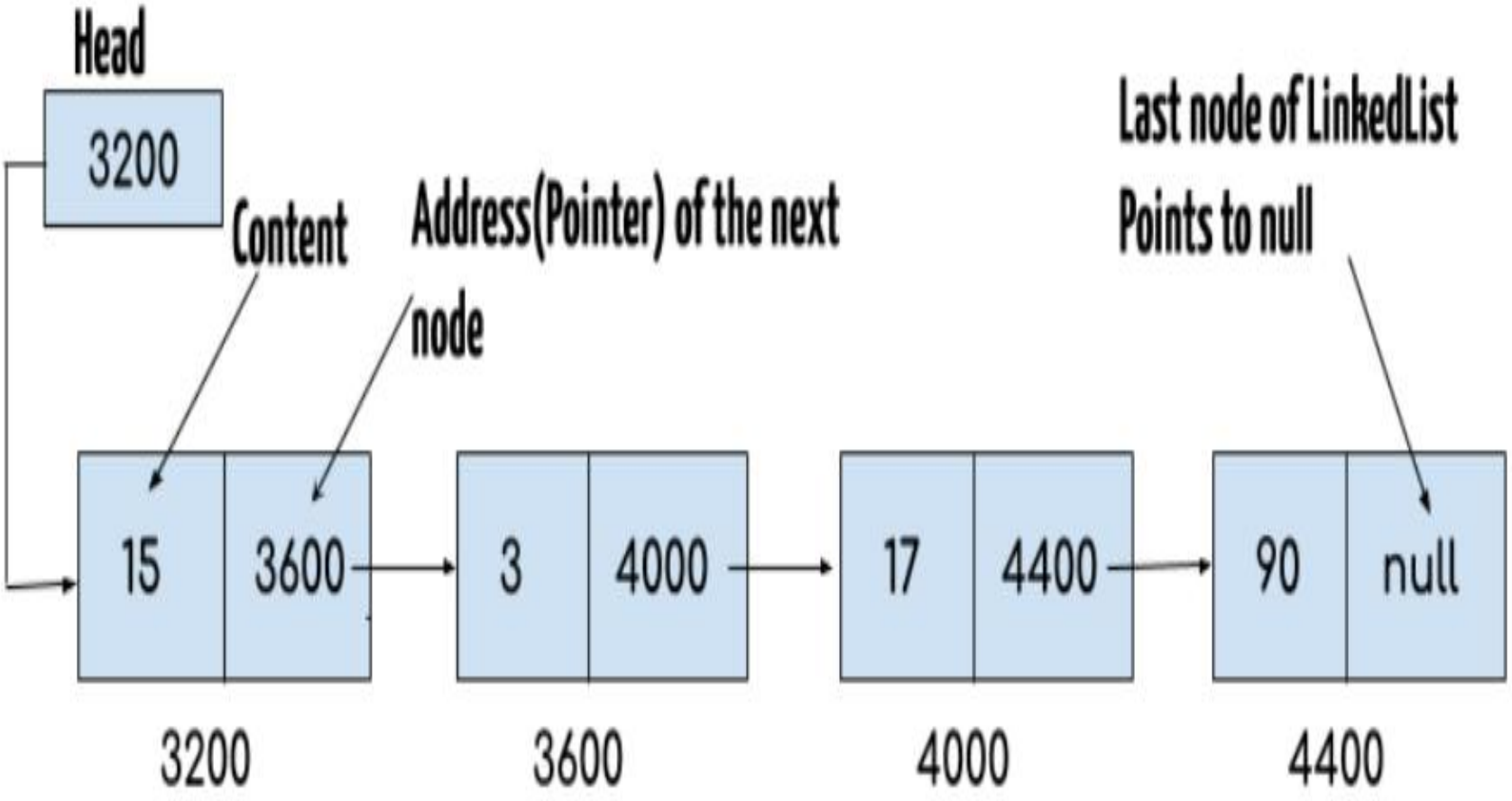
1) Content of the element

2) Pointer/Address/Reference to the Next Node in the LinkedList

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

This is how a LinkedList Looks:



Linked List – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Head of the LinkedList only contains the Address of the **First element** of the List.
The Last element of the LinkedList contains **null** in the pointer part of the node because it is the end of the List so it doesn't point to anything as shown in the above diagram.

The diagram which is shown above represents a **singly linked list**.
There is another complex type variation of LinkedList which is called **doubly linked list**, node of a doubly linked list contains three parts:

1) Pointer to the previous node of the linked list
2) content of the element
3) pointer to the next node of the linked list.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

Why do we need a Linked List?

You must be aware of the arrays which is also a linear data structure but **arrays have certain limitations such as:**

1) **Size of the array is fixed** which is decided when we create an array so it is hard to predict the number of elements in advance, if the declared size fall short then we cannot increase the size of an array and if we declare a large size array and do not need to store that many elements then it is a waste of memory.

2) Array elements **need contiguous memory locations** to store their values.

Linked List – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

3) **Inserting an element in an array is performance wise expensive** as we have to shift several elements to make a space for the new element.

Let's say we have an array that has following elements: 10, 12, 15, 20, 4, 5, 100, now if we want to insert a new element 99 after the element that has value 12 then we have to shift all the elements after 12 to their right to make space for new element.

Similarly **deleting an element** from the array is also a performance wise expensive operation because all the elements after the deleted element have to be shifted left.

Linked List – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

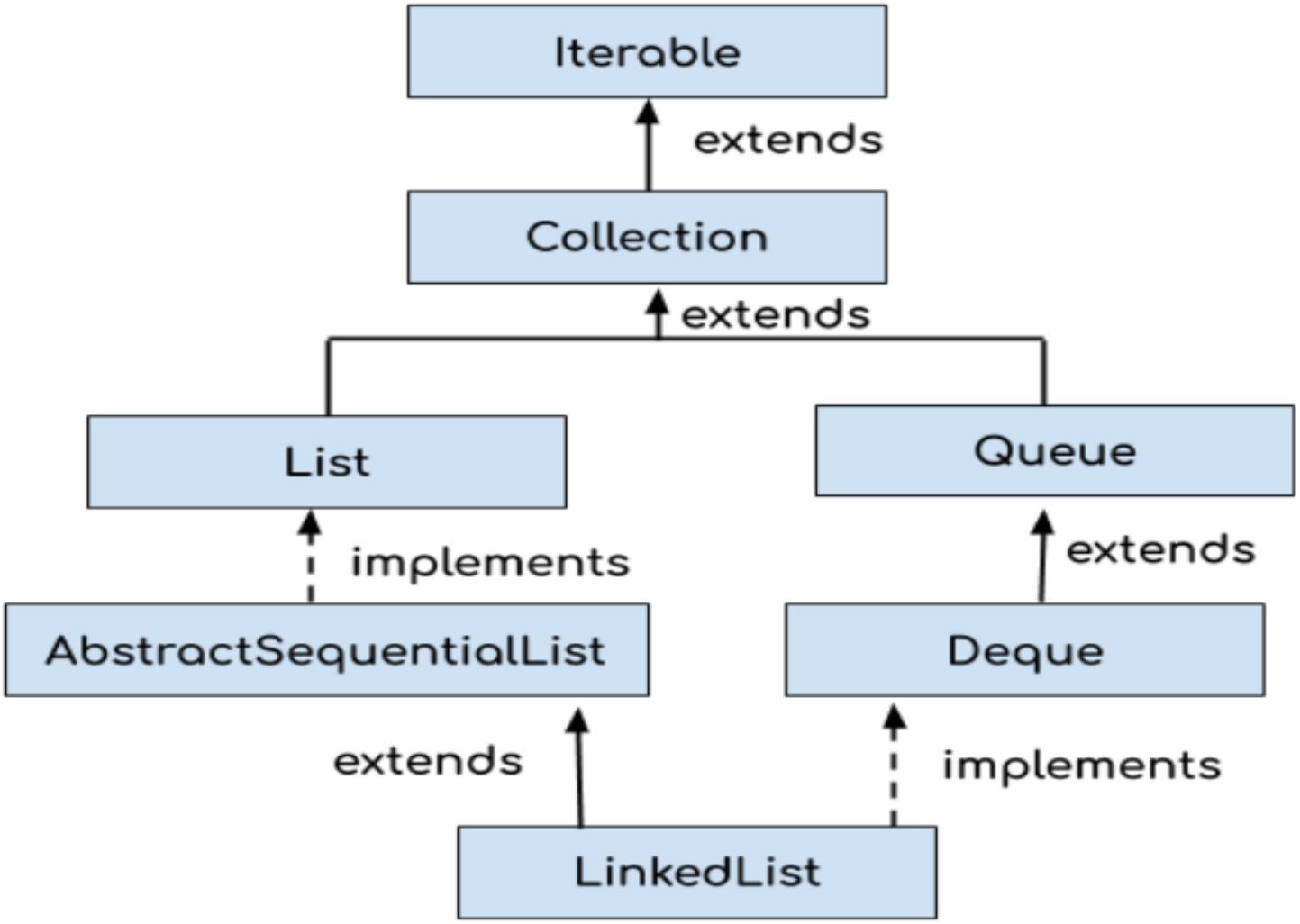
These limitations are handled in the Linked List by providing following features:

1. Linked list allows **dynamic memory allocation**, which means memory allocation is done at the run time by the compiler and we do not need to mention the size of the list during linked list declaration.
2. Linked list elements **don't need contiguous memory locations** because elements are linked with each other using the reference part of the node that contains the address of the next node of the list.
3. Insert and delete operations in the Linked list are not performance wise expensive because adding and deleting an element from the linked list doesn't require element shifting, only the pointer of the previous and the next node requires change.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

Hierarchy of LinkedList class in Java



VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

The Java library provides a linked list class. Now, we will see about that library class usage. The 'LinkedList' class in the java.util package is a generic class, just like the 'ArrayList' class. We can specify the type of the list elements in angle brackets (< >), like as,

LinkedList<String> (or) LinkedList<Product>

Java **Generic Methods and Generic Classes** enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Linked List – Contd.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

We will discuss in detail of Generic Methods and Classes later.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

There are several methods for accessing elements in the LinkedList. The different methods of LinkedList are:

`LinkedList<String> lst = new LinkedList<String>();`	An empty list.
`lst.addLast("Harry")`	Adds an element to the end of the list. Same as `add`.
`lst.addFirst("Sally")`	Adds an element to the beginning of the list. `lst` is now `[Sally, Harry]`.
`lst.getFirst()`	Gets the element stored at the beginning of the list; here `"Sally"`.
`lst.getLast()`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = lst.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `lst` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = lst.listIterator()`	Provides an iterator for visiting all list elements.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

ListIterator – Concepts & Methods

The Java Library had ListIterator, which described a position anywhere inside the linked list.

In concept of ListIterator, we should think of the iterator as pointing between two elements, like as the cursor in a word processor points between two characters.

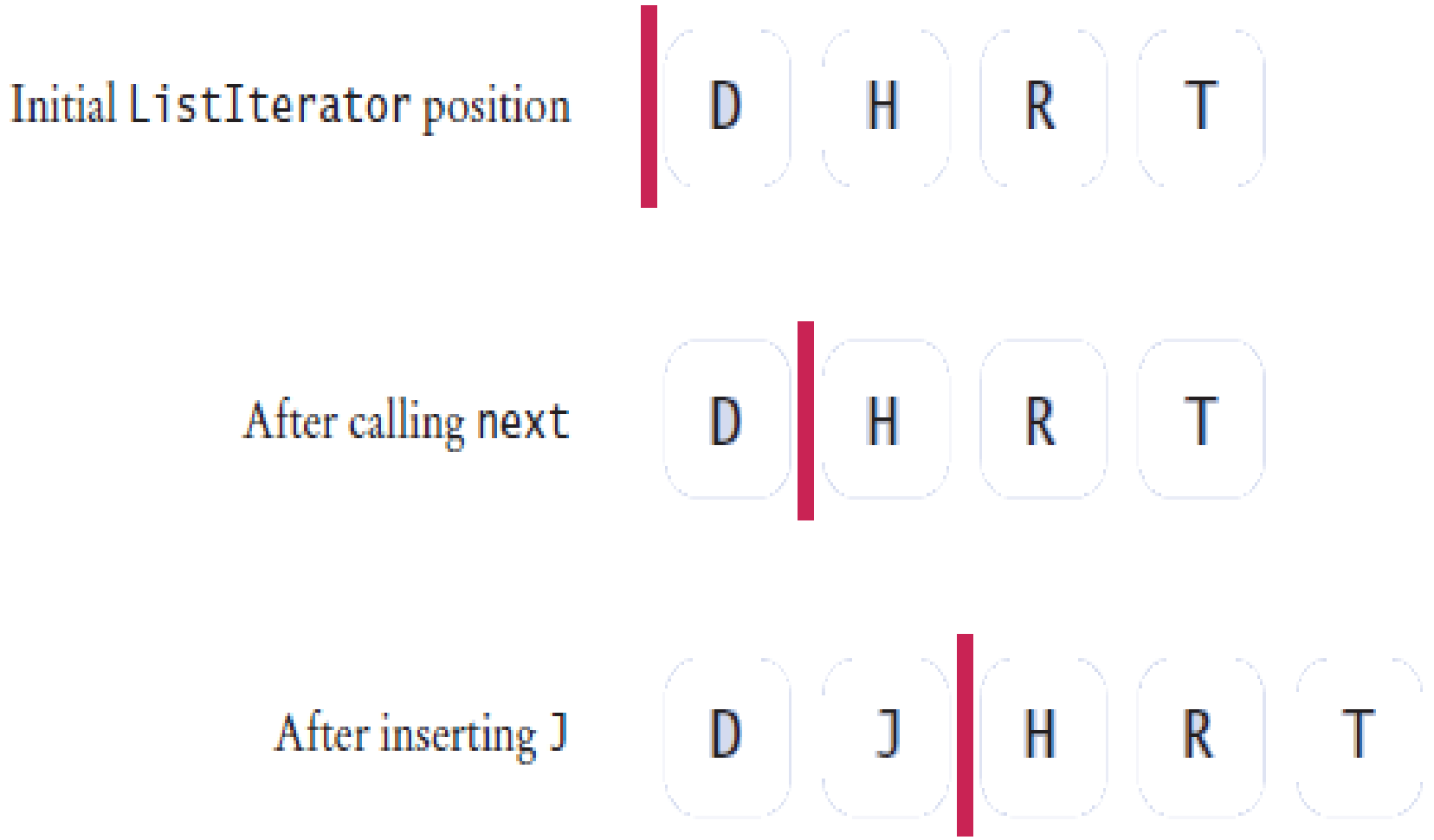
So, we have to think of each element as being like a letter in a word processor, and think of the iterator as being like the blinking cursor between letters.

This concept was disclosed below and also the following table states the different methods of the ListIterator interface.

Linked List – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Conceptual View of the ListIterator is given below:



VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

ListIterator Methods are given below:

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`  s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list.

`iter.add("Diana");`	Adds an element before the iterator position. The list is now `[Diana, Sally]`.
`iter.next();` `iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is again `[Diana]`.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

Here, we will see and practice a simple java program on ListIterator.

```
import java.util.LinkedList;
import java.util.ListIterator;
/** A program that tests the ListIterator in the LinkedList class. */
public class ListTester
{
public static void main(String[] args)
{
LinkedList<String> staff = new LinkedList<String>();
staff.addLast("Diana");
staff.addLast("Harry");
staff.addLast("Romeo");
staff.addLast("Tom");
// | in the comments indicates the iterator position
ListIterator<String> iterator = staff.listIterator(); // |DHRT
iterator.next(); // D|HRT
iterator.next(); // DH|RT
// Add more elements after second element
iterator.add("Juliet"); // DHJ|RT
iterator.add("Nina"); // DHJN|RT
iterator.next(); // DHJNR|T
// Remove last traversed element iterator.remove(); // DHJN|T
// Print all elements
for (String name : staff) System.out.print(name + " ");
System.out.println();
System.out.println("Expected: Diana Harry Juliet Nina Tom");
}
}
```

Expected Result: Diana Harry Juliet Nina Tom
Final Result: Expected: Diana Harry Juliet Nina Tom

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Linked List – Contd.

Implementing the Linked List:

In previous section, we saw how to use the LinkedList class supplied by the Java Library. Now, we will see the implementation of a simple linked list class. This shows us, how the list operations manipulate the links as the list is modified.

We are not implementing all methods of the linked list class. We will implement only single linked list, and the list class will supply direct access only to the first list element, not the last one. Our list will not use a type parameter. We will simply store raw object values and insert casts when retrieving them. The result will be a fully functional list class that shows how the links are updated in add and remove operations and how the iterator traverses the list.

Linked List – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

A ‘Node’ object stores an object and a reference to the next node.

Because the methods of both the list class and the iterator class have frequent access to the Node instance variables, we do not make the instance variables of the ‘Node’ class private.

Instead, we make ‘Node’ a private inner class of the LinkedList class.

Because name of the LinkedList methods returns a ‘Node’ object, it is safe to leave the instance variables public.

Linked List – Contd.

```
public class LinkedList
{
        . . .
        class Node
        {
                public Object data;
                public Node next;
        }
}
```

We can do the following things on LinkedList.

- Adding a Node to the Head of a Linked List.
- Removing the First Node from a Linked List.
- Removing a Node from the Middle of a Linked List using ListIterator methods.
- Adding a Node to the Middle of a Linked List using ListIterator methods.

Linked List – Contd.

The implementation of linked list will be shown by a Java program.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

```
import java.util.*;
public class LinkedListExample {
     public static void main(String args[]) {

       /* Linked List Declaration */
       LinkedList<String> linkedlist = new LinkedList<String>();

       /*add(String Element) is used for adding
        * the elements to the linked list*/
       linkedlist.add("Item1");
       linkedlist.add("Item5");
       linkedlist.add("Item3");
       linkedlist.add("Item6");
       linkedlist.add("Item2");

       /*Display Linked List Content*/
       System.out.println("Linked List Content: " +linkedlist);

       /*Add First and Last Element*/
       linkedlist.addFirst("First Item");
       linkedlist.addLast("Last Item");
       System.out.println("LinkedList Content after addition: " +linkedlist);
```

Linked List – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

```
        /*This is how to get and set Values*/
        Object firstvar = linkedlist.get(0);
        System.out.println("First element: " +firstvar);
        linkedlist.set(0, "Changed first item");
        Object firstvar2 = linkedlist.get(0);
        System.out.println("First element after update by set method: " +firstvar2);

        /*Remove first and last element*/
        linkedlist.removeFirst();
        linkedlist.removeLast();
        System.out.println("LinkedList after deletion of first and last element: " +linkedlist);

        /* Add to a Position and remove from a position*/
        linkedlist.add(0, "Newly added item");
        linkedlist.remove(2);
        System.out.println("Final Content: " +linkedlist);
    }
}
```

Linked List – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Output:

```
Linked List Content: [Item1, Item5, Item3, Item6, Item2]
LinkedList Content after addition: [First Item, Item1, Item5, Item3, Item6, Item2, Last Item]
First element: First Item
First element after update by set method: Changed first item
LinkedList after deletion of first and last element: [Item1, Item5, Item3, Item6, Item2]
Final Content: [Newly added item, Item1, Item3, Item6, Item2]
```

Linked List – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Abstract Data Types:

There are two ways of looking into a linked list. They are:

Concrete View – One way is to think of the concrete implementation of such a list as a sequence of node objects with links between them.

Abstract View – Another way is, we can think of the abstract concept that underlies the linked list. In the abstract, a linked list is an ordered sequence of data items that can be traversed with an iterator.

An abstract data type defines with fundamental operations on the data but does not specify an implementation.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Stack & Queue

These are two common abstract data types that allow insertion and removal of items at the ends only, not in the middle.

Stack:

A Stack lets you insert and remove elements at only one end, called the TOP of the stack.

The basic features of stack was stated in following:

Stack in an ordered list of similar data type.

Stack is a LIFO structure (Last In – First Out).

push( ) function is used to insert new elements into the stack and pop( ) is used to delete an element from the stack.

Stack & Queue – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Both insertion and deletion are allowed at only one end of stack called Top.

The simplest application (or) example of a stack is to reverse a word.

We can push a given word to stack – letter by letter – and then pop letters from the stack.

The stacks are used in Parsing, Expression Conversion (Infix to Postfix, Postfix to Prefix etc.)

Stack & Queue – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Queue:

A Queue is similar to a stack, except that you add items to one end of the queue, called as REAR / TAIL and remove them from the other end of the queue, called as FRONT / HEAD.

This makes queue as FIFO structure (First In – First Out).

The basic features are as follows:

Like stack, queue is also an ordered list of elements of similar data types.

Queue is a FIFO structure.

Stack & Queue – Contd.

Once a new element is inserted into the queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.

add( ) function is used to insert the values at TAIL and peek( ) function is oftenly used to return the value of first element without de-queuing it.

Examples of queue are, serving requests on a single shared resource, like a printer, CPU task scheduling etc.

Stack & Queue – Contd.

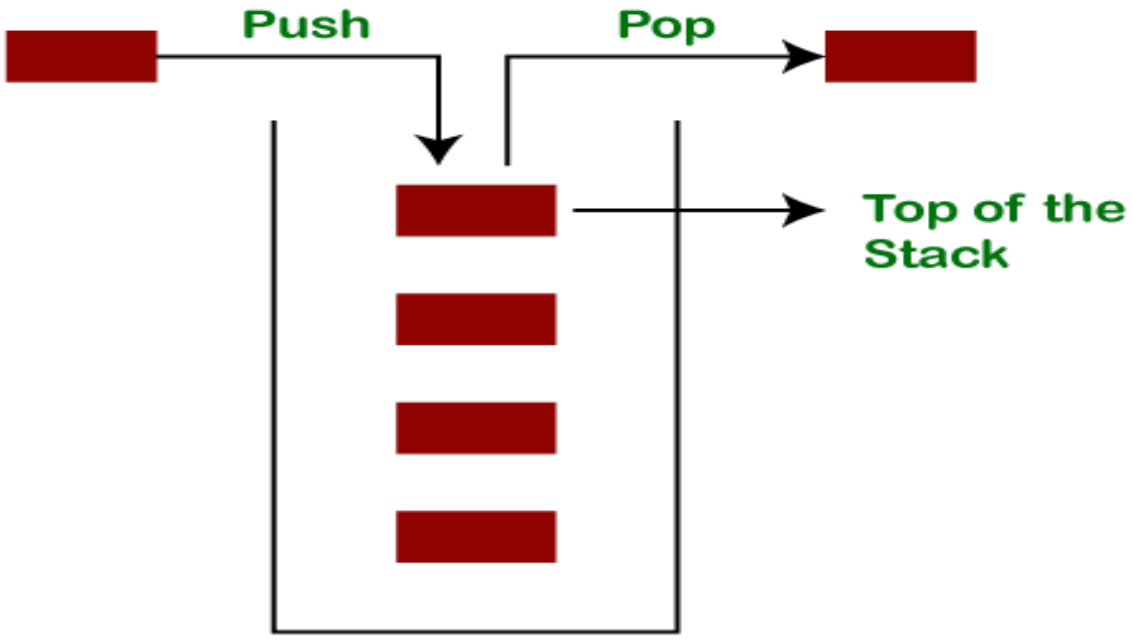
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Stack and Queue methods in Java as follows:

`Queue<Integer> q = new LinkedList<Integer>();`	The `LinkedList` class implements the `Queue` interface.
`q.add(1); q.add(2); q.add(3);`	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.
`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.
`Stack<Integer> s = new Stack<Integer>();`	Constructs an empty stack.
`s.push(1); s.push(2); s.push(3);`	Adds to the top of the stack; `s` is now `[1, 2, 3]`.
`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Stack – Contd.



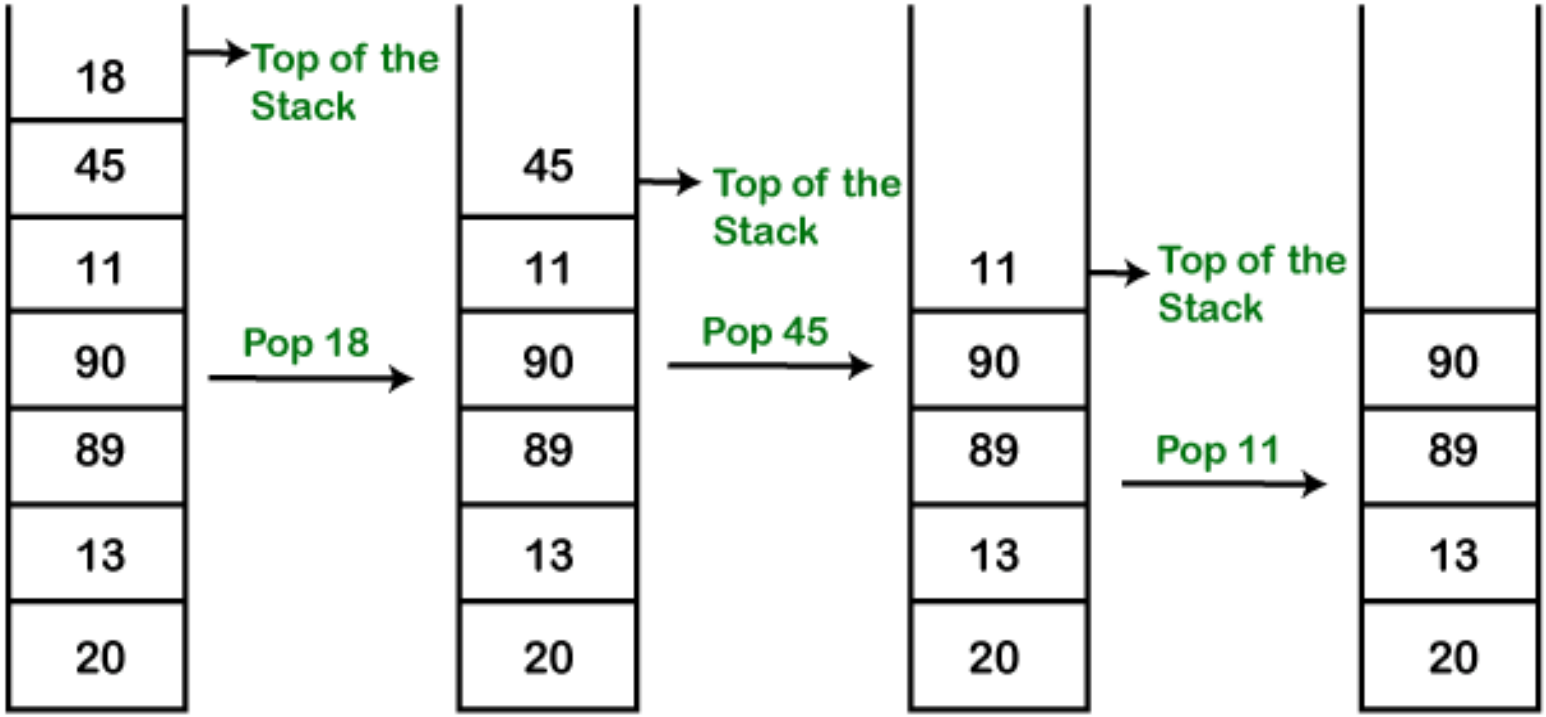
Let's push 20, 13, 89, 90, 11, 45, 18, respectively into the stack.

Stack – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Both insertion and deletion are allowed at only one end of stack called Top.

Let's remove (pop) 18, 45, and 11 from the stack.



Stack – Contd.

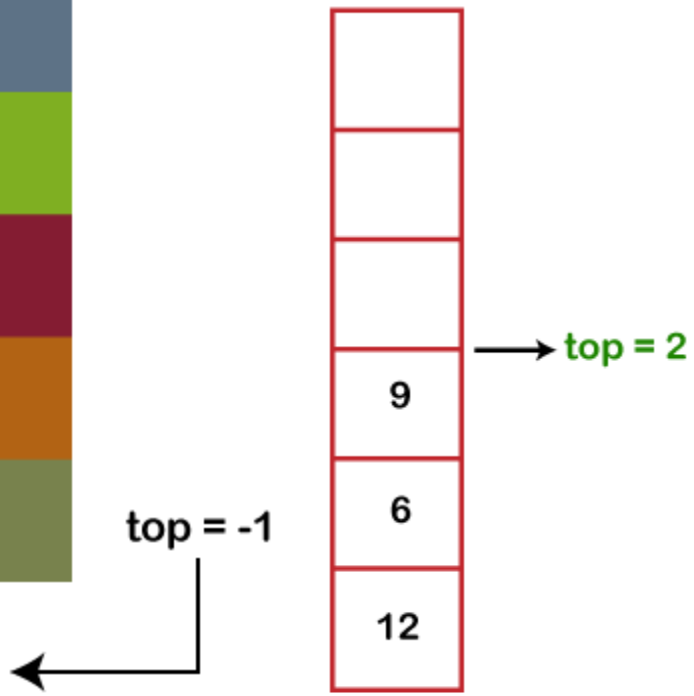
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

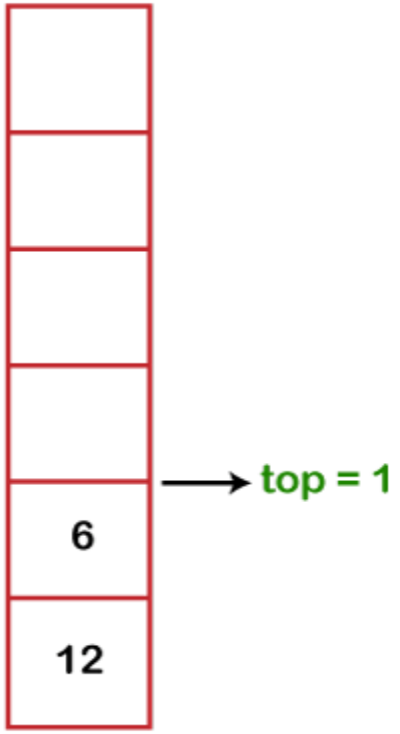
Empty Stack: If the stack has no element is known as an empty stack. When the stack is empty the value of the top variable is -1.

When we push an element into the stack the top is **increased by 1**. In the following figure,

- Push 12, top=0
- Push 6, top=1
- Push 9, top=2

When we pop an element from the stack the value of top is decreased by 1. In the following figure, we have popped 9.





Stack – Contd.

The following table shows the different values of the top.

Top value	Meaning
-1	It shows the stack is empty.
0	The stack has only an element.
N-1	The stack is full.
N	The stack is overflow.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Stack – Contd.

Java Stack Class

In Java, **Stack** is a class that falls under the Collection framework that extends the **Vector** class.

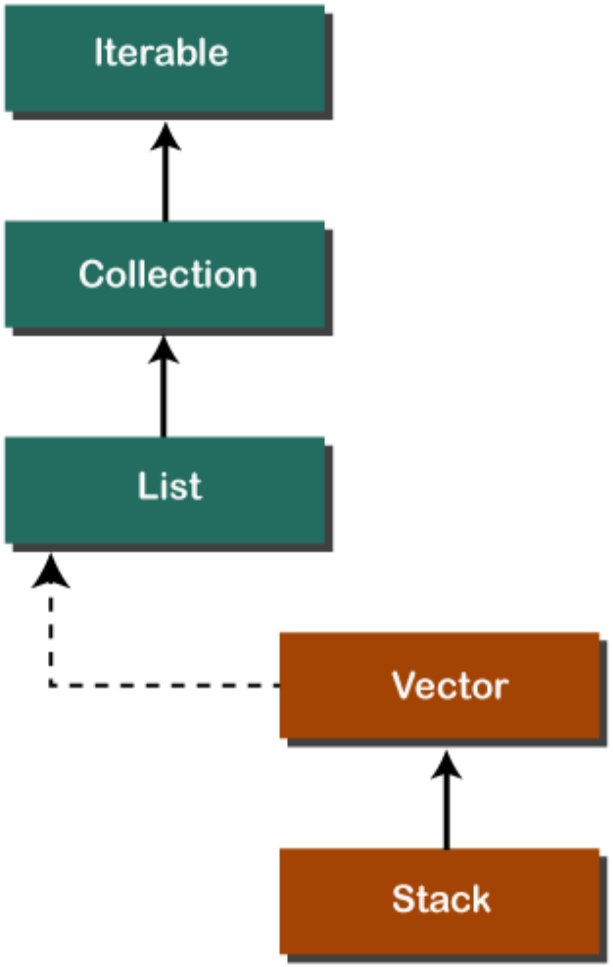
It also implements interfaces **List, Collection, Iterable, Cloneable, Serializable.** It represents the LIFO stack of objects.

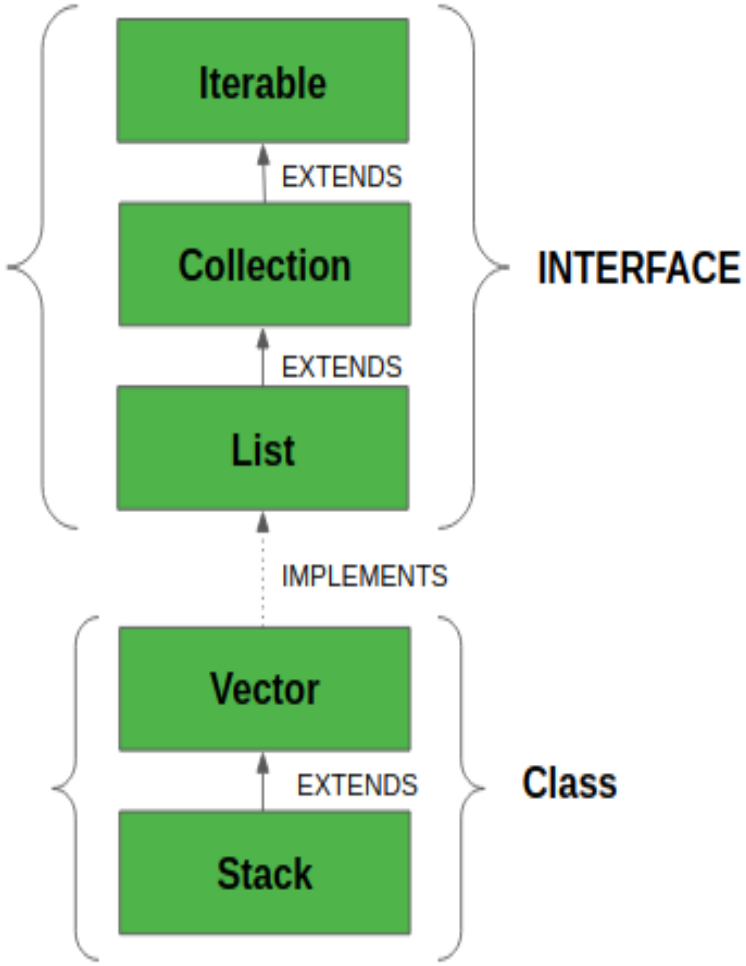
Before using the Stack class, we must import the java.util package.

The stack class arranged in the Collections framework hierarchy, as shown below.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Stack – Contd.





Stack – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

How to Create a Stack?

In order to create a stack, we must import **java.util.stack** package and use the Stack() constructor of this class. The below example creates an empty Stack.

Stack<E> stack = new Stack<E>();

Here E is the type of Object.

Example Java Program for Stack Operations was shown below.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Stack – Contd.

```
// Java code for stack implementation

import java.io.*;
import java.util.*;

class Test
{
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for(int i = 0; i < 5; i++)
        {
            stack.push(i);
        }
    }

    // Popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop Operation:");

        for(int i = 0; i < 5; i++)
        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);
        }
    }
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Stack – Contd.

```
    // Displaying element on the top of the stack
    static void stack_peek(Stack<Integer> stack)
    {
        Integer element = (Integer) stack.peek();
        System.out.println("Element on stack top: " + element);
    }

    // Searching element in the stack
    static void stack_search(Stack<Integer> stack, int element)
    {
        Integer pos = (Integer) stack.search(element);

        if(pos == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element is found at position: " + pos);
    }


    public static void main (String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();

        stack_push(stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);
    }
}
```

Output:

```
Pop Operation:
4
3
2
1
0
Element on stack top: 4
Element is found at position: 3
Element not found
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Queue – Contd.

Operations on Queue:

Mainly the following four basic operations are performed on queue:

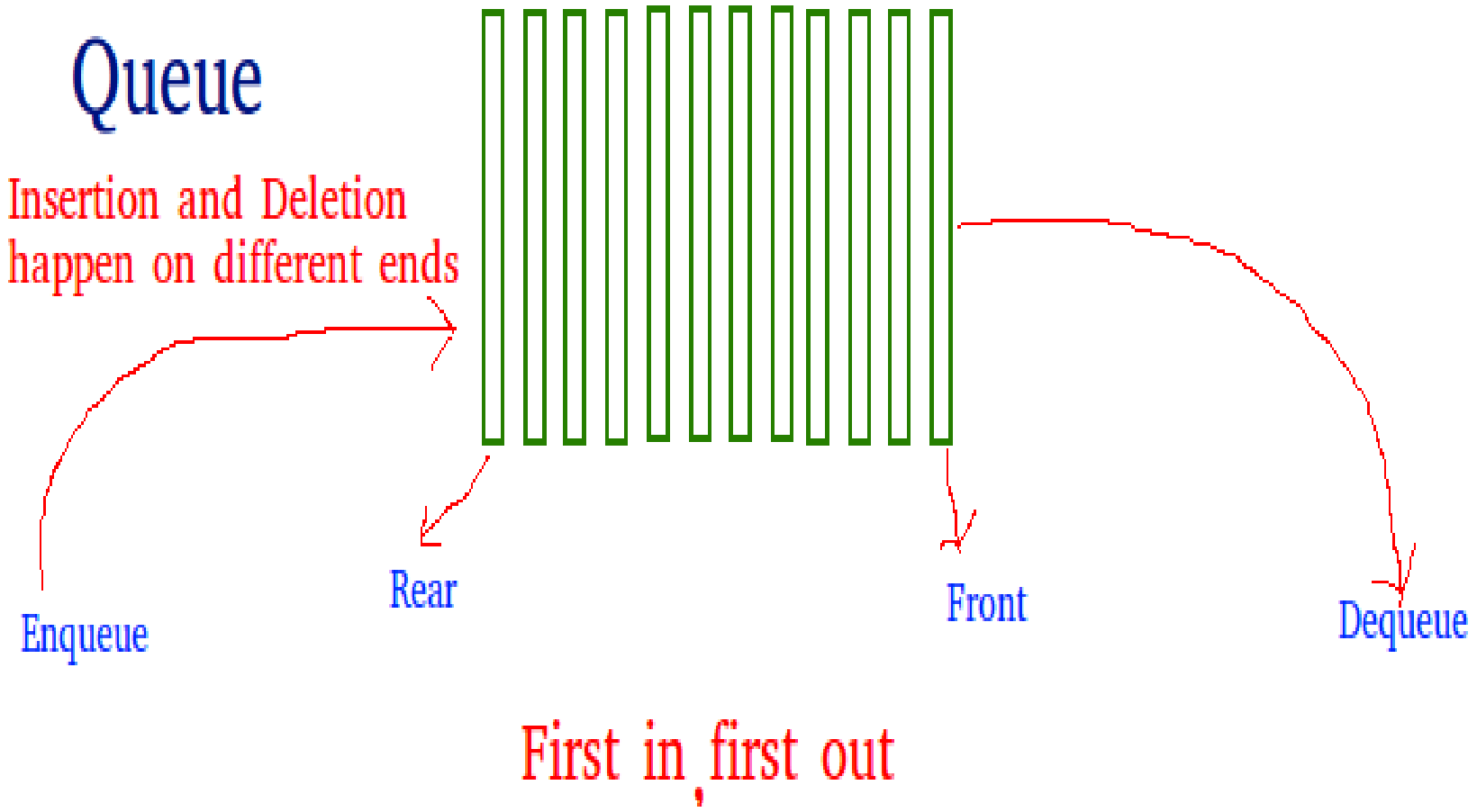
Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

Front: Get the front item from queue.

Rear: Get the last item from queue.

Queue – Contd.


Queue
Insertion and Deletion happen on different ends
Enqueue
Rear
Front
Dequeue
First in, first out

Queue – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Example Java Program for Queue Operations was shown below.

```
import java.util.LinkedList;
import java.util.Queue;
public class Example {
   public static void main(String[] args) {
      Queue<Integer> q = new LinkedList<>();
      q.add(6);
      q.add(1);
      q.add(8);
      q.add(4);
      q.add(7);
      System.out.println("The queue is: " + q);
      int num1 = q.remove();
System.out.println("The element deleted from the head is: " + num1);
System.out.println("The queue after deletion is: " + q);
int head = q.peek();
System.out.println("The head of the queue is: " + head);
int size = q.size();
System.out.println("The size of the queue is: " + size);
}
}
```

Queue – Contd.

The output of the above Java Program for Queue Operations was shown below.

Output

```
The queue is: [6, 1, 8, 4, 7]
The element deleted from the head is: 6
The queue after deletion is: [1, 8, 4, 7]
The head of the queue is: 1
The size of the queue is: 4
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Queue – Contd.

Queue using Array Implementation →

https://www.geeksforgeeks.org/queue-set-1introduction-and-array-implementation/?ref=rp

Queue using Linked List Implementation →

https://www.geeksforgeeks.org/queue-linked-list-implementation/?ref=rp

List Data Structures

List in Java provides the facility to maintain the *ordered collection*.

It contains the index-based methods to insert, update, delete and search the elements.

It can have the duplicate elements also. We can also store the null elements in the list.

The List interface is found in the java.util package and inherits the Collection interface.

List Data Structures

It is a factory of ListIterator interface. Through the ListIterator, we can iterate the list in forward and backward directions.

The implementation classes of List interface are ArrayList, LinkedList, Stack and Vector.

The ArrayList and LinkedList are widely used in Java programming.

List Interface declaration

public interface List<E> **extends** Collection<E>

List Data Structures – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

List in detail:

https://www.javatpoint.com/java-list

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Set & Map Data Structures

SETs:

When comparing arrays and list data structures, both have one common characteristic, i.e. they keep the elements in the same order in which we inserted them.

But, in many applications, we don't really care about the order of the elements in collection.

Let us, take an example, a server may keep a collection of objects like printers.

The order of the objects doesn't really matter.

In mathematics, such an unordered collection is called a 'Set'.

So, a set is an unordered collection of distinct elements. Elements can be added, located and removed.

Set & Map Data Structures

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Set:

If the data structure is no longer responsible for remembering the order of element insertion, can it give us better performance for some of its operations? For this purpose, we can use sets.

The fundamental operations on a set are as follows:
Adding an element.
Removing an element.
Locating an element. (if the set contain a given object)
Listing all elements. (not necessarily in the order in which they were added)

Set & Map Data Structures

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Set:

Sets don't have duplicates. Adding a duplicate of an element that is already present is silently ignored.
As well as, attempting to remove an element that isn't in the set is also silently ignored.
We could use a linked list or array list to implement a set.
But adding, removing and containment testing would be O(n) operations, because they are have to do a linear search through the list (adding required a search through the list to make sure that we don't add a duplicate).

O(n) is Big O Notation and refers to the complexity of a given algorithm. n refers to the size of the input, in your case it's the number of items in your list. O(n) means that your algorithm will take on the order of n operations to insert an item.

Set & Map Data Structures

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Set:

To handle these operations in a set as quick manner, there are two different data structures.

They are ‘hash tables’ and ‘trees’.

Set & Map Data Structures

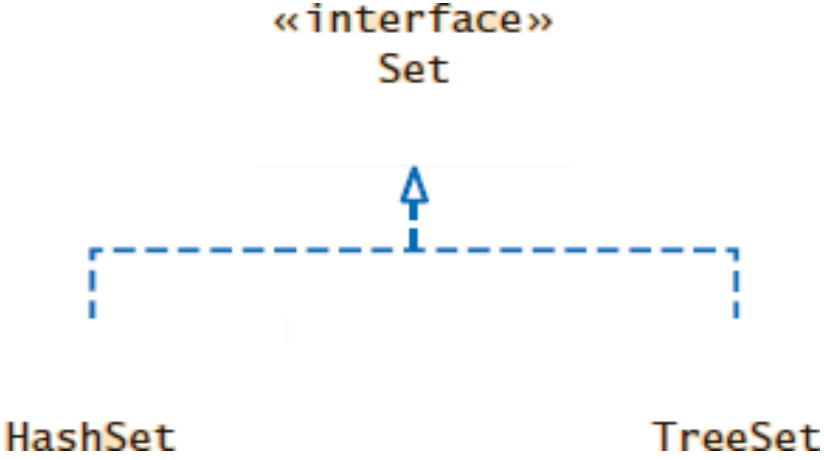
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Set:

The standard Java library provides set implementation based on both data structures, i.e. HashSet and TreeSet.

Both of these data structures implement the set interface and the below figure show this hierarchy.

Set & Map Data Structures



Set:

If we want to use a set in our program, we must choose between these implementation.

In order to use a HashSet, the elements must provide a hashcode method.

Many classes in the standard library implements these methods.

For example, String, Integer, Point, Rectangle, Color and all the collection classes.

Therefore, we can form a HashSet<String>, HashSet<Rectangle>, or even a HashSet<HashSet<Integer>>.

Set & Map Data Structures

Set:

The TreeSet class uses a different strategy for arranging its elements.

Elements are kept in sorted order. In order to use a TreeSet, the element type should implement the comparable interface.

The String and Integer classes fulfill this requirement, but many other classes do not.

We can also construct a TreeSet with a Comparator.

As a rule of thumb, use a hash set unless you want to visit the set elements in sorted order.

Now, we can construct the set by using any one of the following:

Set<String> names = new HashSet<String> ( );

Set<String> names = new TreeSet<String> ( );

Set & Map Data Structures

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Set:

Note that, we store the reference to the HashSet<String> or TreeSet<String> object in a Set<String> variable.
Once, we construct the collection object, the implementation no longer matters; only the interface is important.
Adding and removing Set elements is straightforward:

names.add("Romeo"); names.remove("Juliet");

To visit all elements in a set, use an iterator. A Set iterator visits elements in apparently random order (HashSet) or sorted order (TreeSet). We cannot add an element to a set at an iterator position.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Set & Map Data Structures

SET using Java

Prog-1:

```
import java.util.*;

public class SetDemo
{
public static void main(String args[])
{
int count[] = {34, 22,10,60,30,22};

Set<Integer> set = new HashSet<Integer>();
try
{
for(int i = 0; i<5; i++)
{
set.add(count[i]);
}

System.out.println(set);
TreeSet sortedSet = new TreeSet<Integer>(set);
System.out.println("The sorted list is:");
System.out.println(sortedSet);
System.out.println("The First element of the set is: "+ (Integer)sortedSet.first());
System.out.println("The last element of the set is: "+ (Integer)sortedSet.last());
}catch(Exception e)
{
}
}
}
```

Output:-

```
$ java SetDemo [34, 30, 60, 10, 22]
The sorted list is: [10, 22, 30, 34, 60]
The First element of the set is: 10
The last element of the set is: 60
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Set & Map Data Structures

Prog-2:

```
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

public class Main
{
public static void main(String args[])
{
Set<String> hs = new HashSet<String>();
hs.add("one");
hs.add("two");
hs.add("three");
System.out.println("Here is the HashSet: " + hs);

if (!hs.add("three"))
System.out.println("Attempt to add duplicate. " + "Set is unchanged: "+ hs);

TreeSet<Integer> ts = new TreeSet<Integer>();
ts.add(8);
ts.add(19);
ts.add(-2);
ts.add(3);

System.out.println(ts);
System.out.println("First element in ts: " + ts.first());
System.out.println("Last element in ts: " + ts.last());
System.out.println("First element > 15: " + ts.higher(15));
System.out.println("First element < 15: " + ts.lower(15));

}
}
```

Output:-

```
Here is the HashSet: [one, two, three]
Attempt to add duplicate. Set is unchanged: [one, two, three]
[-2, 3, 8, 19]
First element in ts: -2
Last element in ts: 19
First element > 15: 19
First element < 15: 8
```

Set & Map Data Structures

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

HashSet:

https://www.javatpoint.com/java-hashset

TreeSet:

https://www.javatpoint.com/java-treeset

Set & Map Data Structures

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

MAPs:

A Map is a data type that keeps associates between 'keys' and 'values'.

The following Figure shows a typical example:

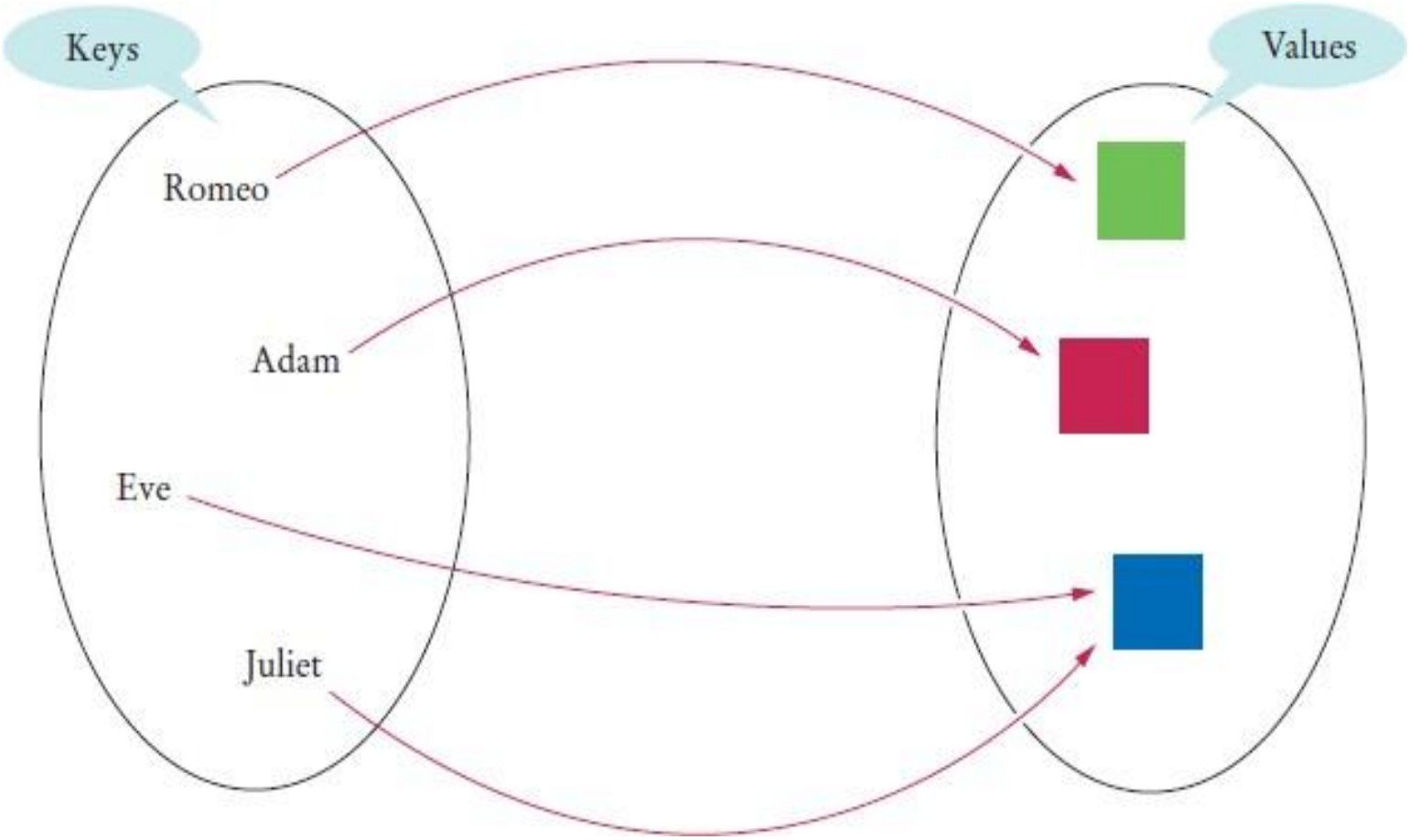
A map that associates names with colors.

This map might describe the favorite colors of various people.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Set & Map Data Structures

Map

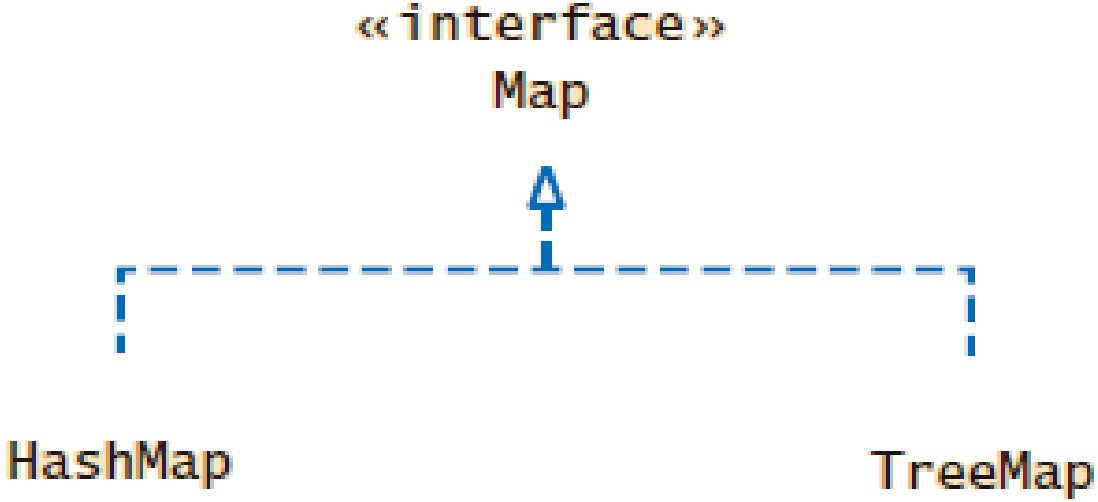


Set & Map Data Structures

Map

From the above Fig, a map is a function from one set, the 'key set', to another set, the 'value set'.
Every key in the map has a unique value, but a value may be associated with several keys.
Just as there are two kinds of set implementations, the Java library has two implementations for maps also.
HashMap and TreeMap. Both of them implement the Map interface which is shown in below Fig.
As with sets, we need to decide which of the two to use. As a rule of thumb, use a HashMap unless we want to visit the keys in sorted order.

Set & Map Data Structures



Map:

After constructing a HashMap or TreeMap, we should store the reference to the map object in a Map reference:

Map<String, Color> favoriteColors = new HashMap<String, Color> ( );

(or)

Map<String, Color> favoriteColors = new TreeMap<String, Color> ( );

Set & Map Data Structures

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Map:

Use the put method to add an association:
favoriteColors.put("Juliet",Color,RED);

We can change the value of an existing association, simply by calling put method again: favoriteColors.put("Juliet",Color,BLUE);

The get method returns the value association with a key.
Color JulietsfavoriteColor = favoriteColors.get("Juliet");

If a key that isn't associated with any values, then the get method returns null.

To remove a key and its associated value, use the remove method: favoriteColors.remove("Juliet");

To find all key and values in a map, iterate through the key set and find the values that correspond to the keys.

Set & Map Data Structures

Map using Java

```
/*  Simple Java HashMap example

This simple Java Example shows how to use Java HashMap. It also describes how to add
something to HashMap and how to retrieve the value added from HashMap. */

import java.util.HashMap;

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

//create object of HashMap
 HashMap<String, Integer> hMap = new HashMap<String, Integer>();

/*Add key value pair to HashMap usingObject put(Object key, Object value) method of Java
HashMap class, where key and value both are objectsput method returns Object which is either
the value previously tied to the key or null if no value mapped to the key.*/

hMap.put("One", new Integer(1));

hMap.put("Two", new Integer(2));

/* Please note that put method accepts Objects. Java Primitive values CAN NOTbe added directly
to HashMap. It must be converted to corresponding wrapper class first.  */

//retrieve value using Object get(Object key) method of Java HashMap class

Object obj = hMap.get("One");

System.out.println(obj);

/* Please note that the return type of get method is an Object. The value must be casted to the
original class. */
}
}
```

/* Output of the program would be 1 */

Set & Map Data Structures

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Map

https://www.javatpoint.com/java-map

HashMap

https://www.javatpoint.com/java-hashmap

TreeMap

https://www.javatpoint.com/java-treemap

Generic Classes and Type Parameters

Generic Programming = Programming with Classes and Methods Parameterized with Types

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array, or an array of any type that supports ordering.

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods, or with a single class declaration, a set of related types, respectively.

Generic Classes and Type Parameters – Contd.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept, we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Classes and Type Parameters – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Generic Classes:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas.

These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Generic Classes and Type Parameters – Contd.

Generic programming is the creation of programming constructs that can be used with many different types.

For example, the java library programmers who implemented the ArrayList class used the technique if generic programming. As a result, you can form array lists that collect elements of different types, such as **ArrayList<String>, ArrayList<BankAccount>**, and so on.

The LinkedList class is an example of generic programming. we can store objects of any class inside a LinkedList. That LinkedList class achieves genericity by using inheritance. It uses references of type object and is therefore capable of storing objects of any class.

Generic Classes and Type Parameters – Contd.

In contrast, the **ArrayList** class is a generic class: a class with a type parameter that is used to specify the type of the objects that we want to store.

Here, we have to note that only our **LinkedList** implementation uses inheritance in general. The standard java library has a generic **LinkedList** class that uses type parameters.

So, in java, generic programing can be achieved with inheritance or with type parameters.

A generic class has one or more type parameters. When declaring a generic class, we specify a type variable for each type parameter.

Generic Classes and Type Parameters – Contd.

The following syntax gives the declaration of **ArrayList** class by the standard java library using the type variable 'E' for the element type:

```
public class ArrrayList<E>
{
    public ArrayList( )
    {
        . . .
    }
    public void add (E element)
    {
        . . .
    }
    . . .
}
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Generic Classes and Type Parameters – Contd.

Here, **E** is a type variable, not a Java reserved word. We could use another name, such as ElementType, instead of E.

However, it is customary to use short, uppercase names for type parameters.

In order to use a generic class, you need to instantiate the type parameter, that is, supply can actual type.

We can supply any class or interface type, for example:

ArrayList<BankAccount>

ArrayList<Measurable>

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Generic Classes and Type Parameters – Contd.

Type parameters can be instantiated with class or interface types.

However, we couldn't substitute any of the eight primitive type for a type parameter.

It would be an error to declare an **ArrayList<double>**.

Use the corresponding wrapper class instead, such as **ArrayList<Double>**.

When we instantiate a generic class, the type that you supply replaces all occurrences of the type variable in the declaration of the class.

For example, the add method for **ArrayList<BankAccount>** has the type variable 'E' replaced with the type BankAccount.

Generic Classes and Type Parameters – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

public void add (BankAccount element)

Dissimilarity with the add method of the **LinkedList** class:

public void add (Object element)

The add method of the generic **ArrayList** class is safer. It is impossible to add a String Object into an **ArrayList<BankAccount>**, but we can accidentally add a String into a **LinkedList** that is intended to hold bank accounts.

Generic Classes and Type Parameters – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

```
Arraylist<BankAccount> accounts1 = new
                                ArrayList<BankAccount> ( );

LinkedList accounts2 = new LinkedList ( ); // Should
                                hold BankAccount objects.

accounts1.add ("my savings"); // Compile-time error.

accounts2.add ("my savings"); // Not declared at
                                Compile-time.
```

Generic Classes and Type Parameters – Contd.

The latter will result in a class cast exception when some other part of the code retrieves the string, believing it to be a bank account:

```
BankAccount account = (BankAccount) accounts2.getFirst ();
                                        // Run-time error
```

Code that uses the generic **ArrayList** class is also easier to read. When you spot an **ArrayList<BankAccount>**, we know right way that is must contain bank accounts.

When we see a LinkedList, we have to study the code to find out what is contains. We used inheritance to implement generic linked lists, has tables, and binary trees, because we already knew the concept of inheritance. But using type parameters, we requires new syntax and additional techniques. So, Type Parameters make generic code safer and easier to read.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Implementing Generic Types

We will learn how to implement our own generic classes. We will write a very simple generic class that stores pairs of objects, each of which can have an arbitrary type. For example,

```
Pair<String, Integer> result = new Pair<String, Integer>
                                ("Harry Morgan", 1729);
```

The **getFirst** and **getSecond** methods retrieve the first and second values of the Pair.

```
String name = result.getFirst ( );
Integer number = result.getSecond ( );
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Implementing Generic Types – Contd.

This class can be useful when you implement a method that computes two values at the same time.

A method cannot simultaneously return a String and an Integer, but it can return a single object of type **Pair<String, Integer>.** The generic Pair class requires two type parameters, one for the type of the first element and one for the type of the second element.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Implementing Generic Types – Contd.

We need to choose variables for the type parameters. It is considered good from to use short uppercase names for type variables shown in below Table.

Type Variable	Meaning
E	Element type in a collection
K	Key type in a Map
V	Value type in a Map
T	General type
S, U	Additional general types

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Implementing Generic Types – Contd.

We place the type variables for a generic class after the class name, enclosed in angle brackets (< >).

public class Pair<T,S>

When we declare the instance variable and methods of the Pair class, use the variable 'T' for the first element type and 'S' for the second element type.

Use type parameters for the types of generic instance variables, method parameters, and return values.

Implementing Generic Types – Contd.

Following example illustrates how we can define a generic class:

```
public class Box<T> {
   private T t;

   public void add(T t) {
      this.t = t;
   }

   public T get() {
      return t;
   }

   public static void main(String[] args) {
      Box<Integer> integerBox = new Box<Integer>();
      Box<String> stringBox = new Box<String>();

      integerBox.add(new Integer(10));
      stringBox.add(new String("Hello World"));

      System.out.printf("Integer Value :%d\n\n", integerBox.get());
      System.out.printf("String Value :%s\n", stringBox.get());
   }
}
```

Output

```
Integer Value :10
String Value :Hello World
```

Generic Methods

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately.

Following are the rules to define Generic Methods:

1) All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).

Generic Methods – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

2) Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
3) The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
4) A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types, not primitive types (like int, double and char).

Generic Methods – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Example to print an array of different type using a single Generic method

```
public class GenericMethodTest {
    // generic method printArray
    public static < E > void printArray( E[] inputArray ) {
        // Display array elements
        for(E element : inputArray) {
            System.out.printf("%s ", element);
        }
        System.out.println();
    }

    public static void main(String args[]) {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println("Array integerArray contains:");
        printArray(intArray);   // pass an Integer array

        System.out.println("\nArray doubleArray contains:");
        printArray(doubleArray);   // pass a Double array

        System.out.println("\nArray characterArray contains:");
        printArray(charArray);   // pass a Character array
    }
}
```

Output

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

Array doubleArray contains:
1.1 2.2 3.3 4.4

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

Generic Methods – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

A generic method is a method with a type parameter.

Such a method can occur in a class that in itself is not generic.

We can think of it as a template for a set of methods that differ only be one or more types.

For example, we may want to declare a method that can print an array of any type:

Generic Methods – Contd.


```
public class ArrayUtil
{
        /** Prints all elements in an array.
         @ 'a' the array to print
        */
        Public <T> static void print (T[ ] a)
        {
                ...
        }
        . . .
}
```

Generic Methods – Contd.

Based on previous topics, it is often easier to see how to implement a generic method by starting with a concrete example.

This method prints the elements in array of strings.

```
public class ArrayUtil
{
public static void print(String[] a)
{
for (String e : a)
System.out.print (e + " ");
System.out.println ( );
}
. . .
}
```

Generic Methods – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

In order to make the method into a generic method, replace string with a type parameter, say E, to denote the element type of the array.

Add a type parameter list, enclosed in angle brackets, between the modifiers (public static) and the return type (void).

```
public static <E> void print(E[] a)
{
for (E e : a)
System.out.print(e + " ");
System.out.println();
}
```

Generic Methods – Contd.

When we wants call the generic method, we need not specify which type to use for the type parameter. So, generic methods are differ from generic classes.

We can call the method with appropriate parameters, and the compiler will match up the type parameters, with the parameter types.

So, when calling a generic method, we need not instantiate the type parameters.

Like generic classes, we cannot replace type parameters with primitive types (8 primitive types).

Generic Classes and Type Parameters – Contd.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Bounded Type Parameters:

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses.

This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Generic Classes and Type Parameters – Contd.

Following example illustrates how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

This example is Generic method to return the largest of three Comparable objects.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Generic Classes and Type Parameters – Contd.

```
public class MaximumTest {
   // determines the largest of three Comparable objects

   public static <T extends Comparable<T>> T maximum(T x, T y, T z) {
      T max = x;   // assume x is initially the largest

      if(y.compareTo(max) > 0) {
         max = y;   // y is the largest so far
      }

      if(z.compareTo(max) > 0) {
         max = z;   // z is the largest now
      }
      return max;   // returns the largest object
   }

   public static void main(String args[]) {
      System.out.printf("Max of %d, %d and %d is %d\n\n",
         3, 4, 5, maximum( 3, 4, 5 ));

      System.out.printf("Max of %.1f,%.1f and %.1f is %.1f\n\n",
         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ));

      System.out.printf("Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum("pear", "apple", "orange"));
   }
}
```

Output

```
Max of 3, 4 and 5 is 5

Max of 6.6,8.8 and 7.7 is 8.8

Max of pear, apple and orange is pear
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

WRAPPER CLASSES:

Java is an object-oriented language and can view everything as an object.

A simple file can be treated as an object (with java.io.File), an address of a system can be seen as an object (with java.util.URL), an image can be treated as an object (with java.awt.Image) and a simple data type can be converted into an object (with wrapper classes).

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Wrapper classes are used to convert any data type into an object.

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself.

Sometimes, it is required to convert data types into objects in Java language.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Let us take an example, upto JDK1.4, the data structures accept only objects to store.

So, a data type is to be converted into an object and then added to a Stack or Vector etc.

For this purpose of conversion, the designers / programmers introduced the concept of wrapper classes.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

What are Wrapper Classes?

As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance.

Wherever, the data type is required as an object, this object can be used.

Wrapper classes include methods to unwrap the object and give back the data type.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

It can be compared with a Biscuits / Chocolates.

The manufacturer wraps the Biscuits / Chocolates with some foil or paper to prevent from the outside pollution.

The User / Consumer takes Biscuits / Chocolates, removes the foil / paper (i.e. wrapper) and eats it.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

We will see the following conversion to understand the Concept of Wrapper Classes:

```
int k = 10;
Integer it1 = new Integer (k);
```

Here, the *int* data type ‘k’ is converted into an object, it1 using *Integer* class.

The ‘it1’ object can be used in Java Programming wherever ‘k’ is needed an object.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Now, we will do the reverse process by the following conversion which can be used to unwrap (getting back *int* from *Integer* object) to object it1.

```
int m = it1.intValue();
System.out.println(m*m); //prints 100.
```

Here, intValue() is a method of Integer class that return an *int* data type.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

List of Wrapper Classes:

In the above stated example code, *Integer* class is known as wrapper class, because it wraps around *int* data type to give it an impression of object.

To wrap or convert each primitive data type, there comes a wrapper class.

So, in Java, Eight wrapper classes exist in *java.lang* package that represent 8 data types.

The following Table shows the 8 primitive data types and corresponding 8 wrapper classes.

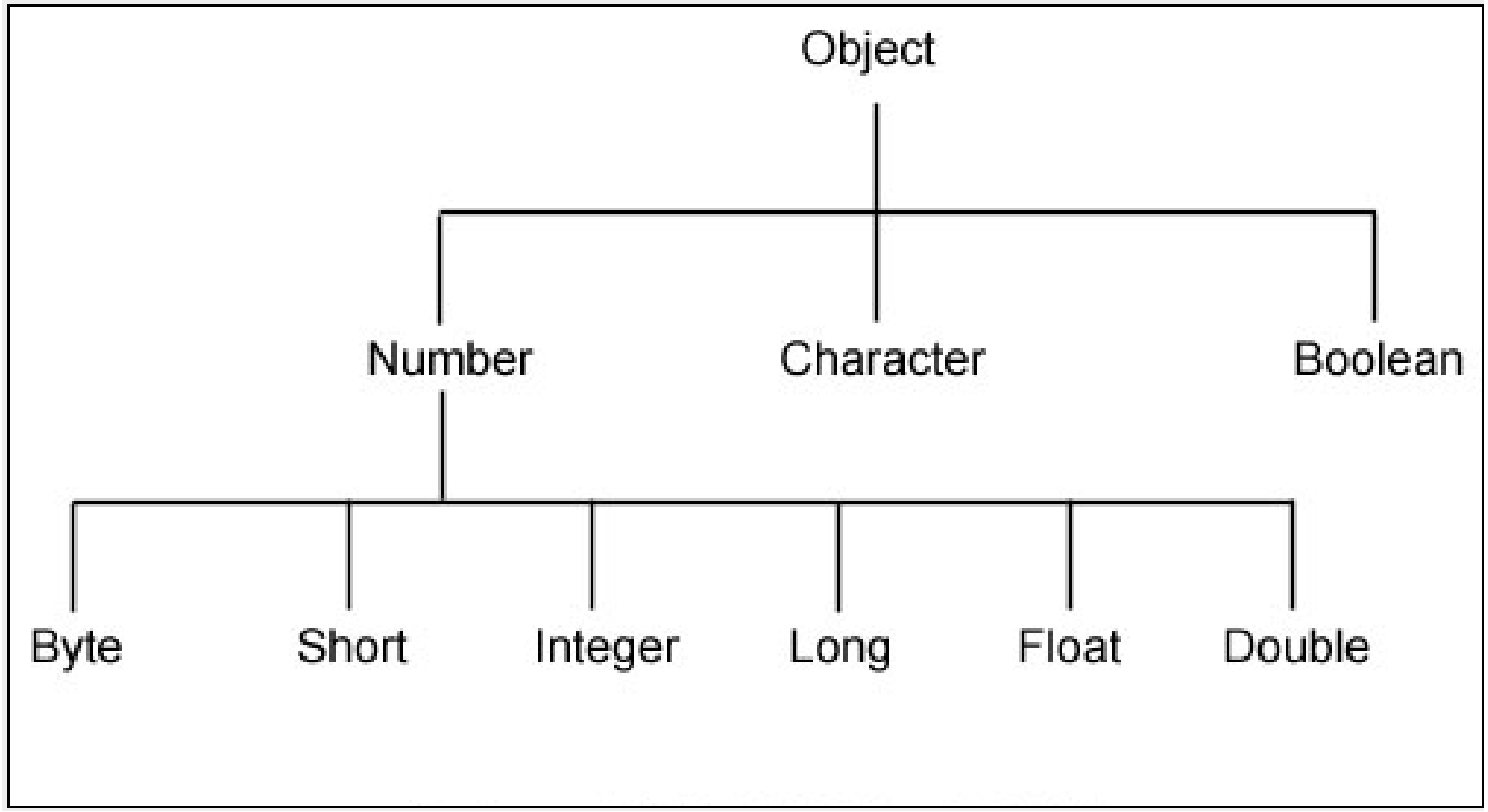
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Wrapper Classes – Wrapping & Unwrapping

And the following Figure shows the Hierarchy of the above Wrapper Classes.



VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

All the 8 wrapper classes are placed in *java.lang* package so that they are implicitly imported and made available to the programmer.

According to the above hierarchy figure, the super class of all numeric wrapper classes is *'Number'* and the super class for *'Character'* and '*Boolean'* is "*Object"*.

All the wrapper classes are defined as final and thus programmers / designers prevented them from inheritance.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Why do we need a wrapper class in Java?

Wrapper Class will convert primitive data types into objects. The objects are necessary if we wish to modify the arguments passed into the method (because primitive types are passed by value).

The classes in java.util package handles only objects and hence wrapper classes help in this case also.

Data structures in the Collection framework such as ArrayList and Vector store only the objects (reference types) and not the primitive types.

The object is needed to support synchronization in multithreading.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Importance of Wrapper Classes:

There are mainly two uses with wrapper classes.

1) To convert simple data types into objects, that is, to give object form to a data type; here constructors are used.

2) To convert strings into data types (known as parsing operations), here methods of type parseXXX() are used.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

The demonstration of wrapping and unwrapping was shown in following Java Program.

In that program we have to observe that, constructors of wrapper classes are used to convert data types into objects and the methods of the form XXXValue() are used to retrieve back the data type from the objects.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

```
public class WrappingUnwrapping
{

public static void main(String args[])
{

// data types
byte grade = 2;
int marks = 50;
float price = 8.6f; // observe a suffix of <strong>f</strong> for float
double rate = 50.5;
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

```
// data types to objects
Byte g1 = new Byte(grade); // wrapping
Integer m1 = new Integer(marks);
Float f1 = new Float(price);
Double r1 = new Double(rate);

// let us print the values from objects
System.out.println("Values of Wrapper objects (printing as objects)");
System.out.println("Byte object g1: " + g1);
System.out.println("Integer object m1: " + m1);
System.out.println("Float object f1: " + f1);
System.out.println("Double object r1: " + r1);
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

```
// objects to data types (retrieving data types from objects)
byte bv = g1.byteValue(); // unwrapping
int iv = m1.intValue();
float fv = f1.floatValue();
double dv = r1.doubleValue();

// let us print the values from data types
System.out.println("Unwrapped values (printing as data
types)");
System.out.println("byte value, bv: " + bv);
System.out.println("int value, iv: " + iv);
System.out.println("float value, fv: " + fv);
System.out.println("double value, dv: " + dv);
}
}
```

Wrapper Classes – Wrapping & Unwrapping

The ***expected result*** of the above program was given below.

```
C:\java\javalang>java WrappingUnwrapping
Values of Wrapper objects (printing as objects)
Byte object g1:  2
Integer object m1:  50
Float object f1:  8.6
Double object r1:  50.5
Unwrapped values (printing as data types)
byte value, bv: 2
int value, iv: 50
float value, fv: 8.6
double value, dv: 50.5
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Autoboxing:

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Wrapper class Example: Primitive to Wrapper

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class WrapperExample1
{
public static void main(String args[])
{
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);
}
}
```

Output:

20 20 20

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing.

It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Wrapper class Example: Wrapper to Primitive

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class WrapperExample2
{
public static void main(String args[])
{
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int explicitly
int j=a;//unboxing, now compiler will write a.intValue() internally
System.out.println(a+" "+i+" "+j);
}
}
```

Output:

3 3 3

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

Another Example program for Wrapping & Unwrapping was given below.

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;

//Autoboxing: Converting primitives into objects
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

```
//Printing objects
System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Wrapper Classes – Wrapping & Unwrapping

```
//Printing primitives
System.out.println("---Printing primitive values---");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intvalue);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
}}
```

Output:

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

Concept of Serialization

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Serialization is a process in which current state of object will be saved in stream of bytes. As byte stream creates platform neutral hence once objects created in one system can be deserialized in other platform.

(OR)

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

Concept of Serialization – Contd.

After a serialized object has been written into a file, it can be read from the file and deserialized, that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Concept of Serialization – Contd.

Classes *'ObjectInputStream'* and *'ObjectOutputStream'* are high level streams that contain the methods for serializing and deserializing an object.

Now, we will see the methods related to these two streams for serialization as well as deserialization of objects.

Concept of Serialization – Contd.

The *ObjectOutputStream* class contains many write methods for writing various data types, but one method in particular stands out:

public final void writeObject(Object x) throws IOException

The above method serializes an object and sends it to the output stream. Similarly, the *ObjectInputStream* class contains the following method for deserializing an object:

public final Object readObject() throws IOException, ClassNotFoundException

The above method retrieves the next object out of the stream and deserializes it. The return value is object, so you will need to cast it to its appropriate data type.

Concept of Serialization – Contd.

To demonstrate how serialization works in Java, we taken an example like employee class and the process was given in below code.

```
public class Employee implements java.io.Serializable

{
public String name;
public String address;
public transient int SSN;
public int number;
public void mailCheck()

{

System.out.println("Mailing a check to " + name + " " + address);

}

}
```

Concept of Serialization – Contd.

In the above code, a class to be serialized successfully, when two conditions must be met:

i) The class must implement the java.io.Serializable interface.

ii) All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient [lasting only for a short time, impermanent].

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Concept of Serialization – Serializing an Object

The ObjectOutputStream class is used to serialize an Object.

The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created.

The program does not generate any output, but study the code and try to determine what the program is doing.

Note: When serializing an object to a file, the standard convention in Java is to give the file a '.ser' extension.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Concept of Serialization – Serializing an Object

```
import java.io.*;

public class SerializeDemo

{

    public static void main(String [] args)

    {
        Employee e = new Employee();  e.name = "abc";
        e.address = "street, city";  e.SSN = 520007;
        e.number = 2020;  try

        {
        FileOutputStream fileOut = new FileOutputStream
                                        ("/tmp/employee.ser");
        ObjectOutputStream out = new ObjectOutputStream
                                        (fileOut);
```

Concept of Serialization – Serializing an Object

```
            out.writeObject(e);
            out.close();
            fileOut.close
            ();

            System.out.printf("Serialized data is saved in
            /tmp/employee.ser");

            } catch(IOException i)

            {

                  i.printStackTrace();

            }

      }

}
```

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Concept of Serialization – Deserializing an Object

The following DeserializeDemo program deserializes the Employee object created in above SerializeDemo program. Observe, Study the program and try to determine its output.

```
import java.io.*;
public class DeserializeDemo
{
public static void main(String [] args)
{
Employee e = null;
try
{
FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
ObjectInputStream in = new ObjectInputStream(fileIn);
e = (Employee) in.readObject();
```

Concept of Serialization – Deserializing an Object

```
in.close();
fileIn.close();
} catch(IOException i)
{
i.printStackTrace();
return;
} catch(ClassNotFoundException c)
{
System.out.println("Employee class not found");
c.printStackTrace();
return;
}
System.out.println("Deserialized Employee...");
System.out.println("Name: " + e.name);
System.out.println("Address: " + e.address);
System.out.println("SSN: " + e.SSN);
System.out.println("Number: " + e.number);
}
}
```

Expected Output:

Deserialized Employee...
Name: abc
Address: street, city
SSN: 0
Number: 2020

Concept of Serialization – Deserializing an Object

Here, the following important points to be noted from the above code and its expected output:

i) The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Concept of Serialization – Deserializing an Object

ii) Notice that the return value of readObject() is cast to an Employee reference.

iii) The value of the SSN field was 520007 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Raw Types

- It's also possible to instantiate generic class `Stack` without specifying a type argument, as follows:

```
// no type-argument specified
Stack objectStack = new Stack( 5 );
```

  - `objectStack` is said to have a raw type
  - The compiler implicitly uses type `Object` throughout the generic class for each type argument.
  - The preceding statement creates a `Stack` that can store objects of any type.
  - Important for backward compatibility with prior versions of Java.
  - Raw-type operations are unsafe and could lead to exceptions.

```
1  // Fig. 21.11: RawTypeTest.java
2  // Raw type test program.
3  public class RawTypeTest
4  {
5     public static void main( String[] args )
6     {
7        Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5 };
8        Integer[] integerElements = {  1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
9
10       // Stack of raw types assigned to Stack of raw types variable
11       Stack rawTypeStack1 = new Stack( 5 );
12
13       // Stack< Double > assigned to Stack of raw types variable
14       Stack rawTypeStack2 = new Stack< Double >( 5 );
15
16       // Stack of raw types assigned to Stack< Integer > variable
17       Stack< Integer > integerStack = new Stack( 10 );
18
```

```
19          testPush( "rawTypeStack1", rawTypeStack1, doubleElements );
20          testPop( "rawTypeStack1", rawTypeStack1 );
21          testPush( "rawTypeStack2", rawTypeStack2, doubleElements );
22          testPop( "rawTypeStack2", rawTypeStack2 );
23          testPush( "integerStack", integerStack, integerElements );
24          testPop( "integerStack", integerStack );
25       } // end main
26
27       // generic method pushes elements onto stack
28       public static < T > void testPush( String name, Stack< T > stack,
29          T[] elements )
30       {
31          System.out.printf( "\nPushing elements onto %s\n", name );
32
33          // push elements onto Stack
34          for ( T element : elements )
35          {
36             System.out.printf( "%s ", element );
37             stack.push( element ); // push element onto stack
38          } // end for
39       } // end method testPush
40
```

```
41      // generic method testPop pops elements from stack
42      public static < T > void testPop( String name, Stack< T > stack )
43      {
44         // pop elements from stack
45         try
46         {
47            System.out.printf( "\nPopping elements from %s\n", name );
48            T popValue; // store element removed from stack
49
50            // remove elements from Stack
51            while ( true )
52            {
53               popValue = stack.pop(); // pop from stack
54               System.out.printf( "%s ", popValue );
55            } // end while
56         } // end try
57         catch( EmptyStackException emptyStackException )
58         {
59            System.out.println();
60            emptyStackException.printStackTrace();
61         } // end catch EmptyStackException
62      } // end method testPop
63   } // end class RawTypeTest
```

```
Pushing elements onto rawTypeStack1
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack1
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:20)

Pushing elements onto rawTypeStack2
1.1 2.2 3.3 4.4 5.5
Popping elements from rawTypeStack2
5.5 4.4 3.3 2.2 1.1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:22)

Pushing elements onto integerStack
1 2 3 4 5 6 7 8 9 10
Popping elements from integerStack
10 9 8 7 6 5 4 3 2 1
EmptyStackException: Stack is empty, cannot pop
        at Stack.pop(Stack.java:32)
        at RawTypeTest.testPop(RawTypeTest.java:53)
        at RawTypeTest.main(RawTypeTest.java:24)
```

Wildcards

- A wildcard is a syntactic construct that is used to denote a family of types in a generic class/method **instantiation**. There are three different kinds of wildcards:
  - " `?` " : the unbounded wildcard. It stands for the family of all types.
  - " `? extends SuperType` " : a wildcard with an upper bound. It stands for the family of all types that are `SuperType` itself or subtypes of `SuperType`.
  - " `? super SubType` " - a wildcard with a lower bound. It stands for the family of all types that are `SubType` itself or supertypes of `SubType`.

Wildcards in Methods That Accept Type Parameters

- We have a powerful generics concept known as wildcards.
- Suppose that you'd like to implement a generic method sum that totals the numbers in an ArrayList.
  - You'd begin by inserting the numbers in the collection.
  - The numbers would be autoboxed as objects of the type-wrapper classes—any int value would be autoboxed as an Integer object, and any double value would be autoboxed as a Double object.
  - We'd like to be able to total all the numbers in the ArrayList regardless of their type.
  - For this reason, we'll declare the ArrayList with the type argument Number, which is the superclass of both Integer and Double.
  - In addition, method sum will receive a parameter of type ArrayList<Number> and total its elements.

```
1  // Fig. 21.13: TotalNumbers.java
2  // Totaling the numbers in an ArrayList<Number>.
3  import java.util.ArrayList;
4
5  public class TotalNumbers
6  {
7     public static void main( String[] args )
8     {
9        // create, initialize and output ArrayList of Numbers containing
10       // both Integers and Doubles, then display total of the elements
11       Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
12       ArrayList< Number > numberList = new ArrayList< Number >();
13
14       for ( Number element : numbers )
15          numberList.add( element ); // place each number in numberList
16
17       System.out.printf( "numberList contains: %s\n", numberList );
18       System.out.printf( "Total of the elements in numberList: %.1f\n",
19          sum( numberList ) );
20    } // end main
21
```

```
22      // calculate total of ArrayList elements
23      public static double sum( ArrayList< Number > list )
24      {
25         double total = 0; // initialize total
26
27         // calculate sum
28         for ( Number element : list )
29            total += element.doubleValue();
30
31         return total;
32      } // end method sum
33   } // end class TotalNumbers
```

```
numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Wildcards in Methods That Accept Type Parameters (cont.)

- In method sum:
  - The `for` statement assigns each `Number` in the `ArrayList` to variable `element`, then uses `Number` method `doubleValue` to obtain the `Number`'s underlying primitive value as a `double` value.
  - The result is added to `total`.
  - When the loop terminates, the method returns the `total`.

Wildcards in Methods That Accept Type Parameters (cont.)

- Given that method `sum` can total the elements of an `ArrayList` of `Number`s, you might expect that the method would also work for `ArrayList`s that contain elements of only one numeric type, such as `ArrayList<Integer>`.
- Modified class `TotalNumbers` to create an `ArrayList-` of `Integer`s and pass it to method `sum`.
- When we compile the program, the compiler issues the following error message:
- `sum(java.util.ArrayList<java.lang.Number>) in TotalNumbersErrors cannot be applied to (java.util.ArrayList<java.lang.Integer>)`
- Although `Number` is the superclass of `Integer`, the compiler does not consider the parameterized type `ArrayList<Number>` to be a superclass of `ArrayList<Integer>`.
- If it were, then every operation we could perform on `ArrayList<Number>` would also work on an `ArrayList<Integer>`.

Wildcards in Methods That Accept Type Parameters (cont.)

- To create a more flexible version of the `sum` method that can total the elements of any `ArrayList` containing elements of any subclass of `Number` we use wildcard-type arguments.
- Wildcards enable you to specify method parameters, return values, variables or fields, and so on, that act as supertypes or subtypes of parameterized types.
- In the following program's method `sum`'s parameter is declared in line 50 with the type:
  - `ArrayList< ? extends Number >`
- A wildcard-type argument is denoted by a question mark (`?`), which by itself represents an "unknown type."
  - In this case, the wildcard extends class `Number`, which means that the wildcard has an upper bound of `Number`.
  - Thus, the unknown-type argument must be either `Number` or a subclass of `Number`.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

```
1  // Fig. 21.14: WildcardTest.java
2  // Wildcard test program.
3  import java.util.ArrayList;
4
5  public class WildcardTest
6  {
7     public static void main( String[] args )
8     {
9        // create, initialize and output ArrayList of Integers, then
10       // display total of the elements
11       Integer[] integers = { 1, 2, 3, 4, 5 };
12       ArrayList< Integer > integerList = new ArrayList< Integer >();
13
14       // insert elements in integerList
15       for ( Integer element : integers )
16          integerList.add( element );
17
18       System.out.printf( "integerList contains: %s\n", integerList );
19       System.out.printf( "Total of the elements in integerList: %.0f\n\n",
20          sum( integerList ) );
21
```

```
22          // create, initialize and output ArrayList of Doubles, then
23          // display total of the elements
24          Double[] doubles = { 1.1, 3.3, 5.5 };
25          ArrayList< Double > doubleList = new ArrayList< Double >();
26
27          // insert elements in doubleList
28          for ( Double element : doubles )
29             doubleList.add( element );
30
31          System.out.printf( "doubleList contains: %s\n", doubleList );
32          System.out.printf( "Total of the elements in doubleList: %.1f\n\n",
33             sum( doubleList ) );
34
35          // create, initialize and output ArrayList of Numbers containing
36          // both Integers and Doubles, then display total of the elements
37          Number[] numbers = { 1, 2.4, 3, 4.1 }; // Integers and Doubles
38          ArrayList< Number > numberList = new ArrayList< Number >();
39
```

```
40          // insert elements in numberList
41          for ( Number element : numbers )
42             numberList.add( element );
43
44          System.out.printf( "numberList contains: %s\n", numberList );
45          System.out.printf( "Total of the elements in numberList: %.1f\n",
46             sum( numberList ) );
47       } // end main
48
49       // total the elements; using a wildcard in the ArrayList parameter
50       public static double sum( ArrayList< ? extends Number > list )
51       {
52          double total = 0; // initialize total
53
54          // calculate sum
55          for ( Number element : list )
56             total += element.doubleValue();
57
58          return total;
59       } // end method sum
60    } // end class WildcardTest
```

```
integerList contains: [1, 2, 3, 4, 5]
Total of the elements in integerList: 15

doubleList contains: [1.1, 3.3, 5.5]
Total of the elements in doubleList: 9.9

numberList contains: [1, 2.4, 3, 4.1]
Total of the elements in numberList: 10.5
```

Wildcards in Methods That Accept Type Parameters (cont.)

- Because the wildcard (`?`) in the method's header does not specify a type-parameter name, you cannot use it as a type name throughout the method's body (i.e., you cannot replace `Number` with `?` in line 55).
- You could, however, declare method `sum` as follows:
  - `public static <T extends Number> double sum( ArrayList< T > list )`
  - allows the method to receive an `ArrayList` that contains elements of any `Number` subclass.
  - You could then use the type parameter `T` throughout the method body.
- If the wildcard is specified without an upper bound, then only the methods of type `Object` can be invoked on values of the wildcard type.
- Also, methods that use wildcards in their parameter's type arguments cannot be used to add elements to a collection referenced by the parameter.

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Summary of UNIT - I

The main objective of this chapter is introducing 'Java concepts' required for developing map reduce programs for 'Big Data'.

By learnt this chapter, we acquired knowledge about the data structures in Java such as Linked List, Stacks, Queues and also example programs in Java.

We also had a basic overview of Sets and Maps in Java, Generic classes and Type parameters, Implementing Generic Types, Generic Methods, Wrapper Classes concepts and Concept of Serialization.

The example java programs are also given for self exercises with expected results.

After this chapter and exercises, the student might be proficient in data structures concepts.