JAVA PROGRAMMING

Chap 7 : Generics and Collections

Generics

- Imagine, wouldn't it 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.
- The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces.
- Using Generics, it is possible to create classes that work with different data types.
- An entity such as class, interface, or method that operates on a parameterized type is a generic entity.
- There are mainly 3 advantages of generics. They are as follows:
  1) **Type-safety:** We can hold only a single type of objects in generics. It doesn't allow to store other objects.
  2) **Type casting is not required**: There is no need to typecast the object.
  3) **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

Types of Java Generics

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.
- Generic Java method takes a parameter and returns some value after performing a task.
- It is exactly like a normal function, however, a generic method has type parameters that are cited by actual type.
- This allows the generic method to be used in a more general way.
- The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.
- 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.
  - 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).**

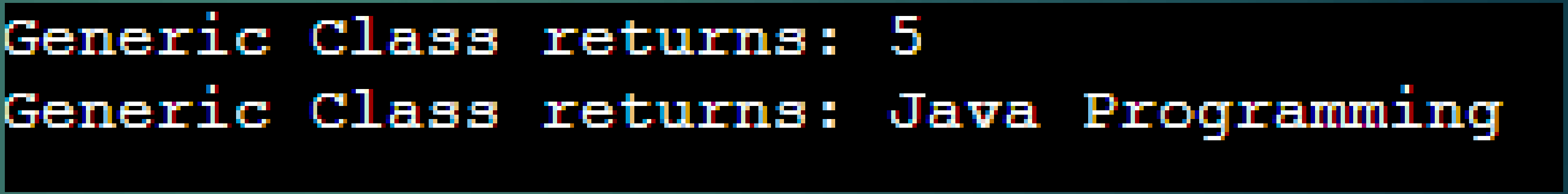
Types of Java Generics

Generic Classes

- A generic class is implemented exactly like a non-generic class.
- The only difference is that it contains a type parameter section.
- There can be more than one type of parameter, separated by a comma.
- The classes, which accept one or more parameters, are known as parameterized classes or parameterized types.
- we use the T type parameter to create the generic class of specific type.
- The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:
  - T - Type
  - E - Element
  - K - Key
  - N - Number
  - V - Value

Generics Class Example

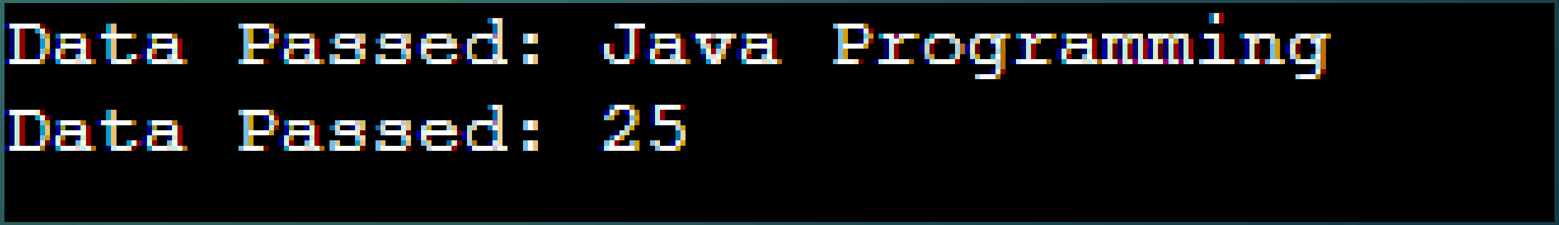
```
class GenericsClass<T> {                // create a Generic class
private T data;                         // variable of T type
public GenericsClass(T data) {      // constructor of Generic class
    this.data = data;
  }
public T getData() {                    // method that return T type variable
    return this.data;
  }
}

class Main {
public static void main(String[] args) {
GenericsClass<Integer> intObj = new GenericsClass<>(5);          // initialize generic class with Integer data
System.out.println("Generic Class returns: " + intObj.getData());

GenericsClass<String> stringObj = new GenericsClass<>("Java Programming"); // initialize generic class with String data
    System.out.println("Generic Class returns: " + stringObj.getData());
  }}
```



```
Generic Class returns: 5
Generic Class returns: Java Programming
```

Generics Method Example

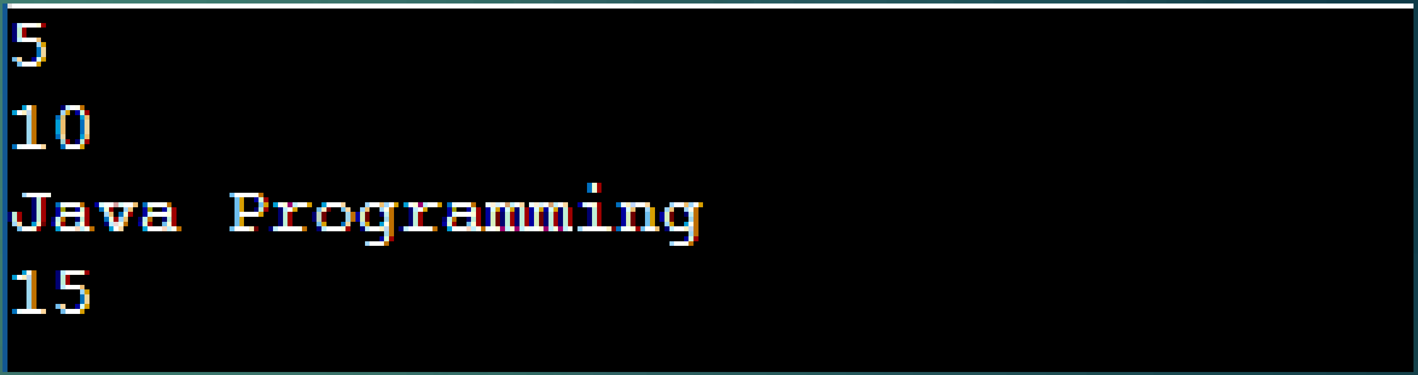
```
class DemoClass {
public <T> void genericsMethod(T data) {                    // create a generics method
System.out.println("Data Passed: " + data);
  }
}

class Main {
  public static void main(String[] args) {
   DemoClass demo = new DemoClass();
   demo.<String>genericsMethod("Java Programming");    // generics method working with String
  demo.genericsMethod(25);       // generics method working with integer without including the type parameter
  }
}
```



```
Data Passed: Java Programming
Data Passed: 25
```

Generics Class Example 2

```
class GenericsClass<T> {                 // create a Generic class
private T data, data1;                          // variables of T type
public GenericsClass(T data, T data1) {     // constructor of Generic class
    this.data = data;
    this.data1 = data1;
  }
public <T> void getData() {                    // method that prints T type variable
    System.out.println(this.data);
    System.out.println(this.data1);
  }
 }
class Main {
public static void main(String[] args) {
GenericsClass intObj = new GenericsClass(5,"10");           // initialize generic class with different data
intObj.getData();

GenericsClass stringObj = new GenericsClass("Java Programming",15);
   stringObj.getData();
  }}
```



```
5
10
Java Programming
15
```

Bounded Types

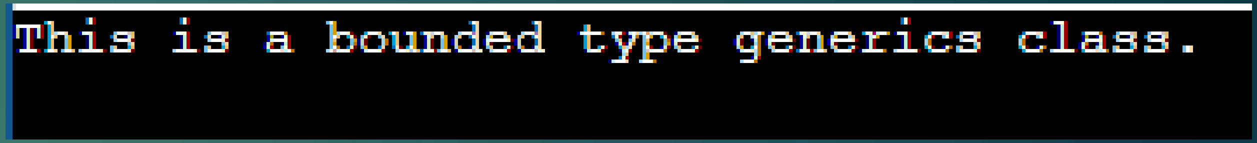
- In general, the type parameter can accept any data types.
- However, if we want to use generics for some specific types (such as accept data of number types) only, then we can use bounded types.
- In the case of bound types, we use the extends keyword.
- Syntax : <T extends A>

  This means T can only accept data that are subtypes of A.
- Eg : class GenericsClass <T extends Number>

  Here, GenericsClass is created with bounded type. This means GenericsClass can only work with data types that are subtypes of Number (Integer, Double, and so on).

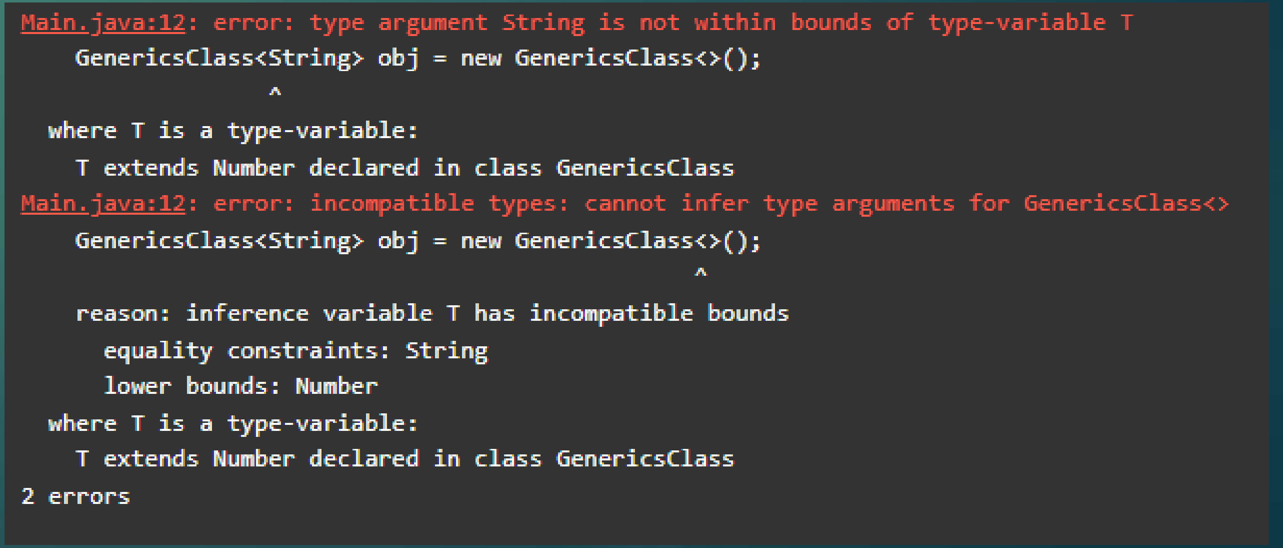
Bounded Type Example

```
class GenericsClass <T extends Number> {
  public void display() {
    System.out.println("This is a bounded type generics class.");
  }}
class Main {
  public static void main(String[] args) {
    // create an object of GenericsClass
    GenericsClass<Integer/Float/Double/Long/Short> obj = new GenericsClass<>();
     obj.display();
  }}
```



```
This is a bounded type generics class.
```

```
class GenericsClass <T extends Number> {
  public void display() {
    System.out.println("This is a bounded type generics class.");
  }}
class Main {
  public static void main(String[] args) {
    // create an object of GenericsClass
    GenericsClass<String> obj = new GenericsClass<>();
     obj.display();
  }}
```



```
Main.java:12: error: type argument String is not within bounds of type-variable T
    GenericsClass<String> obj = new GenericsClass<>();
                  ^
  where T is a type-variable:
    T extends Number declared in class GenericsClass
Main.java:12: error: incompatible types: cannot infer type arguments for GenericsClass<>
    GenericsClass<String> obj = new GenericsClass<>();
                                                 ^
    reason: inference variable T has incompatible bounds
      equality constraints: String
      lower bounds: Number
  where T is a type-variable:
    T extends Number declared in class GenericsClass
2 errors
```

Generic Restrictions

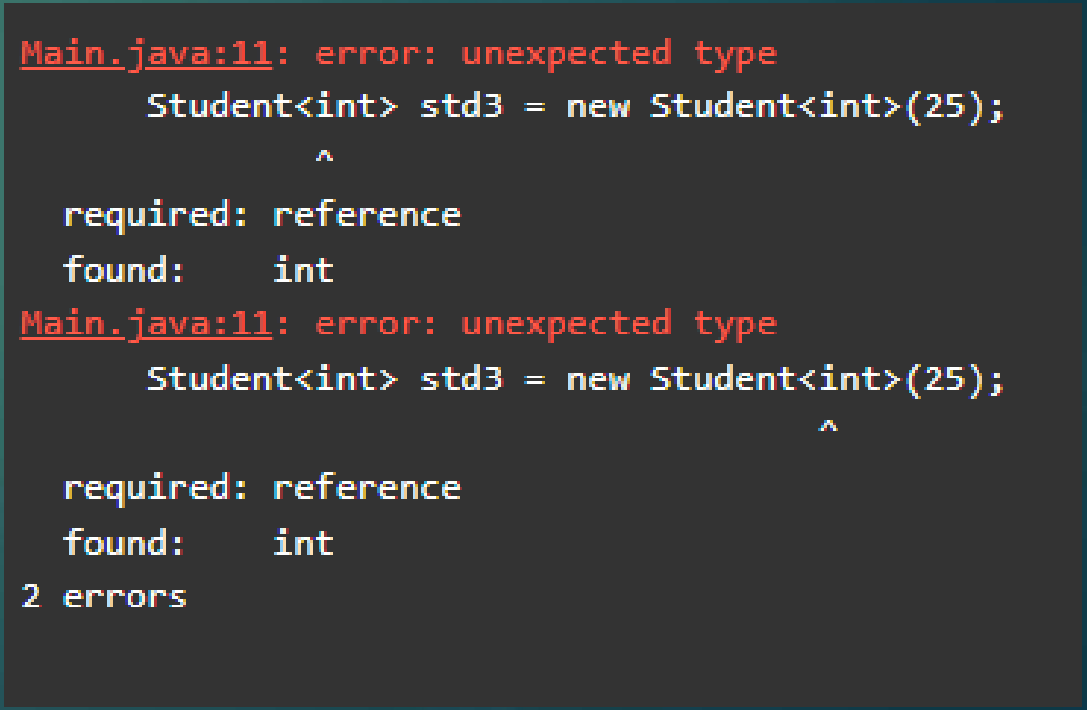
- You cannot use generics in certain ways and in certain scenarios as listed below –
  - You cannot use datatypes with generics.
  - You cannot instantiate the generic parameters.
  - The generic type parameter cannot be static.
  - You cannot cast parameterized type of one datatype to other.
  - You cannot create an array of generic type objects.
  - A generic type class cannot extend the throwable class therefore, you cannot catch or throw these objects.

Generic Restrictions

- You cannot use primitive datatypes with generics.

```
class Student<T>{
  T age;
  Student(T age){
    this.age = age;
  }
}
public class GenericsExample {
  public static void main(String args[]) {
    Student<Float> std1 = new Student<Float>(25.5f);
    Student<String> std2 = new Student<String>("25");
    Student<int> std3 = new Student<int>(25);
  }
}
```

Int is a primitive datatype and hence cannot be used with generics.
We need to mention Integer in generics to work with integer data.

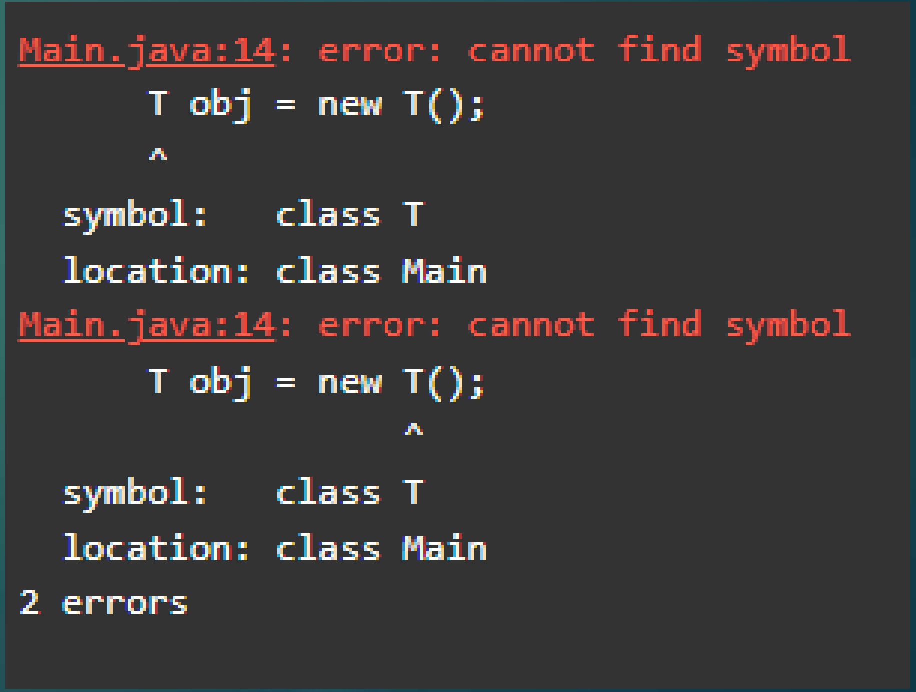


```
Main.java:11: error: unexpected type
        Student<int> std3 = new Student<int>(25);
                ^
  required: reference
  found:    int
Main.java:11: error: unexpected type
        Student<int> std3 = new Student<int>(25);
                                        ^
  required: reference
  found:    int
2 errors
```

Generic Restrictions

- You cannot instantiate the generic parameters.

```
class Student<T>{
  T age;
  Student(T age){
    this.age = age;
  }
  public void display() {
    System.out.println("Value of age: "+this.age);
  }
}
public class Main {
  public static void main(String args[]) {
    Student<Float> std1 = new Student<Float>(25.5f);
    std1.display();
    T obj = new T();
  }
}
```

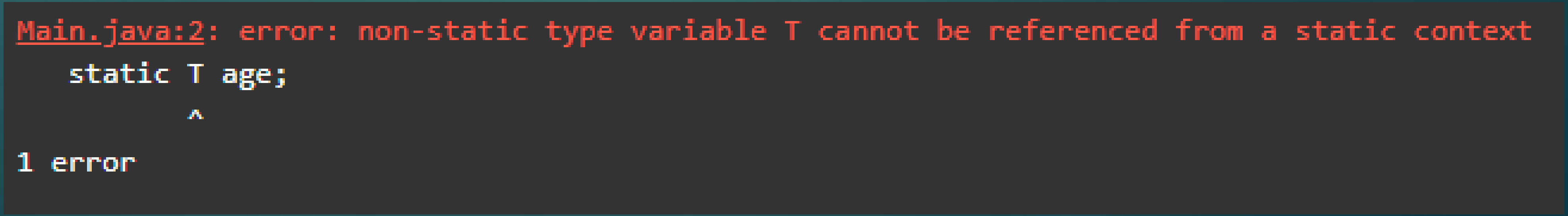


```
Main.java:14: error: cannot find symbol
      T obj = new T();
      ^
  symbol:   class T
  location: class Main
Main.java:14: error: cannot find symbol
      T obj = new T();
                  ^
  symbol:   class T
  location: class Main
2 errors
```

Generic Restrictions

- The generic type parameter cannot be static.

```
class Student<T>{
   static T age;
   Student(T age){
      this.age = age;
   }
   public void display() {
      System.out.println("Value of age: "+this.age);
   }
}
public class Main {
   public static void main(String args[]) {
      Student<Float> std1 = new Student<Float>(25.5f);
      std1.display();
   }
}
```

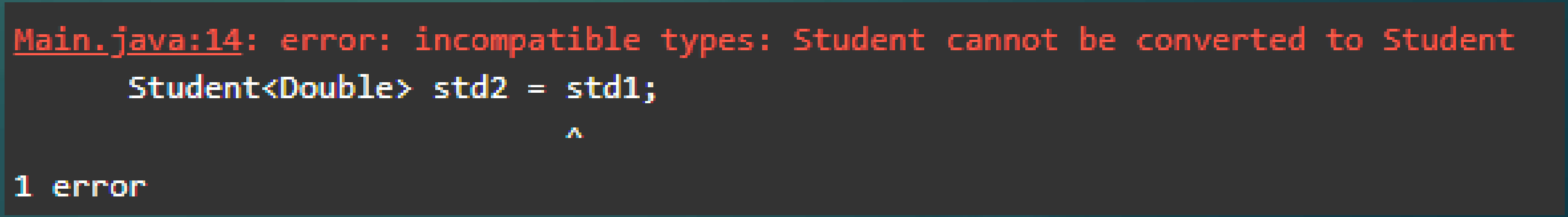


```
Main.java:2: error: non-static type variable T cannot be referenced from a static context
   static T age;
            ^
1 error
```

Generic Restrictions

- You cannot cast parameterized type of one datatype to other.

```
class Student<T>{
   T age;
   Student(T age){
      this.age = age;
   }
   public void display() {
      System.out.println("Value of age: "+this.age);
   }
}
public class Main {
   public static void main(String args[]) {
      Student<Float> std1 = new Student<Float>(25.5f);
      std1.display();
      Student<Double> std2 = std1;
      std2.display();
   }
}
```

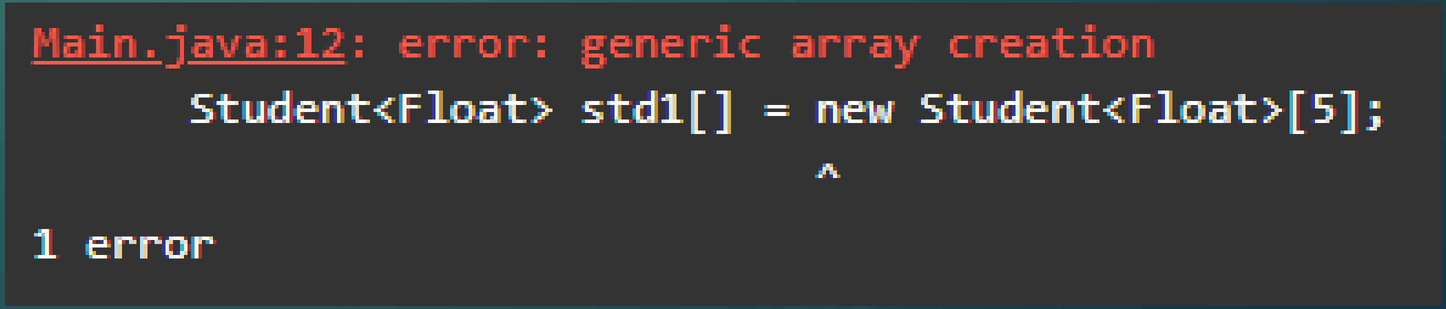


```
Main.java:14: error: incompatible types: Student cannot be converted to Student
      Student<Double> std2 = std1;
                             ^
1 error
```

Generic Restrictions

- You cannot create an array of generic type objects.
- Note : syntax for creating array of objects : Student s[]=new Student[n];

```
class Student<T>{
   T age;
   Student(T age){
      this.age = age;
   }
   public void display() {
      System.out.println("Value of age: "+this.age);
   }
}
public class Main {
   public static void main(String args[]) {
      Student<Float> std1[ ] = new Student<Float>[5];
   }
}
```



```
Main.java:12: error: generic array creation
        Student<Float> std1[] = new Student<Float>[5];
                                ^
1 error
```

Generic Class Hierarchies

- Generic classes can be part of a class hierarchy in just the same way as a non-generic class.
- Thus, a generic class can act as a superclass or be a subclass.
- The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses.
- A subclass can freely add its own type parameters, if necessary.

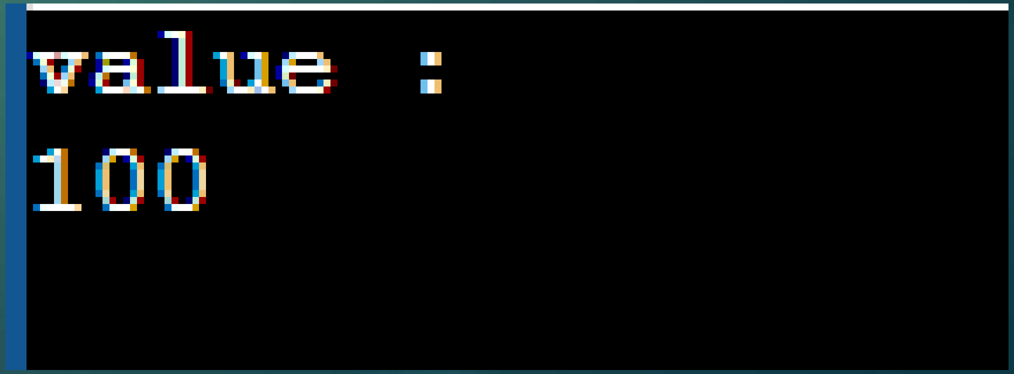
Generic Class Hierarchies

Eg 1 : Generic Super Class and Generic Sub Class

```
import java.io.*;
class Base<T> {                         // Class 1 - Parent class
        T obj;                          // Member variable of parent class
    Base(T o1) {                        // Constructor of parent class
        obj = o1;
    }
    T getobj1() {                       // Member function of parent class that returns an object
        return obj;
    }}

class Child<T> extends Base<T> {     // Class 2 - Child class
     T obj2;                            // Member variable of child class
    Child(T o1, T o2)  {                // Constructor of Child class
        super(o1);                      // Calling super class using super keyword
        obj2 = o2;
    }
    T getobj2() {                       // Member function of child class that returns an object
        return obj2;
    }}
```

Generic Class hierarchies

```
class Main {                                // Class 3 -  Main class
    public static void main(String[] args)
    {
        Child x = new Child("value : ",100);
        System.out.println(x.getobj1());
        System.out.println(x.getobj2());
    }}
```



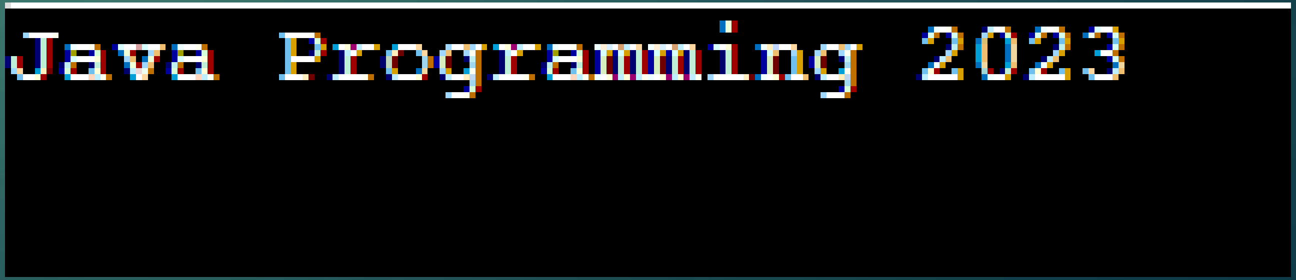
```
value :
100
```

Generic Class Hierarchies

Eg 1 : Non-Generic Super Class and Generic Sub Class

```
import java.io.*;
class Base {                // non-generic super-class
    int n;
   Base(int i) {
       n = i;
  }
    int getval() {
    return n;
    }}

class Child<T> extends Base {      // generic sub-class
    T obj;
    Child(T o1, int i)     {
        super(i);
        obj = o1;
    }
    T getobj() {
    return obj;
    }}
```

```
class Main {
    public static void main(String[] args)
    {
        Child c = new Child("Java Programming", 2023);
       System.out.println(c.getobj() + " " + c.getval());
    }
}
```



```
Java Programming 2023
```

Java Collections

Collections

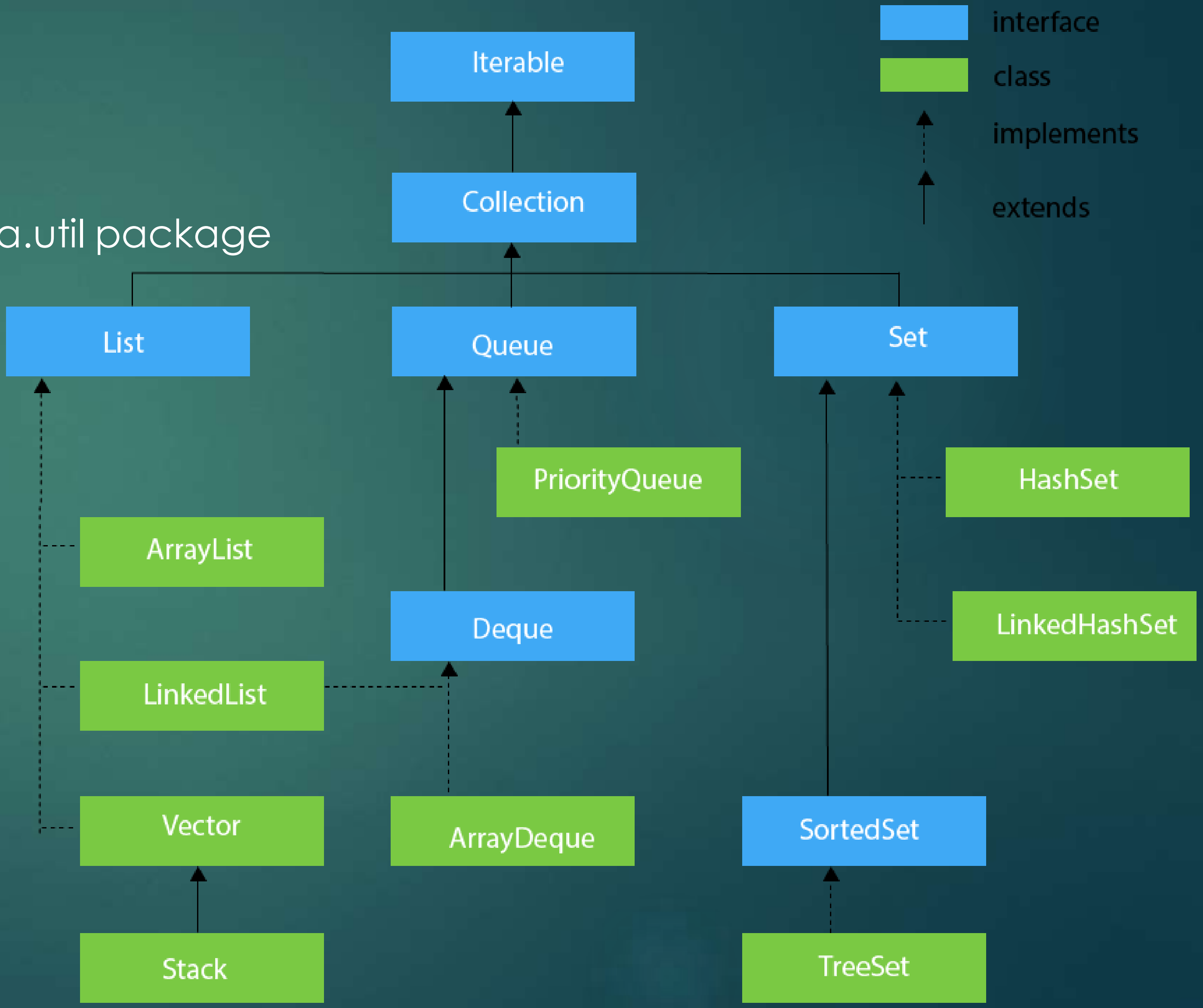
- The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects. A Collection represents a single unit of objects, i.e., a group.
- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

Why to use Java Collections

- There are several benefits of using Java collections such as:
  - Reducing the effort required to write the code by providing useful data structures and algorithms
  - Java collections provide high-performance and high-quality data structures and algorithms thereby increasing the speed and quality
  - Unrelated APIs can pass collection interfaces back and forth
  - Decreases extra effort required to learn, use, and design new API's
  - Supports reusability of standard data structures and algorithms

Collection Framework

- The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:
  - Interfaces and its implementations
  - Classes
  - Algorithm
- The Collections framework is defined in the java.util package



Java Collections : Interface

Iterator interface

- Iterator is an interface that iterates the elements.
- It is used to traverse the list and modify the elements.
- Iterator interface has three methods which are mentioned below:
  - public boolean hasNext() – This method returns true if the iterator has more elements otherwise it returns false.
  - public object next() – It returns the element and moves the cursor pointer to the next element.
  - public void remove() – This method removes the last element returned by the iterator.
- There are three components that extend the collection interface i.e List, Queue and Sets.

Iterable interface

- The Iterable interface is the root interface for all the collection classes.
- The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.
- It contains only one abstract method. i.e., Iterator<T> iterator()
- It returns the iterator over the elements of type T.

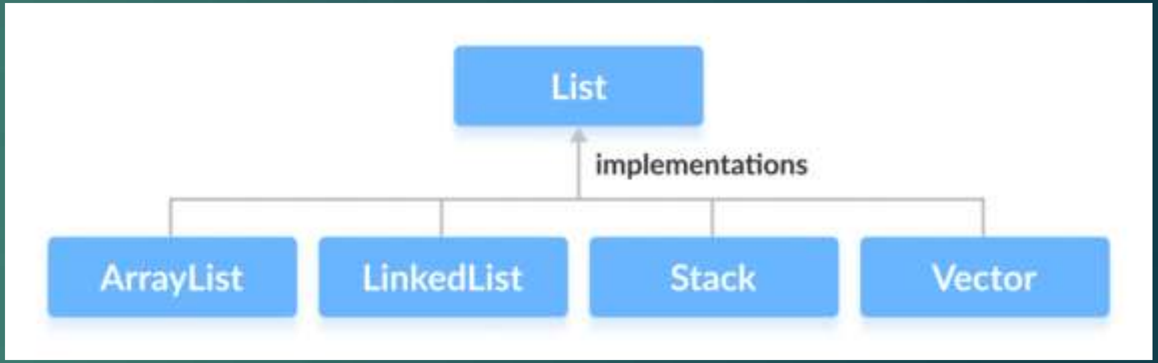
Java Collections : Interface

Collection interface

- The Collection interface is the root interface of the collections framework hierarchy.
- Java does not provide direct implementations of the Collection interface but provides implementations of its sub-interfaces like List, Set, and Queue
- The Collection interface is the interface which is implemented by all the classes in the collection framework.
- the Collection interface builds the foundation on which the collection framework depends.
- **Methods of Collection** : The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its sub interfaces.
  - add() - inserts the specified element to the collection
  - size() - returns the size of the collection
  - remove() - removes the specified element from the collection
  - iterator() - returns an iterator to access elements of the collection
  - addAll() - adds all the elements of a specified collection to the collection
  - removeAll() - removes all the elements of the specified collection from the collection
  - clear() - removes all the elements of the collection

Java Collections : List Interface

- A List is an ordered Collection of elements **which may contain duplicates.**
- It allows us to add and remove elements like an array
- It is an interface that extends the Collection interface.
- Since List is an interface, we cannot create objects from it.
- List interface is implemented by the following classes -
  - ArrayList
  - LinkedList
  - Vectors
  - Stack
- To instantiate the List interface, we must use :
  - List <data-type> list1= new ArrayList();
  - List <data-type> list2 = new LinkedList();
  - List <data-type> list3 = new Vector();
  - List <data-type> list4 = new Stack();

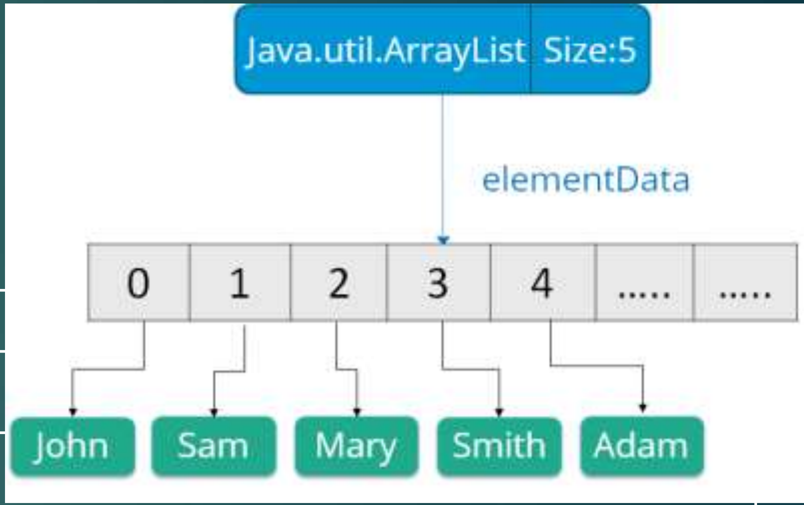


Java Collections : List Interface

- **Methods of List** : The List interface includes all the methods of the Collection interface. Its because Collection is a super interface of List. Some of the commonly used methods of the Collection interface that's also available in the List interface are:
  - add() - adds an element to a list
  - addAll() - adds all elements of one list to another
  - get() - helps to randomly access elements from lists
  - iterator() - returns iterator object that can be used to sequentially access elements of lists
  - set() - changes elements of lists
  - remove() - removes an element from the list
  - removeAll() - removes all the elements from the list
  - clear() - removes all the elements from the list (more efficient than removeAll())
  - size() - returns the length of lists
  - toArray() - converts a list into an array
  - contains() - returns true if a list contains specified element

ArrayList

- ArrayList is the implementation of List Interface where the elements can be dynamically added or removed from the list.
- It uses a dynamic array to store the duplicate element of different data types.
- Also, the size of the list is increased dynamically if the elements are added more than the initial size.
- The ArrayList class maintains the insertion order and is non-synchronized.
- The elements stored in the ArrayList class can be randomly accessed.
- Syntax: `ArrayList object = new ArrayList ();`
- Some of the methods in array list are listed below:

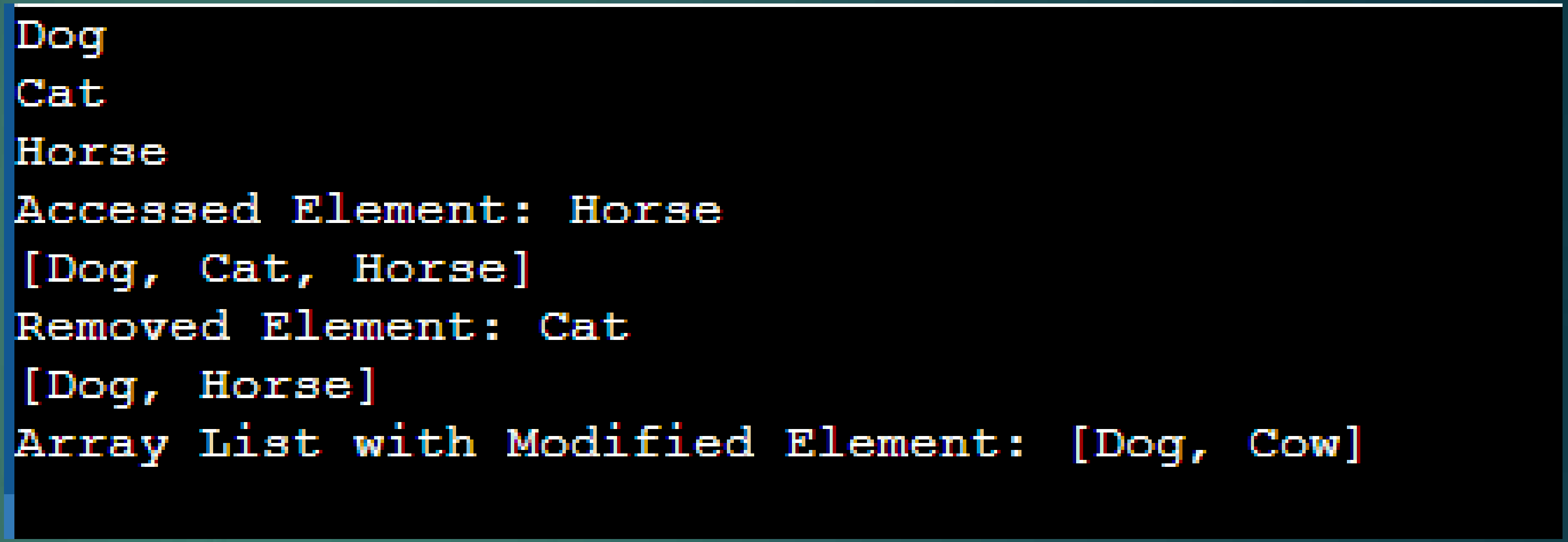


Method	Description
boolean add(Collection c)	Appends the specified element to the end of a list.
void add(int index, Object element)	Inserts the specified element at the specified position.
void clear()	Removes all the elements from this list.
int lastIndexOf(Object o)	Return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element.
Object clone()	Return a shallow copy of an ArrayList.
Object[] toArray()	Returns an array containing all the elements in the list.
void trimToSize()	Trims the capacity of this ArrayList instance to be the list's current size.

ArrayList

```
import java.util.*;
 class Main{
 public static void main(String args[]){
 ArrayList<String> animals=new ArrayList<String>();  // creating array list
 animals.add("Dog");
 animals.add("Cat");
 animals.add("Horse");
 Iterator itr=animals.iterator();
 System.out.println("Array List : ");
 while(itr.hasNext()){
 System.out.println(itr.next());
 }

 String a1 = animals.get(2);                    // Access element from the list
 System.out.println("Accessed Element: " + a1);
 System.out.println(animals);

String r = animals.remove(1);                  // Remove element from the list
System.out.println("Removed Element: " + r);
System.out.println(animals);
```

```
animals.set(1,"Cow");          // Changing element in the list
System.out.println("Array List with Modified Element: " + animals);
}}
```

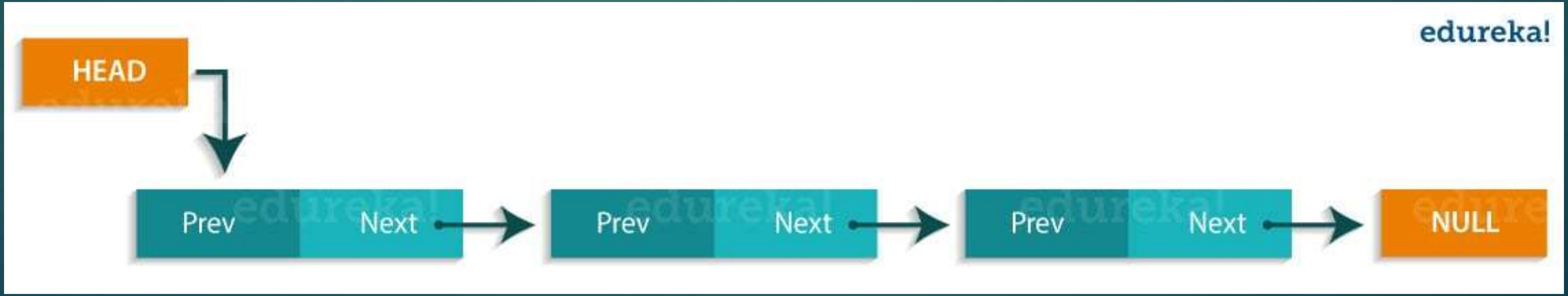


```
Dog
Cat
Horse
Accessed Element: Horse
[Dog, Cat, Horse]
Removed Element: Cat
[Dog, Horse]
Array List with Modified Element: [Dog, Cow]
```

LinkedList

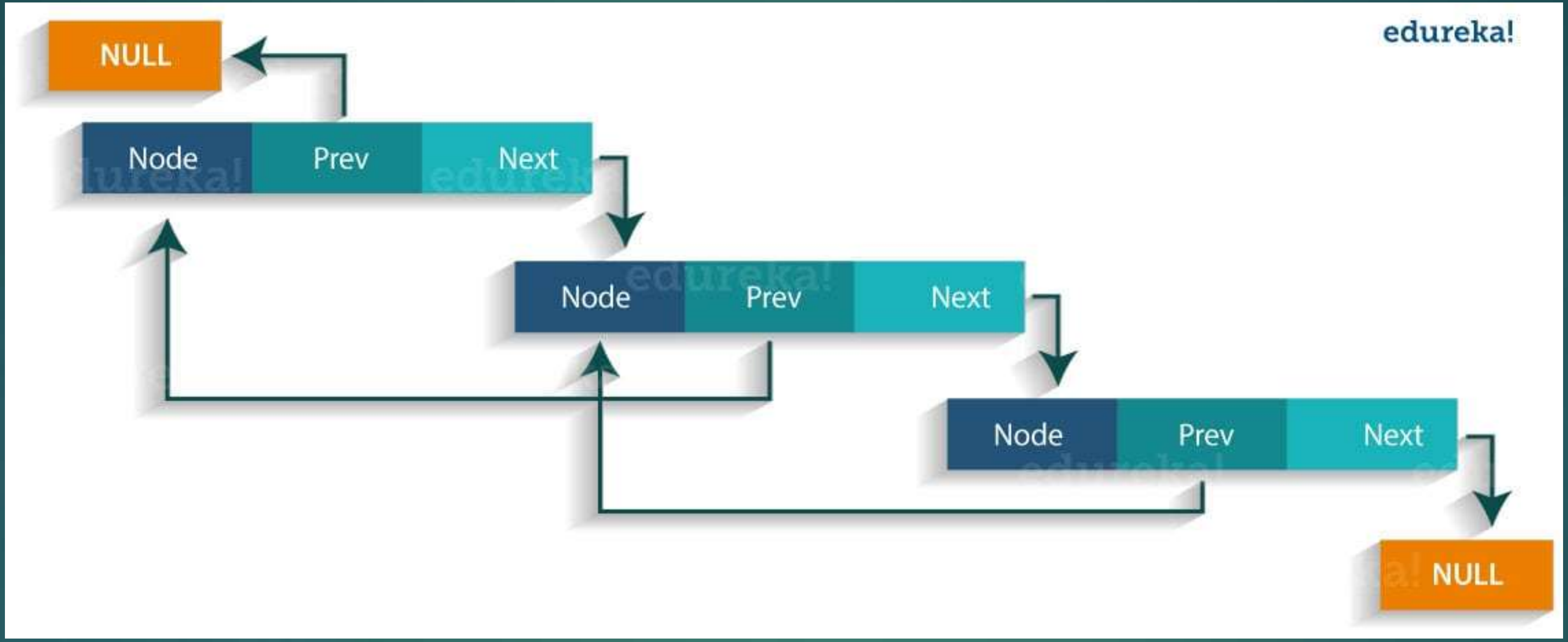
- Linked List is a sequence of links which contains items. Each link contains a connection to another link.
- Linked List implements the Collection interface.
- It uses a doubly linked list internally to store the elements.
- It can store the duplicate elements.
- It maintains the insertion order and is not synchronized.
- In Linked List, the manipulation is fast because no shifting is required.
- Syntax: `Linkedlist object = new Linkedlist();`
- Java Linked List class uses two types of Linked list to store the elements:

1. Singly Linked List: In a singly Linked list each node in this list stores the data of the node and a pointer or reference to the next node in the list. Refer to the below image to get a better understanding of single Linked list.



LinkedList

2. Doubly Linked List: In a doubly Linked list, it has two references, one to the next node and another to previous node. You can refer to the below image to get a better understanding of doubly linked list.



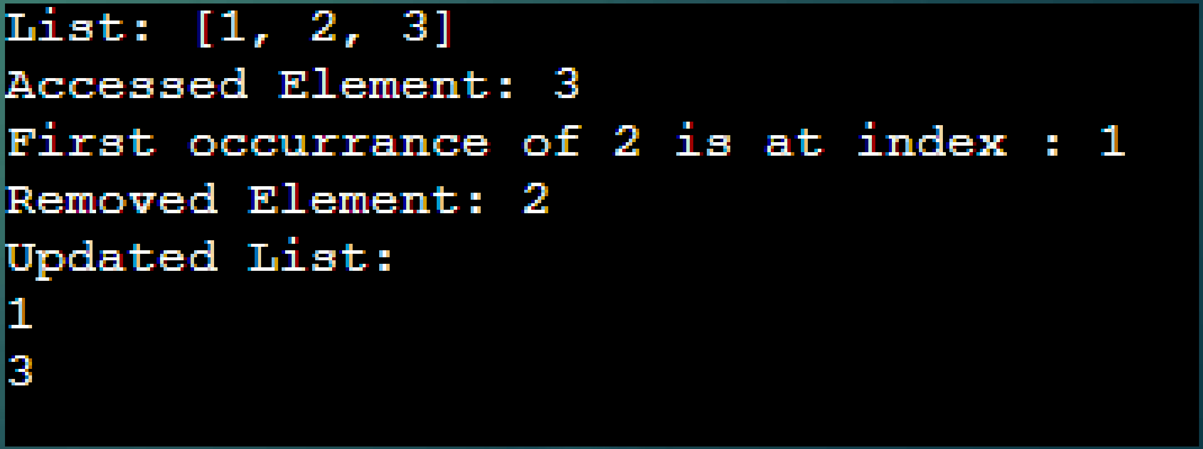
LinkedList

- Some of the methods in the linked list are listed below:

Method	Description
boolean add(Object o)	It is used to append the specified element to the end of the vector.
boolean contains(Object o)	Returns true if this list contains the specified element.
void add (int index, Object element)	Inserts the element at the specified element in the vector.
void addFirst(Object o)	It is used to insert the given element at the beginning.
void addLast(Object o)	It is used to append the given element to the end.
int size()	It is used to return the number of elements in a list
boolean remove(Object o)	Removes the first occurrence of the specified element from this list.
int indexOf(Object element)	Returns the index of the first occurrence of the specified element in this list, or -1.
int lastIndexOf(Object element)	Returns the index of the last occurrence of the specified element in this list, or -1.

LinkedList

```
import java.util.*;
class Main {
    public static void main(String[] args) {
        // Creating list using the LinkedList clas
        List<Integer> numbers = new LinkedList<>();

        // Add elements to the list
        numbers.add(1);
        numbers.add(2);
        numbers.add(3);
        System.out.println("List: " + numbers);

        // Access element from the list
        int number = numbers.get(2);
        System.out.println("Accessed Element: " + number);

        // Using the indexOf() method
        int index = numbers.indexOf(2);
        System.out.println("First occurrance of 2 is at index : " + index);

        // Remove element from the list
        int removedNumber = numbers.remove(1);
        System.out.println("Removed Element: " + removedNumber);

        Iterator itr = numbers.iterator();
        System.out.println("Updated List: ");
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```



```
List: [1, 2, 3]
Accessed Element: 3
First occurrance of 2 is at index : 1
Removed Element: 2
Updated List:
1
3
```

Vector

- Vectors are similar to arrays, where the elements of the vector object can be accessed via an index into the vector.
- Vector implements a dynamic array.
- Also, the vector is not limited to a specific size, it can shrink or grow automatically whenever required. It is similar to ArrayList, but with two differences :
  - Vector is synchronized.
  - Vector contains many legacy methods that are not part of the collections framework.
- Syntax: Vector object = new Vector(size,increment)
- Below are some of the methods of the Vector class:

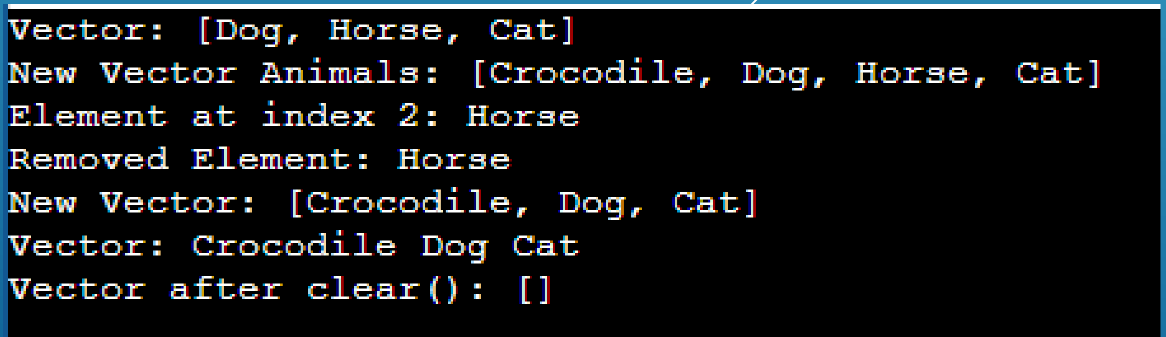
Method	Description
boolean add(Object o)	Appends the specified element to the end of the list.
void clear()	Removes all of the elements from this list.
void add(int index, Object element)	Inserts the specified element at the specified position.
boolean remove(Object o)	Removes the first occurrence of the specified element from this list.
boolean contains(Object element)	Returns true if this list contains the specified element.
int indexOfObject (Object element)	Returns the index of the first occurrence of the specified element in the list, or -1.
int size()	Returns the number of elements in this list.
int lastIndexOf(Object o)	Return the index of the last occurrence of the specified element in the list, or -1 if the list does not contain any element.

Vector

- Vector is synchronized. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.
- It is because when one thread is accessing a vector, and at the same time another thread tries to access it, an exception called ConcurrentModificationException is generated. Hence, this continuous use of lock for each operation makes vectors less efficient.
- However, in array lists, methods are not synchronized. Instead, it uses the Collections.synchronizedList() method that synchronizes the list as a whole.
- Hence, It is recommended to use ArrayList in place of Vector because vectors less efficient.

Vector

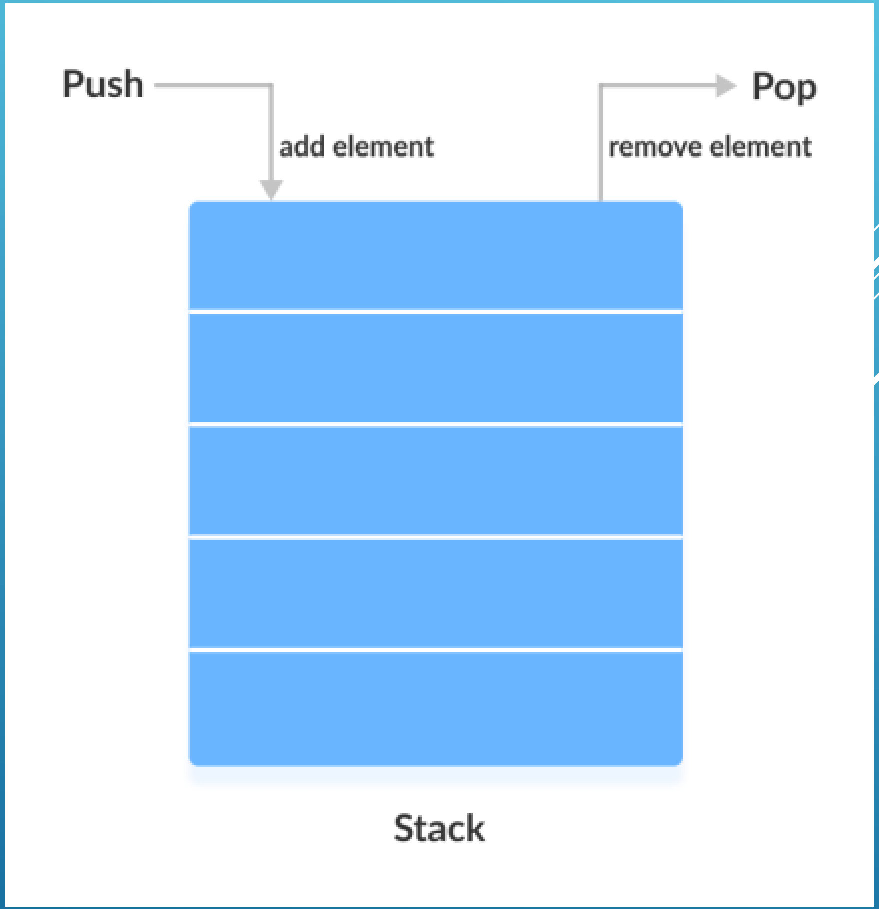
```java
import java.util.*;

class Main {

  public static void main(String[] args) {

    Vector<String> mammals= new Vector<>();


    // Using the add() method

    mammals.add("Dog");

    mammals.add("Horse");

    mammals.add(2, "Cat");   // Using index number

    System.out.println("Vector: " + mammals);


    // Using addAll()

    Vector<String> animals = new Vector<>();

    animals.add("Crocodile");

    animals.addAll(mammals);   // copy mammals vector to animals vector

    System.out.println("New Vector Animals: " + animals);


     String element = animals.get(2);  // access elements from a vector

    System.out.println("Element at index 2: " + element);

    // Remove Element
     System.out.println("Removed Element: " + animals.remove(2));
     System.out.println("New Vector: " + animals);

    // Using iterator()
     Iterator<String> iterate = animals.iterator();
     System.out.print("Vector: ");
     while(iterate.hasNext()) {
     System.out.print(iterate.next() + " ");
     }
     System.out.println();

     animals.clear();    // Using clear()
     System.out.println("Vector after clear(): " + animals);
 }}
```



```
Vector: [Dog, Horse, Cat]
New Vector Animals: [Crocodile, Dog, Horse, Cat]
Element at index 2: Horse
Removed Element: Horse
New Vector: [Crocodile, Dog, Cat]
Vector: Crocodile Dog Cat
Vector after clear(): []
```

Stack

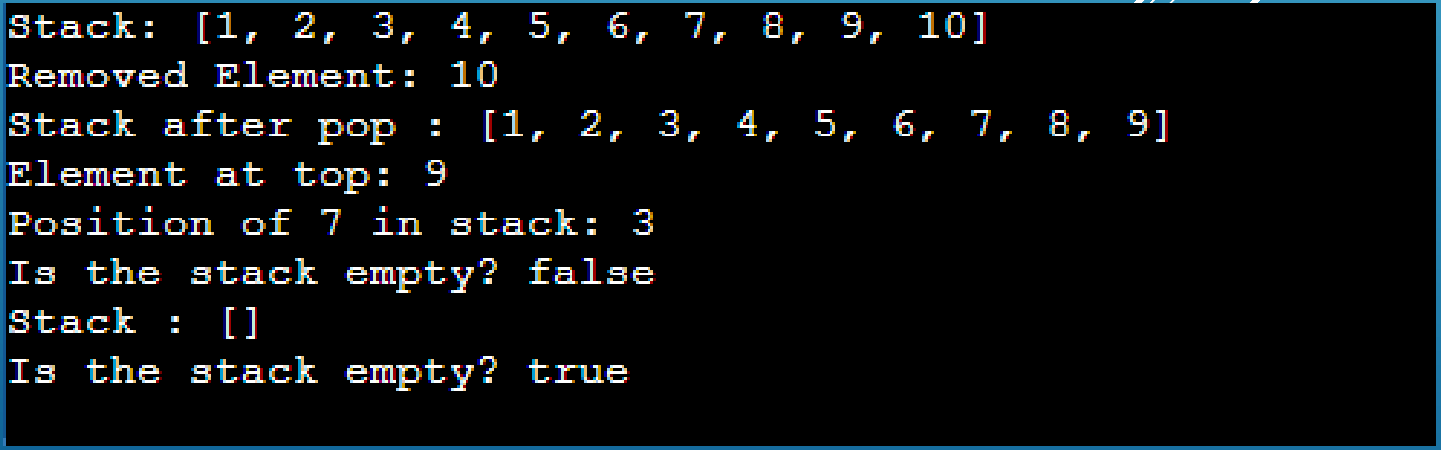
- The stack is the subclass of Vector.
- It implements the last-in-first-out data structure i.e. elements are added to the top of the stack and removed from the top of the stack.
- The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.



Stack

```
import java.util.*;
class Main {
    public static void main(String[] args) {
        Stack<Integer> nos= new Stack<>();

        // Add elements to Stack using push method
        for(int i=1;i<=10;i++)
        {
        nos.push(i);
        }
        System.out.println("Stack: " + nos);

        // Remove element from stack using pop method
        System.out.println("Removed Element: " + nos.pop());
        System.out.println("Stack after pop : " + nos);

        // Access element from the top
        System.out.println("Element at top: " + nos.peek());
```

```
 // Searching an element in the stack
  // search method returns the position of the element from the top of the stack.
  // It returns position and not the index
  int position = nos.search(7);
  System.out.println("Position of 7 in stack: " + position);

  // Check if stack is empty
 System.out.println("Is the stack empty? " + nos.empty());

 nos.clear();   // clear the stack
 System.out.println("Stack : " + nos);
 System.out.println("Is the stack empty? " + nos.empty());
}}
```

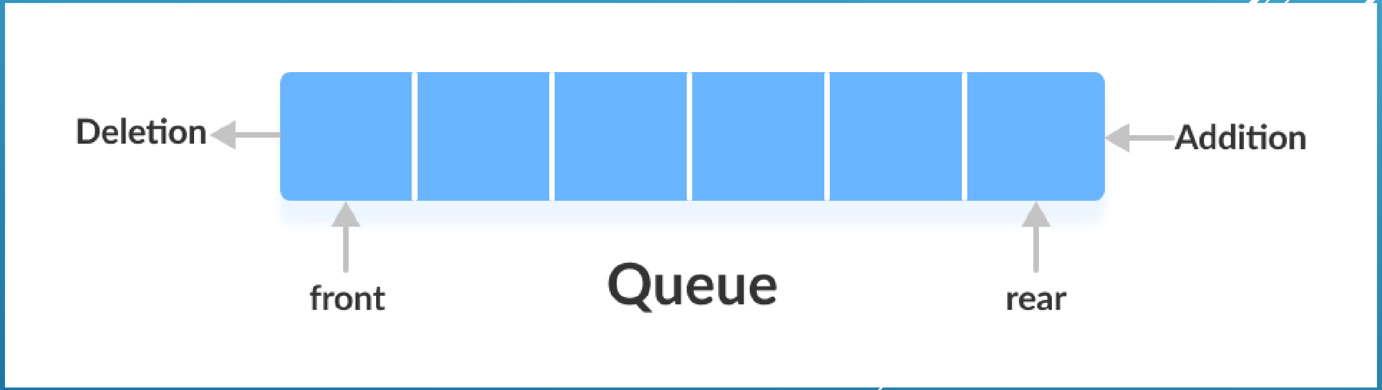


```
Stack: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Removed Element: 10
Stack after pop : [1, 2, 3, 4, 5, 6, 7, 8, 9]
Element at top: 9
Position of 7 in stack: 3
Is the stack empty? false
Stack : []
Is the stack empty? true
```

Java Collections : Queue Interface

- Queue interface maintains the first-in-first-out order.
- It can be defined as an ordered list that is used to hold the elements which are about to be processed.
- There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.
- In a queue, the first element is removed first and last element is removed in the end.
- Since the Queue is an interface, we cannot provide the direct implementation of it.
- In order to use the functionalities of Queue, we need to use classes that implement it:
  - ArrayDeque
  - PriorityQueue
- Queue interface can be instantiated as:

```
Queue<String> q1 = new PriorityQueue();
Queue<String> q2 = new ArrayDeque();
```



Java Collections : Queue Interface

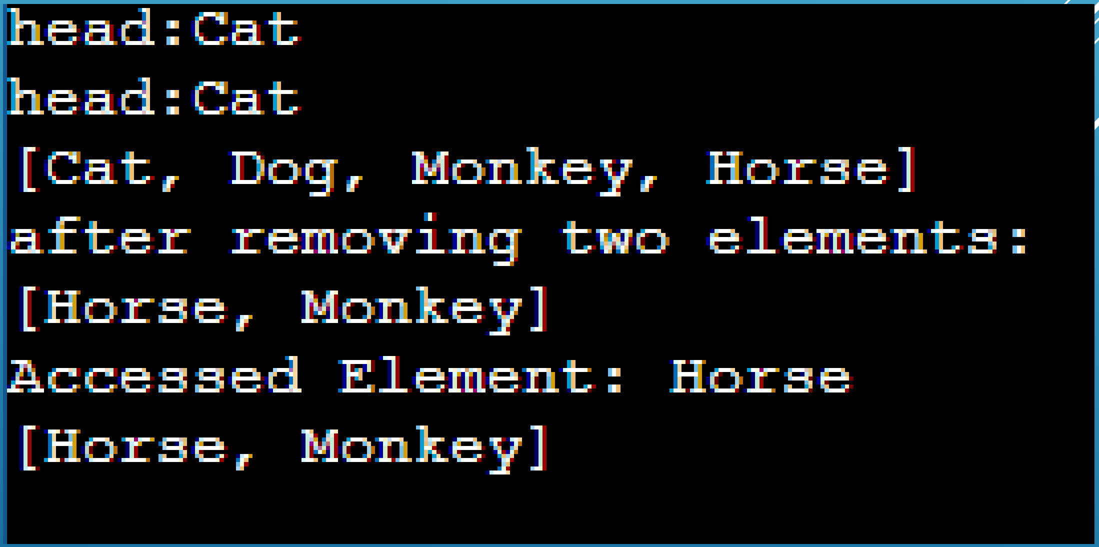
- Some of the commonly used methods of the Queue interface are:
  - add() - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.
  - offer() - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.
  - element() - Returns the head of the queue. Throws an exception if the queue is empty.
  - peek() - Returns the head of the queue. Returns null if the queue is empty.
  - remove() - Returns and removes the head of the queue. Throws an exception if the queue is empty.
  - poll() - Returns and removes the head of the queue. Returns null if the queue is empty.

PriorityQueue Class

- The PriorityQueue class implements the Queue interface.
- It holds the elements or objects which are to be processed by their priorities.
- PriorityQueue doesn't allow null values to be stored in the queue.
- The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at the queue construction time.
- The head of this queue is the least element with respect to the specified ordering.

Priority Queue

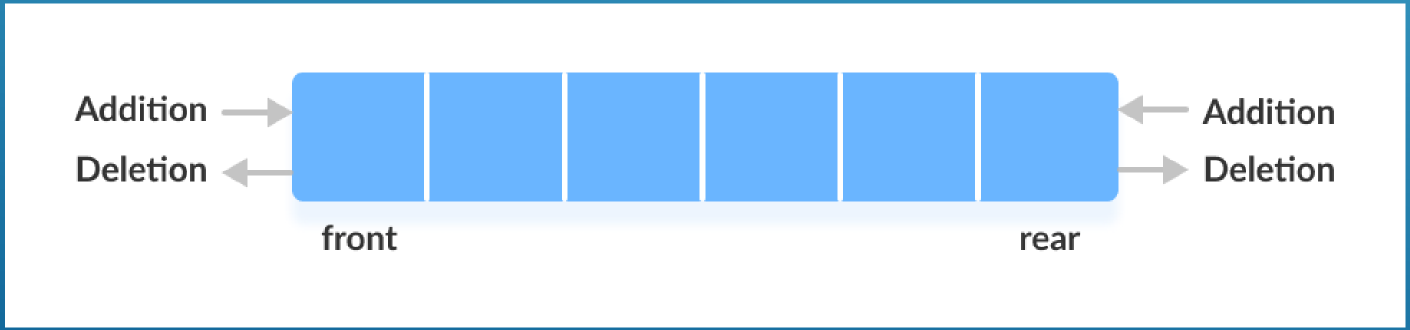
```
import java.util.*;

class Main {

  public static void main(String args[]){

     // creating priority queue

     PriorityQueue<String> queue=new PriorityQueue<String>();



// adding elements

     queue.add("Cat");   // add() Inserts the specified element into the
queue. If the task is successful, it returns true, if not it throws an
exception.

     queue.add("Dog");

     queue.offer("Monkey"); // offer() Inserts the specified element into
the queue. If the task is successful, it returns true, if not it returns false.

     queue.offer("Horse");

     System.out.println("head:"+queue.element()); //element() method
returns head of the queue. It throws an exception if the queue is empty.

     System.out.println("head:"+queue.peek()); // peek() method returns
head of the queue. It returns null if the queue is empty.

      System.out.println(queue);

     queue.remove();  //remove() method returns and removes the head
of the queue. Throws an exception if the queue is empty.

     queue.poll();   // poll() method returns and removes the head of
the queue. Returns null if the queue is empty.

     System.out.println("after removing two elements:");

     System.out.println(queue);


     System.out.println("Accessed Element: " + queue.peek());
System.out.println(queue);

  }}
```

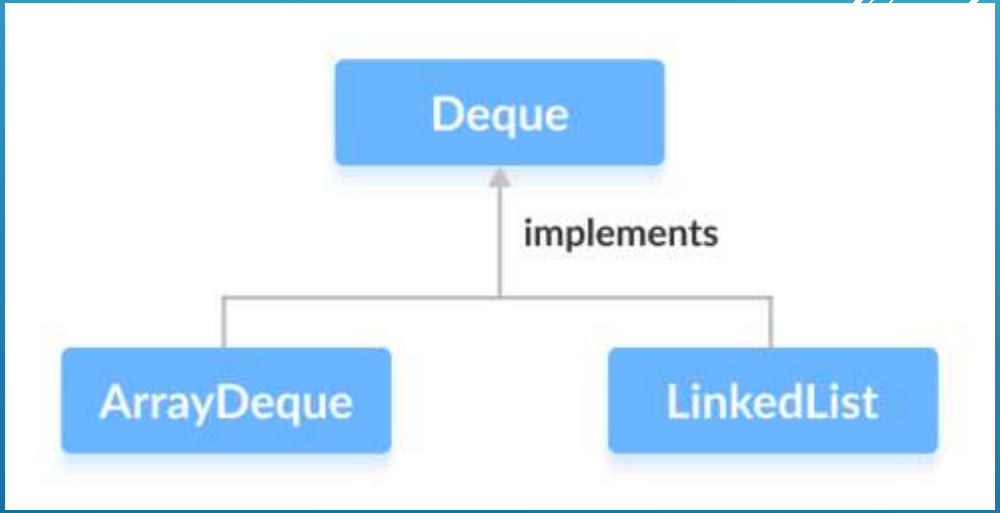


```
head:Cat
head:Cat
[Cat, Dog, Monkey, Horse]
after removing two elements:
[Horse, Monkey]
Accessed Element: Horse
[Horse, Monkey]
```

Deque Interface (Double Ended Queue)

- The Deque interface of the Java collections framework provides the functionality of a double-ended queue.
- It extends the Queue interface.
- In a regular queue, elements are added from the rear and removed from the front.
- However, in a deque, we can insert and remove elements from both front and rear.
- In order to use the functionalities of the Deque interface, we need to use classes that implement it:
  - ArrayDeque
  - LinkedList
- ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.





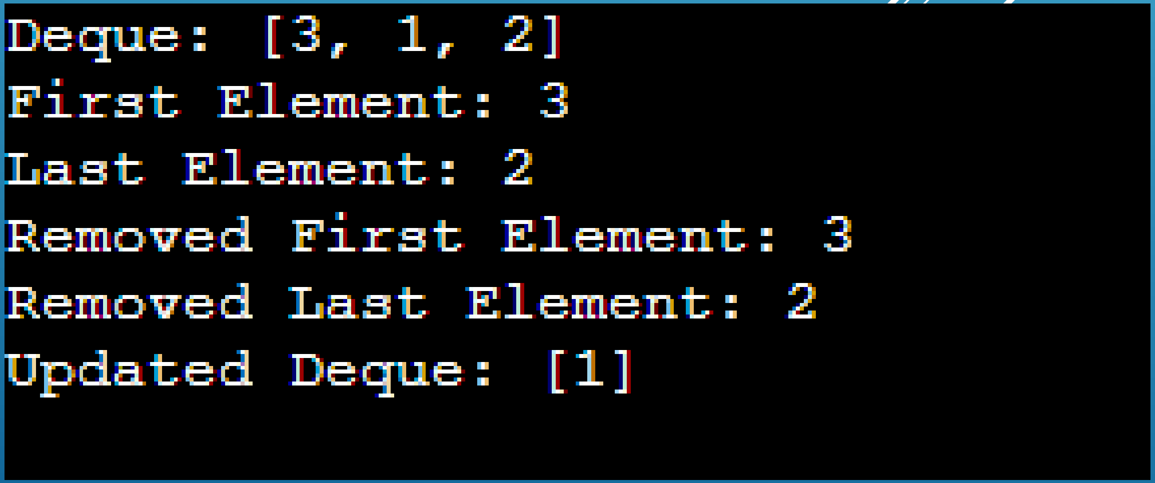
Deque Interface (Double Ended Queue)

- **Methods of Deque :** Since Deque extends the Queue interface, it inherits all the methods of the Queue interface.
- Besides methods available in the Queue interface, the Deque interface also includes the following methods:
  - addFirst() - Adds the specified element at the beginning of the deque. Throws an exception if the deque is full.
  - addLast() - Adds the specified element at the end of the deque. Throws an exception if the deque is full.
  - offerFirst() - Adds the specified element at the beginning of the deque. Returns false if the deque is full.
  - offerLast() - Adds the specified element at the end of the deque. Returns false if the deque is full.
  - getFirst() - Returns the first element of the deque. Throws an exception if the deque is empty.
  - getLast() - Returns the last element of the deque. Throws an exception if the deque is empty.
  - peekFirst() - Returns the first element of the deque. Returns null if the deque is empty.
  - peekLast() - Returns the last element of the deque. Returns null if the deque is empty.
  - removeFirst() - Returns and removes the first element of the deque. Throws an exception if the deque is empty.
  - removeLast() - Returns and removes the last element of the deque. Throws an exception if the deque is empty.
  - pollFirst() - Returns and removes the first element of the deque. Returns null if the deque is empty.
  - pollLast() - Returns and removes the last element of the deque. Returns null if the deque is empty.

Deque interface

```
import java.util.*;

class Main {

    public static void main(String[] args) {
        // Creating Deque using the ArrayDeque class
        Deque<Integer> numbers = new ArrayDeque<>();


        // add elements to the Deque
        numbers.offer(1);
        numbers.offerLast(2);
        numbers.offerFirst(3);
        System.out.println("Deque: " + numbers);


        // Access elements of the Deque
        System.out.println("First Element: " + numbers.peekFirst());
        System.out.println("Last Element: " + numbers.peekLast());


        // Remove elements from the Deque
        System.out.println("Removed First Element: " + numbers.pollFirst());
```

```
System.out.println("Updated Deque: " + numbers);
    }
}
```



```
Deque: [3, 1, 2]
First Element: 3
Last Element: 2
Removed First Element: 3
Removed Last Element: 2
Updated Deque: [1]
```

Java Collections : Set Interface

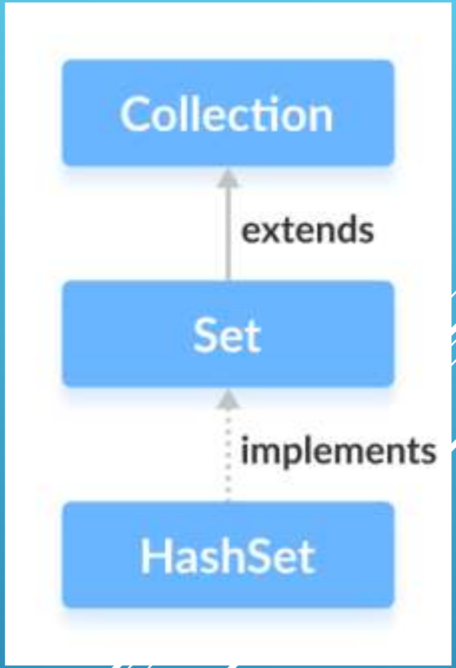
- The Set interface of the Java Collections framework provides the features of the mathematical set in Java.
- It extends the Collection interface.
- Unlike the List interface, sets cannot contain duplicate elements.
- Since Set is an interface, we cannot create objects from it.
- In order to use functionalities of the Set interface, we can use these classes:
    - HashSet
    - LinkedHashSet
    - EnumSet
    - TreeSet
- These classes are defined in the Collections framework and implement the Set interface.
- Syntax : Set<String> abc = new HashSet<>();
- **Set Operations :** The Java Set interface allows us to perform basic mathematical set operations like -
- Union - to get the union of two sets x and y, we can use x.addAll(y)
- Intersection - to get the intersection of two sets x and y, we can use x.retainAll(y)
- Subset - to check if x is a subset of y, we can use y.containsAll(x)

Java Collections : Set Interface

- **Methods of Set**
- The Set interface includes all the methods of the Collection interface. It's because Collection is a super interface of Set.
- Some of the commonly used methods of the Collection interface that's also available in the Set interface are:
  - add() - adds the specified element to the set
  - addAll() - adds all the elements of the specified collection to the set
  - iterator() - returns an iterator that can be used to access elements of the set sequentially
  - remove() - removes the specified element from the set
  - removeAll() - removes all the elements from the set that is present in another specified set
  - retainAll() - retains all the elements in the set that are also present in another specified set
  - clear() - removes all the elements from the set
  - size() - returns the length (number of elements) of the set
  - toArray() - returns an array containing all the elements of the set
  - contains() - returns true if the set contains the specified element
  - containsAll() - returns true if the set contains all the elements of the specified collection
  - hashCode() - returns a hash code value (address of the element in the set)

HashSet Class

- Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.
- The important points about Java HashSet class are:
  - HashSet stores the elements by using a mechanism called hashing.
  - HashSet contains unique elements only.
  - HashSet allows null value.
  - HashSet class is non synchronized.
  - HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
  - HashSet is the best approach for search operations.
  - The initial default capacity of HashSet is 16, and the load factor is 0.75.
- Syntax : HashSet<Integer> numbers = new HashSet<>(8, 0.75);
- The above syntax creates HashSet with 8 capacity and 0.75 load factor
- HashSet is commonly used if we have to access elements randomly. It is because elements in a hash table are accessed using hash codes.
- The hashcode of an element is a unique identity that helps to identify the element in a hash table.
- HashSet cannot contain duplicate elements. Hence, each hash set element has a unique hashcode.

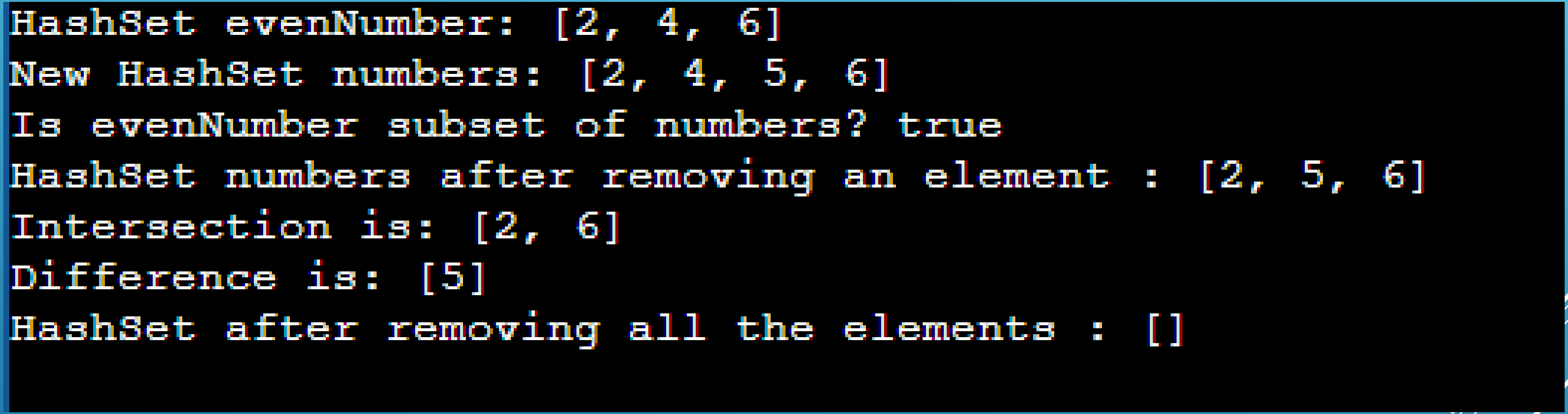


HashSet Class

```
class Main {

    public static void main(String[] args) {

        HashSet<Integer> evenNumber = new HashSet<>();


        // Using add() method

        evenNumber.add(2);

        evenNumber.add(4);

        evenNumber.add(6);

        System.out.println("HashSet evenNumber: " + evenNumber);


        // Using addAll() method

        HashSet<Integer> numbers = new HashSet<>();

        numbers.addAll(evenNumber);     // aka Union operation. Union of
numbers and evenNumber sets

        numbers.add(5);

        System.out.println("New HashSet numbers: " + numbers);


        //to check if a set is a subset of another set or not, we can use the
containsAll() method.

        boolean result = numbers.containsAll(evenNumber);
```

```
//using remove() method

        numbers.remove(4);

        System.out.println("HashSet numbers after removing an
element : " + numbers);


        //intersection of two sets

        numbers.retainAll(evenNumber);

        System.out.println("Intersection is: " + numbers);


        numbers.add(5);

        //difference of two sets

        numbers.removeAll(evenNumber);

        System.out.println("Difference is: " + numbers);


       //using removeAll() method

       numbers.removeAll(numbers);

        System.out.println("HashSet after removing all the elements
: " + numbers);

    }

}
```

HashSet Class



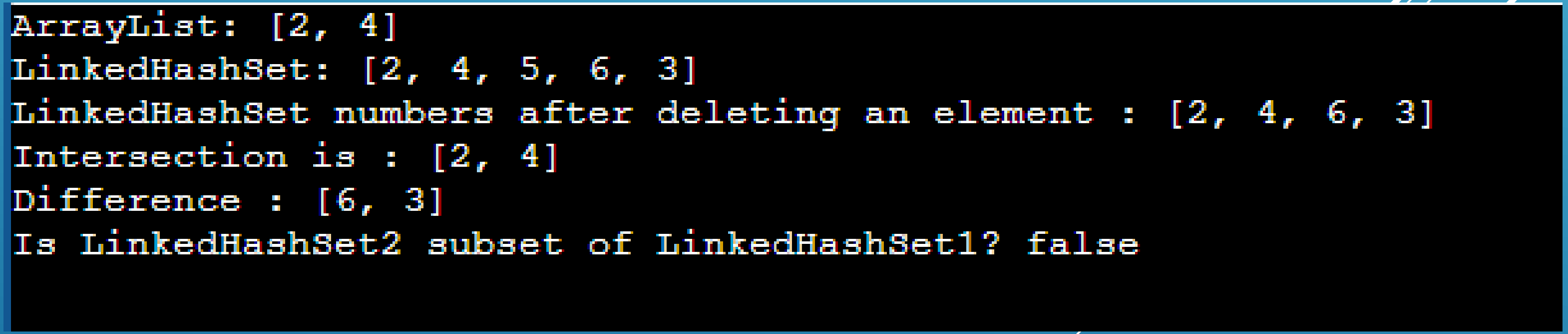
```
HashSet evenNumber: [2, 4, 6]
New HashSet numbers: [2, 4, 5, 6]
Is evenNumber subset of numbers? true
HashSet numbers after removing an element : [2, 5, 6]
Intersection is: [2, 6]
Difference is: [5]
HashSet after removing all the elements : []
```

LinkedHashSet Class

- Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.
- The important points about the Java LinkedHashSet class are:
  - Java LinkedHashSet class contains unique elements only like HashSet.
  - Java LinkedHashSet class provides all optional set operations and permits null elements.
  - Java LinkedHashSet class is non-synchronized.
  - Java LinkedHashSet class maintains insertion order.
- linked hash sets maintain a doubly-linked list internally for all of its elements. The linked list defines the order in which elements are inserted in hash tables.

LinkedHashSet Class

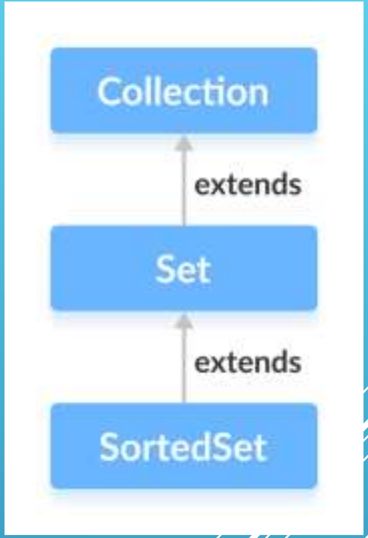
```
class Main {

   public static void main(String[] args) {

      // Creating an arrayList of even numbers

      ArrayList<Integer> evenNumbers = new ArrayList<>();

      evenNumbers.add(2);

      evenNumbers.add(4);

      System.out.println("ArrayList: " + evenNumbers);


      // Creating a LinkedHashSet from an ArrayList

      LinkedHashSet<Integer> numbers = new
LinkedHashSet<>(evenNumbers);

      numbers.add(5);

      numbers.add(6);

      numbers.add(3);

      System.out.println("LinkedHashSet: " + numbers);


      numbers.remove(5);

      System.out.println("LinkedHashSet numbers after deleting an
element : " + numbers);

evenNumbers.retainAll(numbers);
      System.out.println("Intersection is : " + evenNumbers);

      numbers.removeAll(evenNumbers);
      System.out.println("Difference : " + numbers);

      boolean result = numbers.containsAll(evenNumbers);
      System.out.println("Is LinkedHashSet2 subset of LinkedHashSet1? " +
result);
   }
}
```

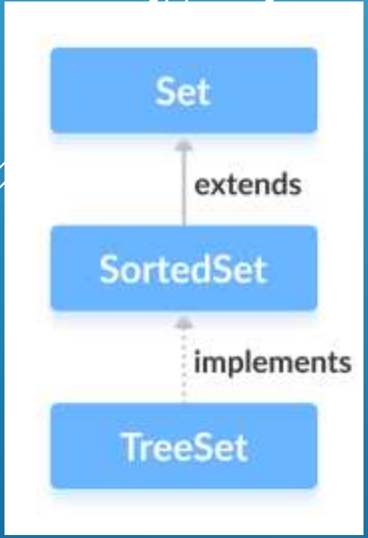


```
ArrayList: [2, 4]
LinkedHashSet: [2, 4, 5, 6, 3]
LinkedHashSet numbers after deleting an element : [2, 4, 6, 3]
Intersection is : [2, 4]
Difference : [6, 3]
Is LinkedHashSet2 subset of LinkedHashSet1? false
```

SortedSet Interface

- The SortedSet interface of the Java Collections framework is used to store elements with some order in a set.
- It extends the Set interface.
- In order to use the functionalities of the SortedSet interface, we need to use the TreeSet class that implements it.
- Syntax : SortedSet<String> abc=new TreeSet<>();
- Here we have used no arguments to create a sorted set. Hence the set will be sorted naturally.
- **Methods of SortedSet**
- The SortedSet interface includes all the methods of the Set interface. It's because Set is a super interface of SortedSet. Besides it, the SortedSet interface also includes these methods:
  - comparator() - returns a comparator that can be used to order elements in the set
  - first() - returns the first element of the set
  - last() - returns the last element of the set
  - headSet(element) - returns all the elements of the set before the specified element
  - tailSet(element) - returns all the elements of the set after the specified element including the specified element
  - subSet(element1, element2) - returns all the elements between the element1 and element2 including element1





TreeSet Class

- Java TreeSet class implements the Set interface that uses a tree for storage. The objects of the TreeSet class are stored in ascending order.
- The important points about the Java TreeSet class are:
  - Java TreeSet class contains unique elements only like HashSet.
  - Java TreeSet class access and retrieval times are quite fast.
  - Java TreeSet class doesn't allow null element.
  - Java TreeSet class is non synchronized.
  - Java TreeSet class maintains ascending order.
  - The TreeSet can only allow those generic types that are comparable.
- Syntax : `TreeSet<Integer> abc=new TreeSet<>();`
- TreeSet is being implemented using a binary search tree

TreeSet Class

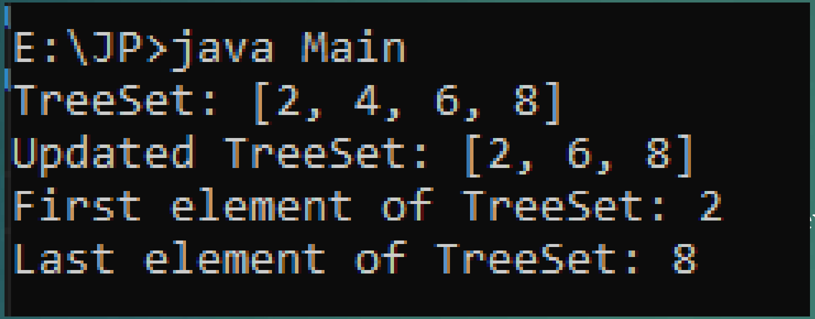
- **Methods of TreeSet :** Apart from the methods of SortedSet, Set and Collections interface, the TreeSet interface also includes these methods:
  - higher(element) - Returns the lowest element among those elements that are strictly greater than the specified element or else returns NULL
  - lower(element) - Returns the greatest element among those elements that are strictly less than the specified element or else returns NULL
  - ceiling(element) - Returns the lowest element among those elements that are greater than or equal to the specified element. If the element passed exists in a tree set, it returns the element passed as an argument or else returns NULL.
  - floor(element) - Returns the greatest element among those elements that are less than or equal to the specified element. If the element passed exists in a tree set, it returns the element passed as an argument or else returns NULL.
  - pollFirst() - returns and removes the first element from the set
  - pollLast() - returns and removes the last element from the set

TreeSet Class

- headSet(element, booleanValue) - The headSet() method returns all the elements of a tree set before the specified element (which is passed as an argument). The booleanValue parameter is optional. Its default value is false. If true is passed as a booleanValue, the method returns all the elements before the specified element **including the specified element**.
- tailSet(element, booleanValue) - The tailSet() method returns all the elements of a tree set after the specified element (which is passed as a parameter) including the specified element. The booleanValue parameter is optional. Its default value is true. If false is passed as a booleanValue, the method returns all the elements after the specified element **without including the specified element.**
- subSet(e1, bv1, e2, bv2) - The subSet() method returns all the elements between e1 and e2 including e1. The bv1 and bv2 are optional parameters. The default value of bv1 is true, and the default value of bv2 is false. If false is passed as bv1, the method returns all the elements between e1 and e2 without including e1. If true is passed as bv2, the method returns all the elements between e1 and e2, including e1.

TreeSet Class

```
import java.util.*;
class Main {
    public static void main(String[] args) {
        TreeSet<Integer> evenNumbers = new TreeSet<>();
        // Using the add() method
        evenNumbers.add(4);
        evenNumbers.add(2);
        evenNumbers.add(6);
        evenNumbers.add(6);
        evenNumbers.add(8);
        System.out.println("TreeSet: " + evenNumbers);
        evenNumbers.remove(4);
        System.out.println("Updated TreeSet: " + evenNumbers);
        System.out.println("First element of TreeSet: " +
evenNumbers.first());
        System.out.println("Last element of TreeSet: " +
evenNumbers.last());
        // Using higher()
        System.out.println("Using higher: " + evenNumbers.higher(4));
      // Using lower()
        System.out.println("Using lower: " + evenNumbers.lower(4));
```

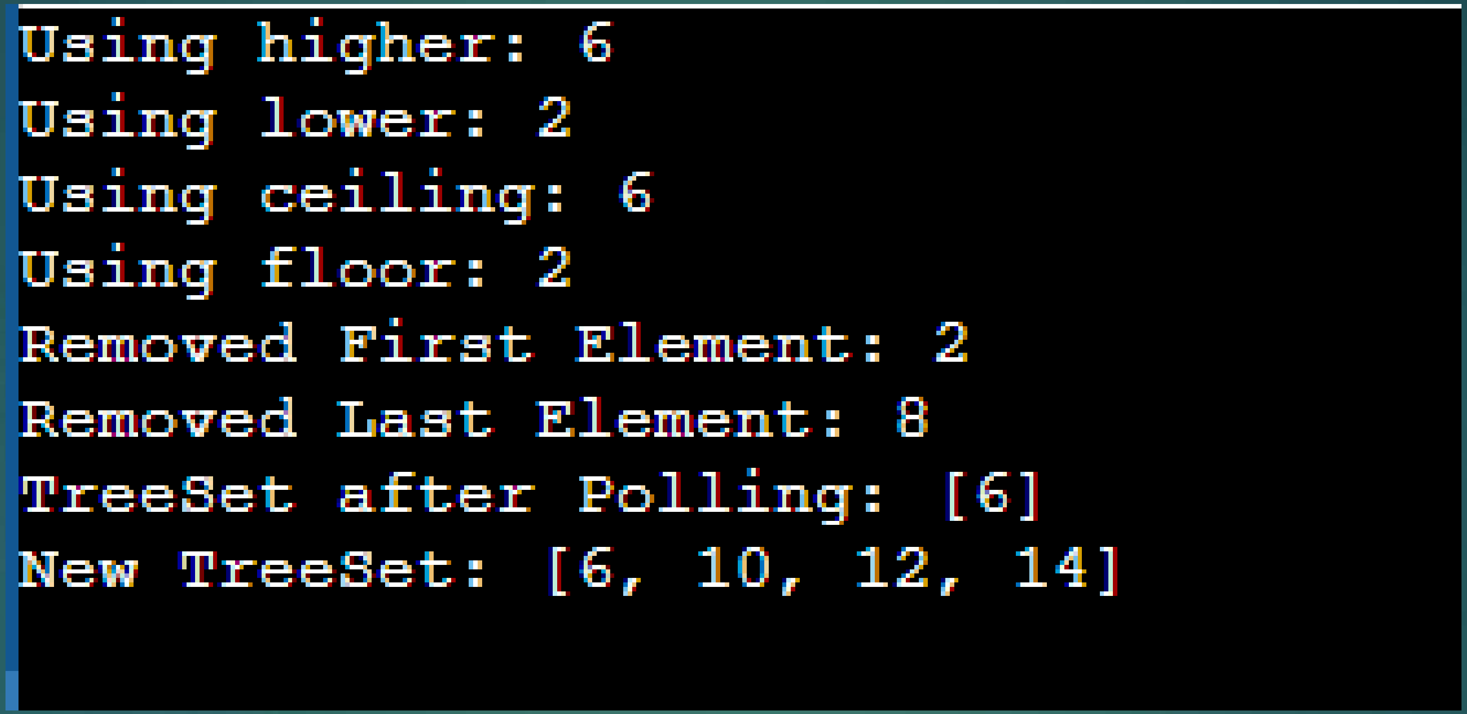


```
E:\JP>java Main
TreeSet: [2, 4, 6, 8]
Updated TreeSet: [2, 6, 8]
First element of TreeSet: 2
Last element of TreeSet: 8
```

```
// Using ceiling()
        System.out.println("Using ceiling: " + evenNumbers.ceiling(4));

        // Using floor()
        System.out.println("Using floor: " + evenNumbers.floor(3));

        // Using pollFirst()
        System.out.println("Removed First Element: " +
venNumbers.pollFirst());

        // Using pollLast()
        System.out.println("Removed Last Element: " +
evenNumbers.pollLast());

        System.out.println("TreeSet after Polling: " + evenNumbers);
        evenNumbers.add(10);
        evenNumbers.add(12);
        evenNumbers.add(14);
        System.out.println("New TreeSet: " + evenNumbers);
```

(code further continued on slide 58)

TreeSet Class

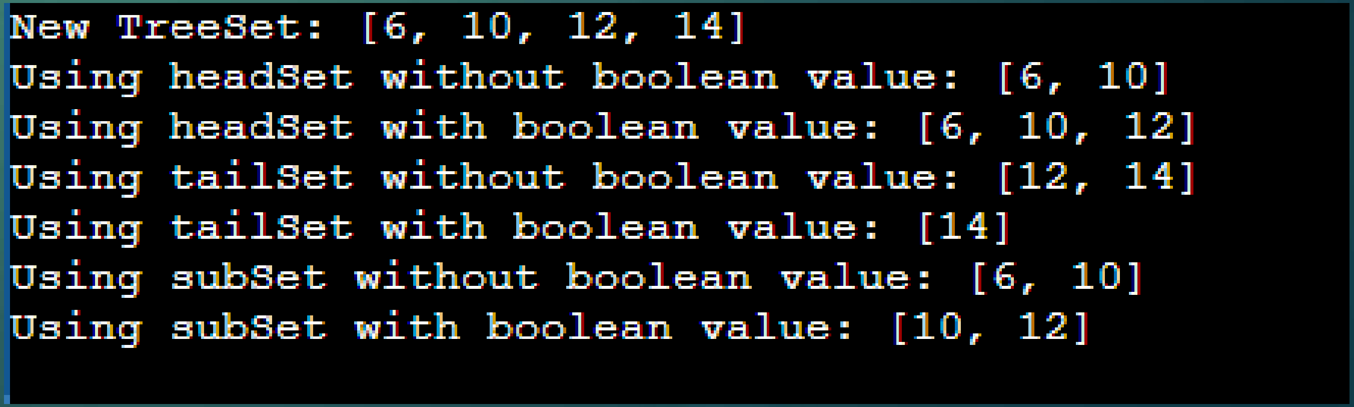


```
Using higher: 6
Using lower: 2
Using ceiling: 6
Using floor: 2
Removed First Element: 2
Removed Last Element: 8
TreeSet after Polling: [6]
New TreeSet: [6, 10, 12, 14]
```

TreeSet Class

```
 System.out.println("New TreeSet: " + evenNumbers);

        // using headSet()

        // Using headSet() with default boolean value
        System.out.println("Using headSet without boolean value: " +
evenNumbers.headSet(12));

        // Using headSet() with specified boolean value
        System.out.println("Using headSet with boolean value: " +
evenNumbers.headSet(12, true));

        // using tailSet()

        // Using tailSet() with default boolean value
        System.out.println("Using tailSet without boolean value: " +
evenNumbers.tailSet(12));

        // Using tailSet() with specified boolean value
        System.out.println("Using tailSet with boolean value: " +
evenNumbers.tailSet(12, false));

//using subSet()

        // Using subSet() with default boolean value
        System.out.println("Using subSet without boolean value: " +
evenNumbers.subSet(6, 12));

        // Using subSet() with specified boolean value
        System.out.println("Using subSet with boolean value: " +
evenNumbers.subSet(6, false, 12, true));
    }
}
```

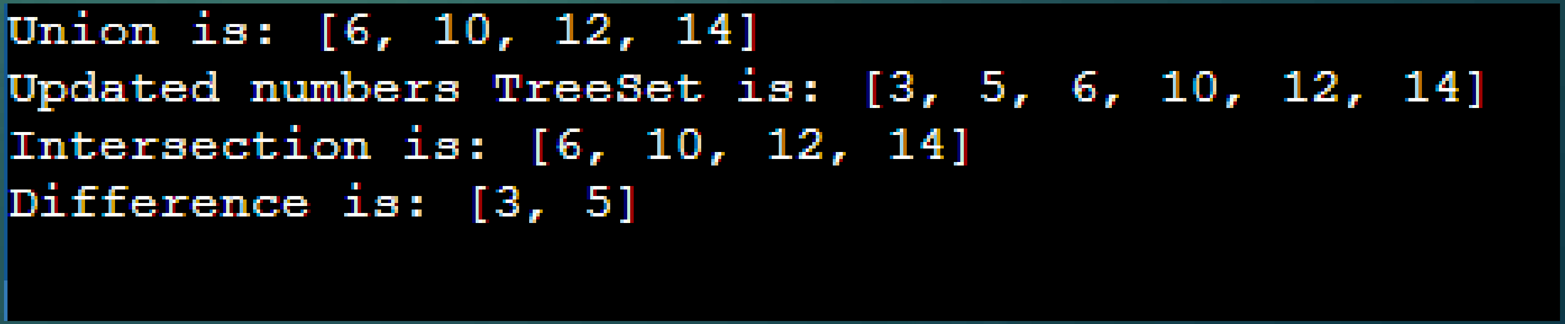
(code further continued on slide 59)



```
New TreeSet: [6, 10, 12, 14]
Using headSet without boolean value: [6, 10]
Using headSet with boolean value: [6, 10, 12]
Using tailSet without boolean value: [12, 14]
Using tailSet with boolean value: [14]
Using subSet without boolean value: [6, 10]
Using subSet with boolean value: [10, 12]
```

TreeSet Class

```
// Union of two sets
       TreeSet<Integer> numbers = new TreeSet<>();
       numbers.addAll(evenNumbers);
       System.out.println("Union is: " + numbers);
       numbers.add(3);
       numbers.add(5);
        System.out.println("Updated numbers TreeSet is: " + numbers);

       // Intersection of two sets
       numbers.retainAll(evenNumbers);
       System.out.println("Intersection is: " + numbers);
       numbers.add(3);
       numbers.add(5);

       // Difference between two sets
       numbers.removeAll(evenNumbers);
       System.out.println("Difference is: " + numbers);

   }
}
```

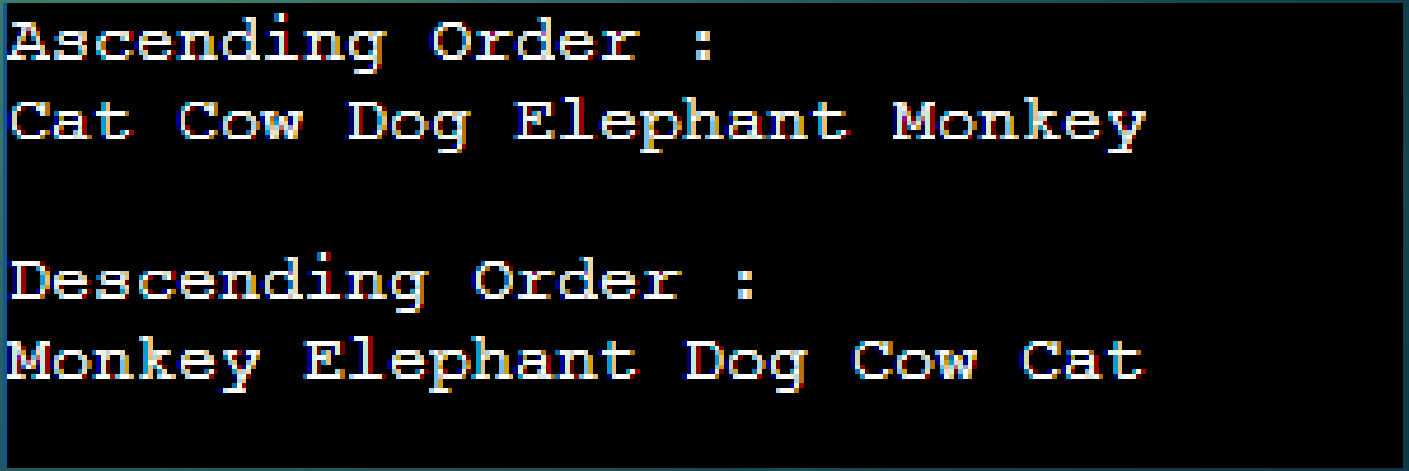


```
Union is: [6, 10, 12, 14]
Updated numbers TreeSet is: [3, 5, 6, 10, 12, 14]
Intersection is: [6, 10, 12, 14]
Difference is: [3, 5]
```

TreeSet Class

- By default, tree set elements are sorted naturally i.e in ascending order. However, we can also traverse the set in descending order

```
import java.util.*;
class Main{
 public static void main(String args[]){
  //Creating and adding elements
  TreeSet<String> animals=new TreeSet<String>();
  animals.add("Dog");
  animals.add("Cat");
  animals.add("Monkey");
  animals.add("Elephant");
  animals.add("Cow");

  //Traversing elements in ascending order
  Iterator<String> itr=animals.iterator();
  System.out.println("Ascending Order : ");
  while(itr.hasNext()){
   System.out.print(itr.next() + " ");
  }
  System.out.println("\n");
  //Traversing elements in descending order
  Iterator<String> des=animals.descendingIterator();
  System.out.println("Descending Order : ");
  while(des.hasNext()){
   System.out.print(des.next() + " ");
  }
 }
}
```



```
Ascending Order :
Cat Cow Dog Elephant Monkey

Descending Order :
Monkey Elephant Dog Cow Cat
```

LinkedHashSet vs HashSet

- Both LinkedHashSet and HashSet implements the Set interface. However, there exist some differences between them.
  - LinkedHashSet maintains a linked list internally. Due to this, it maintains the insertion order of its elements.
  - The LinkedHashSet class requires more storage than HashSet. This is because LinkedHashSet maintains linked lists internally.
  - The performance of LinkedHashSet is slower than HashSet. It is because of linked lists present in LinkedHashSet.

LinkedHashSet Vs. TreeSet

- Here are the major differences between LinkedHashSet and TreeSet:
  - The TreeSet class implements the SortedSet interface. That's why elements in a tree set are sorted. However, the LinkedHashSet class only maintains the insertion order of its elements.
  - A TreeSet is usually slower than a LinkedHashSet. It is because whenever an element is added to a TreeSet, it has to perform the sorting operation.
  - LinkedHashSet allows the insertion of null values. However, we cannot insert a null value to TreeSet.

TreeSet vs HashSet

- Both the TreeSet as well as the HashSet implements the Set interface. However, there exist some differences between them.

  - Unlike HashSet, elements in TreeSet are stored in some order. It is because TreeSet implements the SortedSet interface as well.
  - TreeSet provides some methods for easy navigation. For example, first(), last(), headSet(), tailSet(), etc. It is because TreeSet also implements the NavigableSet interface.
  - HashSet is faster than the TreeSet for basic operations like add, remove, contains and size.

Programming Practice Questions

- Implement a generic stack class GenericStack<T> that supports the following operations:
  - push(T item): Add an item to the stack.
  - pop(): Remove and return the top item from the stack.
  - peek(): Return the top item without removing it.
  - isEmpty(): Check if the stack is empty.
- Extend the above program to implement a generic stack class BoundedStack that can only hold numbers.
- Write a program that creates a List of integers. Populate the list with the first 10 positive integers, then:
  - Print the list.
  - Remove the third element from the list.
  - Add the number 100 at the end of the list.
  - Print the updated list.
- Write a program that accepts a List<Double> of floating-point numbers and sorts the list in descending order. Print the sorted list.
- Write a method that merges two List<Integer> objects into a single list. The resulting list should contain all elements from both lists, but without any duplicates.
- Write a program that creates a Set<Integer> and adds the first 10 positive integers. Then, attempt to add a duplicate integer. Print the set to show that duplicates are not allowed.
- WAP that creates two Set<Integer> objects and
  - returns a new set that contains the union of both sets.
  - returns a new set containing only the elements that are present in both sets.
  - returns a new set containing the difference of both sets.
  - check if one Set is a subset of another Set. Return true if it is, and false otherwise.