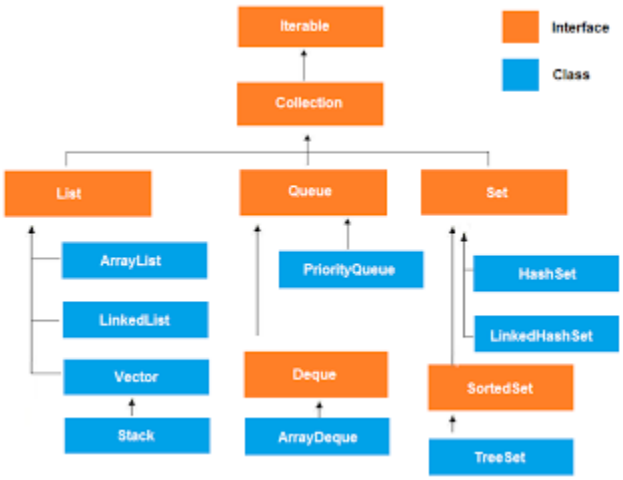
UNIT-I
Data structures in Java

Collection framework provides interfaces and class implementations that enable data handling easy and meaningful. Using collections one can store, retrieve and manipulate the data very effectively and easily.

Before we start lets first discuss the Architecture of Collection framework and hierarchy of Classes and Interfaces.



Collection Interface

The Collection Interface resides at the top of the Hierarchy, although Java does not provides a direct implementation of Collection framework but Collection Interface is being implemented by List and Set Classes.

Data Structure:

A **data structure** is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

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.



Each node in a linked list is connected to the neighboring nodes.

The Structure of Linked Lists:

To understand the inefficiency of arrays and the need for a more efficient data structure, imagine a program that maintains a sequence of employee names. If an employee leaves the company, the name must be removed. In an array, the hole in the sequence needs to be closed up by moving all objects that come after it.

Conversely, suppose an employee is added in the middle of the sequence. Then all names following the new hire must be moved toward the end. Moving a large number of elements can involve a substantial amount of processing time.

A linked list structure avoids this movement. A linked list uses a sequence of *nodes*. A node is an object that stores an element and references to the neighboring nodes in the sequence.
A linked list consists of a number of nodes, each of which has a reference to the next node.



Figure: A Linked List Tom Diana Harry

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

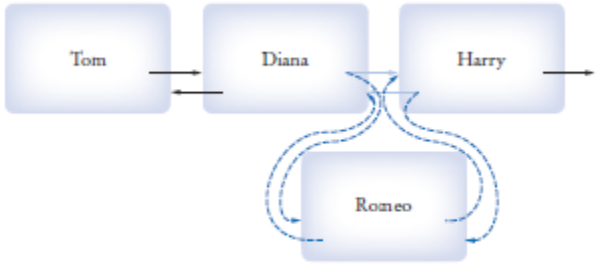


Figure: Inserting a Node into a Linked List

The same is true when you remove a node. What's the catch? Linked lists allow efficient insertion and removal, but element access can be inefficient.



Figure: Removing a Node from a Linked List

For example, suppose you want to locate the fifth element. You must first traverse the first four. This is a problem if you need to access the elements in arbitrary order. The term "random access" is used in computer science to describe an access pattern in which elements are accessed in arbitrary (not necessarily random) order.

In contrast, sequential access visits the elements in sequence. Of course, if you mostly visit all elements in sequence (for example, to display or print the elements), the inefficiency of random access is not a problem. You use linked lists when you are concerned about the efficiency of inserting or removing elements and you rarely need element access in random order.

- The Java library provides a LinkedList class in the java.util package. It is a **generic class**, just like the ArrayList class. That is, you specify the type of the list elements in angle brackets, such as **LinkedList<String>** or **LinkedList<Employee>**.

- The below Table shows important methods of the LinkedList class. As you can see from Table, there are methods for accessing the beginning and the end of the list directly. However, to visit the other elements, you need a list **iterator**.

Working with Linked Lists	
LinkedList<String> list = new LinkedList<String>();	An empty list.
list.addLast("Harry");	Adds an element to the end of the list. Same as add.
list.addFirst("Sally");	Adds an element to the beginning of the list. list is now [Sally, Harry].
list.getFirst();	Gets the element stored at the beginning of the list; here "Sally".
list.getLast();	Gets the element stored at the end of the list; here "Harry".
String removed = list.removeFirst();	Removes the first element of the list and returns it. removed is "Sally" and list is [Harry]. Use removeLast to remove the last element.

List Iterator:

An iterator encapsulates a position anywhere inside the linked list. Conceptually, you should think of the iterator as pointing between two elements, just as the cursor in a word processor points between two characters. In the conceptual view, 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.

You obtain a list iterator with the listIterator method of the LinkedList class:

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

Note that the iterator class is also a generic type. A ListIterator<String> iterates through a list of strings; a ListIterator<Book> visits the elements in a LinkedList<Book>. Initially, the iterator points before the first element.

You can move the iterator position with the next method: **iterator.next();**

The next method throws a **NoSuchElementException** if you are already past the end of the list. You should always call the iterator's **hasNext** method before calling next—it returns true if there is a next element.

```
if (iterator.hasNext())
{
  iterator.next();
}
```

The next method returns the element that the iterator is passing. When you use a **ListIterator<String>**, the return type of the next method is String. In general, the return type of the next method matches the list iterator's type parameter.

You traverse all elements in a linked list of strings with the following loop:

```
while (iterator.hasNext())
{
  String name = iterator.next();
}
```

Methods of the Iterator and ListIterator Interfaces	
ListIterator<String> iter = list.listIterator()	Provides an iterator for visiting all list elements.
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.previous(); iter.set("Juliet");	The set method updates the last element returned by next or previous. The list is now [Juliet].
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. previous and hasPrevious are ListIterator methods.

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

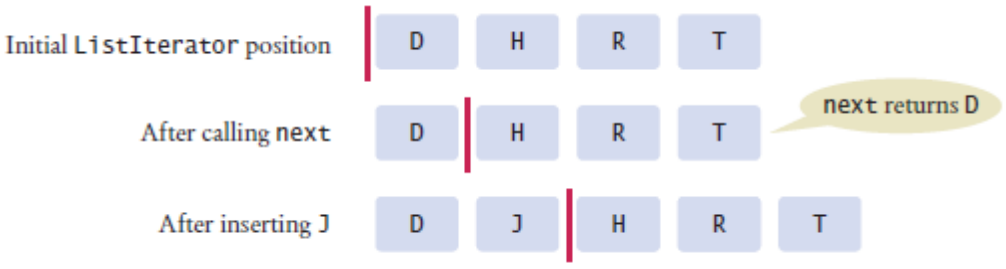


Figure: A Conceptual View of the List Iterator

Example:

```
import java.util.LinkedList;
import java.util.ListIterator;
 /** This program demonstrates the LinkedList class.  */
public class ListDemo
{
    public static void main(String[] args)
   {
    LinkedList<String> staff = new LinkedList<String>();
    staff.add("Diana");
    staff.add("Harry");
    staff.add("Romeo");
    staff.add("Tom");
    System.out.println("Elements are: "+staff);

    // | 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
    System.out.println("After Modifications Elements are: "+staff);
  }
 }
```

Output:

```
Elements are: [Diana, Harry, Romeo, Tom]
After Modifications Elements are: [Diana, Harry, Juliet, Nina, Tom]
```

Stack:

- A **stack** lets you insert and remove elements only at one end, traditionally called the *top* of the stack.
- New items can be added to the top of the stack. Items are removed from 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. With stacks, the addition and removal operations are called push and pop.

```
Stack<String> s = new Stack<String>();
s.push("A"); s.push("B"); s.push("C");
while (s.size() > 0)
{
System.out.print(s.pop() + " "); // Prints C B A
}
```

- There are many applications for stacks in computer science. Consider the undo feature of a word processor. When you select "Undo", the *last* command is undone, then the next-to-last, and so on.
- Another important example is the **run-time stack** that a processor or virtual machine keeps to store the values of variables in nested methods. Whenever a new method is called, its parameter variables and local variables are pushed onto a stack. When the method exits, they are popped off again.
- The Java library provides a simple Stack class with methods **push, pop, and peek,** the latter gets the top element of the stack but does not remove it.

Working with Stacks	
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].
	(Following the toString method of the Stack class, we show the top of the stack at the end.)

int top = s.pop();	Removes the top of the stack; top is set to 3 and s is now [1, 2].
head = s.peek();	Gets the top of the stack without removing it; Head is set to 2;

Example:

```
import java.util.Stack;
public class StackDemo
{
    public static void main(String[] args)
   {
      Stack<String> s = new Stack<String>();
      s.push("Phani"); s.push("Srikanth"); s.push("Srinu");
      System.out.println("Elements are: "+s);
      s.pop();
      System.out.println("After pop Elements are: ");
      while (s.size() > 0)
      {
          System.out.print(s.pop() + " ");
       }
    }
}
```

Output:

```
Elements are: [Phani, Srikanth, Srinu]
After pop Elements are: Srikanth Phani
```

Queue:

A **queue** lets you add items to one end of the queue (the *tail*) and remove them from the other end of the queue (the *head*).

- Queues yield items in a *first-in, first-out* or *FIFO* fashion.
- Items are removed in the same order in which they were added.

A typical application is a print queue. A printer may be accessed by several applications, perhaps running on different computers. If each of the applications tried to access the printer at the same time, the printout would be garbled. Instead, each application places its print data into a file and adds that file to the print queue.

When the printer is done printing one file, it retrieves the next one from the queue. Therefore, print jobs are printed using the "first-in, first-out" rule, which is a fair arrangement for users of the shared printer.

- The Queue interface in the standard Java library has methods add to add an element to the tail of the queue, remove to remove the head of the queue, and peek to get the head element of the queue without removing it.

Working with Queues	
Queue<Integer> q = new LinkedList<Integer>();	The LinkedList class implements the Queue interface.

q.add(1);
q.add(2);
q.add(3);
	Adds to the tail of the queue; q is now [1, 2, 3].

int head = q.remove();	Removes the head of the queue; head is set to 1 and q is [2, 3].
head = q.peek();	Gets the head of the queue without removing it; head is set to 2.

- The LinkedList class implements the Queue interface. Whenever you need a queue, simply initialize a Queue variable with a LinkedList object:

```
Queue<String> q = new LinkedList<String>();
q.add("A"); q.add("B"); q.add("C");
while (q.size() > 0)
{
    System.out.print(q.remove() + " ");   // Prints A B C
}
```

Set:

- The Set interface in the standard Java library has the same methods as the Collection interface.
- There is an essential difference between arbitrary collections and sets.

- A set does not admit duplicates. If you add an element to a set that is already present, the insertion is ignored.
- The Java platform contains three general-purpose Set implementations: **HashSet, TreeSet, and LinkedHashSet.**
- **HashSet:** which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.
- **TreeSet:** which stores its elements in a red-black tree, orders its elements based on ascending order, it is substantially slower than HashSet.
- **LinkedHashSet:** which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).

The methods declared by Set are summarized in the following table:

SN	Methods with Description
1	**add( )** Adds an object to the collection
2	**clear( )** Removes all objects from the collection
3	**contains( )** Returns true if a specified object is an element within the collection
4	**isEmpty( )** Returns true if the collection has no elements
5	**iterator( )** Returns an Iterator object for the collection which may be used to retrieve an object
6	**remove( )** Removes a specified object from the collection
7	**size( )** Returns the number of elements in the collection

Example:

Set have its implementation in various classes like HashSet, TreeSet, LinkedHashSet. Following is the example to explain Set functionality:

```
import java.util.*;

public class SetDemo {

  public static void main(String args[]) {
    int count[] = {34, 22,10,60,30,22};
    Set<Integer> set = new HashSet<Integer>();
    try{
      for(int i = 0; i<5; i++){
        set.add(count[i]);
      }
      System.out.println(set);

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

This would produce the following result:

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

Map Interface:

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.

SN	Methods with Description
1	**void clear( )** Removes all key/value pairs from the invoking map.
2	**boolean containsKey(Object k)** Returns true if the invoking map contains k as a key. Otherwise, returns false.
3	**boolean containsValue(Object v)** Returns true if the map contains v as a value. Otherwise, returns false
4	**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.
5	**boolean equals(Object obj)** Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
6	**Object get(Object k)** Returns the value associated with the key k.
7	**int hashCode( )** Returns the hash code for the invoking map.
8	**boolean isEmpty( )** Returns true if the invoking map is empty. Otherwise, returns false.
9	**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.
10	**Object put(Object k, Object v)** Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
11	**void putAll(Map m)** Puts all the entries from m into this map.

12	**Object remove(Object k)** Removes the entry whose key equals k.
13	**int size( )** Returns the number of key/value pairs in the map.
14	**Collection values( )** Returns a collection containing the values in the map. This method provides a collection-view of the values in the 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<String,Integer> m1 = new HashMap<String,Integer > ();
      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:

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

Generics:

Generics was added in Java 5 to provide **compile-time type checking** and removing risk of **ClassCastException** that was common while working with collection classes. The whole collection framework was re-written to use generics for type-safety. Let's see how generics help us using collection classes safely.

```
List list = new ArrayList();
list.add("abc");
list.add(new Integer(5)); //OK
String str=(String) list.get(0);   //type casting leading to ClassCastException at runtime
String str=(String) list.get(1);
```

Above code compiles fine but throws **ClassCastException** at runtime because we are trying to cast Object in the list to String whereas one of the element is of type Integer. After Java 5, we use collection classes like below.

```
List<String> list1 = new ArrayList<String>(); // java 7 ? List<String> list1 = new ArrayList<>();
list1.add("abc");
//list1.add(new Integer(5)); //compiler error

for(String str : list1){
    //no type casting needed, avoids ClassCastException
}
```

Notice that at the time of list creation, we have specified that the type of elements in the list will be **String**. So if we try to add any other type of object in the list, the program will throw **compile time error**. Also notice that in for loop, we **don't need type casting** of the element in the list, hence removing the **ClassCastException** at runtime.

Generic Class:

We can define our own classes with generics type. A generic type is a class or interface that is parameterized over types. We use angle brackets **(< >)** to specify the **type parameter**.

To understand the benefit, let's see we have a simple class as:

```
public class GenericsTypeOld
{
        private Object t;
        public Object get()
        {
                return t;
        }
        public void set(Object t)
        {
                this.t = t;
        }
         public static void main(String args[])
        {
                GenericsTypeOld type = new GenericsTypeOld();
                type.set("Pankaj");
                String str = (String) type.get(); //type casting, error prone and can cause ClassCastException
        }
}
```

Notice that while using this class, we have to use type casting and it can produce **ClassCastException** at runtime. Now we will use java generic class to rewrite the same class as shown below.

```
public class GenericsType<T>
{
	private T t;
	public T get()
	{
		return this.t;
	}
	public void set(T t1)
	{
		this.t=t1;
	}
	public static void main(String args[])
	{
		GenericsType<String> type = new GenericsType<String>();
		type.set("Pankaj"); //valid
		GenericsType type1 = new GenericsType(); //raw type
		type1.set("Pankaj"); //valid
		type1.set(10); //valid and autoboxing support
	}
}
```

- Notice the use of GenericsType class in the main method. We don't need to **do type-casting** and we can remove **ClassCastException** at runtime.
- If we don't provide the type at the time of creation, compiler will produce a warning that "GenericsType is a raw type. References to generic type GenericsType<T> should be parameterized".
- When we don't provide type, the type becomes Object and hence it's allowing both String and Integer objects but we should always try to avoid this because we will have to use type casting while working on raw type that can produce runtime errors.

Tip: We can use **@SuppressWarnings("rawtypes")** annotation to suppress the compiler warning, check out java annotations tutorial.

- Also notice that it supports java **autoboxing**. That means, the numerical number is automatically convert into String.

Java Generic Type:

- Java Generic Type Naming convention helps us understanding code easily and having a naming convention is one of the best practices of java programming language.
- So generics also come with its own naming conventions.
- Usually type parameter names are **single, uppercase letters** to make it easily distinguishable from java variables.
- The most commonly used type parameter names are:
  - E – Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)

- K – Key (Used in Map)
- N – Number
- T – Type
- V – Value (Used in Map)
- S,U,V etc. – 2nd, 3rd, 4th types

Java Generic Method:

Sometimes we don't want whole class to be parameterized, in that case we can create java generics method. Since constructor is a special kind of method, we can use generics type in constructors too.

Here is a class showing example of java generic method.

```
class GenericsType<T>
{
	private T t;
	public T get()
	{
		return this.t;
	}
	public void set(T t1)
	{
		this.t=t1;
	}
}
public class GenericsMethods
{
	//Java Generic Method
	public static <T> boolean isEqual(GenericsType<T> g1, GenericsType<T> g2)
	{
		return g1.get().equals(g2.get());
	}

	public static void main(String args[])
	{
		GenericsType<String> g1 = new GenericsType<>();
		g1.set("Pankaj");
		GenericsType<String> g2 = new GenericsType<>();
		g2.set("Pankaj");

		boolean isequal = GenericsMethods.<String>isEqual(g1, g2);
		//above statement can be written simply as
		isequal = GenericsMethods.isEqual(g1, g2);
			System.out.println(isequal);
	}
}
```

Notice the isEqual method signature showing syntax to use generics type in methods. Also notice how to use these methods in our java program. We can specify type while calling these methods or we can invoke them like a normal method. Java compiler is smart enough to determine the type of variable to be used, this facility is called as **type inference**.

Wrapper classes:

Introduction:

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 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. For example, upto JDK1.4, the data structures accept only objects to store. A data type is to be converted into an object and then added to a Stack or Vector etc. For this conversion, the designers introduced **wrapper classes**.

- Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes, because they "wrap" the primitive data type into an object of that class.
- The wrapper classes are part of the **java.lang package**, which is imported by default into all Java programs.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int  x = 25;
Integer  y = new Integer(33);
```

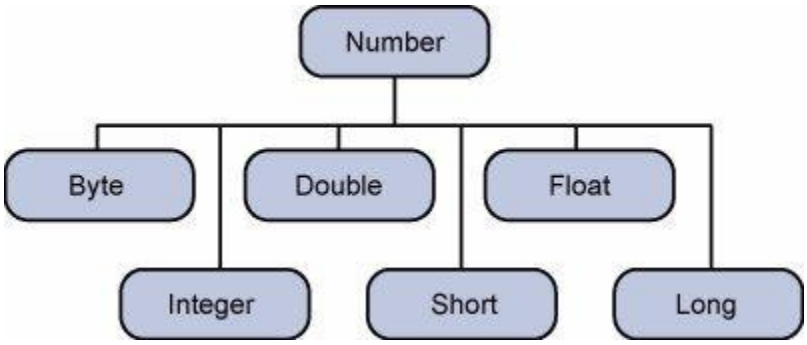
The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.

Below table lists wrapper classes in Java API with constructor details.

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String

double	Double	double or String
long	Long	long or String
short	Short	short or String

Below is wrapper class hierarchy as per Java API



As explain in above table all wrapper classes (except Character) take String as argument constructor. Please note we might get NumberFormatException if we try to assign invalid argument in constructor. For example to create Integer object we can have following syntax.

```
Integer intObj = new Integer (25);
Integer intObj2 = new Integer ("25");
```

Here in we can provide any number as string argument but not the words etc. Below statement will throw run time exception (NumberFormatException)

```
Integer intObj3 = new Integer ("Two");
```

The following discussion focuses on the Integer wrapper class, but applies in a general sense to all eight wrapper classes.

The most common methods of the Integer wrapper class are summarized in below table. Similar methods for the other wrapper classes are found in the Java API documentation.

Method	**Purpose**
parseInt(s)	returns a signed decimal integer value equivalent to string s
toString(i)	returns a new String object representing the integer i
byteValue()	returns the value of this Integer as a byte
doubleValue()	returns the value of this Integer as an double

floatValue()	returns the value of this Integer as a float
intValue()	returns the value of this Integer as an int
shortValue()	returns the value of this Integer as a short
longValue()	returns the value of this Integer as a long
int compareTo(int i)	Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compare(int num1, int num2)	Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.
boolean equals(Object intObj)	Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.

Wrapping and Unwrapping:

```
int marks = 50;
Integer m1 = new Integer(marks);
```

In the above statement, an **int** data type **marks** is given an object form **m1** just by passing the variable to the constructor of **Integer** class. Wherever, **marks** is required as an object, **m1** can be used.

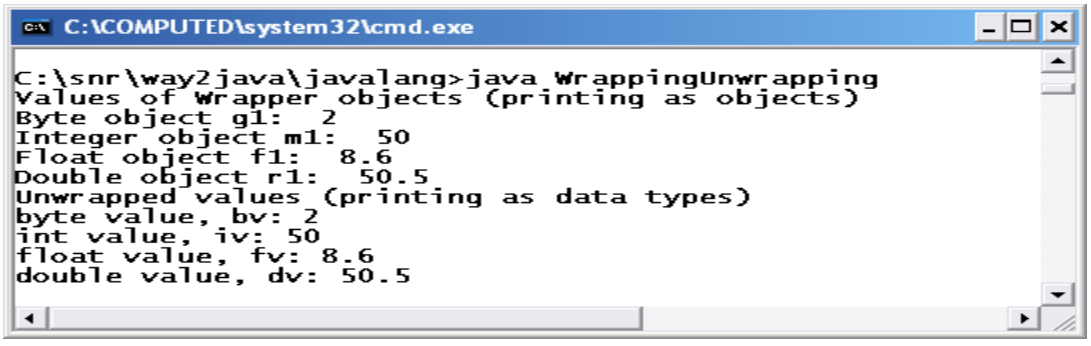
After using the Integer object in programming, now the programmer may require back the data type, as objects cannot be used in arithmetic operations. Now the object **m1** should be unwrapped.

```
int iv = m1.intValue();
```

For unwrapping, the method **intValue()** of Integer class used. The **int** value **iv** can be used in arithmetic operations.

Let's see java program which explain few wrapper classes methods.

```
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);
 }
}
```



As you can observe from the screenshot, 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 object.

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, use the Employee class. Suppose that we have the following Employee class, which implements the Serializable interface:

```
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);
   }
}
```

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 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 = "Phani Kumar";
    e.address = "Ongole";
    e.SSN = 11122333;
    e.number = 101;
    try
    {
      FileOutputStream fileOut = new FileOutputStream("H://employee.ser");
      ObjectOutputStream out = new ObjectOutputStream(fileOut);
      out.writeObject(e);
      out.close();
      fileOut.close();
      System.out.printf("Serialized data is saved in H://employee.ser");
    }
    catch(IOException i)
    {
       i.printStackTrace();
    }
  }
}
```

Deserializing an Object:

The following DeserializeDemo program deserializes the Employee 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)
  {
    Employee e = null;
    try
    {
      FileInputStream fileIn = new FileInputStream("H://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);
  }
}
```

This would produce the following result:

```
Deserialized Employee...
Name: Phani Kumar
Address:Ongole
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 Employee 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 Employee object is 0.