**Wrapper Classes**

**What Is the use of wrapper classes?**

|  |
| --- |
| 1. Wrapper classes are used to convert primitive types to objects and objects to primitive types. |

**What is boxing?**

|  |
| --- |
| 1. Process of converting primitive type to object is called as boxing |

**Example on boxing**

|  |
| --- |
| **package** p1;  **public** **class** WrapperDemo1  {  **public** **static** **void** main(String[] args)  {  **int** a=100; //Integer  //a is a variable of type int  //int is a primitive data type  Integer io=**new** ~~Integer~~(a);  System.***out***.println("a:\t"+a);  System.***out***.println("io:\t"+io.toString());  }  }  **Output**  a: 100  io: 100 |

**AutoBoxing**

|  |
| --- |
| 1. This concept was introduced in java 1.9 version 2. If boxing is done by the automatically then it is called as autobxing   **package** p1;  **public** **class** WrapperDemo1  {  **public** **static** **void** main(String[] args)  { **int** a=100;  Integer io=a; //Auto Boxing  System.***out***.println("a:\t"+a);  System.***out***.println("io:\t"+io.toString());  }  }  **Example on autoboxing**  a: 100  io: 100 |

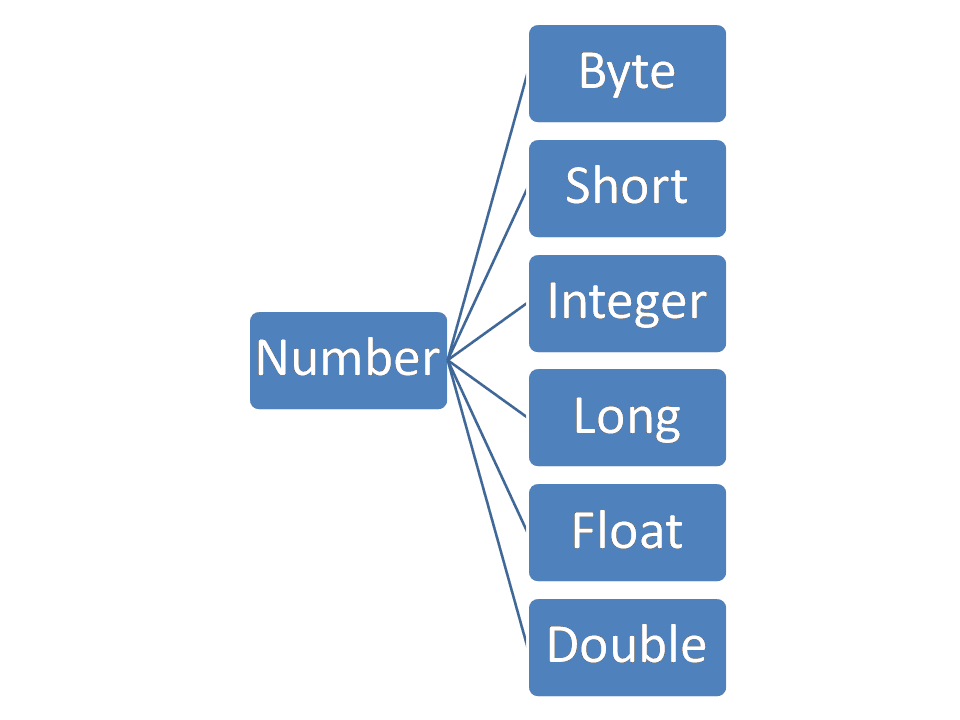
**How many wrapper classes are there in Java?**

|  |
| --- |
| In java every primitive type has a corresponding class  In java we have 8 wrapper classes |

|  |  |
| --- | --- |
| **Primitive type** | **Wrapper class** |
| 1. **byte** | 1. **java.lang.Byte** |
| 1. **short** | 1. **java.lang.Short** |
| 1. **int** | 1. **java.lang.Integer** |
| 1. **long** | 1. **java.lang.Long** |
| 1. **float** | 1. **java.lang.Float** |
| 1. **double** | 1. **java.lang.Double** |
| 1. **char** | 1. **java.lang.Character** |
| 1. **boolean** | 1. **java.lang.Boolean** |

**Number class**

The Number class is the base class for Byte, Short, Integer, Long, Float, Double



**The methods existed in Number classes?**

|  |
| --- |
| 1. Because Number class is abstract class, it contains some abstract methods and some implemented methods.   The list of methods are  Compiled from "Number.java"  public abstract class java.lang.Number implements java.io.Serializable {  public java.lang.Number();  public abstract int intValue();  public abstract long longValue();  public abstract float floatValue();  public abstract double doubleValue();  public byte byteValue();  public short shortValue();  } |

**Unboxing**

|  |
| --- |
| The process of converting object to primitive types is called as Unboxing |

**Example**

|  |
| --- |
| **package** p1;  **public** **class** WrapperDemo1  {  **public** **static** **void** main(String[] args)  { **int** a=100;  Integer io=a;  //the process of converting object into primitive type is called as Un-boxing  //all the methods of Number class can be used to convert objects to primitive types.  **byte** b1=io.byteValue();  **short** s1=io.shortValue();  **int** iv1=io.intValue();  **long** lv=io.longValue();  **float** fv=io.floatValue();  **double** dv=io.doubleValue();  System.***out***.println("b1:\t"+b1);  System.***out***.println("s1:\t"+s1);  System.***out***.println("iv1:\t"+iv1);  System.***out***.println("lv:\t"+lv);  System.***out***.println("fv:\t"+fv);  System.***out***.println("dv:\t"+dv);    }  }  **Output:**  b1: 100  s1: 100  iv1: 100  lv: 100  fv: 100.0  dv: 100.0 |

**AutoUnBoxing**

|  |
| --- |
| If unboxing is performed automatically without calling Number class methods, then it is called as Auto Unboxing  Example on Unboxing  **package** p1;  **public** **class** WrapperDemo1  {  **public** **static** **void** main(String[] args)  { **int** a=100;  Integer io=a;  //Auto Un-Boxing  **int** x=io;  System.***out***.println("a:\t"+a);  System.***out***.println("io:\t"+io);  System.***out***.println("x:\t"+x);    }  }  **Output:**  a: 100  io: 100  x: 100 |

**Where we use Wrapper classes?**

|  |
| --- |
| 1. We use wrapper classes in collection classes, because in Collection classes we can’t use a value as primitive type, we have to use them as objects. 2. In generic classes we can’t use primitive types, we can use only objects so we use wrapper classes in generic classes. 3. At the time of web application development, we use wrapper classes, because to run web applications we need server and servers only understands the values as objects. |

**PROGRAM TO FIND MAX\_VALUE AND MIN\_VALUE which we can store in Login type variable.**

|  |
| --- |
| **package** p1;  **public** **class** WrapperDemo1  {  **public** **static** **void** main(String[] args)  {  Long l1=**new** ~~Long~~(465645654);  System.***out***.println("Max value:\t"+Long.***MAX\_VALUE***);  System.***out***.println("Min value:\t"+Long.***MIN\_VALUE***);  }  }  **Output:**  Max value: 9223372036854775807  Min value: -9223372036854775808 |

**Generic Classes**

1. It is a class which act upon any type
2. It is also called as parameter type

**Example**

|  |
| --- |
| **package** p1;  **public** **class** One<T>  {  T a;  **public** One(T a) //Integer a  { **this**.a=a;  System.***out***.println("Default constructor");  }  **public** String toString()  { **return** "One@[a="+a+"]";  }  }  **Output:**  Default constructor  Default constructor  o1: One@[a=1000]  o2: One@[a=1000.5] |

**Another example on generic types**

**One.java**

|  |
| --- |
| **package** p1;  **public** **class** One<T1,T2>  {  T1 a;  T2 b;  **public** One(T1 a,T2 b) //Integer a  { **this**.a=a;  **this**.b=b;  System.***out***.println("Parameterized constructor");  }    **public** String toString()  { **return** "One@[a="+a+",b="+b+"]";  }  } |

**Main1.java**

|  |
| --- |
| **package** main;  **import** p1.\*;  **class** Emp  {  **int** eno;  String ename;  **float** sal;  **public** Emp(**int** eno, String ename, **float** sal)  { **super**();  **this**.eno = eno;  **this**.ename = ename;  **this**.sal = sal;  }  @Override  **public** String toString() {  **return** "Emp [eno=" + eno + ", ename=" + ename + ", sal=" + sal + "]";  }  }  **public** **class** Main1  {  **public** **static** **void** main(String[] args)  {    One<Integer,Integer> o1=**new** One<Integer,Integer>(100,200);  One<Float,Double> o2=**new** One<Float,Double>(1000.50f,200.00);  One<String,Emp> o3=**new** One<String,Emp>("madhu",**new** Emp(1,"madhu",200000.00f));    System.***out***.println("o1:\t"+o1);  System.***out***.println("o2:\t"+o2);  System.***out***.println("o3:\t"+o3);    }  }  **Output:**  Parameterized constructor  Parameterized constructor  Parameterized constructor  o1: One@[a=100,b=200]  o2: One@[a=1000.5,b=200.0]  o3: One@[a=madhu,b=Emp [eno=1, ename=madhu, sal=200000.0]] |

**Can we restrict the type of object we pass to the class as a parameterized type.**

**One.java**

|  |
| --- |
| **package** p1;  **public** **class** One<T1 **extends** Number,T2 **extends** Number>  {  T1 a;  T2 b;  **public** One(T1 a,T2 b) //Integer a  { **this**.a=a;  **this**.b=b;  System.***out***.println("Parameterized constructor");  }  **public** Float add()  {  **return** a.floatValue()+b.floatValue();  }  **public** String toString()  { **return** "One@[a="+a+",b="+b+"]";  }  } |

**Main1.java**

|  |
| --- |
| **package** main;  **import** p1.\*;  **class** Emp  {  **int** eno;  String ename;  **float** sal;  **public** Emp(**int** eno, String ename, **float** sal)  { **super**();  **this**.eno = eno;  **this**.ename = ename;  **this**.sal = sal;  }  @Override  **public** String toString() {  **return** "Emp [eno=" + eno + ", ename=" + ename + ", sal=" + sal + "]";  }  }  **public** **class** Main1  {  **public** **static** **void** main(String[] args)  {  One<Integer,Integer> o1=**new** One<Integer,Integer>(100,200);  One<Float,Double> o2=**new** One<Float,Double>(1000.50f,200.00);  System.***out***.println("oa:\t"+o1.add());  System.***out***.println("o2:\t"+o2.add());  System.***out***.println("o1:\t"+o1);  System.***out***.println("o2:\t"+o2);  }  }  **Output:**  Parameterized constructor  Parameterized constructor  oa: 300.0  o2: 1200.5  o1: One@[a=100,b=200]  o2: One@[a=1000.5,b=200.0] |

**Generic Methods**

**Main2.java**

|  |
| --- |
| **package** main;  **class** Demo  {  **static** <T1 **extends** Number,T2 **extends** Number>**float** add(T1 a,T2 b)  {  **return** a.floatValue()+b.floatValue();  }  }  **public** **class** Main2 {  **public** **static** **void** main(String[] args)  {  System.***out***.println(Demo.*add*(100,20));  System.***out***.println(Demo.*add*(100.50,20.25));  System.***out***.println(Demo.*add*(20.33f,20.44f));    }  }  Output:  120.0  120.75  40.77 |

**Collection Framework**

Collection framework is used to manage collection of elements easily. We can’t perform all these operations using array because It is having many drawbacks.

**What is an array?**

|  |
| --- |
| 1. Collection of similar data elements stored at contiguous memory locations.   **Advantages:**   1. Memory allocation is fast, because memory allocation will be done in subsequence memory locations. 2. Accessing fast because we access elements by using indexes.   **Drawbacks**   1. Array size is fixed, once the array is created, we can’t change its size. 2. Memory wastage 3. We can’t modify an array |

**Advantages of collection framework**

|  |
| --- |
| **Advantages**   1. We can create dynamically resizable array here 2. We can create LinkedList which are used to modify the elements easily. 3. Easy to manage elements because we are having different types of collection classes here. |

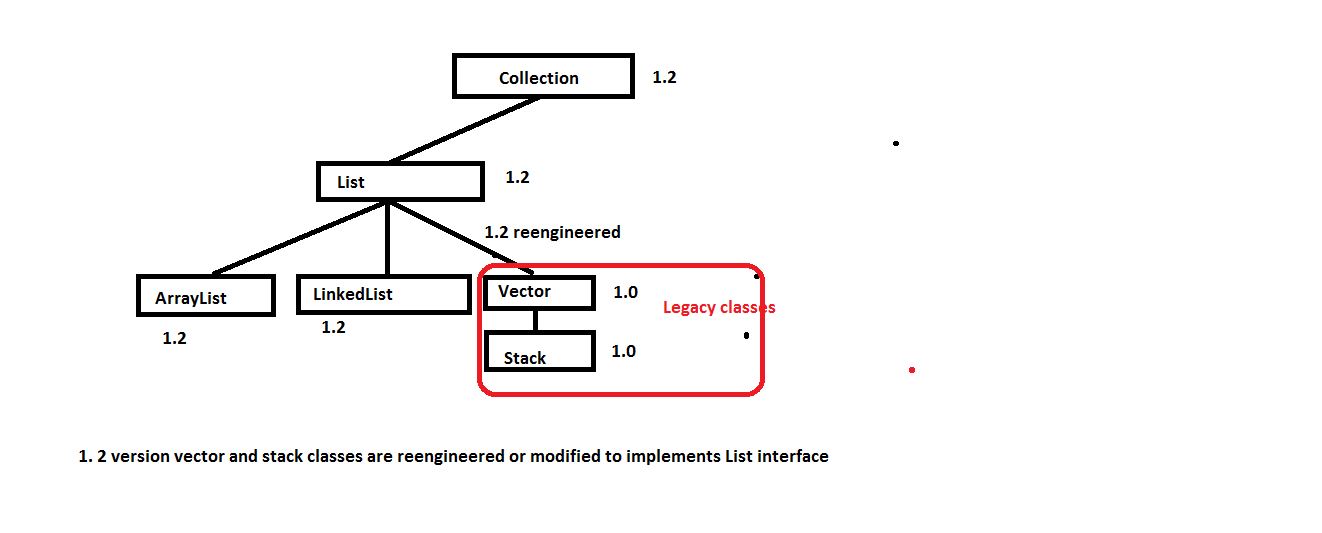
**9 key interfaces**

1. **Collection**
2. **List**
3. **Set**
4. **SortedSet**
5. **NavigableSet**
6. **Map**
7. **SortedMap**
8. **NavigableMap**
9. **Queue**

**What is a collection class?**

|  |
| --- |
| 1. It is a class which implements Collection interface 2. All collection classes are generic types |

**List (I):** It is the child interface of Collection, if we want to represent a group of individual objects as a single entity, where duplicates are allowed and insert order must be preserved then we should go for list.



**ArrayList:**

1. Array List is a dynamic array that stores objects. (Size increases when you add and shrinks when you delete).
2. It can contain duplicate elements
3. It Maintains insertion order
4. It is not synchronized( means not thread safe) i.e. this object can be accessed by any no. of threads at a time.
5. It accepts Random access because arraylist internally uses array and arrays works at the index basis.
6. It uses an array interanally to manage collection of objects.
7. Because array list internally using an array, it is best to store and retrieve elements.
8. Because lots of shifting needs to be occurred better don’t use if you want to modify this object frequently
9. We can store null values in array list

**ArrayListDemo1.java**

|  |
| --- |
| **package** main;  **import** java.util.ArrayList;  **public** **class** ArrayListDemo  {  **public** **static** **void** main(String[] args)  {  //ArrayList<Integer> l1=new ArrayList<Integer>();  **var** l1=**new** ArrayList<Integer>();  System.***out***.println("size of l1:\t"+l1.size());  l1.add(100);  System.***out***.println("size of l1:\t"+l1.size());  l1.add(200);  l1.add(300);  l1.add(400);  l1.add(500);  l1.add(600);  System.***out***.println("size of l1:\t"+l1.size());  l1.remove(**new** ~~Integer~~(100));  System.***out***.println("size of l1:\t"+l1.size());  System.***out***.println("l1:\t"+l1);  l1.set(0, 1000);  System.***out***.println("l1:\t"+l1);  l1.add(1, 2000);  System.***out***.println("l1:\t"+l1);    }  }  Output:  size of l1: 0  size of l1: 1  size of l1: 6  size of l1: 5  l1: [200, 300, 400, 500, 600]  l1: [1000, 300, 400, 500, 600]  l1: [1000, 2000, 300, 400, 500, 600] |

**How to get the elements one by one from a collection object?**

**Way-1: getting elements by using foreach**

**While accessing elements by using forEach is it possible to modify the collection object?**

|  |
| --- |
| No, we will get ConcurrentModificationException |

**While accessing elements by using Iterator object is it possible to modify the collection object?**

|  |
| --- |
| Yes, it is possible |

**Accessing elements from a collection object in different ways**

|  |
| --- |
| **package** main;  **import** java.util.ArrayList;  **import** java.util.Iterator;  **import** java.util.ListIterator;  **public** **class** ArrayListDemo  {  **public** **static** **void** main(String[] args)  { //ArrayList<Integer> l1=new ArrayList<Integer>();  **var** l1=**new** ArrayList<Integer>();  System.***out***.println("size of l1:\t"+l1.size());  l1.add(100);  System.***out***.println("size of l1:\t"+l1.size());  l1.add(200);  l1.add(300);  l1.add(400);  l1.add(500);  l1.add(600);  l1.add(500);  System.***out***.println("size of l1:\t"+l1.size());  //l1.remove(100); //deleting element from index 100  Integer io=**new** ~~Integer~~(100);  l1.remove(io); //deleting a specified element  l1.remove(**new** ~~Integer~~(100));    System.***out***.println("size of l1:\t"+l1.size());  System.***out***.println("l1:\t"+l1);  l1.set(0, 1000);  System.***out***.println("l1:\t"+l1);  l1.add(1, 2000);  System.***out***.println("l1:\t"+l1);    System.***out***.println("Integer objects existed in ArrayList..");  **for**(Integer x:l1)  { System.***out***.println(x);  }    // Iterator<Integer> io1=l1.iterator();  // System.out.println("Getting elements by using Iterator object");  // while(io1.hasNext())  // { Integer ele=io1.next();  // if(ele==500)  // { io1.remove();  // System.out.println(ele+" is removed from l1");  // }  // }  //  // System.out.println("l1:\t"+l1);    ListIterator<Integer> liter=l1.listIterator();  System.***out***.println("Getting elements by using Iterator object");  **while**(liter.hasNext())  { **int** i=liter.nextIndex();  Integer ele=liter.next();  **if**(ele==500)  { liter.remove();  System.***out***.println(ele+" is removed from index "+i);  }**else**  {System.***out***.println(ele+" is located at index "+i);    }    }  //liter.hasPrevious()  //liter.previous()  //liter.set(100);  //liter.hasNext()  //liter.next()  //liter.nextIndex()    }  }  Output:  size of l1: 0  size of l1: 1  size of l1: 7  size of l1: 6  l1: [200, 300, 400, 500, 600, 500]  l1: [1000, 300, 400, 500, 600, 500]  l1: [1000, 2000, 300, 400, 500, 600, 500]  Integer objects existed in ArrayList..  1000  2000  300  400  500  600  500  Getting elements by using Iterator object  1000 is located at index 0  2000 is located at index 1  300 is located at index 2  400 is located at index 3  500 is removed from index 4  600 is located at index 4  500 is removed from index 5 |

**Linked List**

**Linked List:**

* LinkedList class uses doubly linked list to store elements and it extends AbstractSequentialList class and implements List,Deque,Cloneable interfaces.
* It can contains duplicate elements
* Maintain Insertion Order
* It is not synchronized
* No random access
* Manipulation Fast because no shifting needs to be occurred
* It can be used as list, stack, or queue

Linked list includes many convenient methods to manipulate the elements stored. Apart from the methods of super class List... it adds its own methods also of which few are illustrated here

1. Void addFirst(Object obj)
2. Void addLast(Object obj);
3. Object getFirst();
4. Object getLast();
5. Object removeFirst();
6. Object removeLast();

**Example on linked list**

|  |
| --- |
| **package** main;  **import** java.util.ArrayList;  **import** java.util.Iterator;  **import** java.util.LinkedList;  **import** java.util.ListIterator;  **public** **class** LinkedListDemo  {  **public** **static** **void** main(String[] args)  {  LinkedList<Integer> l1=**new** LinkedList<Integer>();  System.***out***.println("size of l1:\t"+l1.size());  l1.add(100);  System.***out***.println("size of l1:\t"+l1.size());  l1.add(200);  l1.add(300);  l1.add(400);  l1.add(500);  l1.add(600);  l1.add(500);  System.***out***.println("size of l1:\t"+l1.size());  //l1.remove(100); //deleting element from index 100  Integer io=**new** ~~Integer~~(100);  l1.remove(io); //deleting a specified element  l1.remove(**new** ~~Integer~~(100));    System.***out***.println("size of l1:\t"+l1.size());  System.***out***.println("l1:\t"+l1);  l1.set(0, 1000);  System.***out***.println("l1:\t"+l1);  l1.add(1, 2000);  System.***out***.println("l1:\t"+l1);    System.***out***.println("Integer objects existed in Collection..");  **for**(Integer x:l1)  { System.***out***.println(x);  }      }  }  Output:  size of l1: 0  size of l1: 1  size of l1: 7  size of l1: 6  l1: [200, 300, 400, 500, 600, 500]  l1: [1000, 300, 400, 500, 600, 500]  l1: [1000, 2000, 300, 400, 500, 600, 500]  Integer objects existed in Collection..  1000  2000  300  400  500  600  500 |