CPE207 Object Oriented Programming

Week 12

Java Collections

Dr. Nehad Ramaha,
Computer Engineering Department
Karabük Universities

These Slides mainly adopted from Assist. Prof. Dr. Ozacar Kasim lecture notes

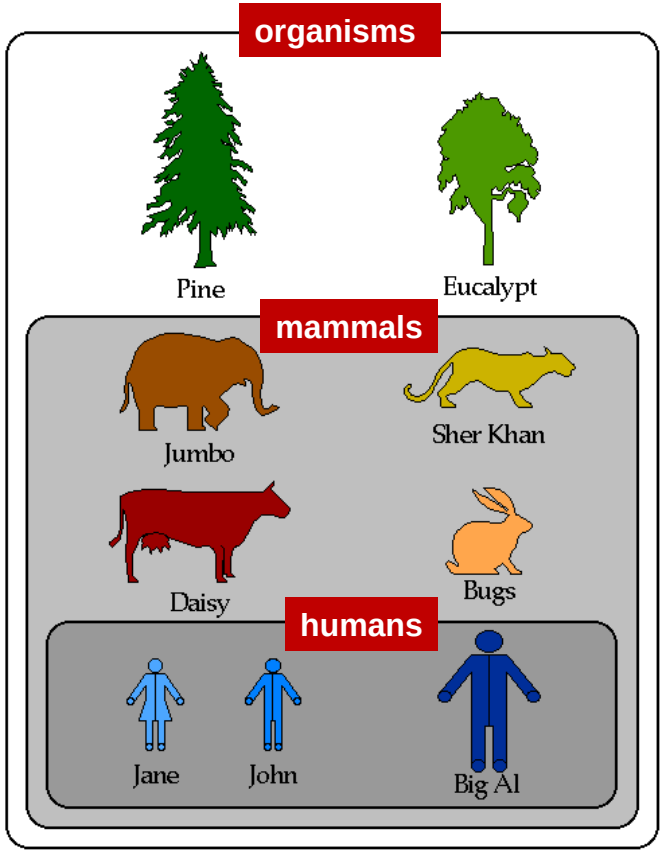
Interfaces: Definition

Week 12

- An interface is a **contract** an implementing class needs to respect
  - interface contains behaviors that a class implements. (Contract)
  - Methods in an interface are by-default public. (Contract)

Abstraction

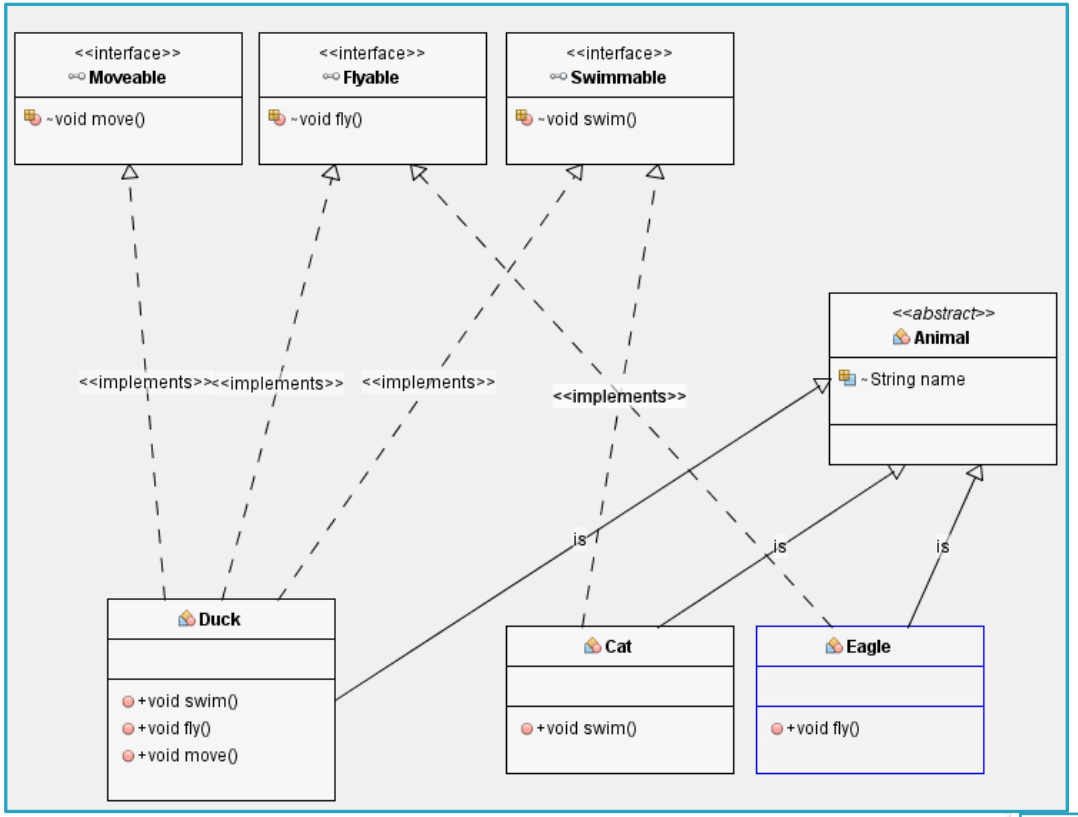
- The selection of significant attributes is *abstraction*.
- We can abstract by appearance, structure, purpose, functionality, privilege etc.
- Abstraction allows us to classify the items such as: organisms, mammals, humans



Multiple inheritance

- Multiple inheritance *is* not allowed in Java. But classes can inherit multiple **interfaces**.

- Multiple inheritance allows a class **to inherit the functionality of more than one interface**
  - Thus allowing for modeling of complex relationships.



Java Collections

Java 2 Collections

- The **Collection** in Java is a framework that provides an architecture to store and manipulate the group of objects.
  - data structures and methods written by pioneers in the field
    - **Joshua Bloch,** who wrote the Java Collections Framework, the java.math package and many more...

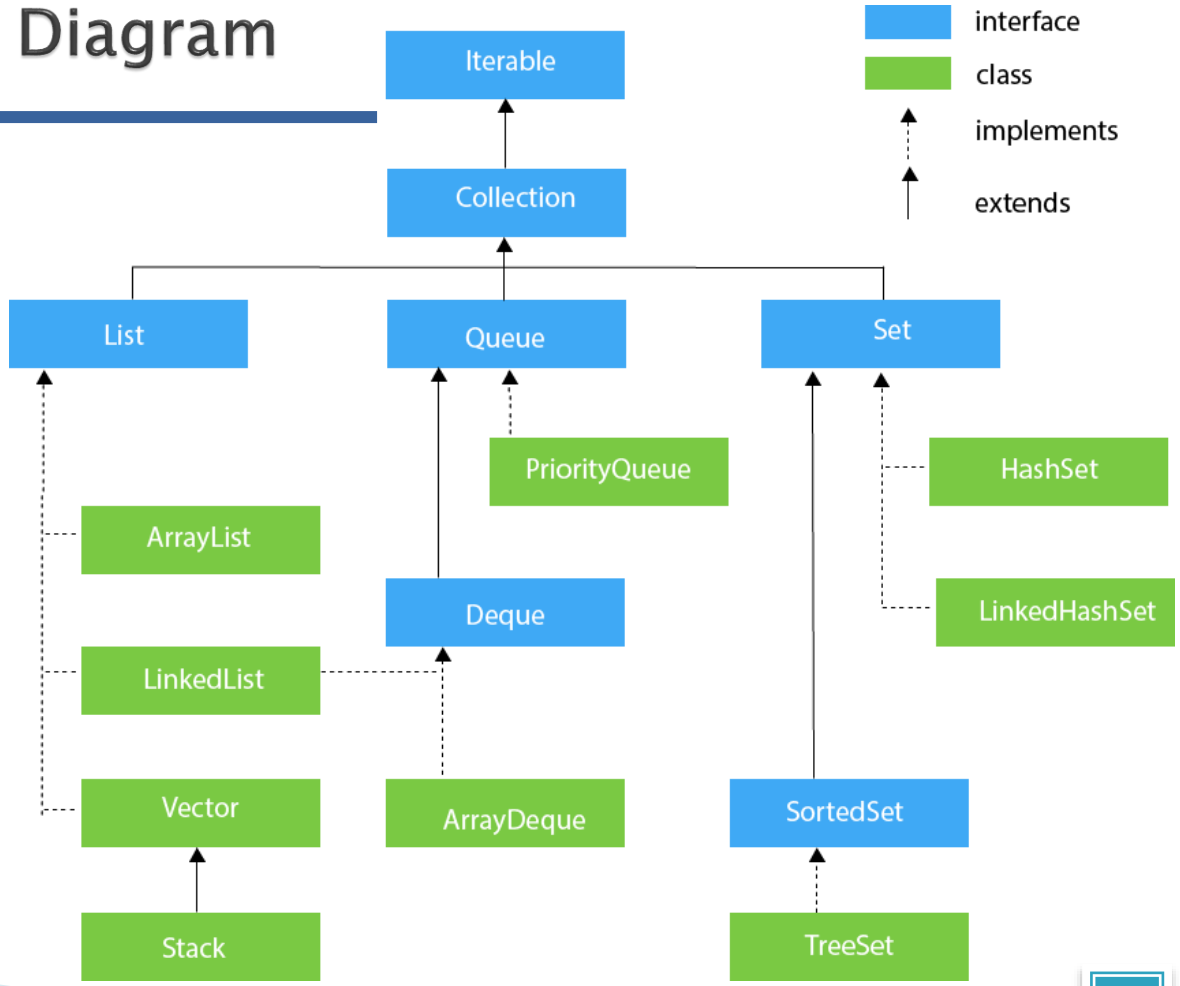
Joshua Bloch

Java Collections Framework

- Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.
- A collections framework contains three things:
  - Interfaces.
    - (Set, List, Queue, Deque)
  - Implementations (classes),
    - (ArrayList, Vector, LinkedList, PriorityQueue, HashSet,LinkedHashSet, TreeSet)
  - Algorithms

Collections Framework Diagram

The **java.util** package contains all the classes and interfaces for the Collection framework.



Source: https://www.javatpoint.com/collections-in-java

Methods of Collection interface

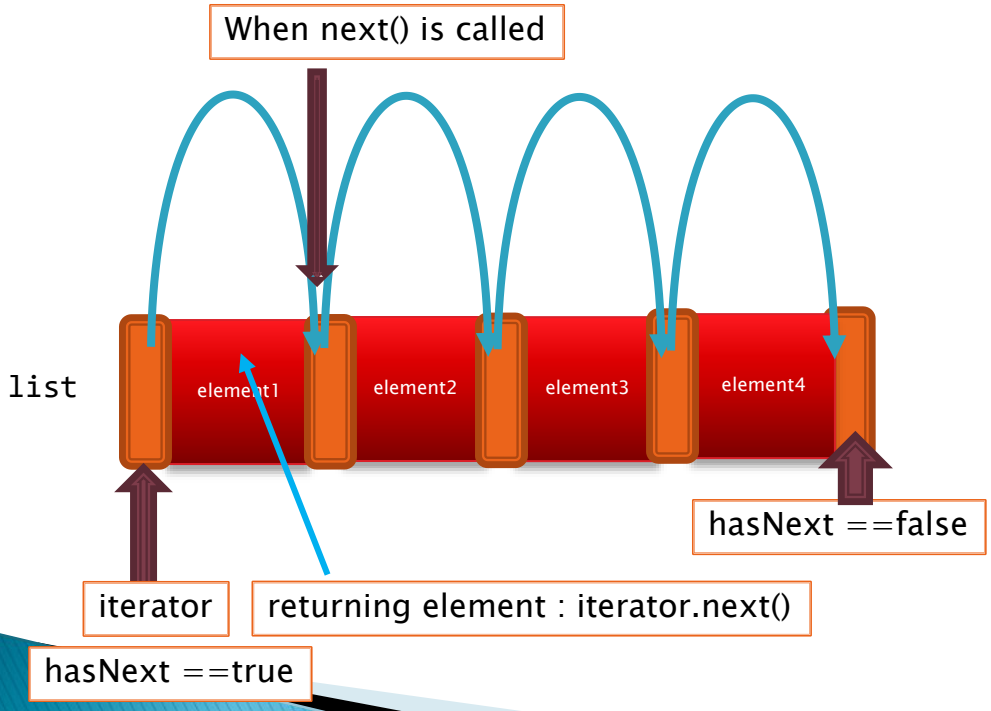
- There are many methods declared in the Collection interface such as:
  - `int size(); //It returns the total number of elements in the collection.`
  - `boolean isEmpty();`
  - `boolean contains(Object element); //It is used to search an element.`
  - `boolean add(Object element);  //to insert an element in this collection.`
  - `boolean remove(Object element); // to delete an element from the collection.`
  - `Iterator iterator();  // what is this?`
- These methods are enough to define the basic behavior of a collection
- This Interface provides an Iterator to step through the elements in the Collection

Iterator Interface

- Defines three fundamental methods
  - **`Object next()`**
    - `It returns the element and moves the cursor pointer to the next element.`
  - **`boolean hasNext()`**
  - **`void remove()`**

- Iterator enables you to cycle through a collection, obtaining or removing elements.
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to **next()** "reads" an element from the collection. Then you can use it or remove it

Iterator Position




```
Iterator iterator = list.iterator();

while(iterator.hasNext()){
 System.out.println(iterator.next());
 iterator.remove();
}
```

When to use Iterators?

- If you only want to scan through the list, then this is enough:

```
for(Object obj : objList) {
    System.out.println(obj.name);
}
```

- If you want to <u>modify</u> the list, then use the Iterator

```
while (iter.hasNext()) {                    //check if there is a next element
      iter.next();                          // go to next element
        iter.remove();              // remove the returning element
       }
```

Example –A Simple Collection

```
public class SimpleCollection  {
    public static void main(String[] args) {
        Collection c= new ArrayList(); // creating a new array list

        c.add(5); //add elements to list
        c.add("hello world"); //more
        c.add(new FullTimeEmployee("jack",10)); //and more

        Iterator iter = c.iterator(); // get the iterator from the collection
        while (iter.hasNext())  //check if there is a next element
            System.out.println(iter.next()); //write the next element.
    }
}
```

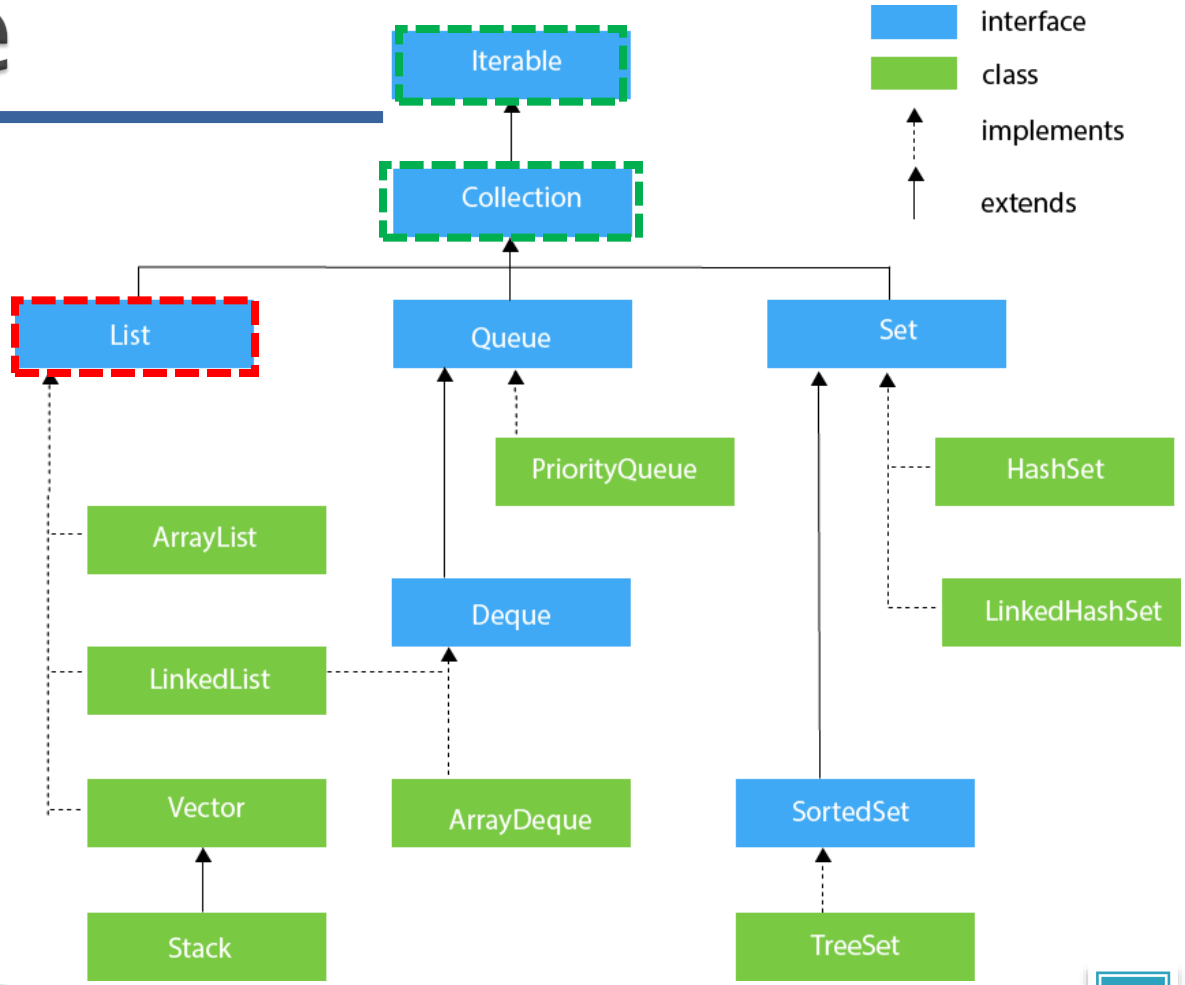
You can put any types of data. But! Don't do this.

Warning:

- **“Don’t use raw types”** Joshua Bloch- Effective Java Third Edition. Ch. 5 Page 117

- Let's say you want to put a single data type and If someone accidentally inserted an object of the wrong type, casts could fail at runtime.
- With generics, you tell the compiler what types of objects are permitted in each collection (**type-safety)**
- **We are not going to use : List elements = new …**
- **Rather we use, List<YourObjectType> elements = new …**
- This will be called as generics later.

List Interface



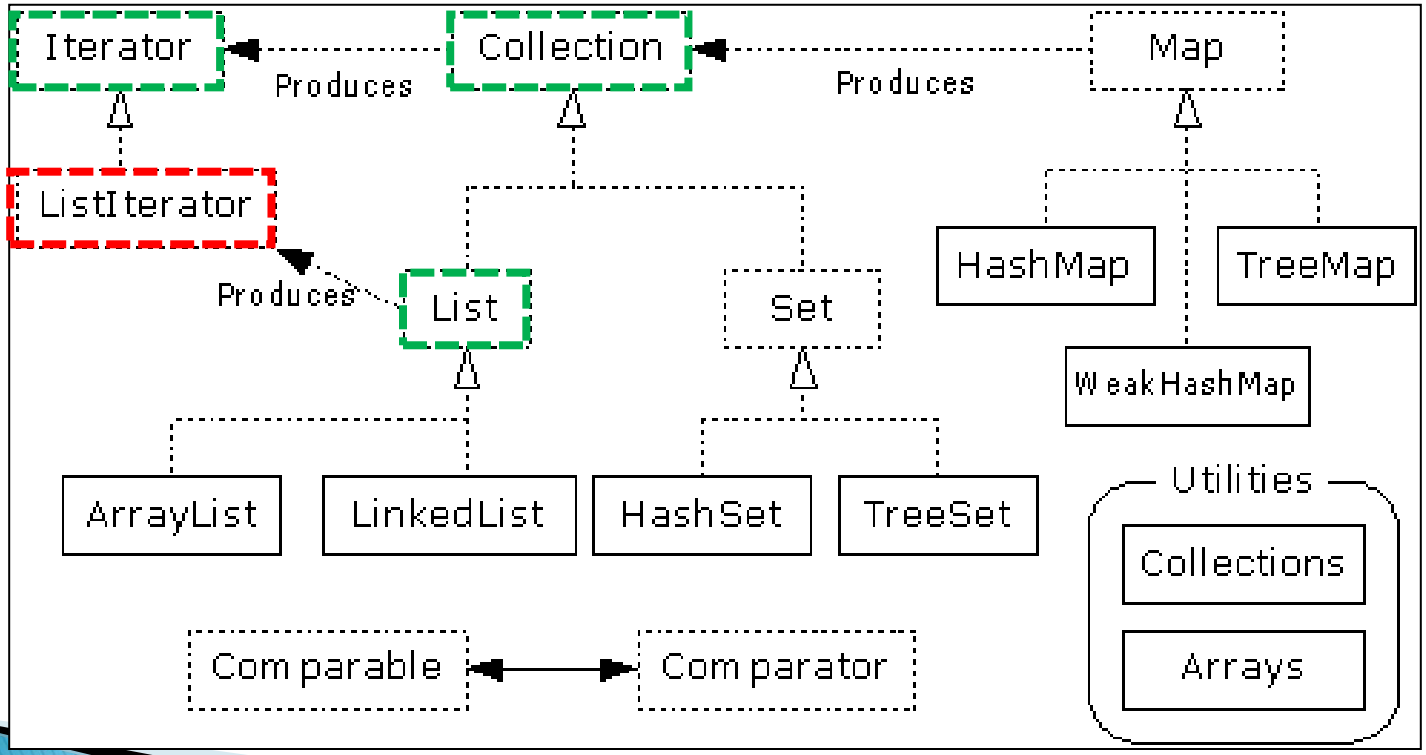
List Interface

- List interface is the child interface of Collection interface.
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
- The List interface adds the concept of *order* to a collection
- The user of a list has control over where an element is added in the collection
- Lists typically allow *duplicate* elements
- Provides a ListIterator to step through the elements in the list.

List Interface

- 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();
- There are various methods in List interface that can be used to insert, delete, and access the elements from the list.

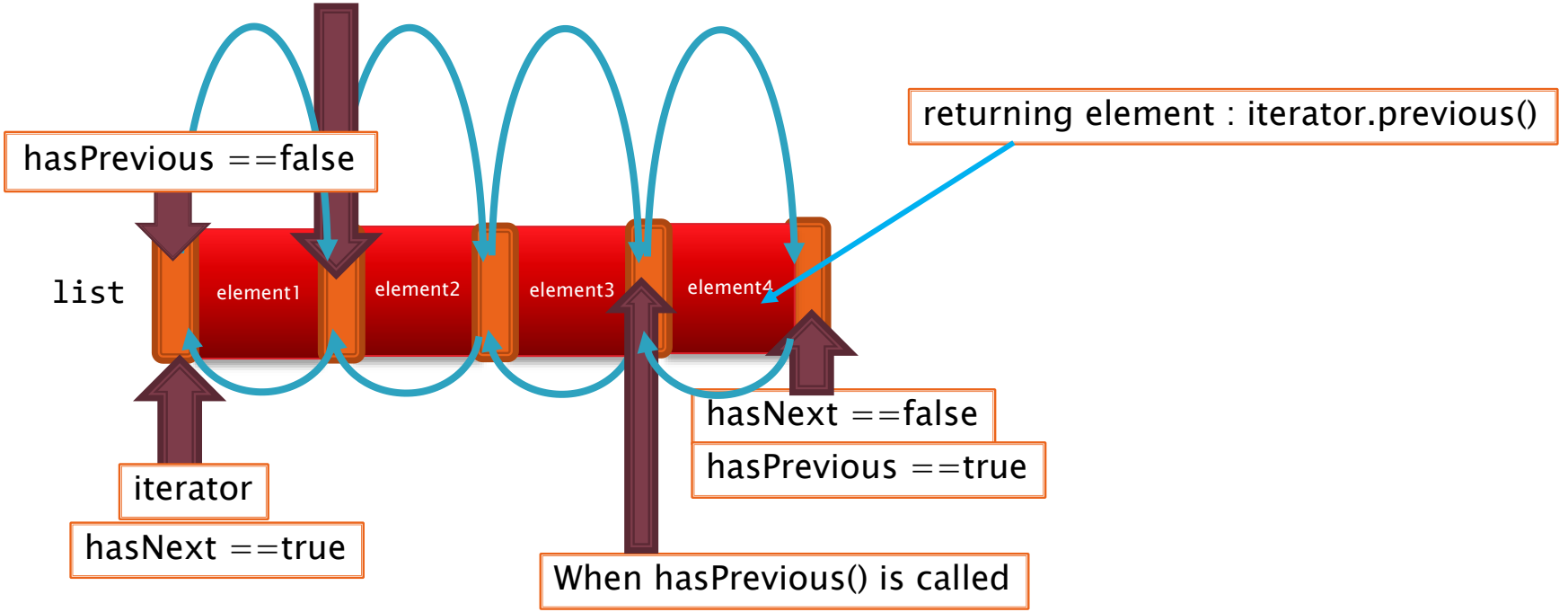
ListIterator Interface



ListIterator Interface

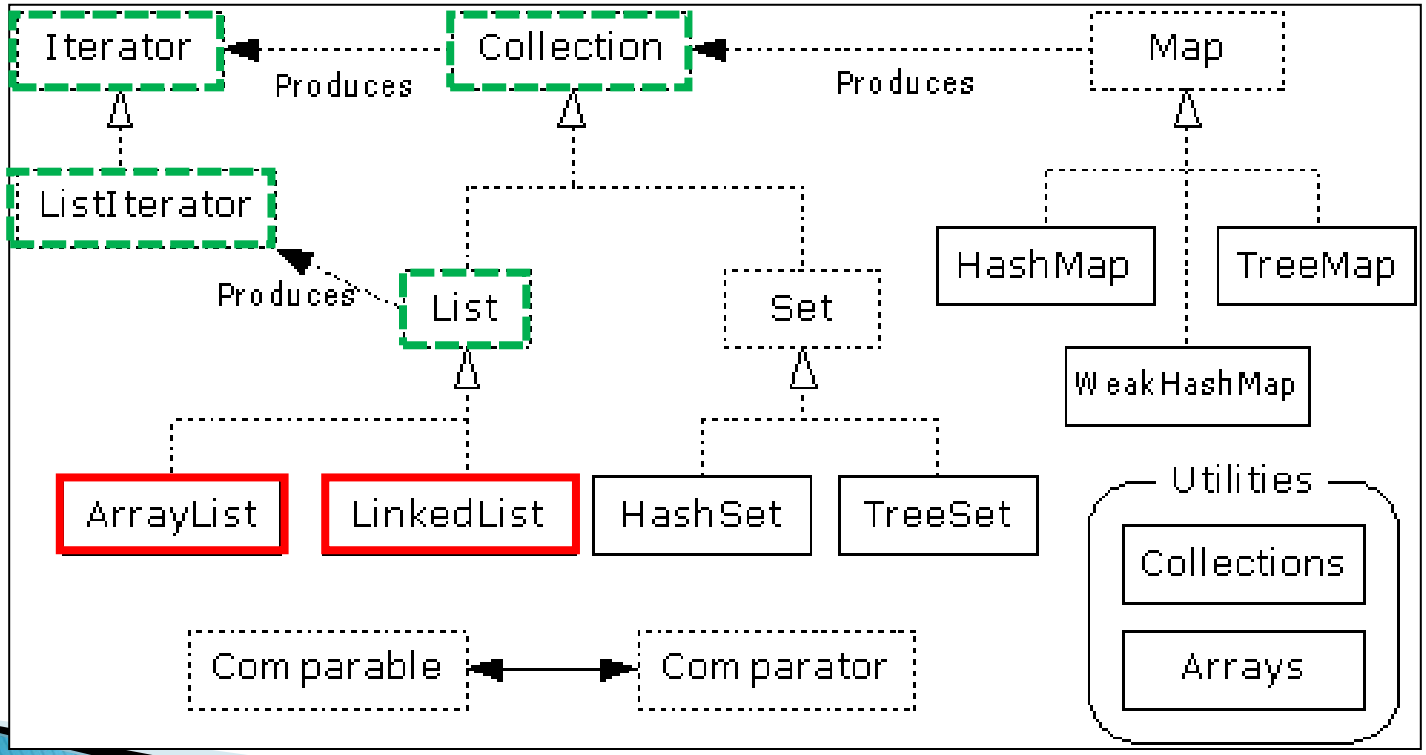
- Extends the Iterator interface
- Defines three fundamental methods
  - `void add(Object o)`
  - `boolean hasPrevious()`
  - `Object previous()`
- The addition of these three methods defines the basic behavior of an ordered list
- A ListIterator knows position within list

ListIterator Position – next(), previous()



```
public class ListIteratorExample {
 public static void main(String[] args) {
    // Create a LinkedList
    LinkedList<String> linkedlist = new LinkedList<String>();

          linkedlist.add(“Abant"); // Add elements to LinkedList
          linkedlist.add(“Mengen");
          linkedlist.add(“Gerede");

          ListIterator listIt = linkedlist.listIterator(); // Obtaining ListIterator
          System.out.println("Forward iteration:");

// Iterating the list in forward direction
    while(listIt.hasNext())
       System.out.println(listIt.next());
// Iterating the list in backward direction

    System.out.println("\nBackward iteration:");
    while(listIt.hasPrevious())
       System.out.println(listIt.previous());
 }
}
```

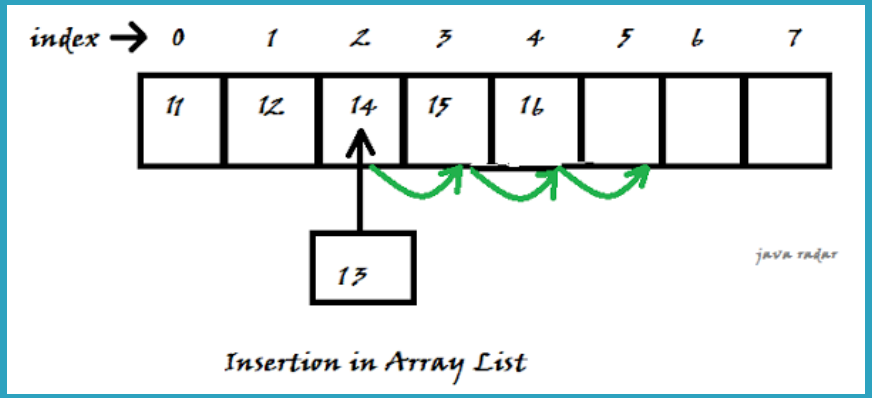
ArrayList and LinkedList



List Implementations

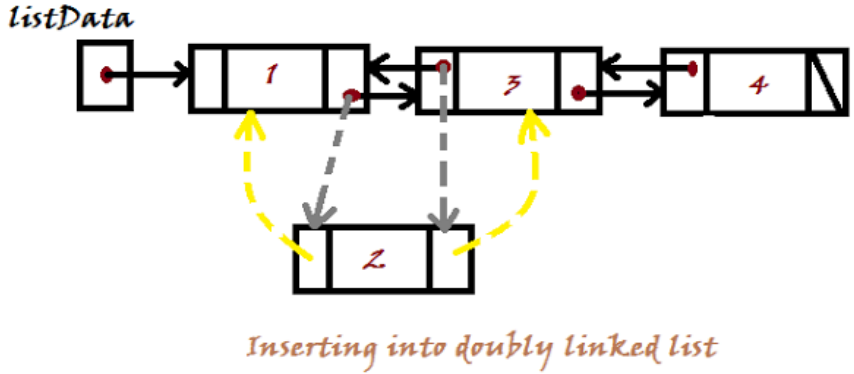
ArrayList

- low cost random access
- high cost insert and delete



LinkedList

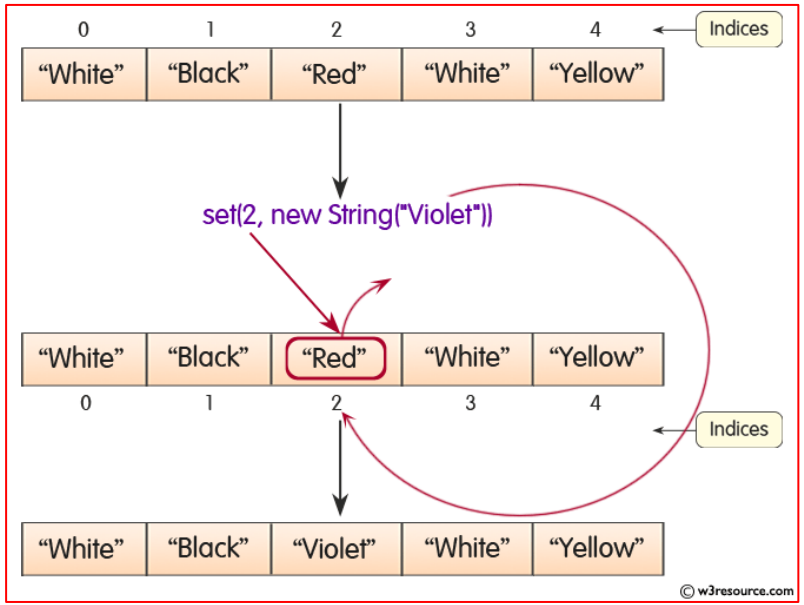
- sequential access
- low cost insert and delete
- high cost random access



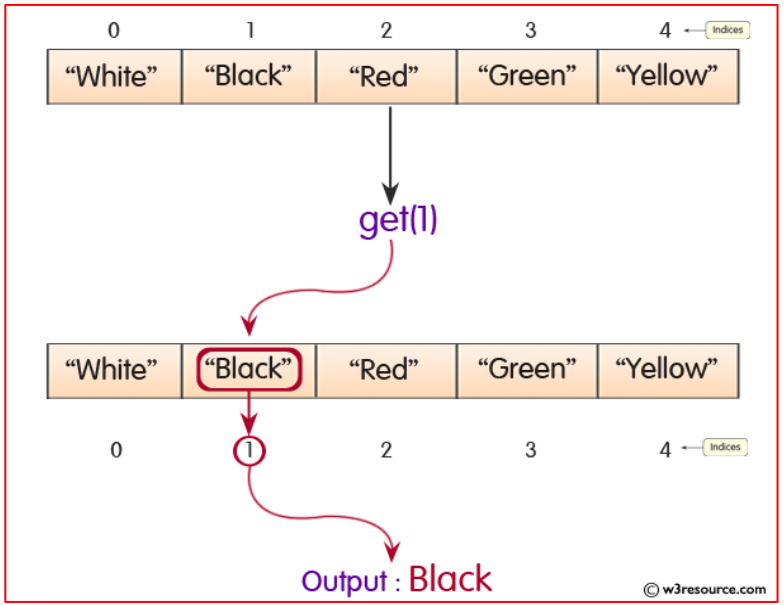
ArrayList overview and methods

- Constant time (**O(1)**) positional access (it's an array)
- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - `Object get(int index)`
  - `Object set(int index, Object element)`
- Indexed add and remove are provided, but can be **costly** if used frequently
  - `void add(int index, Object element)`
  - `Object remove(int index)`
- May want to resize in one shot if adding many elements
  - `void ensureCapacity(int minCapacity)`
- As elements are added to an **ArrayList**, its capacity grows automatically.

Set() and get() methods in ArrayList

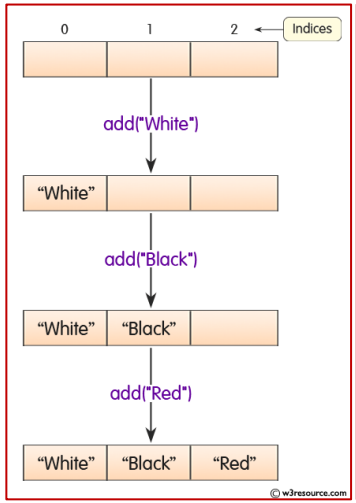


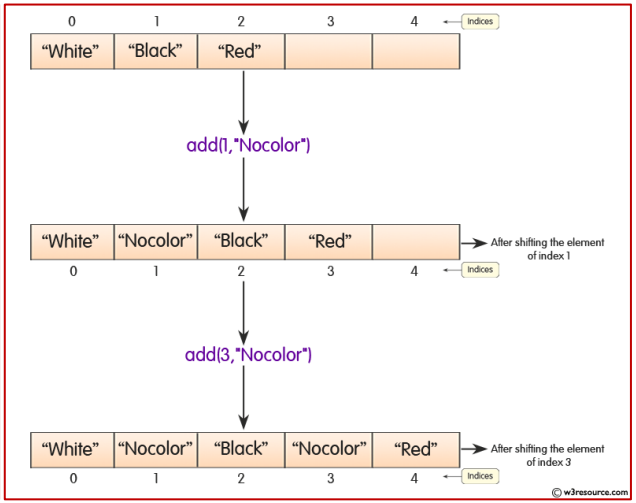
set



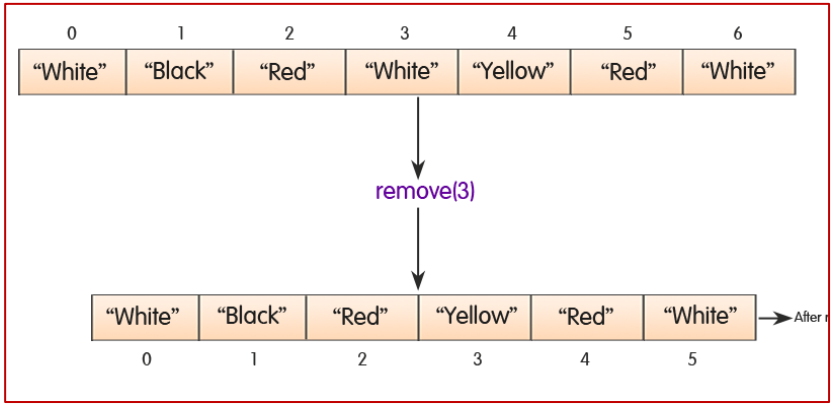
get

add() and remove() methods





add



remove

LinkedList overview

- Stores each element in a node
- Each node stores a link to the next and previous nodes
- Insertion and removal are low-cost
  - just update the links in the surrounding nodes
- Random access is expensive
  - Start from beginning or end and traverse each node while counting

LinkedList methods

- The list is sequential, so access it that way
  - `ListIterator listIterator()`
- ListIterator knows about position
  - use `add()` from ListIterator to add at a position
  - use `remove()` from ListIterator to remove at a position
- LinkedList knows a few things too
  - `void addFirst(Object o), void addLast(Object o)`
  - `Object getFirst(), Object getLast()`
  - `Object removeFirst(), Object removeLast()`

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ListExample {
      public static void main(String[] args) {
            //create ArrayList
            Collection <String> list = new ArrayList<>();
            //add element to ArrayList
             list.add("jack");
             list.add("mike");
             list.add("hulk");

            //create LinkedList
            Collection <String> list2 = new LinkedList<>();
            // add elements to LinkedList
             list2.add("jade");
             list2.add("June");
             list2.add("April");
      }
}
```

Lab exercise

Write a class called BankAccount. The class must have 3 attributes: accountNo(**int**), holderName(**String**) and balance(**double**).

a) Class constructor will have to set these 3 attributes.

b) Create a balanceChange(double amount) method to reduce balance value by given amount.

c) In the Main method:

1. Create 4 objects (a1, a2, a3, a4) from this class and add them all to an ArrayList called myAccounts. List will only accept BankAccount type.
2. Sort objects by holderName in the list. **(Hint: implement Comparable interface)**
3. Reduce the balances by 500 for all the accounts and print sorted objects in the list.

Thanks ☺