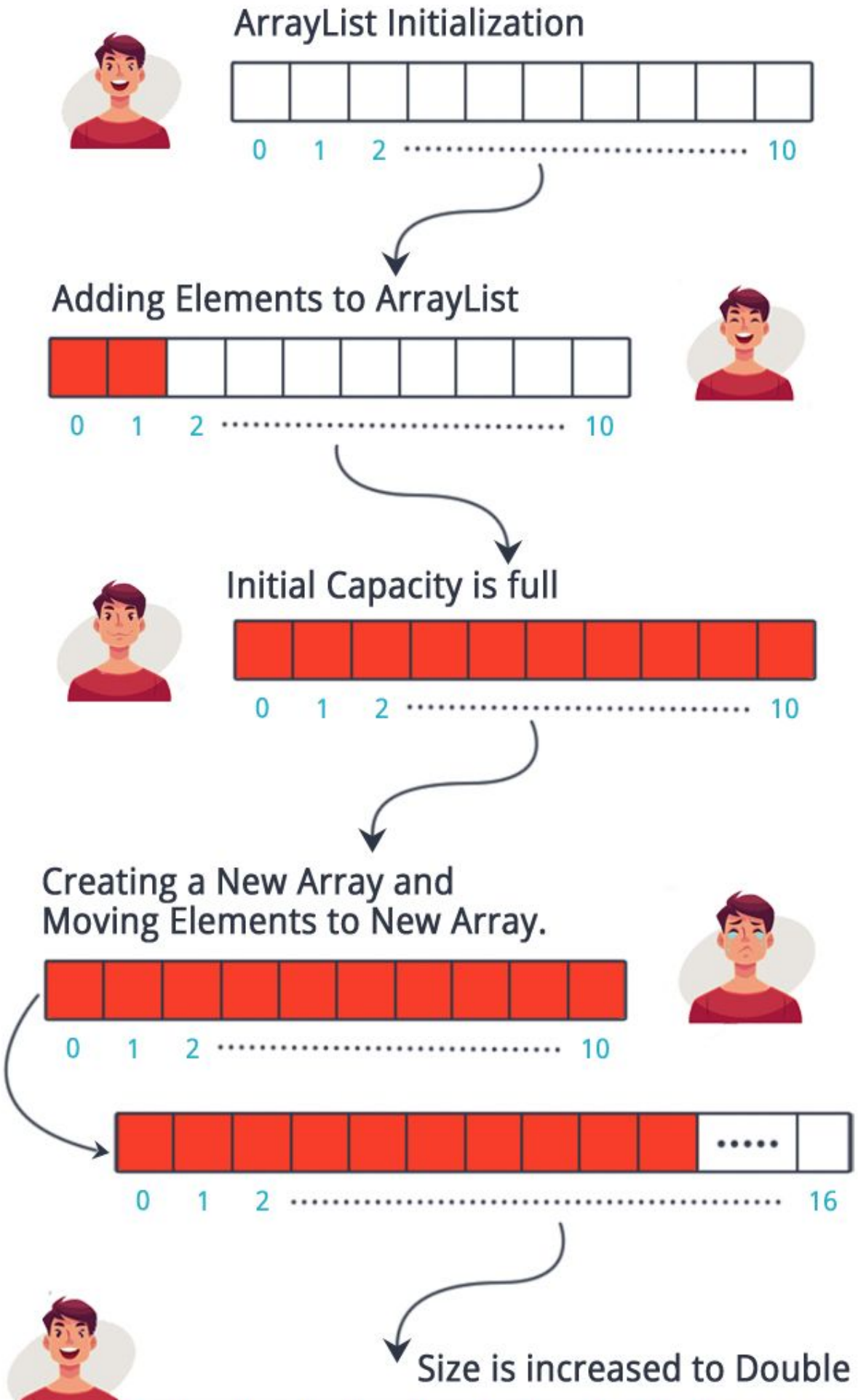
ArrayList implementation :

Internally an ArrayList uses an `Object[]` Array. All the addition, removal and traversal happens on this array.


ArrayList Initialization
0 1 2 10
Adding Elements to ArrayList
0 1 2 10
Initial Capacity is full
0 1 2 10
Creating a New Array and
Moving Elements to New Array.
0 1 2 10
0 1 2 16
Size is increased to Double

Empty List initialization with default capacity

When an object of `ArrayList` is created without initial capacity, the default constructor of the `ArrayList` class is invoked. It uses empty array instance to create the new object.

Here, default capacity of 10 is assigned at a time of empty initialization of ArrayList

In Java 8 or later

```
 /**
      * Shared empty array instance used for empty instances.
      */
    private static final Object[] EMPTY_ELEMENTDATA = {};

    /**
      * Shared empty array instance used for default sized empty instances. We
      * distinguish this from EMPTY_ELEMENTDATA to know how much to inflate when
      * first element is added.
      */
    private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

    /**
      * Constructs an empty list with an initial capacity of ten.
      */
    public ArrayList() {
        this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
    }
```

Here, empty list is inialized with default capacity of 10; ArrayList with `elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA` will be expanded to `DEFAULT_CAPACITY` when the first element is added.

Empty List initialization with initial capacity

When an object of `ArrayList` is created with an initial capacity, the `ArrayList` constructor is invoked to create the array internally.

```
      List<String> arrayList = new ArrayList<String>(30);
```

```
/**
      * Constructs an empty list with the specified initial capacity.
      *
      * @param   initialCapacity   the initial capacity of the list
      * @exception IllegalArgumentException if the specified initial capacity
      *            is negative
      */
    public ArrayList(int initialCapacity) {
            super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
            this.elementData = new Object[initialCapacity];
    }
```

In Java 8 or later

```
    /**
      * Constructs an empty list with the specified initial capacity.
      *
      * @param  initialCapacity  the initial capacity of the list
      * @throws IllegalArgumentException if the specified initial capacity
      *     is negative
      */
    public ArrayList(int initialCapacity) {
        if (initialCapacity > 0) {
            this.elementData = new Object[initialCapacity];
        } else if (initialCapacity == 0) {
            this.elementData = EMPTY_ELEMENTDATA;
        } else {
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        }
    }
```

Here, the size of the array will be equal to the argument passed in the constructor. Then, size of the array will be 30 in above example.

How the size of ArrayList grows dynamically?

In the `add(Object)`, the capacity of the ArrayList will be checked before adding a new element. Here is the implementation of the `add()` method.

As elements are added to an ArrayList, its capacity grows automatically.

In Java 6 or previous

```
/**
    * Appends the specified element to the end of this list.
    *
    * @param e element to be appended to this list
    * @return <tt>true</tt> (as specified by {@link Collection#add})
    */
   public boolean add(E e) {
      ensureCapacity(size + 1);  // Increments modCount!!
      elementData[size++] = e;
      return true;
   }

   /**
    * Increases the capacity of this <tt>ArrayList</tt> instance, if
    * necessary, to ensure that it can hold at least the number of elements
    * specified by the minimum capacity argument.
    *
    * @param   minCapacity   the desired minimum capacity
    */
   public void ensureCapacity(int minCapacity) {
                  modCount++;
                  int oldCapacity = elementData.length;
                  if (minCapacity > oldCapacity) {
                        Object oldData[] = elementData;
                        int newCapacity = (oldCapacity * 3)/2 + 1;
                              if (newCapacity < minCapacity)
                        newCapacity = minCapacity;
                              // minCapacity is usually close to size, so this is a win:
                              elementData = Arrays.copyOf(elementData, newCapacity);
                  }
   }
```

Consider an ArrayList, , with capacity . Once ‘s underlying array is filled to capacity with values, a new underlying array is created that has increased capacity. The elements currently stored in are then copied over to the new, larger capacity array, and the new array replaces ‘s original underlying array. Typically, the increased

capacity is some manner of rough doubling though the actual amount that the capacity increases depends on the implementation of ArrayList you're using. In some Java implementations, the `ensureCapacity` method in `ArrayList` class ensure the new size of the underlying array:

> *int newCapacity = (oldCapacity * 3)/2 + 1;*

Increasing the capacity by what amounts to `1.5X` is still enough to give some guarantees:`O(1)` access and `O(1)` insertion (on average). This has the advantage of wasting slightly less space when the ArrayList is not very full.

In Java 7 or later

```
/**
      * Appends the specified element to the end of this list.
      *
      * @param e element to be appended to this list
      * @return <tt>true</tt> (as specified by {@link Collection#add})
      */
    public boolean add(E e) {
        ensureCapacityInternal(size + 1);  // Increments modCount!!
        elementData[size++] = e;
        return true;
    }

      private void ensureCapacityInternal(int minCapacity) {
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
        }

        ensureExplicitCapacity(minCapacity);
    }

      private void ensureExplicitCapacity(int minCapacity) {
        modCount++;

        // overflow-conscious code
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }

    /**
      * Increases the capacity to ensure that it can hold at least the
      * number of elements specified by the minimum capacity argument.
```

```
     *
     * @param minCapacity the desired minimum capacity
     */
    private void grow(int minCapacity) {
        // overflow-conscious code
        int oldCapacity = elementData.length;
        int newCapacity = oldCapacity + (oldCapacity >> 1);
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;
        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);
        // minCapacity is usually close to size, so this is a win:
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
```

`ensureCapacityInternal()` determines what is the current size of occupied elements and what is the maximum size of the array.

The `grow` method in ArrayList class ensure the new size of the underlying array:

> *int newCapacity = oldCapacity + (oldCapacity >> 1);*

In the above code, `minCapacity` is the size of the current elements (including the new element to be added to the ArrayList).

Performance of ArrayList

The `add` operation runs in amortized constant time, that is, adding n elements requires `O(n)` time. All of the other operations run in linear time.