Lecture Seven

Generic Class, STL and Collection Framework

C++ & Java

© **Dr. Mohammad Mahfuzul Islam, PEng**
Professor, Dept. of CSE, BUET

Generic Class

- Many **classes, interfaces** or **methods** are logically same but uses various types of data like integer, double or string. For example, a stack.

- Generic class, interface or method uses **type parameter** It is recommended that type parameter names be **single character capital letter** like T, V and E.

- Java **does not** allow a **primitive type** as a **type parameter**. However, it's not a serious problem due having **wrappers** to encapsulate each primitive type. Java's **autoboxing** and **auto-unboxing** makes the use of wrapper transparent.

- When code with generic class, interface or method is **compiled**, all generic type information is **removed** and **type parameter** is replaced with **compatible type**. This is known as **Erasure**.

- A generic class cannot extend **Throwable**. This means that you cannot create generic exception classes.

Generic Class

```
#include <iostream>
using namespace std;

template <typename T>
class Gen{
   T ob;
public:
   Gen(T o){ ob = o;}
   T getob(){ return ob;}
   void showType();
};

template <class T>
void Gen<T>::showType(){
   cout << "Type: " <<typeid(ob).name()<<endl;
}
```

C++ Code

OUTPUT:

```
Type of T is: i
Value: 88
Type of T is: d
Value: 22.22
Type of T is: Pc
Value: Template Test
```

```
int main(){
   Gen<int> iob(88);
   iob.showType();
   cout << "Value: " << iob.getob() << endl;

   Gen<double> dob(22.22);
   dob.showType();
   cout << "Value: " << dob.getob() << endl;

   Gen<char *> cob("Template Test");
   cob.showType();
   cout << "Value: " << cob.getob() << endl;

   // iob = cob; // Error: incompatible types

   return 0;
}
```

`template <typename T>`

can also be used instead of

`template <class T>` and Vice-Versa

Generic Class

Java Code

```
class Gen <T>{
    T ob;
    Gen(T o){ ob = o; }
    T getOb(){ return ob;}
    void showType(){
        System.out.println("Type of T: "+ob.getClass().getName());
    }
}
public class Main {
    public static void main(String[] args) {
        Gen<Integer> iOb = new Gen<Integer>(88);
        iOb.showType();
        int v = iOb.getOb();
        System.out.println("Value: " + v);

        Gen<String> strOb = new Gen<String>("Generic Test.");
        strOb.showType();
        String str = strOb.getOb();
        System.out.println("value: "+str);
    }
}
```

OUTPUT:

```
Type of T: java.lang.Integer
Value: 88
Type of T: java.lang.String
value: Generic Test.
```

Type Inference and Local Variable Type Inference in Java

Consider A Declaration:

```
Gen<Integer> iOb = new Gen<Integer>(88);
```

Repetition of Integer can be avoided in two ways:

(a) Type Inference;
(b) Local variable Type Inference

(a) Type Inference:

Repetition of **Integer** can be avoided using **<>**, which is known as the **diamond operator**.

```
Gen<Integer> iOb = new Gen<>(88);
```

(b) Local variable Type Inference:

Repetition of **Integer** can be avoided **local variable var**.

```
var iOb = new Gen<Integer>(88);
```

```
class Gen <T>{
   T ob;
   Gen(T o){ ob = o; }
   T getOb(){ return ob;}
   void showType(){
      System.out.println("Type of T: "+ob.getClass().getName());
   }
}
public class Main {
   public static void main(String[] args) {
      Gen<Integer> iOb = new Gen<>(88);
      iOb.showType();
      int v = iOb.getOb();
      System.out.println("Value: " + v);

      var strOb = new Gen<String>("Generic Test.");
      strOb.showType();
      String str = strOb.getOb();
      System.out.println("value: "+str);
   }
}
```

Using Object instead of Generic Class in Java

```java
class NonGen{
    Object ob;
    NonGen(Object o) { ob = o; }
    Object getOb(){ return ob;}
    void showType(){
        System.out.println("Type: "+ob.getClass().getName());
    }
}

public class Main {
    public static void main(String[] args) {
        NonGen iOb = new NonGen(88);
        iOb.showType();
        int v = (Integer) iOb.getOb();
        System.out.println("Value: "+v);

        NonGen strOb = new NonGen("NonGenDemo");
        strOb.showType();
        String str = (String) strOb.getOb();
        System.out.println("Value: "+str);

        iOb = strOb;
        v = (Integer) iOb.getOb();
    }
}
```

OUTPUT:

```
Type: java.lang.Integer
Value: 88
Type: java.lang.String
Value: NonGenDemo
Exception in thread "main"
java.lang.ClassCastException: class
java.lang.String cannot be cast to
class java.lang.Integer
(java.lang.String and
java.lang.Integer are in module
java.base of loader 'bootstrap')
        at Main.main(Main.java:21)
```

Two Problems:

- ✓ Explicit casts must be employed to retrieve the stored data.
- ✓ Many kinds of type mismatch errors cannot be found until runtime.

Generic Class with More Type Parameters

C++ Code

```
#include <iostream>
using namespace std;

template <class T, class V>
class Gen{
     T ob1;   V ob2;
public:
     Gen(T o1, V o2){ ob1 = o1; ob2 = o2;}
     T getob1(){ return ob1;}
     V getob2(){ return ob2;}
     void showType(){
          cout << "Type of T is: " << typeid(ob1).name() << endl;
          cout << "Type of V is: " << typeid(ob2).name() << endl;
    }
};

int main(){
    Gen<int, char *> iob(88, "Double Type Parameters");
    iob.showType();
    cout << "Value: " << iob.getob1() << endl;
    cout << "Value: " << iob.getob2() << endl;

    return 0;
}
```

Java Code

```
class Gen<T, V>{
   T ob1;   V ob2;
   Gen(T o1, V o2){ ob1 = o1; ob2 = o2; }
   T getOb1(){ return ob1; }
   V getOb2(){ return ob2; }
   void showType(){
      System.out.println("Type of T: "+ob1.getClass().getName());
      System.out.println("Type of V: "+ob2.getClass().getName());
   }
}
public class Main {
   public static void main(String[] args) {
      Gen<Integer, String> ob = new Gen<>(88, "Double Type 
Parameters");
      ob.showType();
      System.out.println("Value: "+ob.getOb1());
      System.out.println("Value: "+ob.getOb2());
   }
}
```

OUTPUT:

```
Type of T is: i
Type of V is: Pc
Value: 88
Value: Double Type Parameters
```

Bounded Type Parameters in Java

How can a method **average()** be added in a generic class?

↓

Bounded Type Parameters

OUTPUT:

```
Integer Avg: 3.0
Double Avg: 4.220000000000001
```

```
class Stats<T extends Number>{
    T[] nums;
    Stats(T[] o){ nums = o; }
    double average(){
        double sum = 0.0;
        for( T num: nums )
            sum += num.doubleValue();
        return sum / nums.length;
    }
}
public class Main {
    public static void main(String[] args) {
       Integer[] inums={1, 2, 3, 4, 5};
       Stats<Integer> iOb = new Stats<>(inums);
       System.out.println("Integer Avg: " + iOb.average());

       Double[] dnums = {2.3, 3.5, 4.3, 1.6, 9.4};
       Stats<Double> dOb = new Stats<>(dnums);
       System.out.println("Double Avg: " + dOb.average());

    //  double[] str ={"One", "Two", "Three"};
    //  Stats<String> strOb = new Stats<String>(str);
    }
}
```

Using Wildcard <?> Arguments

Method “**isSameAvg()**” to check
whether average of Integer array is same as the average of Double array.

```
class Stats<T extends Number>{
    T[] nums;
    Stats(T[] o){ nums = o; }
    double average(){
        double sum = 0.0;
        for(T num: nums)
            sum += num.doubleValue();
        return sum / nums.length;
    }
    boolean isSameAvg(Stats<T> ob){
        if (average() == ob.average())
            return true;
        return false;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Integer[] inums = {2, 4, 6, 8};
        Stats<Integer> iOb = new Stats<>(inums);
        Double[] dnums = {2.0, 4.0, 6.0, 8.0};
        Stats<Double> dOb = new Stats<>(dnums);
        if (iOb.isSameAvg(dOb))
            System.out.println("Same Avg.");
        else System.out.println("Different Avg.");
    }
}
```

OUTPUT:

```
java: incompatible types: Stats<java.lang.Double>
cannot be converted to Stats<java.lang.Integer>
```

Solution: wildcard

```
boolean isSameAvg(Stats<?> ob){
    ……………………………
}
```

Bounded Wildcard <?>

```java
class TwoD{
    int x, y;
    TwoD(int a, int b) { x = a; y = b; }
}

class ThreeD extends TwoD{
    int z;
    ThreeD(int a, int b, int c){
        super(a, b);
        z = c;
    }
}

class FourD extends ThreeD{
    int t;
    FourD(int a, int b, int c, int d){
        super(a, b, c);
        t = d;
    }
}
```

```java
class Coords<T extends TwoD>{
    T[] coords;
    Coords(T[] o) { coords = o; }
}
```

```java
public class Main {

   static void showXY( Coords<?> c){
      System.out.println("X Y Coordinate:");
      for (TwoD coord: c.coords )
          System.out.println(coord.x+" "+coord.y);
   }

   static void showXYZ( Coords<? extends ThreeD> c){
      System.out.println("X Y Z Coordinates:");
      for(ThreeD coord: c.coords){
          System.out.println(coord.x + " "+coord.y+" "+ coord.z );
      }
   }

   static void showAll( Coords<? extends FourD> c){
      System.out.println("X Y Z T Coordinates:");
      for (int i = 0; i < c.coords.length; ++i){
          System.out.println(c.coords[i].x+" "
                    +c.coords[i].y+" "
                    +c.coords[i].z+" "
                    +c.coords[i].t);
      }
   }
```

Bounded Wildcard <?>

```
  public static void main(String[] args) {
      TwoD[] td = { new TwoD(0, 0),
                    new TwoD(4,5)};
      FourD[] fd ={ new FourD(3, 4, 5, 6),
                    new FourD(7, 4, 3, 8),
                    new FourD(9, 3, 6, 2)};

      Coords<TwoD> tdOb = new Coords<>(td);
      Coords<FourD> fdOb = new Coords<>(fd);

      showXY(tdOb);
      //  showXYZ(tdOb);
      //  showAll(tdOb);

      showXY(fdOb);
      showXYZ(fdOb);
      showAll(fdOb);
  }
}
```

OUTPUT:

```
X Y Coordinate:
0 0
4 5
X Y Coordinate:
3 4
7 4
9 3
X Y Z Coordinates:
3 4 5
7 4 3
9 3 6
X Y Z T Coordinates:
3 4 5 6
7 4 3 8
9 3 6 2
```

What Happens?

```
static void showXY( Coords<?
Extends TwoD> c)
```

What Happens?

```
static void showXYZ( Coords<?
Extends TwoD> c)
```

Error:

java: incompatible types: capture#1 of ? extends TwoD cannot be converted to ThreeD

```
showXZY(tdOb);
```

OUTPUT:

```
java: incompatible types: Coords<TwoD> cannot be
converted to Coords<? extends ThreeD>
```

Generic Function / Method

```
#include <iostream>
using namespace std;

template <typename T>
void swapargs(T &a, T &b){
   T temp = a;
   a = b;
   b = temp;
}

void swapargs(int &a, int &b){
   int temp = a;
   a = b;
   b = temp;
   cout << "Non-Generic swap" << endl;
}

class Gen{                //non-generic class
public:
   template <typename T>
   void showType(T &a){
     cout << "Type: " << typeid(a).name() << endl;
   }
};
```

C++ Code

```
int main(){
    int i = 10, j = 20;
    float x = 10.1, y = 23.3;
    char a = 'x', b = 'y’;

    swapargs(i, j);
    cout << i << " " << j << endl;

    swapargs(x, y);
    cout << x << " " << y << endl;

    swapargs(a, b);
    cout << a << " " << b << endl;

    Gen g;
    g.showType(i);
    g.showType(x);
    g.showType(a);

    return 0;
}
```

OUTPUT:

```
Non-Generic swap
20 10
23.3 10.1
y x
Type: i
Type: f
Type: c
```

Generic Method in Java

```
public class Main {
    static <T extends Comparable<T>, V extends T>
    boolean isIn(T x, V[] list){
        for(T y: list)
            if (x.equals(y)) return true;
        return false;
    }

    public static void main(String[] args) {
        Integer[] nums = {2, 4, 6, 7, 8, 3, 5};
        String[] strs ={"One", "Two", "Three"};

        if (isIn(2, nums))
            System.out.println("2 is in nums");

        if (isIn("Two", strs))
            System.out.println("Two is in strs");
    }
}
```

OUTPUT:

```
2 is in nums
Two is in strs
```

- The **Type Parameters** is declared **before** the return type of a method.
- **Comparable** is a **Generic Interface** defined in **java.lang** and its **Type Parameter** specifies the type of objects that that it compares.

Generic Constructor

C++ Code

```cpp
#include <iostream>
using namespace std;

class GenCons{
   double val;
public:
   template <typename T>
   GenCons(T arg){
      val = arg;
   }
   void showVal(){
      cout << "Val: " << val << endl;
   }
};

int main(){
   GenCons iVal(100);
   GenCons dVal(25.6);

   iVal.showVal();
   dVal.showVal();
}
```

OUTPUT:

```
Val: 100
Val: 25.6
```

Java Code

```java
class GenCons{
    private double val;

    <T extends Number>
    GenCons(T arg){
        val = arg.doubleValue();
    }
    public void showVal(){
        System.out.println("Val: "+val);
    }
}
public class Main {
    public static void main(String[] args) {
        GenCons iVal = new GenCons(100);
        GenCons dVal = new GenCons(25.6);
        iVal.showVal();
        dVal.showVal();
    }
}
```

Generic Interface

```
interface MinMax<T extends Comparable<T>>{
   T min();
   T max();
}

class MyClass <T extends Comparable<T>>
implements MinMax<T>{
   T[] list;

   MyClass(T[] cList){ list = cList; }

   public T min(){
      T val = list[0];
      for(T x: list)
         if (x.compareTo(val) < 0) val = x;
      return val;
   }

   public T max(){
      T val = list[0];
      for(T x: list)
         if (x.compareTo(val) > 0) val = x;
      return val;
   }
}
```

```
public class Main {
    public static void main(String[] args) {
        Integer[] iVals = {2, 4, 3, 1, 7, 9, 6};
        String[] sVals = {"One", "Two", "Three", "Four"};

        MyClass<Integer> iOb = new MyClass<>(iVals);
        MyClass<String> sOb = new MyClass<>(sVals);

        System.out.println("Min in iVals: "+ iOb.min());
        System.out.println("Max in iVals: "+ iOb.max());
        System.out.println("Min in sVals: "+ sOb.min());
        System.out.println("Max in sVals: "+ sOb.max());
    }
}
```

OUTPUT:

```
Min in iVals: 1
Max in iVals: 9
Min in sVals: Four
Max in sVals: Two
```

Generic in Inheritance

```
#include <iostream>
using namespace std;

class NonGen{
   int num;
public:
   NonGen(int n){ num = n; }
   void show(){
      cout << "NonGen: " << num << endl;
   }
};

template <class T>
class Gen : public NonGen{
   T ob;
public:
   Gen(int n, T o) : NonGen(n){ ob = o; }
   T getob(){ return ob; }
   void show(){
      cout << "Gen: " << ob << endl;
   }
};
```

C++ Code

```
template <class T, class V>
class Gen2D : public Gen<T>{
    V ob;
public:
    Gen2D(int n, T o, V o2) : Gen<T>(n, o){ ob = o2; }
    V getob(){ return ob; }
    void show(){
        cout << "Gen2D: " << ob << endl;
    }
};

int main(){
    NonGen ob(10);
    Gen<char> ob1(20, 'A’);
    Gen2D<double, char *> ob2(40, 60.5, "Gen2 Object");

    ob.show();
    ob1.show();
    ob2.show();

    cout << ob1.getob() << endl;
    cout << ob2.getob() << endl;
    return 0;
}
```

OUTPUT:

```
NonGen: 10
Gen: A
Gen2D: Gen2 Object
A
Gen2 Object
```

Generic in Inheritance

```
class NonGen{
    private int num;
    NonGen(int n) { num = n;}
    public void show(){
        System.out.println("NonGen: "+num);
    }
}

class Gen<T> extends NonGen{
    private T ob;
    Gen(int n, T o){ super(n); ob = o; }
    public T getOb(){ return ob; }
    public void show(){
        System.out.println("Gen: "+ob);
    }
}

class Gen2<T, V> extends Gen<T>{
    private V ob;
    Gen2(int n, T o1, V o2){ super(n, o1); ob = o2; }
    public V getOb2(){ return ob; }
    public void show(){
        System.out.println("Gen2: "+ob);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        NonGen ob = new NonGen(10);
        Gen<Character> ob1 = new Gen<>(20, 'A');
        Gen2<Double, String> ob2 = new Gen2<>(40, 60.5, "Gen2 Object");

        ob.show();
        ob1.show();
        ob2.show();

        System.out.println(ob1.getOb());
        System.out.println(ob2.getOb2());
    }
}
```

OUTPUT:

```
NonGen: 10
Gen: A
Gen2D: Gen2 Object
A
Gen2 Object
```

Standard Template Library(STL)

C++

STL Items

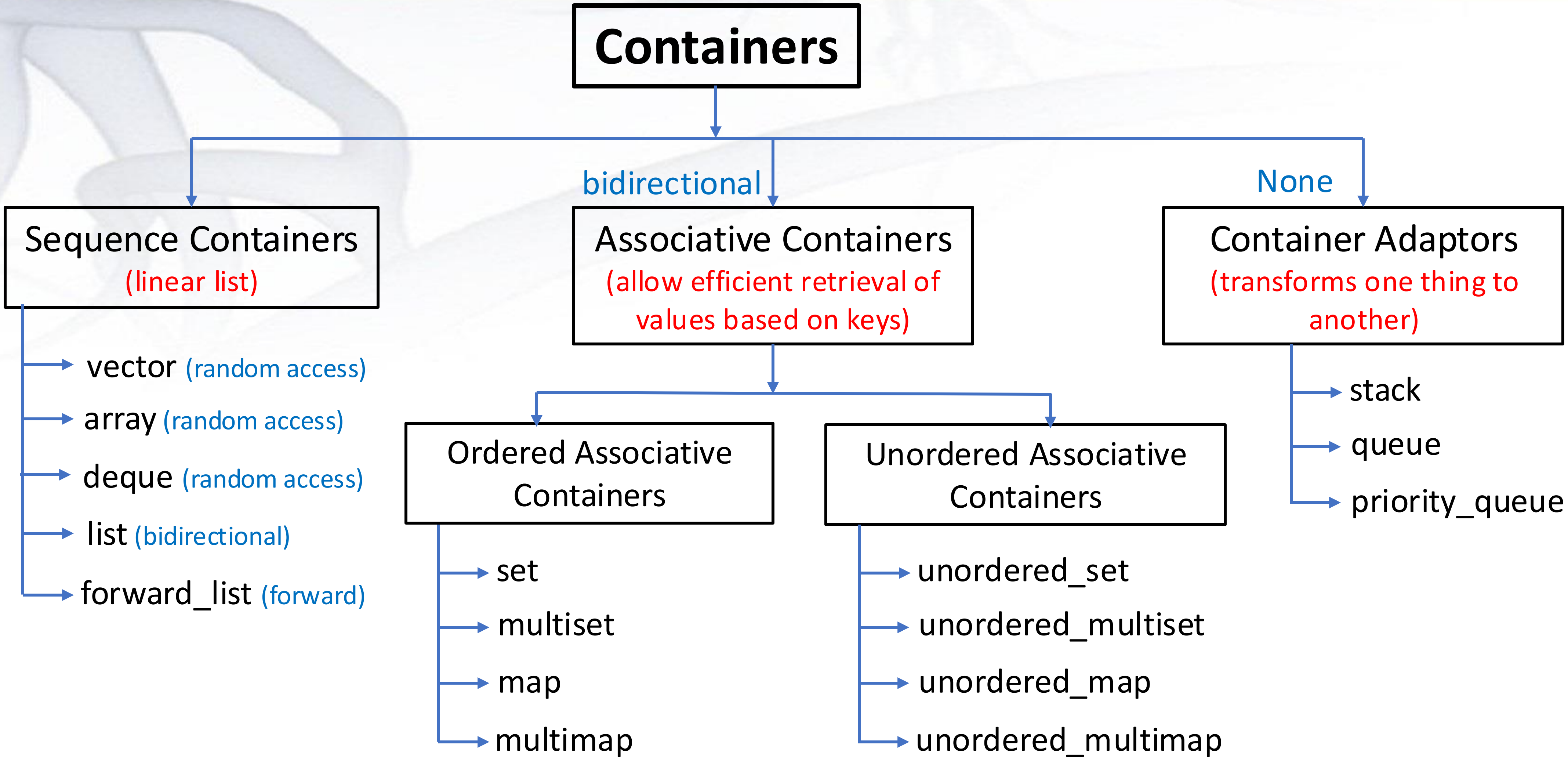
At the core of **Standard Template Library (STL)** are three fundamental Items: **Containers**, **Algorithms** and **Iterators**.

- **Containers:** containers are objects that hold other objects.
- **Algorithms:** algorithms act containers. Their capabilities includes initialization, sorting, searching and transforming the contents of containers.
- **Iterators:** iterators are objects that act like pointer to the contents of a container with the ability to cycle.

- Besides these three Items, STL relies upon some standard components like **allocators**, **predicates**, **comparison functions** and **function objects**.
- In addition to the headers required by the various STL classes, the C++ standard library includes the **<utility>** and **<functional>** headers, which provide support for the STL.

Containers

Some examples of Containers



Containers

- Sequence Containers (linear list)
  - vector (random access)
  - array (random access)
  - deque (random access)
  - list (bidirectional)
  - forward_list (forward)
- bidirectional
  Associative Containers (allow efficient retrieval of values based on keys)
  - Ordered Associative Containers
    - set
    - multiset
    - map
    - multimap
  - Unordered Associative Containers
    - unordered_set
    - unordered_multiset
    - unordered_map
    - unordered_multimap
- None
  Container Adaptors (transforms one thing to another)
  - stack
  - queue
  - priority_queue

Some functions associated with Containers

Some commonly used functions:

Function	Purpose	Containers Applicable
insert()	Insert an elements	Sequence containers, Associative containers
erase()	Remove elements	Sequence containers, Associative containers
push_back()	Add an element at the end.	Sequence containers
pop_back()	Remove an element from the end	Sequence containers
push_front()	Add an element at the start.	**list** and **deque** containers
pop_front	Remove an element from the start	**list** and **deque** containers
size()	Return the current size of the container	Sequence containers, Associative containers
sort()	Sort data within the range in custom order	Array, vector, deque, etc.
merge()	Combine two sorted ranges.	
empty()		
clear()		
make_pair()		

typedef defined in STL

Some of the most common typedef names defined in STL:

	Containers Applicable
size_type	Some type of integer
Reference	A reference to an element
const_reference	A **const** reference to an element
iterator	An iterator
const_iterator	A **const** iterator
reverse_iterator	A reverse iterator
const_reverse_iterator	A **const** reverse iterator
value_type	The type of a value stored in a container
allocator_type	The type of the allocator
key_type	The type of a key
key_compare	The type of a function that compares two keys
value_compare	The type of a function that compares two values

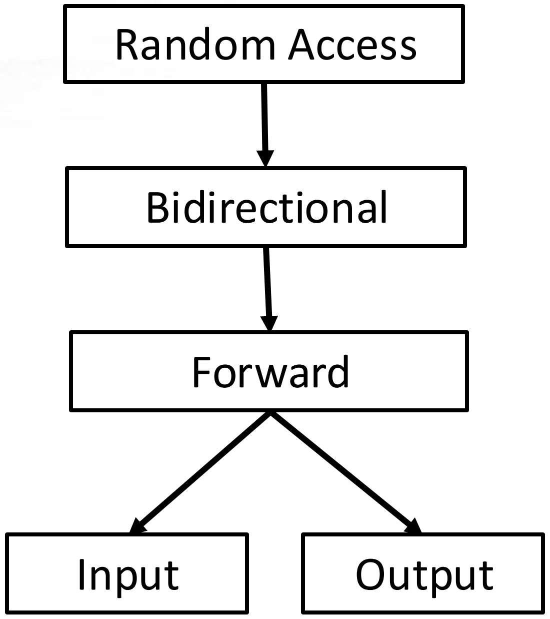
- Overloaded operators <, <=, >, >=, == and != perform element-by-element comparisons.
- Overloaded operators <, <=, > and >= are *not* provided for the *unordered associative containers*.

Iterators

- **Iterators** are objects that like **pointers**.

There are five types of iterators:

Iterator	Actions	Direction
Random Access	Store and Retrieve values	Elements may be accessed randomly.
Bidirectional	Store and Retrieve values	Forward and Backward moving
Forward	Store and Retrieve values	Forward moving only.
Input	Retrieve, but not Store	Forward moving only.
Output	Store, but not Retrieve	Forward moving only.



There are five types of iterators:

Function	Purpose	Containers Applicable
begin()	Return iterator to the start	Sequence Containers, Associative Containers
end()	Return iterator to the end (last element: **end()**-1)	Sequence Containers, Associative Containers
find()	Locate an element given its key	Associative Containers

Some Sample Codes

Sample Definition of a Container (vector):

```
vector<int> iv;           // create zero-length int vector
vector<char> cv(5);       // create 5-element char vector
vector<char> cv(5, 'x’);  // initialize a 5-element char vector
vector<int> iv2(iv);      // create int vector from an int vector
```

Sample Definition of Iterator:

```
vector<char>::iterator p; // create an iterator
```

Sample code for a Container:

```
vector<char> v2;
char str[] = "<Vector>";
for(int i=0; str[i]; i++) v2.push_back(str[i]);
```

Sample insert() and erase() code:

```
v.insert(p, 10, 'X’);               // insert 10 ‘X’ into V starting from p
v.insert(p, v2.begin(), v2.end());  // insert v2 into v
v.erase(p, p+10);                // remove next 10 elements starting from p
```

A Sample Program using vector

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
   vector<char> v;
   vector<char>::iterator p;

   if (v.empty()){
      cout << "Vector is empty" << endl;
   }

   for(int i = 0; i < 10; i++){
      v.push_back(i+'A’);
   }

   p = v.begin();
   while(p != v.end()){
      cout << *p << " ";
      p++;
   }
   cout << endl;

   p = p-3;
   v.insert(p++, 'X’);
   v.insert(p, 3, 'Y’);
   for(int i = 0; i < v.size(); i++){
       cout << v[i] << " ";
   }
   cout << endl;

   p = v.begin();
   p += 3;
   v.erase(p, p+3);
   for(int i = 0; i < v.size(); i++){
      cout << v[i] << " ";
   }
   cout << endl;

   v.clear();
   cout << "Size: " << v.size() << endl;

   return 0;
}
```

OUTPUT:

```
Vector is empty
A B C D E F G H I J
A B C D E F G X Y Y Y H I J
A B C G X Y Y Y H I J
Size: 0
```

A Sample Program using list

```
#include <iostream>
#include <list>
using namespace std;

int main(){
  list<int> lst1, lst2;
  list<int>::iterator p = lst1.begin();

  for(int i = 0; i < 5; i++){
    lst1.push_back(2*i);
    lst2.push_front(2*i+1);
  }

  lst1.merge(lst2);
  p = lst1.begin();
  while(p != lst1.end()){
    cout << *p << " ";
    p++;
  }
  cout << endl;
  cout << "Size1: " << lst1.size() << " ";
  cout << "Size2: " << lst2.size() << endl;

  lst1.reverse();
```

```
p = lst1.begin();
while(p != lst1.end()){
  cout << *p << " ";
  p++;
}
cout << endl;

for(int i = 0; i < 3; i++){
  lst2.push_back(i+20);
}

lst1.splice(lst1.end(), lst2);
cout << "Size1: " << lst1.size() << " ";
cout << "Size2: " << lst2.size() << endl;

p = lst1.begin();
while(p != lst1.end()){
  cout << *p << " ";
  p++;
}
cout << endl;
```

```
  lst1.sort();
  p = lst1.begin();
  while(p != lst1.end()){
    cout << *p << " ";
    p++;
  }
  cout << endl;

  return 0;
}
```

OUTPUT:

```
0 2 4 6 8 9 7 5 3 1
Size1: 10 Size2: 0
1 3 5 7 9 8 6 4 2 0
Size1: 13 Size2: 0
1 3 5 7 9 8 6 4 2 0 20 21 22
0 1 2 3 4 5 6 7 8 9 20 21 22
```

A Sample Program using map

```
#include <iostream>
#include <map>
using namespace std;

int main(){
    map<char, char*> myMap;
    map<char, char*>::iterator p;

    myMap.insert(pair<char, char*>('J', "John"));
    myMap.insert(pair<char, char*>('P', "Paul"));
    myMap.insert(make_pair<char, char*>('L', "Linda"));
    myMap.insert(make_pair<char, char*>('M', "Mike"));

    p = myMap.begin();
    while(p != myMap.end()){
       cout << p->first << " " << p->second << endl;
       p++;
    }
    cout << endl;
```

```
    p = myMap.find('M’);
    if(p != myMap.end()){
        cout << "Found: " << p->first << " "
             << p->second << endl;
    } else {
        cout << "Not found" << endl;
    }

    return 0;
}
```

OUTPUT:

```
J John
L Linda
M Mike
P Paul

Found: M Mike
```

Algorithms

Algorithms act on containers. Header <algorithm> should be included.

Some Algorithms:

Name	Format	Explanation
count()	int n = count(v.begin(), v.end(), val)	Returns the **number of elements** in the sequence beginning at **v.begin()** and ending at **v.end()** that match **val**
count_if()	int n = count_if(v.begin(), v.end(), even)	Returns the **number of elements** in the sequence beginning at **v.begin()** and ending at **v.end()** for which the unary predicate **even** returns **true**
remove_copy()	remove_copy(v.begin(), v.end(), back_inserter(v2), val)	Copies elements from the specified range, i.e., from **v.begin()** to **v.end()** that are not equal to **val** and puts the results into the sequence pointed by **v2.begin()**
reverse()	reverse(v.begin(), v.end())	Reverse the order of the range specified
transform()	transform(v.begin(), v.end(), v.begin(), xform)	Modifies each element in the range specified according to the function **xform**

Example Program using Algorithms

```
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

bool even(int n){
   if (n % 2 == 0) return true;
   return false;
}

int square(int n){
   return n * n;
}

int main(){
   list<int> myList;
   list<int>::iterator p;
   list<int> v2;
   list<int>::iterator q;
```

```
for(int i = 1; i < 6; i++){
      myList.push_back(i * 3);
      myList.push_front(i * 4);
  }

  for(p = myList.begin(); p != myList.end(); p++){
      cout << *p << " ";
  }
  cout << endl;

  int n = count(myList.begin(), myList.end(), 12);
  cout << "12 appears " << n << " times" << endl;

  n = count_if(myList.begin(), myList.end(), even);
  cout << n << " even numbers." << endl;

  remove_copy(myList.begin(), myList.end(),
               back_inserter(v2), 12);
```

Example Program using Algorithms

```
    for(q = v2.begin(); q != v2.end(); q++){
        cout << *q << " ";
    }
    cout << endl;

    reverse(v2.begin(), v2.end());
    for(q = v2.begin(); q != v2.end(); q++){
        cout << *q << " ";
    }
    cout << endl;

    transform(v2.begin(), v2.end(), v2.begin(), square);
    for(q = v2.begin(); q != v2.end(); q++){
        cout << *q << " ";
    }
    cout << endl;

    return 0;
}
```

OUTPUT:

```
20 16 12 8 4 3 6 9 12 15
12 appears 2 times
7 even numbers.
20 16 8 4 3 6 9 15
15 9 6 3 4 8 16 20
225 81 36 9 16 64 256 400
```

Collection Framework

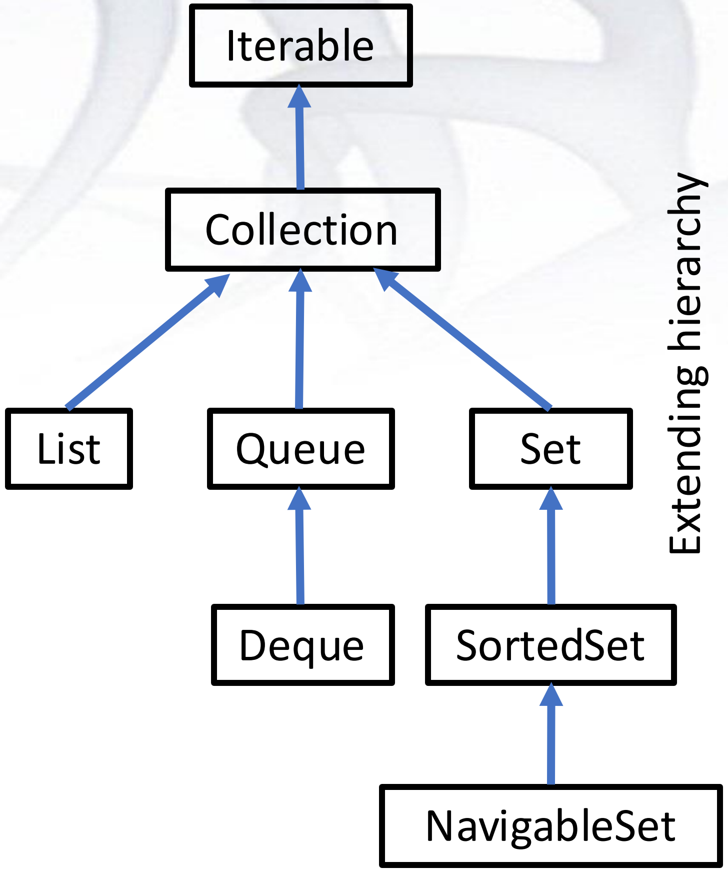
Java

Collection Framework

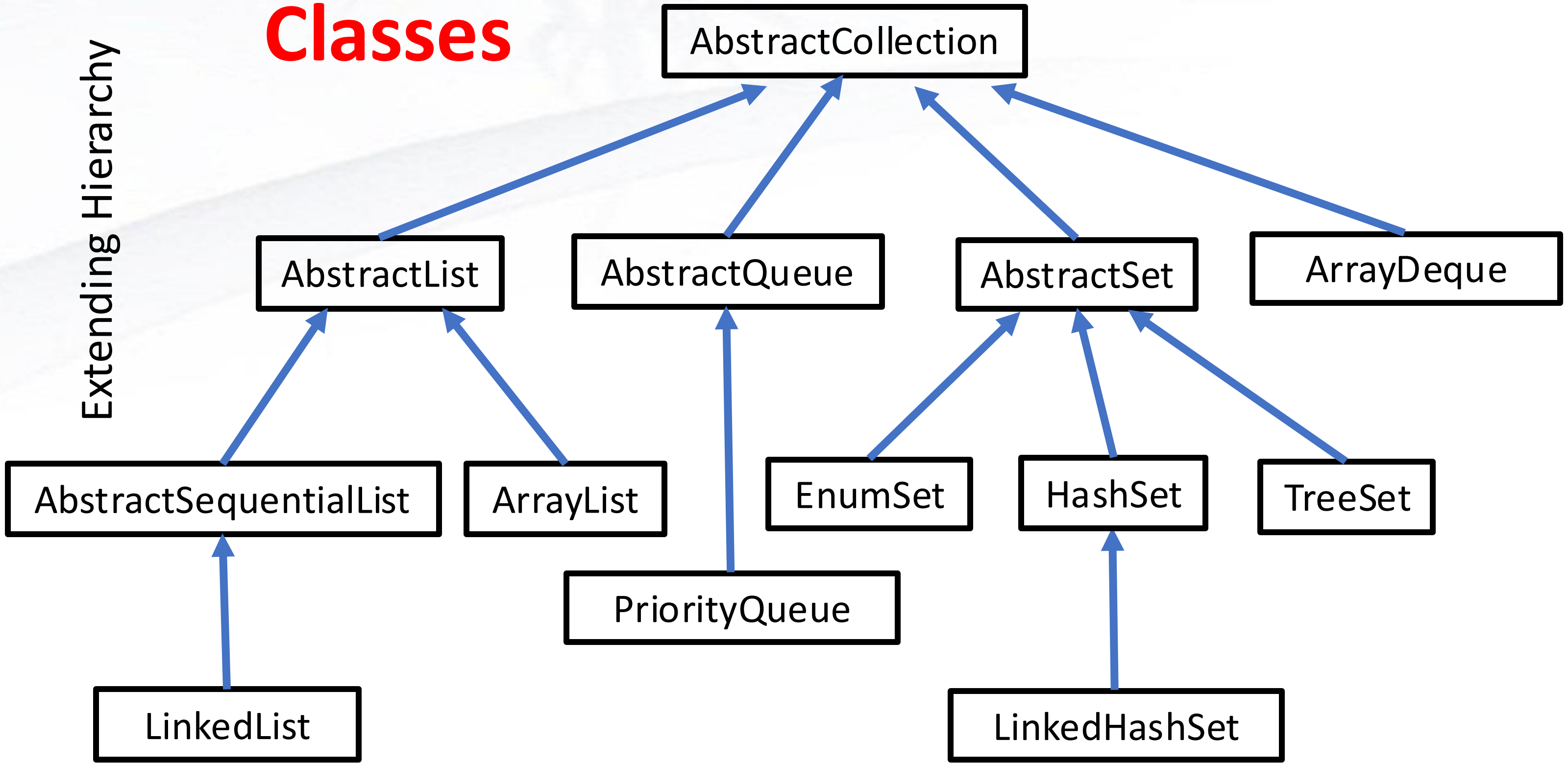
- The **Collections Framework** is a sophisticated hierarchy of **interfaces** and **classes** that provide state-of-the-art technology for managing groups of objects. Package: **java.util**
- **Three items** of Collection Framework: **Collections** (**Interfaces** & **Classes**), **Iterators** and **Algorithms**.
- **Goals** of Collection Framework:
  - ✓ High **efficiency** and high **performance**
  - ✓ Different types of collections to work in a **similar manner** and with a high degree of **interoperability**
  - ✓ **Easy extending**/**adapting** a collection.
- Legacy of ad-hoc classes before Collections: **Dictionary**, **Vector**, **Stack** and **Properties**.
- In addition to Collection **interfaces**, Collections also use **Comparator, RandomAccess, Iterator, ListIterator and Spliterator**.

Collections

Interfaces



Classes



Implementation:

- ✓ AbstractCollection, AbstractList, AbstractQueue and AbstractSet implement Collection, List, Queue and Set, respectively.
- ✓ ArrayDeque implemts Deque.

The Collection Interfaces

- Two Types of methods for interfaces: **modifiable** and **unmodifiable**. All the built-in Collections are modifiable, i.e., allow to modify the contents of the collection.

Some Methods of Collection Interface:

Method	Return Type	Description

add()
addAll()
	boolean	**add()** adds a compatible object into the collection.
		addAll() method adds the entire contents of a collection to another.

remove()
removeAll(c)
removeIf()
retainAll(c)
	boolean	**remove()** removes an object from the collection.
		removeAll() removes all objects of c from the collection.
		removeIf() removes objects that satisfy the condition specified by *predicate*.
		retainAll() removes all objects from the invoking collection except those in *c*.

clear()	void	Clear or remove all objects.

contains()
containAll()
	boolean	**contains()** checks whether a collection contains a specific object.
		containAll() checks whether one collection contains all the members of another.

isEmpty()	boolean	Returns **true** if the invoking collection is empty.
size()	int	Returns the number of elements in the invoking collection.
toString()	Object[]	Returns an array of elements from the invoking collection.

The Collection Interface

Some Methods of Collection Interface:

Method	Return Type	Description
equals()	boolean	Returns true if the invoking collection and object are equal.

iterator()
spliterator()
	Iterator
	Spliterator
		Returns an Iterator for the invoking collection.
		Returns a Spliterator from the invoking collection.

stream()
parallelStream()
	Stream	Returns a stream that uses the invoking collection as its source of elements.
		Returned stream supports parallel operations, if possible.

Some Exceptions of Collection Interface:

Exception Name	Reason for thrown
UnsupportedOperationException	If a collection cannot be modified.
ClassCastException	When one object is incompatible with another.
NullPointerException	If an attempt is made to store a null object in a collection.
IllegalArgumentException	If an invalid argument is used.
IllegalStateException	if an attempt is made to add an element to a fixed-length collection that is full.

The List Interface

Some Methods of List Interface:

Method	Return Type	Description
replaceAll()	void	Modify each element in the collection.
get()	Object	Return the object stored at the specific index.
set()	Object	Assigns *obj* to the location specified by *index* . Returns old object value.

indexOf()
lastIndexOf()
	int	Return the index of the first instance of an object.
		Return the index of the last instance of an object.

subList()	list	To obtain a sublist specifying beginning and ending index.
sort()	void	To sort the list.
of()	list	Creates an unmodifiable list containing the elements specified in *parameter-list*.

Exception of List Interface:

Exception Name	Reason for thrown
IndexOutOfBoundsException	If an invalid index is used.

The Queue Interface

Some Methods of Queue Interface:

Method	Return Type	Description

poll()
remove()
	Object	Obtain an element from the head of the queue. Return **null** if the queue is empty.
		Remove an element from the head of the Queue. Return **Exception** if empty.

peek()
element()
	Object	Obtain an element from the head of the Queue without removing it. **peek()** returns **null** while **element()** throws **exception** if the queue is empty.

offer()	boolean	Add an element to a queue. It fails when fixed-length Queue is full.

Exceptions of Queue Interface:

Exception Name	Reason for thrown
NoSuchElementException	If an attempt is made to remove an element from an empty Queue.

Iterator

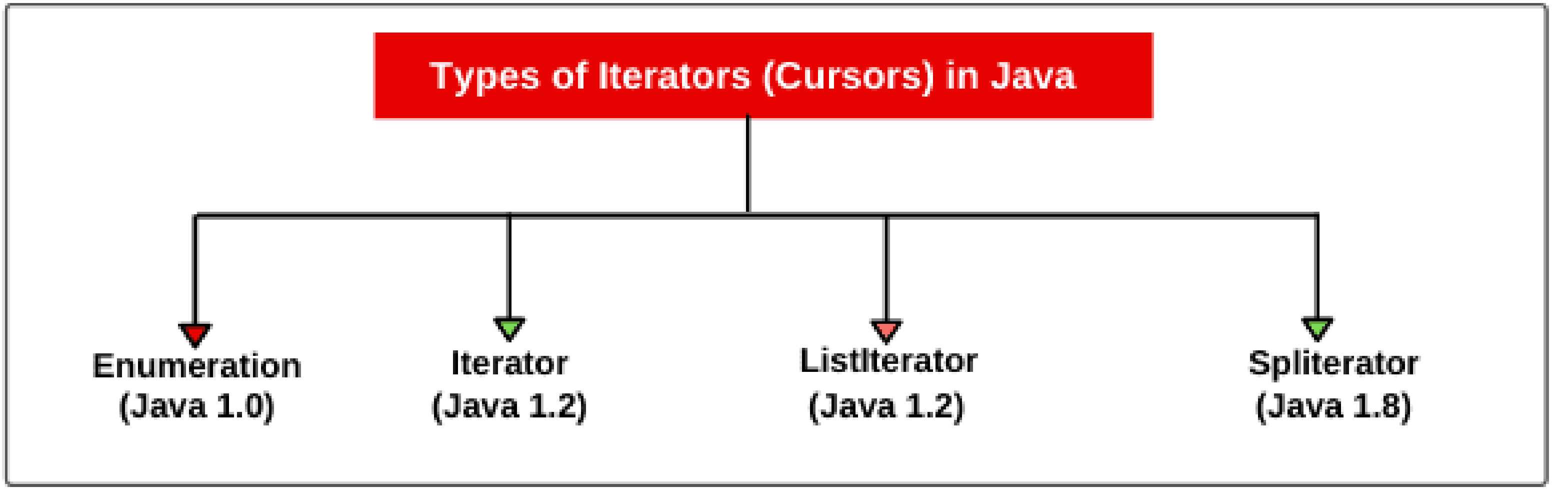
Iterator: A standardized way of accessing the elements within a collection, one at a time. It's an Interface.

ListIterator: It extends Iterator interface and used for all types of list including ArrayList, Vector, LinkedList, Stack, etc. Access the collection either forward or backward directions.

Spliterator: Provide support for parallel iteration.

PrimitiveIterator: Used for primitive data type. For example, PrimitiveIterator.OfDouble

Class **iterator** implements **Iterator** Interface.



Types of Iterators (Cursors) in Java

Enumeration (Java 1.0)

Iterator (Java 1.2)

ListIterator (Java 1.2)

Spliterator (Java 1.8)

Iterator

Methods used in Iterator:

Method	Return Type	Description
hasNext()	boolean	Returns true if the iterator has more elements.
next()	Object	Returns the next element of the iterator. Throws **NoSuchElementException** if no more element is present.
remove()	void	Removes the current element in the collection. If it is not followed by next() method, then throws **IllegalStateException**.

Some **Methods** of **ListIterator:**

Method	Return Type	Description
set(), add()	void	Assign Object to the current and next location, respectively.
previous()	Object	Returns the previous element.
nextIndex(), previosIndex()	int	Returns the index of next and previous element, respectively.
forEachRemaining()	void	The action specified by *action* is executed on each unprocessed element in the collection.

The ArrayList class

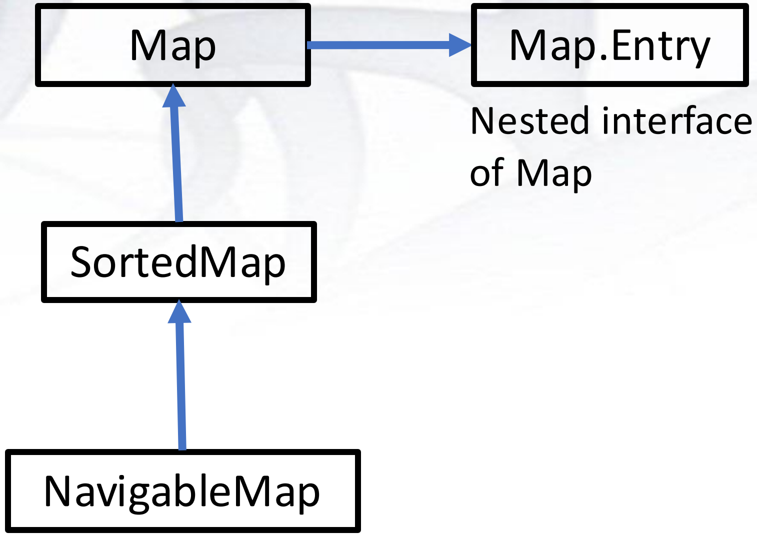
```
import java.util.Iterator;
import java.util.ListIterator;

public class Main {
    public static void main(String[] args) {
        ArrayList <String> al = new ArrayList<>();

        al.add("One");
        al.add("Two");
        al.add("Three");
        System.out.println(al);

        Iterator<String> itr = al.iterator();

        while(itr.hasNext()){
            String str = itr.next();
            System.out.print(str + " ");
        }
        System.out.println();
        itr.remove();
        System.out.println(al) ;
        ListIterator<String> litr = al.listIterator();
        litr.add("Three");
        System.out.println(al);

        while(litr.hasNext()){
            String str = litr.next();
            litr.set(str+"+");
        }
        System.out.println(al);

        while(litr.hasPrevious()){
            String str = litr.previous();
            System.out.print(str+" ");
        }
        System.out.println();

        String[] strA = new String[al.size()];
        strA = al.toArray(strA);
        for(String s: strA) System.out.print(s+" ");
    }
}
```

OUTPUT:

```
[One, Two, Three]
One Two Three
[One, Two]
[Three, One, Two]
[Three, One+, Two+]
Two+ One+ Three
Three One+ Two+
```

Map

Interfaces



Map

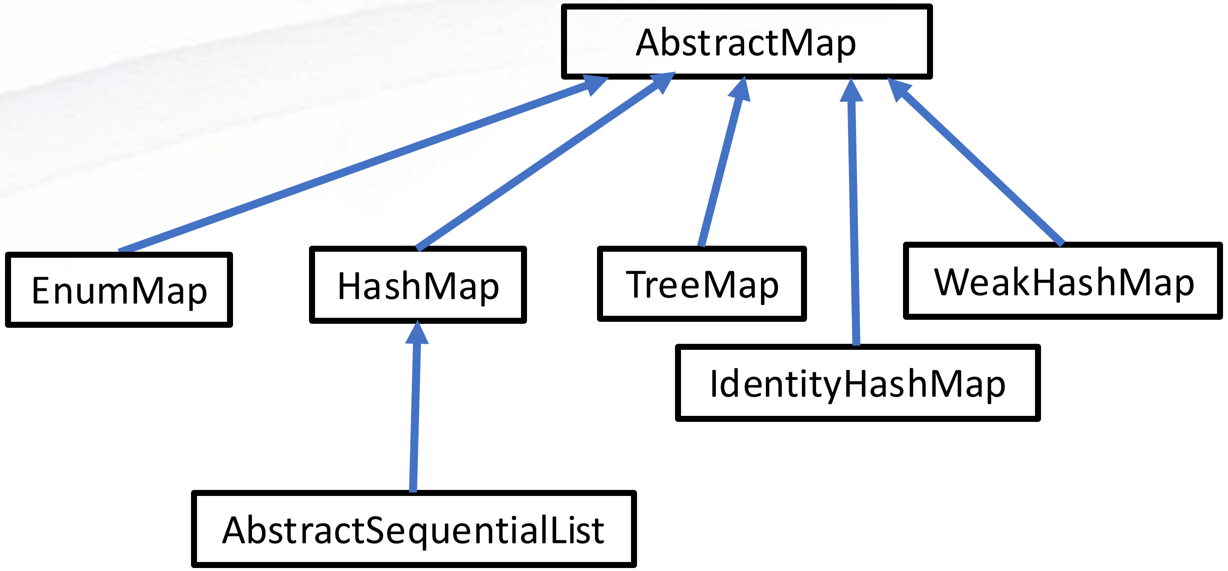
Map.Entry

Nested interface of Map

SortedMap

NavigableMap

Classes



AbstractMap

EnumMap

HashMap

TreeMap

WeakHashMap

IdentityHashMap

AbstractSequentialList

Map

Some **Methods** of **Map** Interface:

Method	Return Type	Description
get()	Value	To obtain a value passing the key as an argument.
put()	Value	To put a value into a Map specifying the key and value. Returns the previous value linked to the key is returned. Returns **null** if the key did not already exist.
entrySet()	Set	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**.
getKey()	Key	To obtain the collection-view of keys.
getValue()	Value	To obtain the collection-view of values.
containsKey()	boolean	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
containsValue()	boolean	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.

Map

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class Main {
   public static void main(String[] args) {
       HashMap<String, Double> hm = new HashMap<>();
       hm.put("John Doe", 3434.34);
       hm.put("Tom Smith", 123.22);
       hm.put("Jane Baker", 1378.00);

       double balance = hm.get("John Doe");
       hm.put("John Doe", balance+1000);
       Set<Map.Entry<String, Double>> set = hm.entrySet();
       for(Map.Entry<String, Double> me: set){
          System.out.print(me.getKey()+" : ");
          System.out.println(me.getValue());
       }

       if (hm.containsKey("Tom Smith")) System.out.println("Found");
   }
}
```

OUTPUT:

```
John Doe : 4434.34
Tom Smith : 123.22
Jane Baker : 1378.0
Found
```

Comparator

- Both **TreeSet** and **TreeMap** store elements in sorted order. However, it is the comparator that defines precisely what "sorted order" means.

Some **Methods** of **Comparator:**

Method	Return Type	Description
compareTo()	int	Tests whether an object equals the invoking comparator.
equals()	int	Two objects (Object1 and Object2) are compared. Returns zero if both are equal. Returns positive value if Object1 is greater than Object2; otherwise returns negative value.
reversed()	Comparator	Reverses the ordering of the comparator.
nullsFirst()	Comparator	Returns a comparator that views **null** values as less than other values.
nullsLast()	Comparator	Returns a comparator that views **null** values as greater than other values.

Comparator

```
import java.util.Comparator;
import java.util.TreeSet;

class MyComp implements Comparator<String> {
    public int compare(String aStr, String bStr){
        return bStr.compareTo(aStr);
    }
}

public class Main {
    public static void main(String[] args) {
        TreeSet<String> ts = new TreeSet<>(new MyComp());
        MyComp mc = new MyComp();

        ts.add("One");
        ts.add("Two");
        ts.add("Three");
        ts.add("Four");
        ts.add("Five");
        System.out.println(ts);
        System.out.println(mc.compare("Dhaka", "Pabna"));
        System.out.println(ts.reversed());
    }
}
```

OUTPUT:

```
[Two, Three, One, Four, Five]
12
[Five, Four, One, Three, Two]
```

```
import java.util.Comparator;
import java.util.TreeSet;

class MyComp implements Comparator<String> {
    public int compare(String aStr, String bStr){
        return aStr.compareTo(bStr);
    }
}

public class Main {
    public static void main(String[] args) {
        TreeSet<String> ts = new TreeSet<>(new MyComp());
        MyComp mc = new MyComp();

        ts.add("One");
        ts.add("Two");
        ts.add("Three");
        ts.add("Four");
        ts.add("Five");

        System.out.println(ts);
        System.out.println(mc.compare("Dhaka", "Jamalpur"));
        System.out.println(ts.reversed());
    }
}
```

OUTPUT:

```
[Five, Four, One, Three, Two]
-6
[Two, Three, One, Four, Five]
```

Algorithms

```java
import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        LinkedList<Integer> ll = new LinkedList<>();
        ll.add(20);
        ll.add(8);
        ll.add(1, 5);
        ll.addFirst(6);
        ll.addLast(-25);
        System.out.println(ll);
        Comparator<Integer> r = Collections.reverseOrder();
        Collections.sort(ll, r);
        System.out.println(ll);
        Collections.shuffle(ll);
        System.out.println(ll);
        System.out.println("Max:" + Collections.max(ll));
        System.out.println("Min: " + Collections.min(ll));
    }
}
```

OUTPUT:

```
[6, 20, 5, 8, -25]
[20, 8, 6, 5, -25]
[8, -25, 6, 5, 20]
Max:20
Min: -25
```