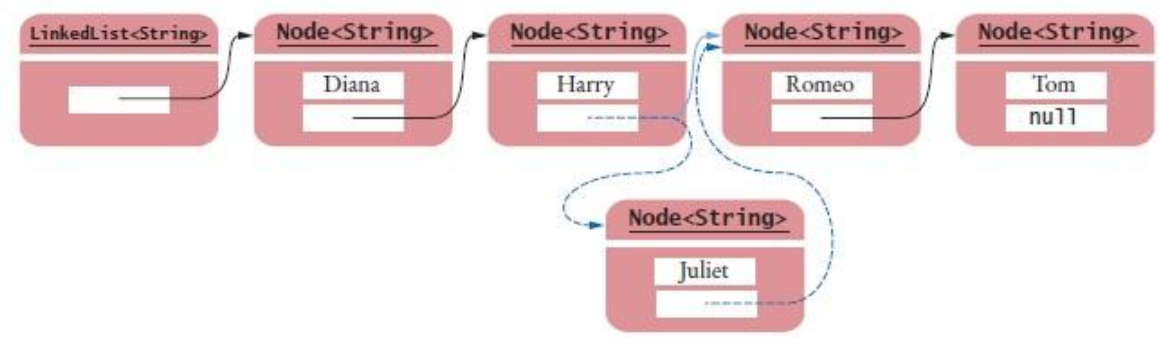
BIG DATA ANALYTICS

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

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.
- A linked list consists of a number of nodes, each of which has a reference to the next node.
- For Example: To understand the need for such a data structure, imagine a program that maintains a sequence of employee objects, sorted by the last names of the employees. When a new employee is hired, an object needs to be inserted into the sequence. Unless the company happened to hire 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 hire must be moved toward the end.



- When you insert a new node into a linked list, only the neighboring node references need to be updated.
- The Java library provides a linked list class.
- The LinkedList class in the java.util package.
- It is a generic class.

- The primitive data types are not supported to use in generic classes. We will convert the primitive types into wrapper classes for using in generic classes.

Syntax:

LinkedList<Type> linkedlist_name = new LinkedList<Type>();

Example:

LinkedList<String> lst = new LinkedList<String>();

It creates empty linked list lst.

Methods:

Method	Description
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

Example Program:

Write a Java Program to create a empty linked list using java library and add 2 elements to the linked list, print the linked list and add 4 more elements to the starting of the list and remove last 2 elements from the list. Finally print the list.

```
import java.util.*;
public class ImplementLinkedList
{
	public static void main(String args[])
	{
		LinkedList<String> ll = new LinkedList<String>();
		ll.addLast("A");
		ll.addLast("B");
		System.out.println(ll);
		ll.addFirst("C");
		ll.addFirst("D");
		ll.addFirst("E");
		ll.addFirst("F");
		System.out.println(ll);
		ll.removeLast();
		ll.removeLast();
		System.out.println(ll);
	}
}
```

Output: [A,B]

[F,E,D,C]

- Visiting the elements of a linked list in sequential order is efficient, but random access is not possible.
- We use a list iterator to access elements inside a linked list.

List Iterator:

- A list iterator describes a position anywhere inside the linked list.
- List Iterator is used to access the elements anywhere in the linked list.
- You obtain a list iterator with the listIterator method of the LinkedList class.

LinkedList class:

LinkedList<String> employeeNames = new LinkedList<String>();

List Iterator:

ListIterator<String> iterator = employeeNames.listIterator();

Methods of the ListIterator Interface:

Method	Description
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.remove();	Removes the element from the list.

Example Program:

```
import java.util.LinkedList;
import java.util.ListIterator;
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");
        ListIterator<String> iterator = staff.listIterator();
        iterator.next(); // D|HRT
        iterator.next(); // DH|RT
        iterator.add("Juliet"); // DHJ|RT
        iterator.add("Nina"); // DHJN|RT
        iterator.next(); // DHJNR|T
        iterator.remove(); // DHJN|T
        for (String name : staff)
            System.out.print(name + " ");
            System.out.println();
    }
}
```

Output: Diana Harry Juliet Nina Tom

Implementing Linked Lists:

- In the last section you saw how to use the linked list class supplied by the Java library. In this section, we will look at the implementation of a simplified version of this class.
- This shows you how the list operations manipulate the links as the list is modified.

Creation of a node:

A Node object stores an object and a reference to the next node. Because the methods of both the linked 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 none of the LinkedList methods returns a Node object, it is safe to leave the instance variables public.

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

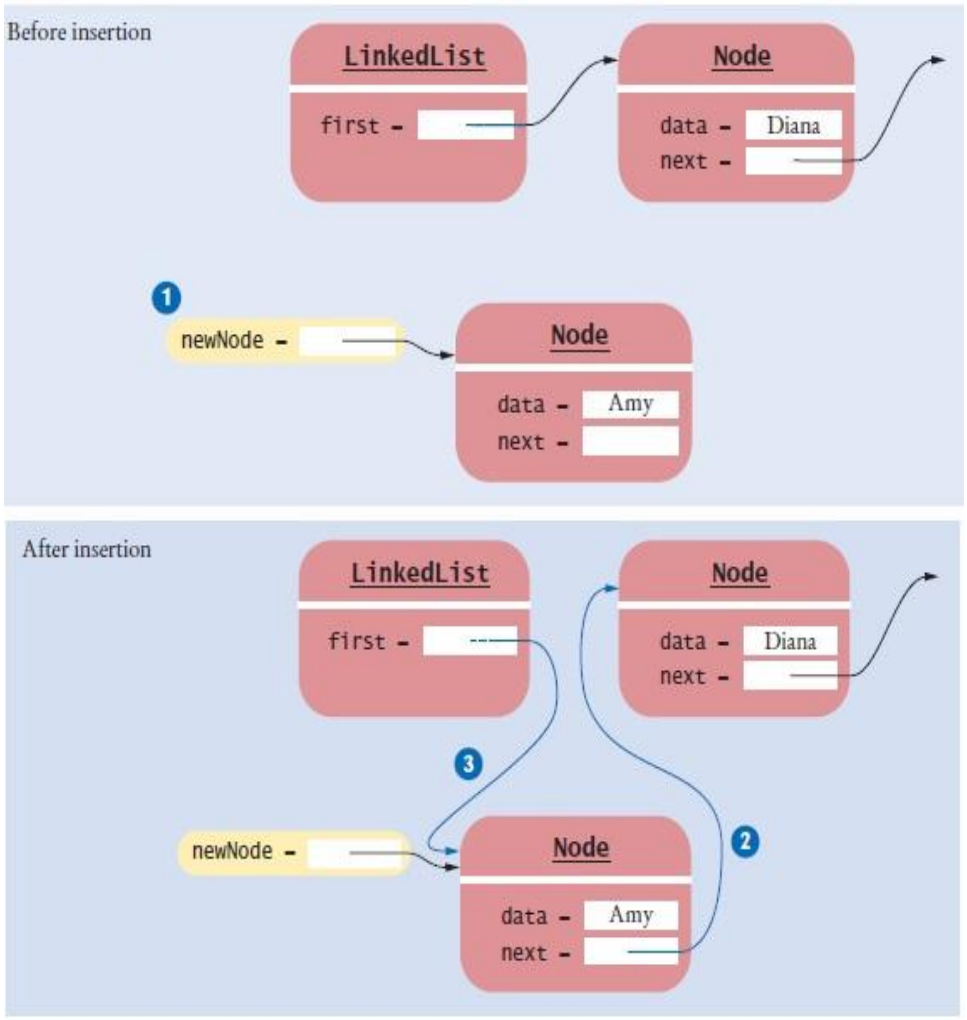
LinkedList class holds a reference first to the first node (or null, if the list is completely empty).

```
public class LinkedList
{
        private Node first;
        . . .
        public LinkedList()
        {
                first = null;
        }
        public Object getFirst()
        {
                if (first == null)
                throw new NoSuchElementException();
                return first.data;
        }
}
```

addFirst() Method:

When a new node is added to the list, it becomes the head of the list, and the node that was the old list head becomes its next node.

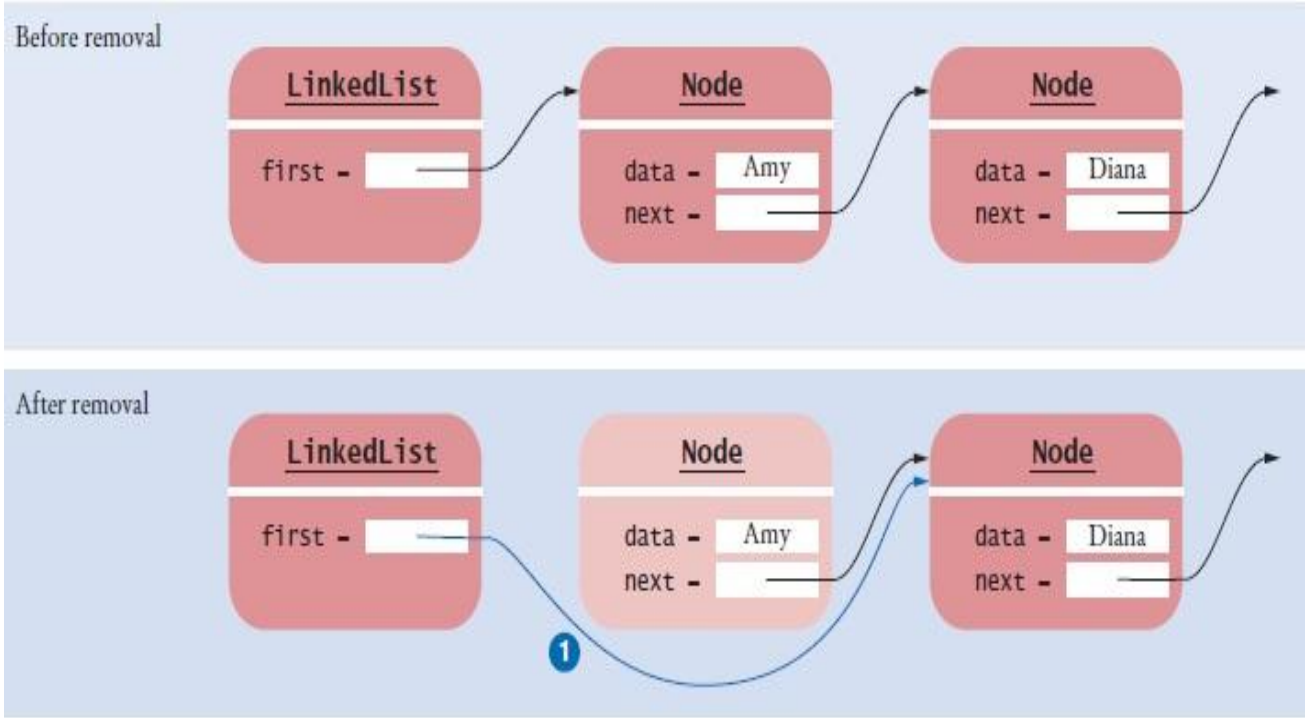
```
public class LinkedList
{
        . . .
        public void addFirst(Object element)
        {
                Node newNode = new Node();
                newNode.data = element;
                newNode.next = first;
                first = newNode;
        }
        . . .
}
```



removeFirst() Method:

Removing the first element of the list works as follows. The data of the first node are saved and later returned as the method result. The successor of the first node becomes the first node of the shorter list. Then there are no further references to the old node, and the garbage collector will eventually recycle it.

```
public class LinkedList
{
. . .
public Object removeFirst()
{
if (first == null)
throw new NoSuchElementException();
Object element = first.data;
first = first.next;
return element;
}
. . .
}
```



Stack:

- A stack is a collection of items with “last in, first out” (LIFO) retrieval.
- A **stack** lets you insert and remove elements at only one end, traditionally called the *top* of the stack.
- To visualize a stack, think of a stack of books

A stack of Books

- New items can be added to the top of the stack. Items are removed at the top of the stack as well. Therefore, they are removed in the order that is opposite from the order in which they have been added, called *last in, first out* or *LIFO* order.
- For example, if you add items A, B, and C and then remove them, you obtain C, B, and A.
- The addition and removal operations are called push and pop.

Syntax:

Stack<Type> stack_name = new Stack<Type>();

Example:

Stack<Integer> s = new Stack<Integer>();

Methods:

Method	Description
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 element of the stack without removing it; head is set to 2.
boolean empty()	Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
int search(Object element)	Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

Example Program:

Write a program to create the stack with four elements and then check the top of the element is 4 or not if its 4 insert two more elements to stack otherwise remove two elements from stack and then check if the element 7 is present in stack or not if its present remove one element from stack otherwise insert two elements into stack and then finally print the stack.

```
import java.util.Stack;
class StackImplementation
{
        public static void main(String args[])
         {
                int ele;
               Stack<Integer> s = new Stack<Integer>();
                s.push(10);
                 s.push(20);
```

```
                s.push(30);
                 s.push(40);
                 if(s.peek()==4)
                {
                        s.push(50);
                         s.push(60);
                 }
                else
                {
                        s.pop();
                        s.pop();
                 }
                if(s.search(7)==-1)
                {
                        s.push(80);
                        s.push(90);
                 }
                else
                        s.pop();
                System.out.println("Stack:"+s);
        }
}
```

Output: [10,20,80,90]

Queue:

- A queue is a collection of items with “first in, first out” (FIFO) retrieval.
- A queue is similar to a stack, except that you add items to one end of the queue (the tail) and remove them from the other end of the queue (the head).
- Queue can’t be created and accessed directly, it can access by using the LinkedList class.

Syntax:

Queue<Type> queue_name = new LinkedList<Type>();

Example:

Queue<Integer> q = new LinkedList<Integer>();

Methods:

Method	Description
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 1and q is [2, 3].
head = q.peek();	Gets the head of the queue without removing it; head is set to 2.
Object element()	It is used to retrieve the head of the queue.
Object poll()	It is used to retrieve and remove the head of the queue, or returns null if the queue is empty.

Example Program:

Write a program to create an empty queue and insert five elements into the queue and then check if the head of the queue is 5 or not if its 5 remove the head element and return the element, otherwise remove two elements from the queue and then finally prints the queue.

```
import java.util.Queue;
import java.util.LinkedList;
class QueueImplementation
{
	public static void main(String args[])
	{
		Queue<Integer> q = new LinkedList<Integer>();
		int head;
		 q.add(5);
		 q.add(20);
		 q.add(13);
		 q.add(47);
		q.add(56);
		 if(q.peek()==5)
		{
			 head = q.poll();
			System.out.println("Removed:"+head);
		}
		else
		 {
			q.remove();
			q.remove();
		}
		System.out.println("Queue: "+q);
}	}
```

Output: Removed: 5

Queue: [20,13,47,56]

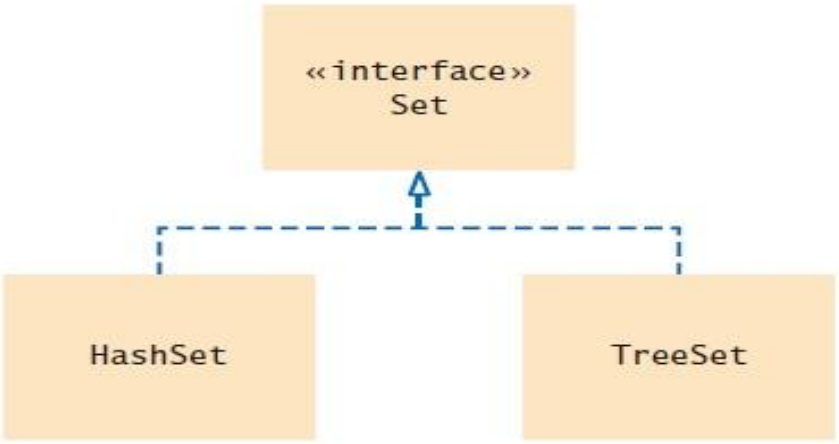
Sets:

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

Fundamental operations of a set:

- Adding an element
- Removing an element
- Locating an element (Does the set contain a given object?)
- Listing all elements (not necessarily in the order in which they were added)

The standard Java library provides set implementations based on both data structures, called HashSet and TreeSet.



First major difference between HashSet and TreeSet is performance. HashSet is faster than TreeSet and should be preferred choice if sorting of element is not required.

Most important difference between HashSet and TreeSet is ordering. HashSet doesn't guaranteed any order while TreeSet maintains objects in Sorted order .

Syntax:

HashSet:

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

TreeSet:

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

Methods:

Method	Description
add( )	Adds an object to the collection.
clear( )	Removes all objects from the collection.
contains( )	Returns true if a specified object is an element within the collection.
isEmpty( )	Returns true if the collection has no elements.
iterator( )	Returns an Iterator object for the collection, which may be used to retrieve an object.
remove( )	Removes a specified object from the collection.
size( )	Returns the number of elements in the collection.

Example Program:

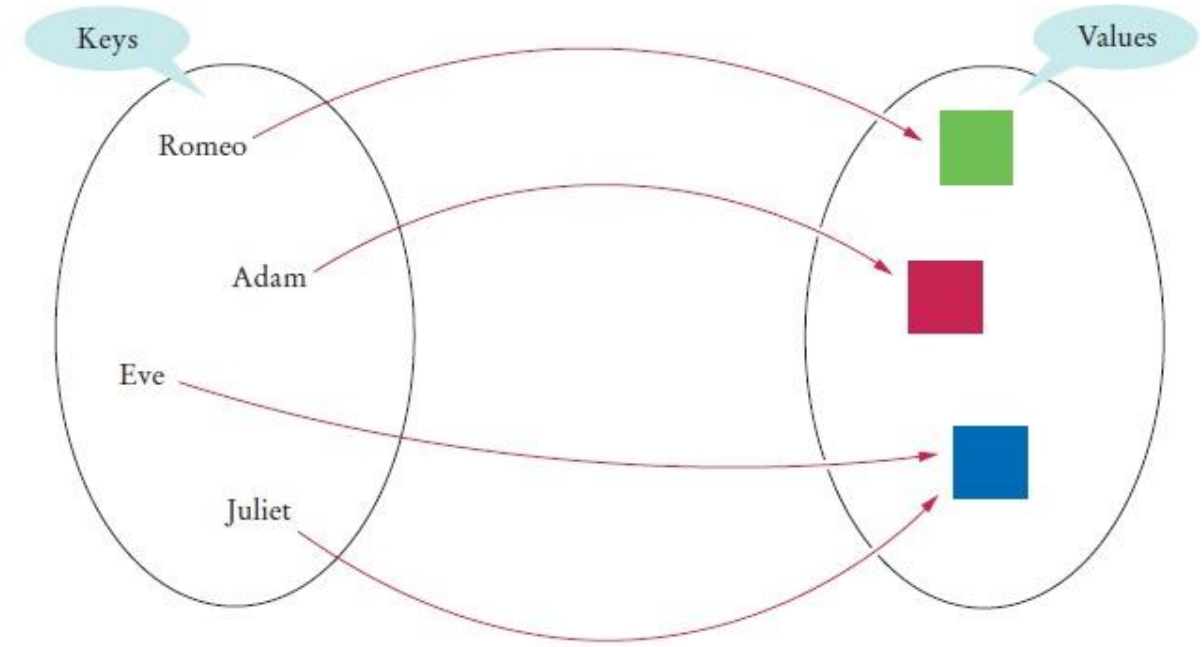
Write a program to create set with given elements order and then add two more elements to set then sort the data and move to another set and check if the size is 7 or not if it's 7 add the two more elements to the sorted set and print the set otherwise remove the two elements from set and print the set.

```
public class set
{
	public static void main(Stirng args[])
	{
		Set<Integer> a=-new Hashset(3,1,2,4,5);
		a.add(6);
		a.add(7);
		set<Integer>b=new Traceset<Integer>(a);
		int k=b.size();
		if(k==7)
		{
			b.add(8);
			b.add(9);
		}
		else
		{
			b.remove();
			b.remove();
		}
		System.out.println(b);
	}
}
```

Output: [1,2,3,4,5,6,7,8,9]

Maps:

- A map is a data type that keeps associations between keys and values.
- 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.



In Map, Keys must be a unique.

Values may be a duplicate.

- Java library has two implementations for maps: HashMap and TreeMap.

Syntax:

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

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

- **HashMap** is no ordering on keys or values.
- **TreeMap** is ordered by the key.

Methods:

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
boolean containsKey(Object k)	Returns true if the invoking map contains k as a key. Otherwise, returns false.
boolean containsValue(Object v)	Returns true if the map contains v as a value. Otherwise, returns false.
Set entrySet( )	Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. This method provides a set-view of the invoking map.
boolean equals(Object obj)	Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
Object get(Object k)	Returns the value associated with the key k.
boolean isEmpty( )	Returns true if the invoking map is empty. Otherwise, returns false.
Set keySet( )	Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
Object put(Object k, Object v)	Puts an entry in the invoking map, overwriting any previous value associated with the key.
void putAll(Map m)	Puts all the entries from m into this map.
Object remove(Object k)	Removes the entry whose key equals k.
int size( )	Returns the number of key/value pairs in the map.

Collection values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Example Program:

```
import java.util.*;
public class CollectionsDemo
{
        public static void main(String[] args)
        {
                Map m1 = new HashMap();
                m1.put("Zara", "8");
                m1.put("Mahnaz", "31");
                m1.put("Ayan", "12");
                m1.put("Daisy", "14");
                System.out.println(" Map Elements");
                System.out.print("\t" + m1);
        }
}
```

Output:

Map Elements

{Daisy = 14, Ayan = 12, Zara = 8, Mahnaz = 31}

Generic Classes and Type Parameters:

- **Generic programming** is the creation of programming constructs that can be used with many different types.
- Java library programmers who implemented the ArrayList class used the technique of 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 that we implemented in previous section is also an example of generic programming—you can store objects of any class inside a LinkedList.
- A generic class has one or more type parameters.
- When declaring a generic class, you specify a type variable for each type parameter.
- Java library declares the ArrayList class, using the type Variable E for the element type

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

- Here, E is a type variable.
- In order to use a generic class, you need to *Instantiate* the type parameter, that is, supply an actual type.
    - ArrayList<BankAccount>
    - ArrayList<Measurable>

- you cannot substitute any of the eight primitive types for a type parameter. It would be an error to declare an
    - ArrayList<double>
- Use the corresponding wrapper class instead, such as
    - ArrayList<Double>

- For Ex: If we decalre ArrayList<BankAccount> has the type variable E replaced with the type BankAccount
    - public void add(BankAccount element)

Implementing Generic Types:

- 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();
- Some people find it simpler to start out with a regular class, choosing some actual types instead of the type parameters.

```
public class Pair // Here we start out with a pair of String and Integer values
{
        private String first;
        private Integer second;
        public Pair(String firstElement, Integer secondElement)
        {
                first = firstElement;
                second = secondElement;
        }
        public String getFirst()
         {
                 return first;
         }
        public Integer getSecond()
        {
                return second;
```

```
        }
}
```

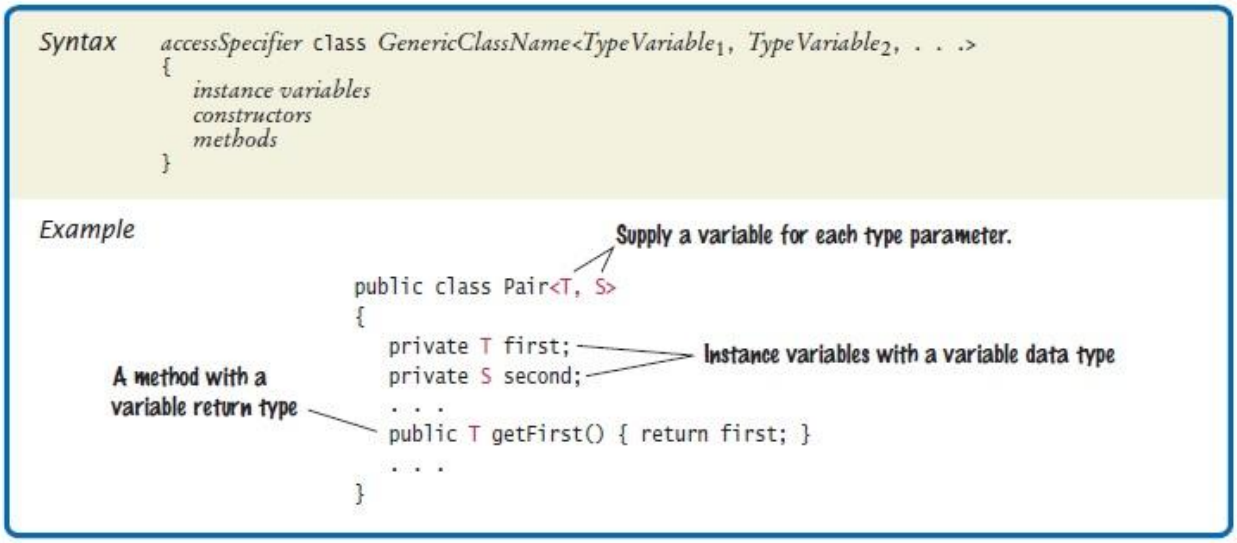
- The generic Pair class requires two type parameters, one for the type of the first element and one for the type of the second element.
- We need to choose variables for the type parameters. It is considered good form to use short uppercase names for type variables.
- You place the type variables for a generic class after the class name,
    - enclosed in angle brackets (< and >):
    - public class Pair<T, S>

Type Variables:

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

- When you declare the instance variables and methods of the Pair class, use the variable T for the first element type and S for the second element type:

Declaring a Generic Class:



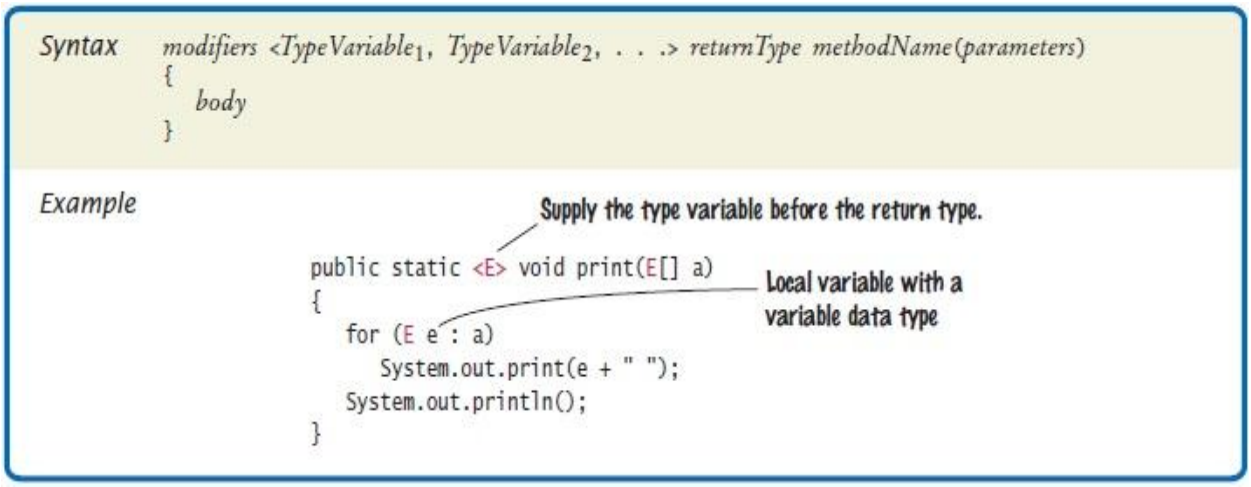
Generic class Example:

```
public class Pair<T, S>
{
        private T first;
        private S second;
        public Pair(T firstElement, S secondElement)
        {
                first = firstElement;
                second = secondElement;
        }
        public T getFirst()
        {
                return first;
        }
        public S getSecond()
        {
                return second;
        }
}
```

Generic Methods:

- A generic method is a method with a type parameter.
- You can think of it as a template for a set of methods that differ only by one or more types.
- For example, we may want to declare a method that can print an array of any type.

Declaring a Generic Method:



Generic Method Example:

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

Wrapper Class:

- A Wrapper class is a class whose object wraps or contains primitive data types.
- Wrapper classes are used for converting primitive data types into objects.

Need of Wrapper Classes:

- They convert primitive data types into objects.
- 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 objects (reference types) and not primitive types.

Primitive Data types and their Corresponding Wrapper classes:

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

Autoboxing:

- Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing.
- For example – conversion of int to Integer, long to Long, double to Double etc.

```
public class JavaExample{
                public static void main(String args[])
                {
                 object int num=100;
                 Integer obj=Integer.valueOf(num);     System.out.println(num+ " "+ obj);
                 }
}
```

Output: 100 100

Unboxing:

- It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing.
- For example – conversion of Integer to int, Long to long, Double to double, etc.

```
public class JavaExample
{
                 public static void main(String args[])
                 {
                         Integer obj = new Integer(100);
                         int num = obj.intValue();
                System.out.println(num+ " "+ obj);
                 }
}
```

Output: 100 100

Serialization:

- Serialization in Java is a mechanism of writing the state of an object into a byte-stream.
- The reverse operation of serialization is called deserialization where byte-stream is converted into an object.
- The serialization and deserialization process is platform-independent, it means you can serialize an object in a platform and deserialize in different platform.
- For serializing the object, we call the **writeObject()** method ObjectOutputStream.
- For deserialization we call the **readObject()** method of ObjectInputStream.

ObjectOutputStream class:

- The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

Constructor:

public ObjectOutputStream(OutputStream out) throws IOException {}

creates an ObjectOutputStream that writes to the specified OutputStream.

Methods:

Method	Description
public final void writeObject(Object obj) throws IOException {}	writes the specified object to the ObjectOutputStream.
public void flush() throws IOException {}	flushes the current output stream.
public void close() throws IOException {}	closes the current output stream.

Example of Java Serialization:

```
import java.io.*;
class Persist{
        public static void main(String args[])
        {
        try
        {
             Student s1 =new Student(211,"ravi");
             FileOutputStream fout=new FileOutputStream("f.txt");
            ObjectOutputStream out=new ObjectOutputStream(fout);
            out.writeObject(s1);
            out.flush();
            out.close();
            System.out.println("success");
         }catch(Exception e)
        {
            System.out.println(e);
        }
        }
}
```

Output: success

ObjectInputStream class:

- An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Constructor:

public ObjectInputStream(InputStream in) throws IOException {}

creates an ObjectInputStream that reads from the specified InputStream.

Methods:

Method	Description
public final Object readObject() throws IOException, ClassNotFoundException{}	**reads an object from the input stream.**
public void close() throws IOException {}	**closes ObjectInputStream.**

Example of Java Deserialization:

```
import java.io.*;
class Depersist{
        public static void main(String args[]){
        try{
          ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
          Student s=(Student)in.readObject();
          System.out.println(s.id+" "+s.name);
          in.close();
         }catch(Exception e){System.out.println(e);}
        }
}
```

Output: 211 ravi