

# UNIT - 1

## DATA STRUCTURES IN JAVA

### INTRODUCTION TO DATA STRUCTURES:

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.

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.

### 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. They are known as Primitive Data Structures.

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.

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 1.1.

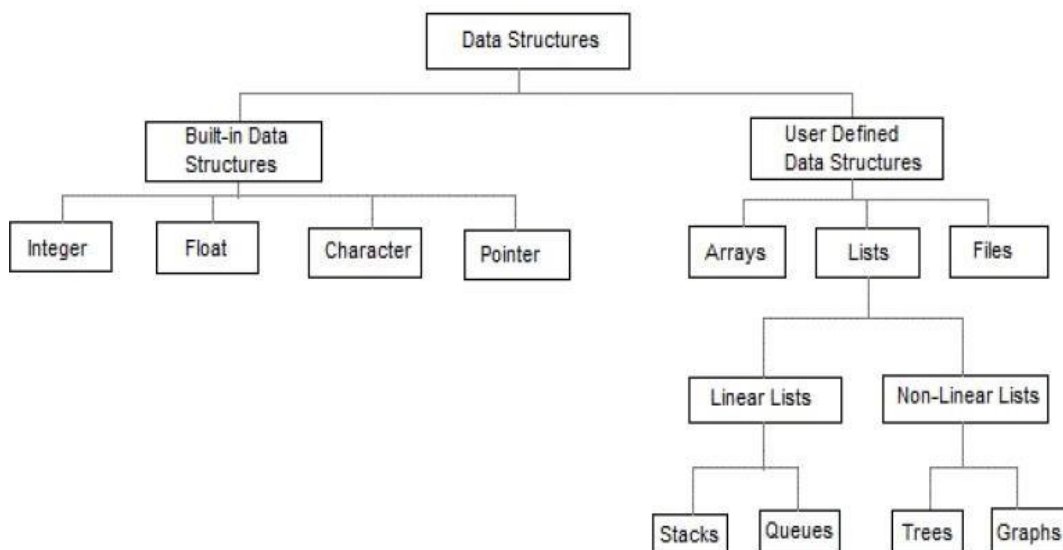
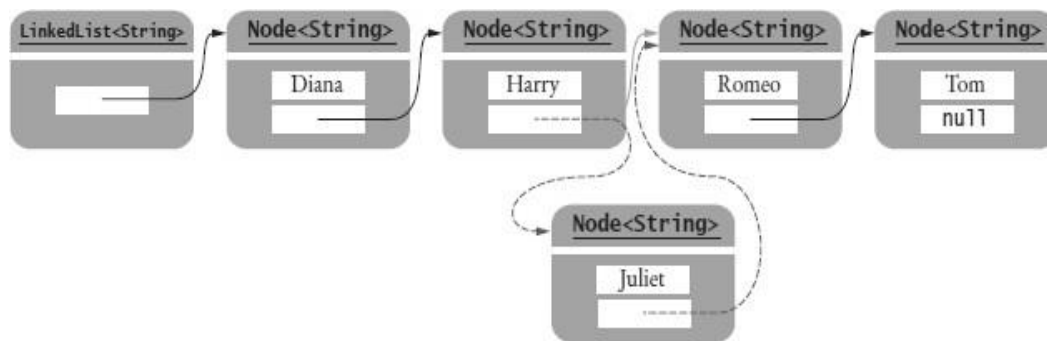


Fig.1.1: Classification of Data Structures

## **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.

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 1.2.



*Fig.1.2: Inserting an element in a Linked List*

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:

- **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.
- **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.

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>

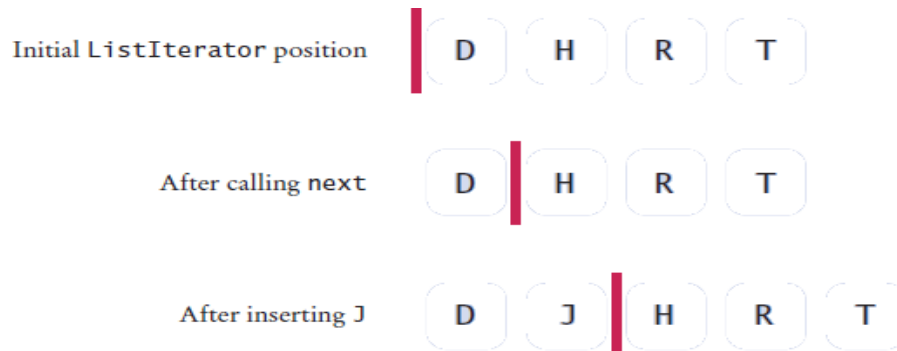
There are several methods for accessing elements in the LinkedList. The following Table 1.1 shows the different methods of LinkedList.

*Table.1.1: Linked List Methods*

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

- **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 shown in Fig.1.3 and the below Table 1.2 states the different methods of the ListIterator interface.



*Fig.1.3: Conceptual View of the ListIterator*

*Table.1.2: ListIterator Methods*

<code>String s = iter.next();</code>	Assume that <code>iter</code> points to the beginning of the list [Sally] before calling <code>next</code> . After the call, <code>s</code> is "Sally" and the iterator points to the end.
<code>iter.hasNext()</code>	Returns false because the iterator is at the end of the collection.
<pre>if (iter.hasPrevious()) {     s = iter.previous(); }</pre>	<code>hasPrevious</code> returns true because the iterator is not at the beginning of the list.
<code>iter.add("Diana");</code>	Adds an element before the iterator position. The list is now [Diana, Sally].
<pre>iter.next(); iter.remove();</pre>	<code>remove</code> removes the last element returned by <code>next</code> or <code>previous</code> . The list is again [Diana].

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 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  
Expected: Diana Harry Juliet Nina Tom

**Implementing 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 the add and remove operations and how the iterator traverses the list.

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, 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.

```

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.

The implementation of linked list will be shown by a Java program in **Lab Exercise** programs.

**Abstract Data Types:**

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

- i) 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.
- ii) 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.

## **STACKS AND QUEUES:**

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

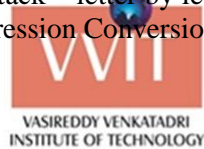
### **Stack:**

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:

- i) Stack is an ordered list of similar data type.
- ii) Stack is a LIFO structure (Last In – First Out).
- iii) push( ) function is used to insert new elements into the stack and pop( ) is used to delete an element from the stack.
- iv) 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.)

### **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:

- i) Like stack, queue is also an ordered list of elements of similar data types.
- ii) Queue is a FIFO structure.
- iii) 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.
- iv) 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.

The following Table 1.3 shows, how to use the Stack and Queue methods in Java.

*Table.1.3: Stack & Queue Methods / Functions*

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

## **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.

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:

- i) Adding an element.
- ii) Removing an element.
- iii) Locating an element. (if the set contain a given object)
- iv) Listing all elements. (not necessarily in the order in which they were added)

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 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). To handle these operations in a set as quick manner, there are two different data structures. They are 'hash tables' and 'trees'.

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 Fig. 1.4 show this hierarchy.

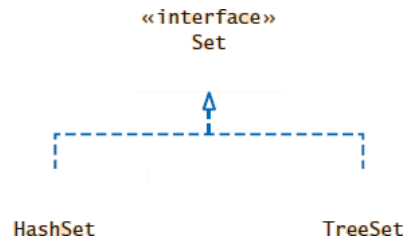


Fig.1.4: Conceptual View of the ListIterator

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>>`.

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> ();
  
```

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.

## MAPs:

A `Map` is a data type that keeps associates between 'keys' and 'values'. The following Fig. 1.5 shows a typical example: a map that associates names with colors. This map might describe the favorite colors of various people.

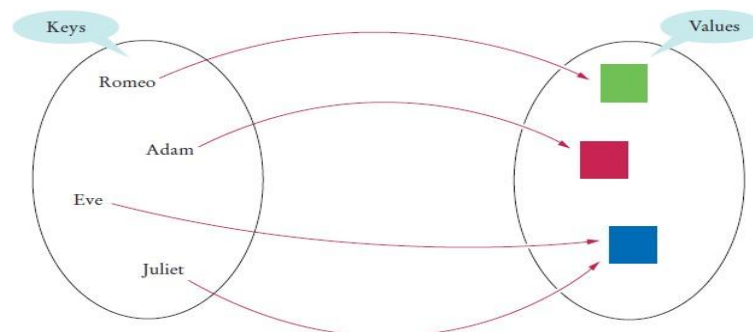




Fig.1.5: A Map

From the above example Fig. 1.5, 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 Fig. 1.6. 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.

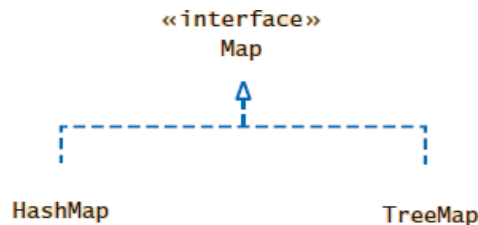



Fig.1.6: Map Classes & Interfaces in Standard Library

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> ();
```

- 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.

## **GENERIC CLASSES AND TYPE PARAMETERS:**

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.

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

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

```
ArrayList<BankAccount>
ArrayList<Measurable>
```

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.

```
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.

```
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.
```

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

## **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 ( );
```

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. We need to choose variables for the type parameters. It is considered good form to use short uppercase names for type variables shown in below Table 1.4:

*Table.1.4: Type Variables & Meaning*

<u>Type Variable</u>	<u>Meaning</u>
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

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.

## **GENERIC METHODS:**

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 by one or more types. For example, we may want to declare a method that can print an array of any type:

```
public class ArrayUtil
{
    /** Prints all elements in an array.
     * @ param a the array to print
     */
    Public <T> static void print (T[ ] a)
```

```

    {
        ...
    }
    ...
}

```

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 ( );
    }
    ...
}

```

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();
}

```

When we want to call the generic method, we need not specify which type to use for the type parameter. So, generic methods 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).

## **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). 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. 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 introduced the concept of wrapper classes.

## **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. 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.

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.

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.

#### 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 1.5 shows the 8 primitive data types and corresponding 8 wrapper classes.

Table 1.5: Wrapper Classes for 8 Primitive Data Types

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

And the following Figure.1.7 shows the Hierarchy of the above Wrapper Classes.

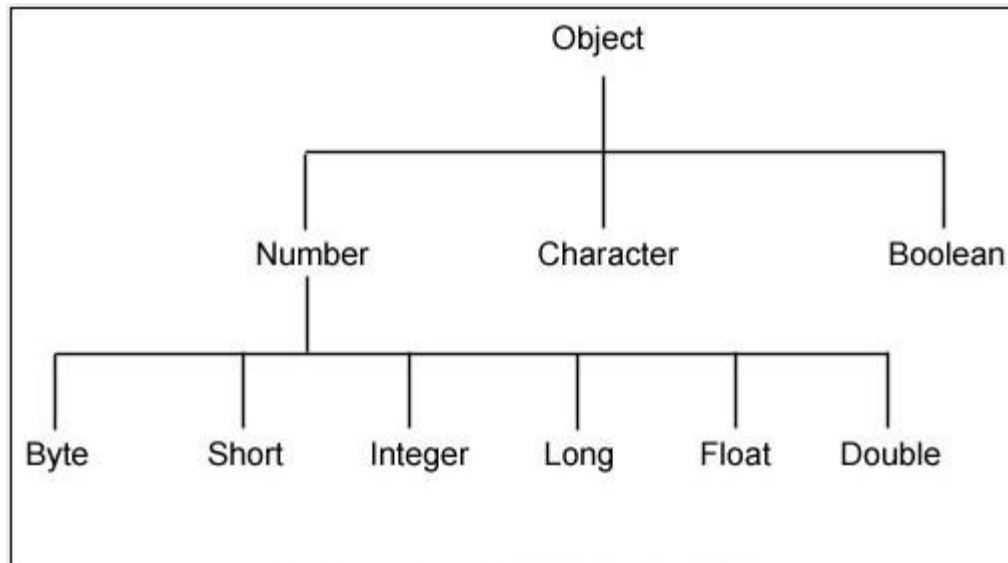


Fig.1.7: Wrapper Classes Hierarchy

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.

### **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.

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.

```

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;

        // 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);

// 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);
}
}

```



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

```

## **CONCEPT OF SERIALIZATION:**

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.

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. 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.

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.

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);
    }
}
```

In the above code, a class to be serialized successfully, 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].



### **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.

```
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 = 521180;
        e.number = 2016;
        try
        {
            FileOutputStream fileOut = new
FileOutputStream("/tmp/employee.ser");
            ObjectOutputStream out = new ObjectOutputStream(fileOut);
            out.writeObject(e);
            out.close();
            fileOut.close();
            System.out.printf("Serialized data is saved in /tmp/employee.ser");
        } catch(IOException i)
        {
            i.printStackTrace();
        }
    }
}
```

### **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();
        }
    }
}
```

```

        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: 2016



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`.
- ii) Notice that the return value of `readObject()` is cast to an `Employee` reference.
- iii) The value of the SSN field was 11122333 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.