UNIT 1

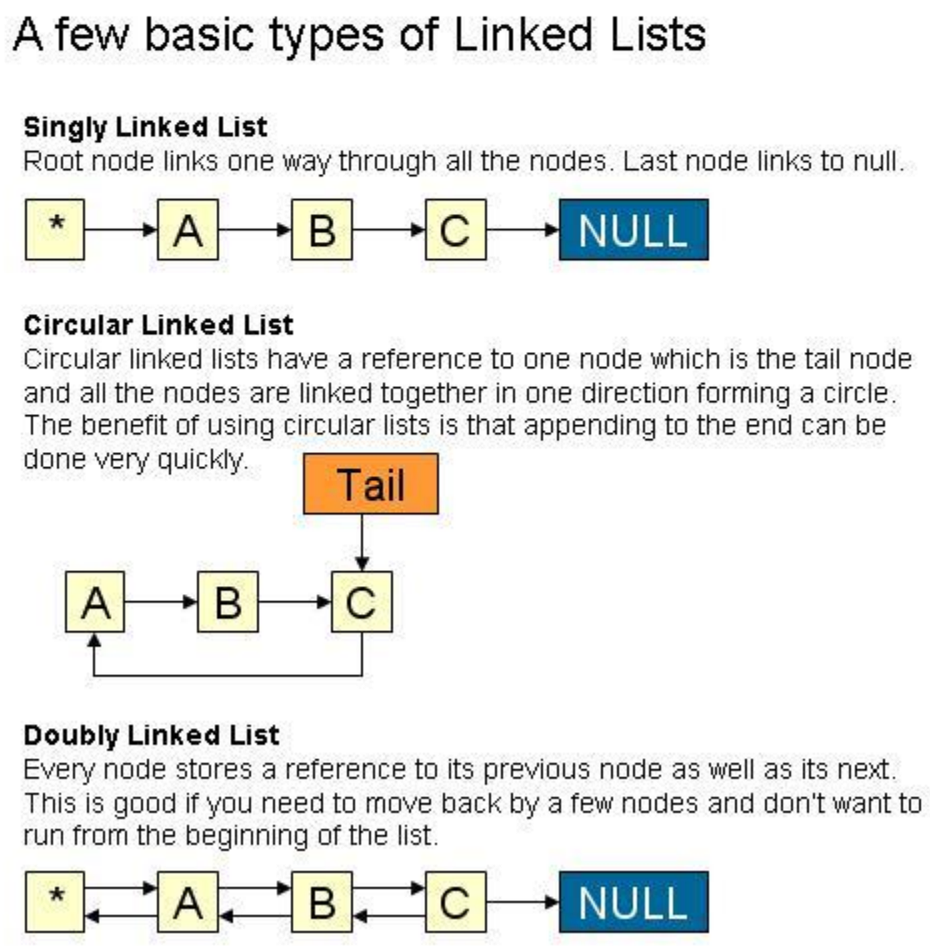
Hadoop and Big Data

1 Data structures in Java:

- Linked List:

Linked Lists are a very common way of storing arrays of data. The major benefit of linked lists is that you do not specify a fixed size for your list. The more elements you add to the chain, the bigger the chain gets.

There is more than one type of a linked list, although for the purpose of this tutorial, we'll stick to singly linked lists (the simplest one). If for example you want a doubly linked list instead, very few simple modifications will give you what you're looking for. Many data structures (e.g. Stacks, Queues, Binary Trees) are often implemented using the concept of linked lists. Some different types of linked lists are shown below


A few basic types of Linked Lists
Singly Linked List
Root node links one way through all the nodes. Last node links to null.
* A B C NULL
Circular Linked List
Circular linked lists have a reference to one node which is the tail node and all the nodes are linked together in one direction forming a circle. The benefit of using circular lists is that appending to the end can be done very quickly.
Tail
A B C
Doubly Linked List
Every node stores a reference to its previous node as well as its next. This is good if you need to move back by a few nodes and don't want to run from the beginning of the list.
* A B C NULL

The following java program demonstrate about implementation of linked list in java

```
import java.util.Scanner
/* Class Node */
class Node
{
    protected int data;
    protected Node link;

    public Node()

    {link = null; data =
                0;
    }

    public Node(int d,Node n)
    {data = d;
               link = n;
    }
    public void setLink(Node n)
    {
        link = n;
    }

    /* Function to set data to current Node */
    public void setData(int d)
    {
        data = d;
    }
    /* Function to get link to next node */
    public Node getLink()
    {
        return link;
    }
    /* Function to get data from current Node */
    public int getData()
    {
        return data;
    }
}
```

/* Class linkedList */

```
class linkedList

{protected Node start; protected
      Node end ;
    public int size ;

    /* Constructor */
    public linkedList()
```

```
{start = null; end =
            null;
            size = 0;
}
public boolean isEmpty()
{return start == null;
}
/* Function to get size of list */
public int getSize()
{return size;
}
/* Function to insert an element at begining */
public void insertAtStart(int val)
{Node nptr = new Node(val, null);
            size++ ;
   if(start == null)
   {start = nptr;
               end = start;
   }
   else
   {
      nptr.setLink(start);
      start = nptr;
   }
}
public void insertAtEnd(int val)
{Node nptr = new Node(val,null); size++
            ;
   if(start == null)
   {
     start = nptr;
      end = start;
   }
   else
   {
      end.setLink(nptr);
      end = nptr;
   }
}
public void insertAtPos(int val , int pos)
{
   Node nptr = new Node(val, null);
   Node ptr = start;
   pos = pos - 1 ;
   for (int i = 1; i < size; i++)
   {
      if (i == pos)
      {Node tmp = ptr.getLink() ;
                ptr.setLink(nptr);
                nptr.setLink(tmp);
         break;
       }
      ptr = ptr.getLink();
```

```
            }
            size++ ;
        }
        /* Function to delete an element at position */
        public void deleteAtPos(int pos) {

            if (pos == 1)
            {start = start.getLink();
                             size--;
                        return ;
            }
            if (pos == size)
            {Node s = start;
                        Node t = start;
                while (s != end)
                {t = s;
                            s = s.getLink();
                }
                end = t;
                end.setLink(null);
                size --;
                return;
    }
            Node ptr = start;
            pos = pos - 1 ;
            for (int i = 1; i < size - 1; i++)
            {
                if (i == pos)
                {Node tmp = ptr.getLink(); tmp =
                    tmp.getLink(); ptr.setLink(tmp);

                    break;
               }
                ptr = ptr.getLink();
            }
            size-- ;

        }

       public void display()
        {
            System.out.print("\nSingly Linked List = ");
            if (size == 0)
            {
                System.out.print("empty\n");
                return;
            }
            if (start.getLink() == null)
            {
                System.out.println(start.getData() );
                return;
            }
            Node ptr = start;
        System.out.print(start.getData()+ "->");
        ptr = start.getLink();
        while (ptr.getLink() != null)
        {
            System.out.print(ptr.getData()+ "->");
            ptr = ptr.getLink();
        }
        System.out.print(ptr.getData()+ "\n");
    }
}

/* Class SinglyLinkedList */
public class SinglyLinkedList
{
    public static void main(String[] args)
    {
        Scanner scan = new Scanner(System.in); /*
        Creating object of class linkedList */ linkedList
        list = new linkedList();
        System.out.println("Singly Linked List
        Test\n"); char ch;
        /* Perform list operations */
        do
        {
            System.out.println("\nSingly Linked List Operations\n");
            System.out.println("1. insert at begining");
            System.out.println("2. insert at end");
            System.out.println("3. insert at position");
            System.out.println("4. delete at position");
            System.out.println("5. check empty");
            System.out.println("6. get size"); int choice =
            scan.nextInt();
            switch (choice)
            {
            case 1 :
                System.out.println("Enter integer element to insert");
                list.insertAtStart( scan.nextInt() ); break;

            case 2 :
                System.out.println("Enter integer element to insert");
                list.insertAtEnd( scan.nextInt() ); break;

            case 3 :
                System.out.println("Enter integer element to insert");
                int num = scan.nextInt() ;
                System.out.println("Enter position");
                int pos = scan.nextInt() ;
                if (pos <= 1 || pos > list.getSize() )
                    System.out.println("Invalid position\n");
                else
                    list.insertAtPos(num, pos);
                break;
            case 4 :
                System.out.println("Enter position");
                int p = scan.nextInt() ;
                if (p < 1 || p > list.getSize() )
                    System.out.println("Invalid position\n");
                else
                    list.deleteAtPos(p);
                break;
            case 5 :
                System.out.println("Empty status = "+
                list.isEmpty()); break;
            case 6 :
                System.out.println("Size = "+ list.getSize() +" \n");
                break;
             default :
                System.out.println("Wrong Entry \n ");
                break;
            }
            /* Display List */
            list.display();
           System.out.println("\nDo you want to continue (Type y or n) \n");
            ch = scan.next().charAt(0);
        } while (ch != 'N'|| ch != 'n');
    }
}
```

- **Stacks**

A **stack** is a data structure that allows data to be inserted (a 'push' operation), and removed (a 'pop' operation). Many **stacks** also support a read ahead (a 'peek' operation), which reads data without removing it. A **stack** is a LIFO-queue, meaning that the last data to be inserted will be the first data to be removed.

Implementation of stack in java:

```
public class MyStack {
   private int maxSize;
   private long[] stackArray;
   private int top;
   public MyStack(int s) {
      maxSize = s;
      stackArray = new long[maxSize];
      top = -1;
   }
   public void push(long j) {
      stackArray[++top] = j;
   }
   public long pop() {
      return stackArray[top--];
   }
   public long peek() {
      return stackArray[top];
```

```
  }
  public boolean isEmpty() {
    return (top == -1);
  }
  public boolean isFull() {
    return (top == maxSize - 1);
  }
  public static void main(String[] args) {
    MyStack theStack = new MyStack(10);
    theStack.push(10);
    theStack.push(20);
    theStack.push(30);
    theStack.push(40);
    theStack.push(50);
    while (!theStack.isEmpty()) {
      long value = theStack.pop();
      System.out.print(value);
      System.out.print(" ");
    }
    System.out.println("");
  }
}
```

- **Queues**

Queue collection implements First-In Fist-Out behavior on items stored within it.

```
import java.util.*;
public class QueueDemo {
  static String newLine = System.getProperty("line.separator");
  public static void main(String[] args) {
  System.out.println(newLine + "Queue in Java" + newLine);
  System.out.println("-----------------------" + newLine);
   System.out.println("Adding items to the Queue" + newLine);
   //Creating queue would require you to create instannce of LinkedList and
   assign //it to Queue
   //Object. You cannot create an instance of Queue as it is
   abstract Queue queue = new LinkedList();

   //you add elements to queue using add method
   queue.add("Java");
   queue.add(".NET");

   queue.add("Javascript");
   queue.add("HTML5");
   queue.add("Hadoop");

   System.out.println(newLine + "Items in the queue..." + queue + newLine);

   //You remove element from the queue using .remove method //This would
   remove the first element added to the queue, here Java
   System.out.println("remove element: " + queue.remove() + newLine);

   //.element() returns the current element in the queue, here when "java" is removed
```

```
    //the next most top element is .NET, so .NET would be printed.
    System.out.println("retrieve element: " + queue.element() + newLine);

    //.poll() method retrieves and removes the head of this queue
    //or return null if this queue is empty. Here .NET would be printed and then would
    //be removed
    //from the queue
    System.out.println("remove and retrieve element, null if empty: " + queue.poll() +
    newLine);

    //.peek() just returns the current element in the queue, null if empty //Here it
    will print Javascript as .NET is removed above System.out.println("retrieve
    element, null is empty " + queue.peek() + newLine);
  }
}
```

- **Sets**

Set ia a collection of elements. In Java HashSet class is used for maintaining Set type data.

The following program implements the set data structure.

```
public class Family {

    public static void main(String[] args)
    {

        HashSet hashset = new HashSet();
        hashset.add(3);
        hashset.add("srinivas");
        hashset.add("shoba");
             hashset.add("srithan sai");
             hashset.add("sai krithik");
        System.out.println("Set is "+hashset);
    }
}
```
Output:

- **Maps**

The Map interface maps unique keys to values. A key is an object that you use to retrieve a value at a later date.

Given a key and a value, you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.

- Several methods throw a NoSuchElementException when no items exist in the invoking map.

- A ClassCastException is thrown when an object is incompatible with the elements in a map.

- A NullPointerException is thrown if an attempt is made to use a null object and null is not allowed in the map.

- An UnsupportedOperationException is thrown when an attempt is made to change an unmodifiable map.

Example:

Map has its implementation in various classes like HashMap. Following is the example to explain map functionality:

```
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();
     System.out.println(" Map Elements");
     System.out.print("\t" + m1);
  }
}
```

This would produce the following result:

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

2 Generics:

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.

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 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:

- 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).

- 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.

- 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.

- 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).

Example:

Following example illustrates how we can print 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
   }
}
```

This would produce the following result:

```
Array integerArray contains:
123456

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
HELLO
```

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.

Example:

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:

```
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( "Maxm 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" ) );
  }
}
```

This would produce the following result:

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

maximum of 6.6, 8.8 and 7.7 is 8.8

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

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.

Example:

Following example illustrates how we can define a generic class:

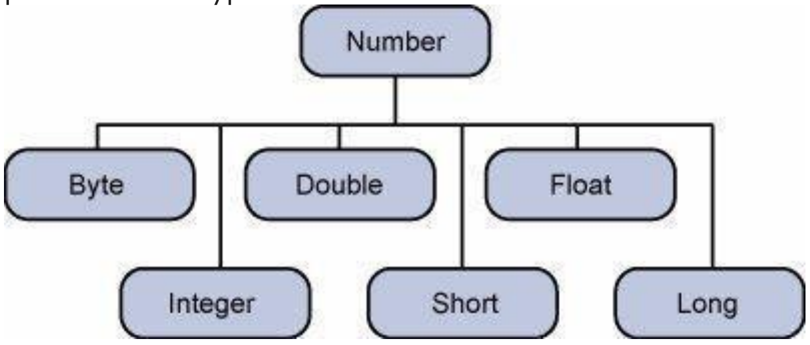
```
public class Box<T> {


  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());
  }
}
```

This would produce the following result:

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

3 Wrapper Classes

All the **wrapper classes** (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract **class**Number. The object of the **wrapper class** contains or wraps its respective primitive data type.



Wrapper class in java provides the mechanism *to convert primitive into object and object into primitive*.

Since J2SE 5.0, **autoboxing** and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known and autoboxing and vice-versa unboxing.

One of the eight classes of *java.lang* package are known as wrapper class in java. The list of eight wrapper classes are given below:

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

Wrapper class Example: Primitive to Wrapper

```
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
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

```
20 20 20
```

Wrapper class Example: Wrapper to Primitive

```
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
int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

```
3 3 3
```

4 Concept of Serialization

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.

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
```

This 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, I am going to use the Student class that we discussed early on in the book. Suppose that we have the following Student class, which implements the Serializable interface:

```
public class Student 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);
  }
}
```

Notice that for a class to be serialized successfully, two conditions must be met:

- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object:

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Student object and serializes it to a file.

When the program is done executing, a file named Student.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)
   {
      Student e = new Student();
      e.name = "srinivas";
      e.address = "ibm vijayawada";
      e.SSN = 11122333;
      e.number = 101;

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

Deserializing an Object:

The following DeserializeDemo program deserializes the Student object created in the SerializeDemo program. Study the program and try to determine its output:

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

```
  {
     Student e = null;
     try
     {
        FileInputStream fileIn = new
        FileInputStream("/tmp/Student.ser"); ObjectInputStream in = new
        ObjectInputStream(fileIn); e = (Student) in.readObject();
        in.close();
        fileIn.close();
     }catch(IOException i)
     {
        i.printStackTrace();
        return;
     }catch(ClassNotFoundException c)
     {
        System.out.println("Student class not found");
        c.printStackTrace();
        return;
     }
     System.out.println("Deserialized Student...");
     System.out.println("Name: " + e.name);
     System.out.println("Address: " + e.address);
     System.out.println("SSN: " + e.SSN);
     System.out.println("Number: " + e.number);
   }
}
```

This would produce the following result:

```
Deserialized Student...
Name: srinivas
Address:ibm,vijayawada
SSN: 0
Number:101
```

Here are following important points to be noted:

- 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.

- Notice that the return value of readObject() is cast to an Student reference.

- 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 Student object is 0.